Digital Architecture Engineering

Donald Scott

DIGITAL ARCHITECTURE ENGINEERING

DIGITAL ARCHITECTURE ENGINEERING

Donald Scott

BIBLIOTEX
Digital Library

Digital Architecture Engineering
by Donald Scott

Ebook ISBN: 9781984663979

Published by:
Bibliotex
Canada
Website: www.bibliotex.com

Contents

1

Computer Architecture

To understand digital signal processing systems, we must understand a little about how computers compute. The modern definition of a *computer* is an electronic device that performs calculations on data, presenting the results to humans or other computers in a variety of (hopefully useful) ways.

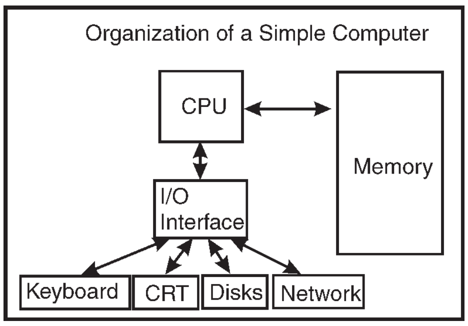


Fig. Generic Computer Hardware Organization.

The generic computer contains *input* devices (keyboard, mouse, A/D (analog-to-digital) converter, etc.), a *computational unit*, and output devices (monitors, printers, D/A converters). The computational unit is the computer's heart, and usually consists of a *central processing unit* (CPU), a *memory*, and an input/output (I/O) interface. What I/O devices might be present on a given computer vary greatly.

A simple computer operates fundamentally in discrete time: Computers are *clocked* devices, in which computational steps occur periodically according to ticks of a clock. This description belies clock speed: When you say "I have a 1 GHz computer," you mean that your computer takes 1 nanosecond to perform each step. That is incredibly fast! A "step" does not, unfortunately, necessarily mean a computation like an addition; computers break such computations down into several stages, which means that the clock speed need not express the computational speed. Computational speed is expressed in units of millions of instructions/second (Mips). Your 1 GHz computer (clock speed) may have a computational speed of 200 Mips.

Computers perform integer (discrete-valued) computations: Computer calculations can be numeric (obeying the laws of arithmetic), logical (obeying the laws of an algebra), or symbolic (obeying any law you like). Each computer instruction that performs an elementary numeric calculation — an addition, a multiplication, or a division — does so only for integers. The sum or product of two integers is also an integer, but the quotient of two integers is likely to not be an integer. How does a computer deal with numbers that have digits to the right of the decimal point? This problem is addressed by using the so-called *floating-point* representation of real numbers. At its heart, however, this representation relies on integer-valued computations.

Representing Numbers

Focusing on numbers, all numbers can represented by the *positional notation system.* 2 The *b*-ary positional representation system uses the position of digits ranging from 0 to *b*-1 to denote a number. The quantity *b* is known as the *base* of the number system. Mathematically, positional systems represent the positive integer *n* as

$$\forall d_k, d_k \in <apply>\{0,\ldots,b-1\}.</apply>:$$
$$($$
$$n = \sum_{k=0}^{\infty} \left(d_k b^k\right)$$
$$)$$
$$\forall d_k, d_k \in <apply>\{0,\ldots,b-1\}.</apply>:$$
$$($$
$$f = \sum_{k=-\infty}^{-1} \left(d_k b^k\right)$$
$$)$$

and we succinctly express *n* in base-*b* as $n_b = d_N d_{N'1} \ldots d_0$. The number 25 in base 10 equals $2 \times 10^1 + 5 \times 10^0$, so that the*digits* representing this number are d_0=5, d_1=2, and all other d_k equal zero. This same number in *binary* (base 2) equals 11001 ($1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$) and 19 in hexadecimal (base 16). Fractions between zero and one are represented the same way.

All numbers can be represented by their sign, integer and fractional parts. Complex numbers can be thought of as two real numbers that obey special rules to manipulate them.

Humans use base 10, commonly assumed to be due to us having ten fingers. Digital computers use the base 2 or *binary* number representation, each digit of which is known as a *bit* (*b*inary dig*it*).

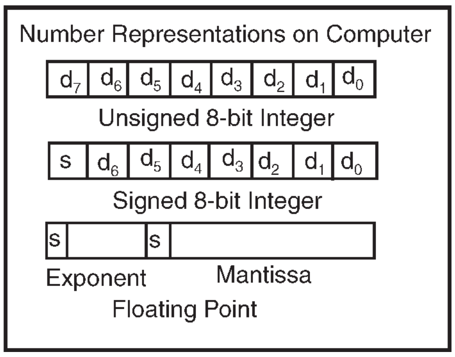


Fig. The Various ways Numbers are Represented in binary are Illustrated. The Number of Bytes for the Exponent and Mantissa Components of floating Point Numbers Varies.

Here, each bit is represented as a voltage that is either "high" or "low," thereby representing "1" or "0," respectively. To represent signed values, we tack on a special bit—the *sign bit*—to express the sign. The computer's memory consists of an ordered sequence of *bytes*, a collection of eight bits. A byte can therefore represent an unsigned number ranging from 0 to255. If we take one of the bits and make it the sign bit, we can make the same byte to represent numbers ranging from "128to 127. But a computer cannot represent *all* possible real numbers. The fault is not with the binary number system; rather having only a finite number of bytes is the problem. While a gigabyte of memory may seem to be a lot, it takes an infinite number of bits to represent *ð*. Since we want to store many numbers in a computer's memory, we are restricted to those that have a *finite*binary representation.

Large integers can be represented by an ordered sequence of bytes. Common lengths, usually expressed in terms of the number of bits, are 16, 32, and 64. Thus, an unsigned 32-bit number can represent integers ranging between 0 and$2^{32"}$1 (4,294,967,295), a number almost big enough to enumerate every human in the world.

Computer Arithmetic and Logic

The binary addition and multiplication tables are:

(

0 + 0 = 0

0 + 1= 1

1 + 1 = 10

1 + 0 = 1

0 × 0 = 0

0 × 1 = 0

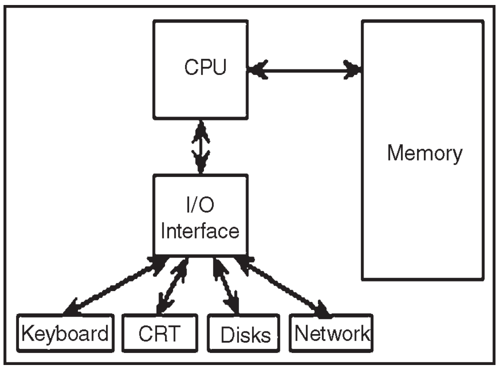
1 × 1 = 1

1 × 0 = 0

)

Note that if carries are ignored, subtraction of two single-digit binary numbers yields the same bit as addition. Computers use high and low voltage values to express a bit, and an array of such voltages express numbers akin to positional notation. Logic circuits perform arithmetic operations.

Buses and Architecture



Bus

A set of parallel conductors, which allow devices attached to it to communicate with the CPU.

The bus consists of three main parts:

- Control lines
- Address lines
- Data lines

Control Lines

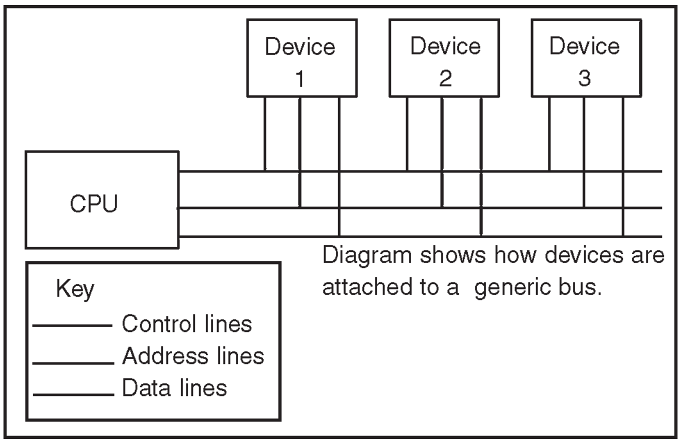
These allow the CPU to control which operations the devices attached should perform, *I.E.* read or write.

Address Lines

Allows the CPU to reference certain (Memory) locations within the device.

Data Lines

The meaningful data which is to be sent or retrieved from a device is placed on to these lines.



The Bus is set to run at a specified speed which is measured in MHz.

Types of Buses

Expansion Bus

Expansion buses (sometimes called *peripheral buses*) are buses that have connectors that allow you to add expansion cards (peripherals) to a computer.

There are different types of standard internal buses that are characterized by:

- Their shape
- The number of connector pins
- The type of signals (frequency, data, etc.)

ISA Bus

The original version of the ISA bus (*Industry Standard Architecture*) that appeared in 1981 with PC XT was an 8-bit bus with a clock speed of 4.77 MHz. In 1984, with the appearance of PC AT (the *Intel 286* processor), the bit was expanded into a 16-bit bus and the clock speed went from 6 to 8 MHz and finally to 8.33 MHz, offering a maximum transfer rate of 16 Mb/s (in practice only 8 Mb/s because one cycle out of every two was used for addressing).

The ISA bus permitted bus mastering, *i.e.* it enabled controllers connected directly to the bus to communicate directly with the other peripherals without going through the processor. One of the consequences of *bus mastering* is direct memory access (DMA). However, the ISA bus only allows hardware to address the first 16 megabytes of RAM.

Up until the end of the 1990s, almost all PC computers were equipped with the ISA bus, but it was progressively replaced by the PCI bus, which offered a better performance.

Fig. 8-bit ISA Connector

Fig.16-bit ISA Connector

MCA Bus

The MCA bus (*Micro Channel Architecture*) is an improved proprietary bus designed by IBM in 1987 to be used in their PS/2 line of computer. This 16 to 32-bit bus was incompatible with the ISA bus and could reach a throughput of 20 Mb/s.

EISA Bus

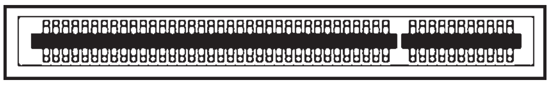
The EISA bus (*Extended Industry Standard Architecture*) was developed in 1988 by a consortium of companies (AST, Compaq, Epson, Hewlett-Packard, NEC, Olivetti, Tandy, Wyse and Zenith) in order to compete with the MCA proprietary bus that was launched by IBM the previous year. The EISA bus used connectors that were the same size as the ISA connector but with 4 rows of contacts instead of 2, for 32-bit addressing. The EISA connectors were deeper and the additional rows of contacts were placed below the rows of ISA contacts. Thus, it was possible to plug an ISA expansion board into an EISA connector. However, they did not plug as deep into the connector (because of the bezels) and thus only used the top rows (ISA) of contacts.

Local Bus

Traditional I/O buses, such as ISA, MCA our EISA buses, are directly connected to the main bus and there are forced to work at the same frequency. However, some I/O peripherals need a very low bandwidth while other need higher bandwidths. Therefore there are bottlenecks on the bus. In order to solve this problem, the "local bus" architecture offers to take advantage of the system bus, or front side bus (*FSB*), by interfacing directly with it.

VLB Bus

In 1992, the VESA local bus (VLB) was developed by the *VESA* (*Video Electronics Standard Association*under the aegis of the company *NEC*) in order to offer a local bus dedicated to graphics systems. The VLB is a 16-bit ISA connector with an added 16-bit connector:



The VLB bus is a 32-bit bus initially intended to work a bandwidth of 33 MHz (the bandwidth of the first PC 486s at that time). The VESA local bus was used on the following 486 models (40 and 50 MHz, respectively) as well as on the very first Pentium processors, but it was quickly replaced by the PCI bus.

Registers

Registers are fast memory, almost always connected to circuitry that allows various arithmetic, logical, control, and other manipulations, as well as possibly setting internal flags.

Most early computers had only one data register that could be used for arithmetic and logic instructions. Often there would be additional special purpose registers set aside either for temporary fast internal storage or assigned to logic circuits to implement certain instructions. Some early computers had one or two address registers that pointed to a memory location for memory accesses (a pair of address registers typically would act as source and destination pointers for memory operations). Computers soon had multiple data registers, address registers, and sometimes other special purpose registers. Some computers have general purpose registers that can be used for both data and address operations. Every digital computer using a von Neumann architecture has a register (called the program counter) that points to the next executable instruction. Many computers have additional control registers for implementing various control capabilities. Often some or all of the internal flags are combined into a flag or status register.

Accumulators

Accumulators are registers that can be used for arithmetic, logical, shift, rotate, or other similar operations. The first computers typically only had one accumulator. Many times there were related special purpose registers that contained the source data for an accumulator. Accumulators were replaced with data registers and general purpose registers. Accumulators reappeared in the first microprocessors.

- *Intel* 8086/80286: One word (16 bit) accumulator; named AX (high order byte of the AX register is named AH and low order byte of the AX register is named AL)
- *Intel* 80386: One double word (32 bit) accumulator; named EAX (low order word uses the same names as the accumulator on the Intel 8086 and 80286 [AX] and low order and high order bytes of the low order words of four of the registers use the same names as the accumulator on the Intel 8086 and 80286 [AH and AL])

- *Mix*: One accumulator; named A-register; five bytes plus sign

Data registers

Data registers are used for temporary scratch storage of data, as well as for data manipulations (arithmetic, logic, etc.). In some processors, all data registers act in the same manner, while in other processors different operations are performed are specific registers.

- *Mix*: One extension register; named X-register; five bytes plus sign; can be concatenated on the right hand side of the A-register (accumulator)
- *Motorola* 680 × 0, 68300: 8 long word (32 bit) data registers; named D0, D1, D2, D3, D4, D5, D6, and D7

Address registers

Address registers store the addresses of specific memory locations. Often many integer and logic operations can be performed on address registers directly (to allow for computation of addresses).

Sometimes the contents of address register(s) are combined with other special purpose registers to compute the actual physical address. This allows for the hardware implementation of dynamic memory pages, virtual memory, and protected memory.

The number of bits of an address register (possibly combined with information from other registers) limits the maximum amount of addressable memory. A 16-bit address register can address 64K of physical memory. A 24-bit address register can address address 16 MB of physical memory. A 32-bit address register can address 4 GB of physical memory. A 64-bit address register can address 1.8446744 x 10^{19} of physical memory. Addresses are always unsigned binary numbers.

- *Mix*: One jump registers; named J-register; two bytes and sign is always positive
- *Motorola* 680 × 0, 68300: 8 long word (32 bit) address registers; named A0, A1, A2, A3, A4, A5, A6, and A7 (also called the stack pointer)

General purpose registers

General purpose registers can be used as either data or address registers.

- *DEC Vax*: 16 word (32 bit) general purpose registers; named R0 through R15
- *IBM* 360/370: 16 full word (32 bit) general purpose registers; named 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A (or 10), B (or 11), C (or 12), D (or 13), E (or 14), and F (or 15)
- *Intel* 8086/80286: 8 word (16 bit) general purpose registers; named AX, BX, CX, DX, BP, SP, SI, and DI (high order bytes of the AX, BX, CX, and DX registers have the names AH, BH, CH, and DH and low order bytes of the AX, BX, CX, and DX registers have the names AL, BL, CL, and DL)
- *Intel* 80386: 8 double word (32 bit) general purpose registers; named EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI (low order words use the same names as the general purpose registers on the Intel 8086 and 80286 and low order and high order bytes of the low order words

of four of the registers use the same names as the general purpose registers on the Intel 8086 and 80286)

- *Motorola* 88100: 32 word (32 bit) general purpose registers; named r0 through r31

Constant registers

Constant registers are special read-only registers that store a constant. Attempts to write to a constant register are illegal or ignored. In some RISC processors, constant registers are used to store commonly used values (such as zero, one, or negative one) — for example, a constant register containing zero can be used in register to register data moves, providing the equivalent of a clear instruction without adding one to the instruction set. Constant registers are also often used in floating point units to provide such value as pi or e with additional hidden bits for greater accuracy in computations.

- *Motorola* 88100: r0 (general purpose register 0) contains the constant 32 bit integer zero

Floating point registers

Floating point registers are special registers set aside for floating point math.

Index registers

Index registers are used to provide more flexibility in addressing modes, allowing the programmer to create a memory address by combining the contents of an address register with the contents of an index register (with displacements, increments, decrements, and other options). In some processors, there are specific index registers (or just one index register) that can only be used only for that purpose.

In some processors, any data register, address register, or general register (or some combination of the three) can be used as an index register.

- *IBM* 360/370: Any of the 16 general purpose registers may be used as an index register
- *Intel* 80 × 86: 7 of the 8 general purpose registers may be used as an index register (the ESP is the exception)
- *Mix*: Five index registers; named I-registers I1, I2, I3, I4, and I5; five bytes plus sign
- *Motorola* 680 × 0, 68300: Any of the 8 data registers or the 8 address registers may be used as an index register

Base registers

Base registers or segment registers are used to segment memory. Effective addresses are computed by adding the contents of the base or segment register to the rest of the effective address computation. In some processors, any register can serve as a base register. In some processors, there are specific base or segment registers (one or more) that can only be used for that purpose. In some processors with multiple base or segment registers, each base or segment register is used for different kinds of memory accesses (such as a segment register for data accesses and a different segment register for program accesses).

- *IBM* 360/370: Any of the 16 general purpose registers may be used as a base register
- *Intel* 80 × 86: 6 dedicated segment registers: CS (code segment), SS (stack segment), DS (data segment), ES (extra segment, a second data segment register), FS

(third data segment register), and GS (fourth data segment register)

- *Motorola* 680 × 0, 68300: Any of the 8 address registers may be used as a base register

Control registers

Control registers control some aspect of processor operation. The most universal control register is the program counter.

Program counter

Almost every digital computer ever made uses a program counter. The program counter points to the memory location that stores the next executable instruction. Branching is implemented by making changes to the program counter. Some processor designs allow software to directly change the program counter, but usually software only indirectly changes the program counter (for example, a JUMP instruction will insert the operand into the program counter). An assembler has a location counter, which is an internal pointer to the address (first byte) of the next location in storage (for instructions, data areas, constants, etc.) while the source code is being converted into object code.

The VAX uses the 16th of 16 general purpose registers as the program counter (PC). Almost the entire instruction set can directly manipulate the program counter, allowing a very rich set of possible kinds of branching.

The program counter in System/360 and 370 machines is contained in bits 40-63 of the program status word (PSW), which is directly accessible by some instructions.

- *IBM* 360/370: Program counter is bits 40-63 of the program status word (PSW)
- *Intel* 8086/80286: 16-bit instruction pointer (IP)
- *Intel* 80386: 32-bit instruction pointer (EIP)
- *Motorola* 680 × 0, 68300: 32-bit program counter (PC)

Processor flags

Processor flags store information about specific processor functions. The processor flags are usually kept in a flag register or a general status register.

This can include result flags that record the results of certain kinds of testing, information about data that is moved, certain kinds of information about the results of compations or transformations, and information about some processor states.

Closely related and often stored in the same processor word or status register (although often in a privileged portion) are control flags that control processor actions or processor states or the actions of certain instructions.

- *IBM* 360/370: Program status word (PSW)
- *Intel* 8086/80286: 16-bit flag register (FLAGS); system flags, control flag, and status flags)
- *Intel* 80386: 32-bit flag register (EFLAGS); system flags, control flag, and status flags)
- *Mix*: An overflow toggle and a comparison indicator
- *Motorola* 680 × 0, 68300: 16-bit status register (SR); high byte is system byte and requires privileged access, low byte is user byte or condition code register (CCR)

A few typical result flags (with processors that include them):

- Auxilary carry Set if a carry out of the most significant bit of a BCD operand occurs (binary coded decimal addition). Also commonly set if a borrow occurs in a BCD subtract. Used in Intel 80x86 [AF].

- Carry Set if a carry out of the most significant bit of an operand occurs (addition). Also commonly set if a borrow occurs in a subtract. Used in Digital VAX [C], Intel 80x86 [CF], Motorola 680x0 [C], Motorola 68300 [C], Motorola M68HC16 [C].
- Comparison indicator contains one of three values: less, equal, or greater. Used in MIX.
- Extend Set to the value of the carry bit for arithmetic operations (used to support implementation of multi-byte arithmetic larger than that implemented directly by the hardware. Used in Motorola 680x0 [X], Motorola 68300 [X].
- Half carry Set if a carry out of bit 3 of an operand occurs during BCD addition. Used in Motorola M68HC16.
- Negative Set if the most significant bit of a result is set. Used in Digital VAX [N], Motorola 680x0 [N], Motorola 68300 [N], Motorola M68HC16 [N].
- Overflow Set if arithmetic overflow occurs. Used in Digital VAX [V], Intel 80x86 [OF], Motorola 680x0 [V], Motorola 68300 [V], Motorola M68HC16 [V].
- Overflow toggle a single bit that is either on or off. Used in MIX.
- Parity For odd parity machines, set to make an odd number of one bits; for an even parity machine, set to make an even number of one bits. Used in Intel 80x86 [PF]. The IBM 360/370 has odd parity on memory.

- Sign Set for negative sign. Used in Intel 80x86 [SF].
- Trap Set for traps. Used in Intel 80x86 [TF].
- Zero Set if a result equals zero. Used in Digital VAX [Z], Intel 80x86 [ZF], Motorola 680x0 [Z], Motorola 68300 [Z], Motorola M68HC16 [Z].

Some conditions are determined by combining multiple flags. For example, if a processor has a negative flag and a zero flag, the equivalent of a positive flag is the case of both the negative and zero flags both simultaneously being cleared.

A few typical control flags (with processors that include them):

- Decimal overflow trap enable Set if decimal overflow occurs (or conversion error on a VAX). Used in Digital VAX [DV].
- Direction flag Determines the direction of string operations (set for autoincrement, cleared for autodecrement). Used in Intel 80x86 [DF].
- Floating underflow trap enable Set if floating underflow occurs. Used in Digital VAX [FU].
- Integer overflow trap enable Set if integer overflow occurs (or conversion error on a VAX). Used in Digital VAX [IV].
- Interupt enable Set if interrupts enabled. Used in Intel 80x86 [IF].
- i/o privilege level Used to control access to I/O instructions and hardware (thereby seperating control over I/O from other supervisor/user states). Two bits. Used in Intel 80x86 [IO PL].
- Nested task flag Used in Intel 80x86 [NF].
- Resume flag Used in Intel 80x86 [RF].

- Virtual 8086 mode Used to switch to virtual 8086 emulation. Used in Intel 80x86 [VM].

Stack pointer

Stack pointers are used to implement a processor stack in memory. In many processors, address registers can be used as generic data stack pointers and queue pointers. A specific stack pointer or address register may be hardwired for certain instructions. The most common use is to store return addresses, processor state information, and temporary variables for subroutines.

- IBM 360/370: any of the 16 general purpose registers may be used as a stack pointer
- Intel 8086/80286: dedicated stack pointer (SP) combined with stack segment pointer (SS) to create address of stack
- Intel 80386: dedicated stack pointer (ESP) combined with stack segment pointer (SS) and the stack-frame base pointer (EBP) to create address of stack
- Motorola 680x0, 68300: dedicated user stack pointer (USP, A7) and system stack pointer (SSP, A7) for implicit stack pointer operations, as well as allowing any of the 8 address registers to be as explicit stack pointers.

ARCHITECTURE AND STRUCTURE

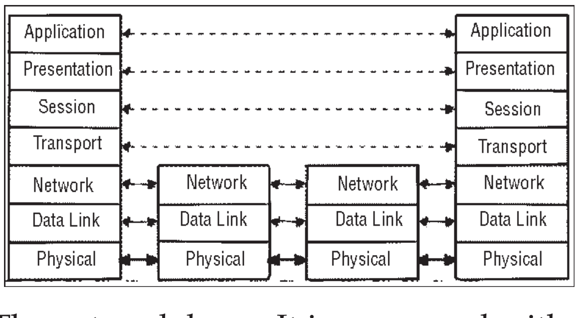
The importance of having standardised networking protocols have led to the evolution of several networking reference models. The Open System Interconnection (OSI) and the TCP/IP models will be looked at. These models allow us to more easily study the structure of networks.

OSI Reference Model

The OSI model is based on a proposal by the International Standards Organisation. The OSI reference model has a layered architecture, whereby each layer encapsulates a function of the overall complex task. The proposed model has 7 layers. The key idea is that although data is transmitted vertically, each layer is programmed horizontally. Each layer ignores the complications of the lower layers and assumes that the information it passes out is immediately seen by destination host. At each layer, appropriate headers are appended onto the data when sending, and are removed as the data is decoded. Since the headers from the lower levels are removed before the next higher level looks at the data, the concept of horizontal coding is preserved.

Layer 1: The physical layer, and is concerned with transmitting raw bits over a communication channel.

Layer 2: The data link layer, handles error correction and flow regulation.



Layer 3: The network layer. It is concerned with the operation of the subnet, that is, it determines how packets are routed from source to destination. Routes can be based on hardwired static tables, or can be highly dynamic, where each packet is assigned a different route through the network, depending on the network load. In broadcast networks, routing is simple, and hence the network layer is often thin or non-existent.

Layer 4: The transport layer. Data accepted from the session layer is split into smaller units if need be and passed to the network layer. It is also in charge of ensuring that the information arrives correctly on the other end. Connections across the network, are established and deleted from this layer. There are mechanisms to regulate flow of information between fast and slow host machines. This flow control is distinct from the flow control talked about in layer 2.

Layer 5: The session layer implements a set of rules for establishing and terminating data streams between hosts. Services provided include dialogue control(full of half duplex), token management and other end to end data control.

Layer 6: The presentation layer, and it performs certain functions requested sufficiently often enough to warrant finding a general solution for them, rather than letting each user solve them. It manages between different abstract data types like ASCII, Unicode, integer representation, one's and two's complements etc. This allows computers with different data representations to communicate.

Layer 7: The application layer and is charged with translating between different applications. For example, for a screen editor to work over a network, it has to know the different escape codes of different terminals.

Similarly with FTP. Different systems have different file naming conventions, different ways of representing text. The applications layer takes care of these complications.

The TCP/IP model was born out of the U.S. Department of Defence ARPANET. It consist of 4 layers and is named after the two protocols used.

- *Layer 4 is the application layer*: The TCP/IP model does not have a session or presentation layer. No need for them were perceived. Hence the application layer here corresponds to layers 5-7 in the OSI model. It contains higher protocols such as FTP, TELNET, HTTP etc.
- *Layer 3, the transport layer corresponds to the layer 4 in the OSI model*: The transport layer was designed to allow peer entities on the source and destination hosts to establish a conversation. Two end to end protocols are defined here. Transmission Control Protocol (TCP) is a reliable connection oriented protocol that allows a byte stream originating from one machine to de delivered without error to any other machine in the Internet. Incoming messages are fragmented into discrete messages before being transmitted. At the destination, TCP reassembles the received messages into the output stream. The second protocol, User Datagram Protocol (UDP) is an unreliable connectionless protocol. It is used for applications that do not want TCP's sequencing or flow control and wish to implement their own. It is widely used for one shot, client server type request-reply queries and applications in which prompt delivery is more important than accurate delivery. *E.g.* speech or video streams.

- *Layer 2 is the Internet layer*: The requirement that the network should survive a loss of subnet hardware, led to the choice of a packet switching network based on a connectionless layer. The job of the Internet layer is to permit hosts to inject packets into the network and have them travel independently to the destination. The Internet layer defines an official format and protocol called Internet Protocol (IP). The primary job here is one of packet routing and avoidance of congestion. It is very similar to the OSI network layer.
- *The final layer is the host to network layer*: The TCP/IP model does not say much about what occurs here, except to point out that the host has to connect to the network using some protocol so it can send IP packets to the network.

Hardware Components

Here we will examine the different hardware that makes up a network. The most basic hardware that makes up a network is the transmission medium. The oldest and most common transmission medium is twisted pair wires. It consists of two insulated copper wires, typically about 1mm thick. The purpose of twisting is to reduce electrical interference.

The bandwidth depends on the thickness of the wires and distance travelled, but several megabytes/sec can be achieved for a few kilometres. Twisted pair cabling comes in several varieties, one of which is the Unshielded Twisted Pair (UTP) category 5 cabling. Category 5 UTP wires have more twists per centimetre than other twisted pair cabling, and has Teflon insulation. This results in less cross talk and a better quality signal over longer distances.

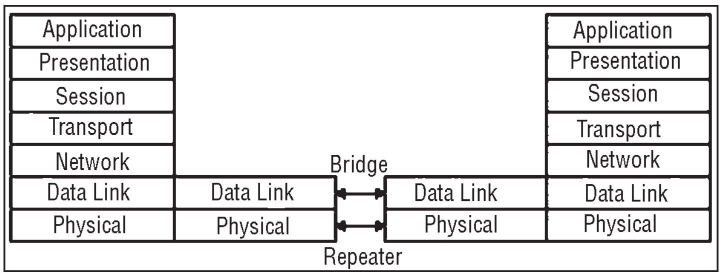
Coaxial cables are another popular form of transmission medium. It has better shielding than UTP so it can span for longer distances and at higher speeds. A coax cable is made up of a copper core, covered by a layer of insulating material, a braided outer conductor and finally a protective plastic covering. For 1 kilometre cables, bandwidths of 1 to 2 Gbps are feasible.

Fibre optic cables have a glass core centre, from which light propagates. It is next enclosed by a cladding of glass with a lower index of refraction than the core. This keeps the light within the core. A plastic jacket is the last layer and is to protect the cladding. Fibres are typically grouped into bundles, protected by another outer sheath.

The achievable bandwidth (about 1 Gbps) of fibre is current restricted by the ability to convert between electrical to optical signals. Speeds of 100Gbps are feasible and terabit performance is just a few years down the road. Fibre has many advantages over copper. It can handle higher bandwidths, due to lower attenuation, it only requires repeaters every 30 km as opposed to 5 km for copper. Fibre is also not affected by power surges, electromagnetic interference or power failures. Fibre does not leak light and thus is difficult to tap into. Fibre is light weight and its lower installation cost makes it a good choice over copper.

A repeater operates at the physical layer. It regenerates a signal received on one cable segment and retransmits it on another cable segment. A repeater can be used to extend the coverage of a network, and also to connect networks. Although a network can be extended with a repeater, a network is still constrained by the maximum permissible length of that LAN. Since a repeater functions at the physical layer, it is transparent to data flow and hence is limited to connecting two similar networks. Repeaters also cause problems with traffic.

If two networks are connected via a repeater, all messages from one network is passed to the other network, regardless of the intended recipient. If implemented without knowledge concerning the traffic flow on each LAN, performance problems will arise.



Bridges are more intelligent than repeaters. A bridge connects at the data link layer, where it can examine each frame that passes through it.

The bridge looks at the destination address of the incoming frame; if the destination is for the network across the bridge, it translates the frame, and retransmits it on the other network. If the frame is for a destination from the source network, the bridge repeats back the frame. There are typically two types of bridges. Transparent bridge and translating bridge. A transparent bridge connects two networks that employ the same protocol at the data link layer, whereas a translating bridge provides a connection capability between two networks that have differing protocols at the data link layer.

Routers operate one level higher, at the network layer. By operating at the network layer, routers are able to make decisions about how packets are routed between networks. Although multiported bridges can be said to have routing capabilities, this is usually for one point to point link within a network. A router has the ability to fragment and dynamically re-route the message across networks, making use of the most efficient paths. Routers are known to workstations that use their service, hence packets can be sent directly to routers. This means that routers do not have to examine in detail every packet it receives, and this makes them more efficient. However, the higher functionality of routers over bridges means that on average, routers perform a half to two thirds more processing over bridges.

Gateways function through all layers. Essentially, gateways perform protocol translation between networks. Gateways are generally designed and used for LAN-WAN connections and not for inter LAN communications.

The idea of intelligent hubs came about because running cables all over the building made it hard to configure and repair networks. Instead, collapse the LAN topology and put it in a box; terminate all devices in the network at the box using separate wires for each device. This centralised location makes it more convenient for network configuration, management and monitoring. Intelligent hubs are a level above normal hubs and wiring cabinets. Intelligent hubs are becoming key network management points as functionality such as bridges, routers and servers are built into them. Some intelligent hubs provide remote management capabilities that makes it easier to diagnose problems at distant locations and isolate faults from the rest of the network.

2

Internet Architecture

Introduction

What is the *Internet* architecture? It is by definition a meta-network, a constantly changing collection of thousands of individual networks intercommunicating with a common protocol.

The Internet's architecture is described in its name, a short from of the compound word "inter-networking". This architecture is based in the very specification of the standard *TCP/IP* protocol, designed to connect any two networks which may be very different in internal hardware, software, and technical design. Once two networks are interconnected, communication with TCP/IP is enabled end-to-end, so that any node on the Internet has the near magical ability to communicate with any other no matter where they are. This openness of design has enabled the Internet architecture to grow to a global scale. In practice, the Internet technical architecture looks a bit like a multi-dimensional river system, with small tributaries feeding medium-sized streams feeding large rivers. For example, an individual's access to the Internet is often from home over a modem to a local Internet service provider who connects to a regional network connected to a national network. At the office, a desktop computer might be connected to a local area network with a company connection to a corporate Intranet connected to several national Internet service providers.

In general, small local Internet service providers connect to medium-sized regional networks which connect to large national networks, which then connect to very large bandwidth networks on the Internet *backbone*. Most Internet service providers have several redundant network cross-connections to other providers in order to ensure continuous availability.

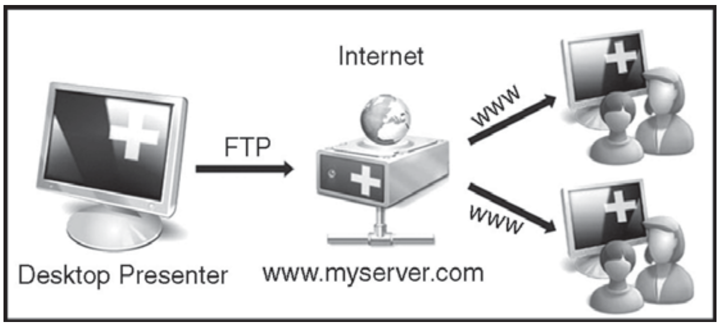
The companies running the Internet backbone operate very high bandwidth networks relied on by governments, corporations, large organizations, and other Internet service providers. Their technical infrastructure often includes global connections through underwater cables and satellite links to enable communication between countries and continents.

As always, a larger scale introduces new phenomena: the number of packets flowing through the switches on the backbone is so large that it exhibits the kind of complex non-linear patterns usually found in natural, analog systems like the flow of water or development of the rings of Saturn (RFC 3439, S2.2).

Each communication *packet* goes up the hierarchy of Internet networks as far as necessary to get to its destination network where local *routing* takes over to deliver it to the addressee.

In the same way, each level in the hierarchy pays the next level for the bandwidth they use, and then the large backbone companies settle up with each other. Bandwidth is priced by large Internet service providers by several methods, such as at a fixed rate for constant availability of a certain number of megabits per second, or by a variety of use methods that amount to a cost per gigabyte. Due to economies of scale and efficiencies in management, bandwidth cost drops dramatically at the higher levels of the architecture.

Definition of Internet



The Internet is a global network of computers. Every computer that is connected to the Internet is considered a part of that network. This means even your home computer. It's all a matter of degrees, you connect to your ISP's network, then your ISP connects to a larger network and so on. At the top of the tree is the high-capacity backbones, all of these interconnect at 'Network Access Points' 'NAPs' at important regions around the world. The entire Internet is based on agreements between these backbone providers who set in place all the fibre optics lines and other technical aspects of the Internet. The first high speed backbone was created by the 'National Science Foundation' in 1987.

The Internet was first created by the Advanced Research Projects Agency (ARPA) of the U.S. government in 1960's, and was first known as the ARPANet. At this stage the Internet's first computers were at academic and government institutions. They were mainly used for accessing files and to send email. From 1983 onwards the Internet as we know it today started to form with the introduction of the communication protocol TCP/IP to ARPANet.

Since 1983 the Internet has accommodated alot of changes and continues to keep developing. The last two decades has seen the Internet accommodate such things as network LANs and ATM and frame switched services. The Internet continues to evolve with it becoming available on mobile phones and pagers and possibly on televisions in the future. The actual term "Internet" was finally defined in 1995 by FNC (The Federal Networking Council). The resolution created by the The Federal Networking Council (FNC) agrees that the following language reflects our definition of the term "Internet". "Internet" refers to the global information system that,

The Evolution of the Internet

The underpinnings of the Internet are formed by the global interconnection of hundreds of thousands of otherwise independent computers, communications entities and information systems. What makes this interconnection possible is the use of a set of communication standards, procedures and formats in common among the networks and the various devices and computational facilities connected to them. The procedures by which computers communicate with each other are called “protocols.” While this infrastructure is steadily evolving to include new capabilities, the protocols initially used by the Internet are called the “TCP/IP” protocols, named after the two protocols that formed the principal basis for Internet operation.

On top of this infrastructure is an emerging set of architectural concepts and data structures for heterogeneous information systems that renders the Internet a truly global information system.

In essence, the Internet is an architecture, although many people confuse it with its implementation. When the Internet is looked at as an architecture, it manifests two different abstractions. One abstraction deals with communications connectivity, packet delivery and a variety of end-end communication services. The other abstraction deals with the Internet as an information system, independent of its underlying communications infrastructure, which allows creation, storage and access to a wide range of information resources, including digital objects and related services at various levels of abstraction.

Interconnecting computers is an inherently digital problem. Computers process and exchange digital information, meaning that they use a discrete mathematical “binary” or “two-valued” language of 1s and 0s. For communication purposes, such information is mapped into continuous electrical or optical waveforms.

The use of digital signaling allows accurate regeneration and reliable recovery of the underlying bits. We use the terms "computer," "computer resources" and "computation" to mean not only traditional computers, but also devices that can be controlled digitally over a network, information resources such as mobile programs and other computational capabilities.

The telephone network started out with operators who manually connected telephones to each other through "patch panels" that accepted patch cords from each telephone line and electrically connected them to one another through the panel, which operated, in effect, like a switch. The result was called circuit switching, since at its conclusion, an electrical circuit was made between the calling telephone and the called telephone. Conventional circuit switching, which was developed to handle telephone calls, is inappropriate for connecting computers because it makes limited use of the telecommunication facilities and takes too long to set up connections. Although reliable enough for voice communication, the circuit-switched voice network had difficulty delivering digital information without errors.

For digital communications, packet switching is a better choice, because it is far better suited to the typically "burst" communication style of computers. Computers that communicate typically send out brief but intense bursts of data, then remain silent for a while before sending out the next burst. These bursts are communicated as packets, which are very much like electronic postcards.

The postcards, in reality packets, are relayed from computer to computer until they reach their destination. The special computers that perform this forwarding function are called variously “packet switches” or “routers” and form the equivalent of many bucket brigades spanning continents and oceans, moving buckets of electronic postcards from one computer to another. Together these routers and the communication links between them form the underpinnings of the Internet.

Without packet switching, the Internet would not exist, as we now know it. Going back to the postcard analogy, postcards can get lost. They can be delivered out of order, and they can be delayed by varying amounts. The same is true of Internet packets, which, on the Internet, can even be duplicated. The Internet Protocol is the postcard layer of the Internet. The next higher layer of protocol, TCP, takes care of re-sending the “postcards” to recover packets that might have been lost, and putting packets back in order if they have become disordered in transit.

Of course, packet switching is about a billion times faster than the postal service or a bucket brigade would be. It also has to operate over many different communications systems, or substrata. The authors designed the basic architecture to be so simple and undemanding that it could work with most communication services. Many organizations, including commercial ones, carried out research using the TCP/IP protocols in the 1970s. Email was steadily used over the nascent Internet during that time and to the present. It was not until 1994 that the general public began to be aware of the Internet by way of the World Wide Web application, particularly after Netscape Communications was formed and released its browser and associated server software.

Thus, the evolution of the Internet was based on two technologies and a research dream. The technologies were packet switching and computer technology, which, in turn, drew upon the underlying technologies of digital communications and semiconductors. The research dream was to share information and computational resources. But that is simply the technical side of the story. Equally important in many ways were the other dimensions that enabled the Internet to come into existence and flourish. This aspect of the story starts with cooperation and far-sightedness in the U.S. Government, which is often derided for lack of foresight but is a real hero in this story.

It leads on to the enthusiasm of private sector interests to build upon the government funded developments to expand the Internet and make it available to the general public. Perhaps most important, it is fueled by the development of the personal computer industry and significant changes in the telecommunications industry in the 1980s, not the least of which was the decision to open the long distance market to competition.

The role of workstations, the Unix operating system and local area networking (especially the Ethernet) are themes contributing to the spread of Internet technology in the 1980s into the research and academic community from which the Internet industry eventually emerged.

Many individuals have been involved in the development and evolution of the Internet covering a span of almost four decades if one goes back to the early writings on the subject of computer networking by Kleinrock, Licklider, Baran, Roberts, and Davies. The ARPANET, described below, was the first wide-area computer network. The NSFNET, which followed more than a decade later under the leadership of Erich Bloch, Gordon Bell, Bill Wulf and Steve Wolff, brought computer networking into the mainstream of the research and education communities. It is not our intent here to attempt to attribute credit to all those whose contributions were central to this story, although we mention a few of the key players

Computer Network Hierarchy

Every computer that is connected to the Internet is part of a network, even the one in your home. For example, you may use a modem and dial a local number to connect to an Internet Service Provider (ISP). At work, you may be part of a local area network (LAN), but you most likely still connect to the Internet using an ISP that your company has contracted with. When you connect to your ISP, you become part of their network. The ISP may then connect to a larger network and become part of their network. The Internet is simply a network of networks.

Most large communications companies have their own dedicated backbones connecting various regions. In each region, the company has a Point of Presence (POP). The POP is a place for local users to access the company's network, often through a local phone number or dedicated line. The amazing thing here is that there is no overall controlling network. Instead, there are several high-level networks connecting to each other through Network Access Points or NAPs.

Internet Network Example

Here's an example. Imagine that Company A is a large ISP. In each major city, Company A has a POP. The POP in each city is a rack full of modems that the ISP's customers dial into. Company A leases fibre optic lines from the phone company to connect the POPs together.

Imagine that Company B is a corporate ISP. Company B builds large buildings in major cities and corporations locate their Internet server machines in these buildings. Company B is such a large company that it runs its own fibre optic lines between its buildings so that they are all interconnected.

In this arrangement, all of Company A's customers can talk to each other, and all of Company B's customers can talk to each other, but there is no way for Company A's customers and Company B's customers to intercommunicate. Therefore, Company A and Company B both agree to connect to NAPs in various cities, and traffic between the two companies flows between the networks at the NAPs.

In the real Internet, dozens of large Internet providers interconnect at NAPs in various cities, and trillions of bytes of data flow between the individual networks at these points. The Internet is a collection of huge corporate networks that agree to all intercommunicate with each other at the NAPs. In this way, every computer on the Internet connects to every other.

The Function of an Internet Router

All of these networks rely on NAPs, backbones and routers to talk to each other. What is incredible about this process is that a message can leave one computer and travel halfway across the world through several different networks and arrive at another computer in a fraction of a second!

The routers determine where to send information from one computer to another.

Routers are specialized computers that send your messages and those of every other Internet user speeding to their destinations along thousands of pathways. A router has two separate, but related, jobs:

- It ensures that information doesn't go where it's not needed. This is crucial for keeping large volumes of data from clogging the connections of "innocent bystanders."
- It makes sure that information does make it to the intended destination.

In performing these two jobs, a router is extremely useful in dealing with two separate computer networks. It joins the two networks, passing information from one to the other. It also protects the networks from one another, preventing the traffic on one from unnecessarily spilling over to the other. Regardless of how many networks are attached, the basic operation and function of the router remains the same. Since the Internet is one huge network made up of tens of thousands of smaller networks, its use of routers is an absolute necessity.

Internet Backbone

The National Science Foundation (NSF) created the first high-speed backbone in 1987. Called NSFNET, it was a T1 line that connected 170 smaller networks together and operated at 1.544 Mbps (million bits per second). IBM, MCI and Merit worked with NSF to create the backbone and developed a T3 (45 Mbps) backbone the following year.

Backbones are typically fibre optic trunk lines. The trunk line has multiple fibre optic cables combined together to increase the capacity. Fibre optic cables are designated OC for optical carrier, such as OC-3, OC-12 or OC-48. An OC-3 line is capable of transmitting 155 Mbps while an OC-48 can transmit 2,488 Mbps (2.488 Gbps). Compare that to a typical 56K modem transmitting 56,000 bps and you see just how fast a modern backbone is.

Today there are many companies that operate their own high-capacity backbones, and all of them interconnect at various NAPs around the world. In this way, everyone on the Internet, no matter where they are and what company they use, is able to talk to everyone else on the planet. The entire Internet is a gigantic, sprawling agreement between companies to intercommunicate freely.

Internet Protocol IP Addresses

Every machine on the Internet has a unique identifying number, called an IP Address. The IP stands for Internet Protocol, which is the language that computers use to communicate over the Internet. A protocol is the pre-defined way that someone who wants to use a service talks with that service. The "someone" could be a person, but more often it is a computer program like a Web browser.

A typical IP address looks like this:

To make it easier for us humans to remember, IP addresses are normally expressed in decimal format as a *dotted decimal number* like the one above. But computers communicate in binary form. Look at the same IP address in binary:

The four numbers in an IP address are called octets, because they each have eight positions when viewed in binary form. If you add all the positions together, you get 32, which is why IP addresses are considered 32-bit numbers. Since each of the eight positions can have two different states (1 or zero), the total number of possible combinations per octet is 2^8 or 256. So each octet can contain any value between zero and 255. Combine the four octets and you get 2^{32} or a possible 4,294,967,296 unique values!

Out of the almost 4.3 billion possible combinations, certain values are restricted from use as typical IP addresses. For example, the IP address 0.0.0.0 is reserved for the default network and the address 255.255.255.255 is used for broadcasts.

The octets serve a purpose other than simply separating the numbers. They are used to create classes of IP addresses that can be assigned to a particular business, government or other entity based on size and need. The octets are split into two sections: Net and Host. The Net section always contains the first octet. It is used to identify the network that a computer belongs to. Host (sometimes referred to as Node) identifies the actual computer on the network. The Host section always contains the last octet. There are five IP classes plus certain special addresses.

Domain Name System

When the Internet was in its infancy, it consisted of a small number of computers hooked together with modems and telephone lines. You could only make connections by providing the IP address of the computer you wanted to establish a link with. For example, a typical IP address might be 216.27.22.162. This was fine when there were only a few hosts out there, but it became unwieldy as more and more systems came online.

The first solution to the problem was a simple text file maintained by the Network Information Centre that mapped names to IP addresses. Soon this text file became so large it was too cumbersome to manage. In 1983, the University of Wisconsin created the Domain Name System (DNS), which maps text names to IP addresses automatically.

URL: Uniform Resource Locator

When you use the Web or send an e-mail message, you use a domain name to do it. For example, the Uniform Resource Locator (URL) "http://www.yahoo.com" contains the domain name howstuffworks.com. So does this e-mail address: example@yahoo.com. Every time you use a domain name, you use the Internet's DNS servers to translate the human-readable domain name into the machine-readable IP address. Top-level domain names, also called first-level domain names, include.COM.ORG.NET.EDU and.GOV. Within every top-level domain there is a huge list of second-level domains.

Every name in the.COM top-level domain must be unique. The left-most word, like www, is the host name. It specifies the name of a specific machine (with a specific IP address) in a domain. A given domain can, potentially, contain millions of host names as long as they are all unique within that domain.

DNS servers accept requests from programs and other name servers to convert domain names into IP addresses.

When a request comes in, the DNS server can do one of four things with it:

- It can answer the request with an IP address because it already knows the IP address for the requested domain.
- It can contact another DNS server and try to find the IP address for the name requested. It may have to do this multiple times.
- It can say, "I don't know the IP address for the domain you requested, but here's the IP address for a DNS server that knows more than I do."
- It can return an error message because the requested domain name is invalid or does not exist.

A DNS Example

Let's say that you type the URL www.yahoo.com into your browser. The browser contacts a DNS server to get the IP address. A DNS server would start its search for an IP address by contacting one of the root DNS servers. The root servers know the IP addresses for all of the DNS servers that handle the top-level domains (.COM.NET.ORG, etc.). Your DNS server would ask the root for www.howstuffworks.com, and the root would say, "I don't know the IP address for www.howstuffworks.com, but here's the IP address for the.COM DNS server." Your name server then sends a query to the.COM DNS server asking it if it knows the IP address for www.howstuffworks.com. The DNS server for the COM domain knows the IP addresses for the name servers handling the www.yahoo.com domain, so it returns those.

Your name server then contacts the DNS server for www.yahoo.com and asks if it knows the IP address for www.yahoo.com. It actually does, so it returns the IP address to your DNS server, which returns it to the browser, which can then contact the server for www.yahoo.com to get a Web page.

One of the keys to making this work is redundancy. There are multiple DNS servers at every level, so that if one fails, there are others to handle the requests. The other key is caching. Once a DNS server resolves a request, it caches the IP address it receives.

Once it has made a request to a root DNS server for any.COM domain, it knows the IP address for a DNS server handling the.COM domain, so it doesn't have to bug the root DNS servers again for that information. DNS servers can do this for every request, and this caching helps to keep things from bogging down.

Even though it is totally invisible, DNS servers handle billions of requests every day and they are essential to the Internet's smooth functioning. The fact that this distributed database works so well and so invisibly day in and day out is a testimony to the design.

Internet Servers and Clients

Internet servers make the Internet possible. All of the machines on the Internet are either servers or clients. The machines that provide services to other machines are servers. And the machines that are used to connect to those services are clients. There are Web servers, e-mail servers, FTP servers and so on serving the needs of Internet users all over the world.

When you connect to www.yahoo.com to read a page, you are a user sitting at a client's machine. You are accessing the yahoo Web server. The server machine finds the page you requested and sends it to you. Clients that come to a server machine do so with a specific intent, so clients direct their requests to a specific software server running on the server machine. For example, if you are running a Web browser on your machine, it will want to talk to the Web server on the server machine, not the e-mail server.

A server has a static IP address that does not change very often. A home machine that is dialing up through a modem, on the other hand, typically has an IP address assigned by the ISP every time you dial in. That IP address is unique for your session — it may be different the next time you dial in. This way, an ISP only needs one IP address for each modem it supports, rather than one for each customer.

Ports and HTTP

Any server machine makes its services available using numbered ports — one for each service that is available on the server. For example, if a server machine is running a Web server and a file transfer protocol (FTP) server, the Web server would typically be available on port 80, and the FTP server would be available on port 21. Clients connect to a service at a specific IP address and on a specific port number.

Once a client has connected to a service on a particular port, it accesses the service using a specific protocol. Protocols are often text and simply describe how the client and server will have their conversation. Every Web server on the Internet conforms to the hypertext transfer protocol (HTTP). Networks, routers, NAPs, ISPs, DNS and powerful servers all make the Internet possible. It is truly amazing when you realise that all this information is sent around the world in a matter of milliseconds! The components are extremely important in modern life— without them, there would be no Internet. And without the Internet, life would be very different indeed for many of us.

Internet and World Wide Web

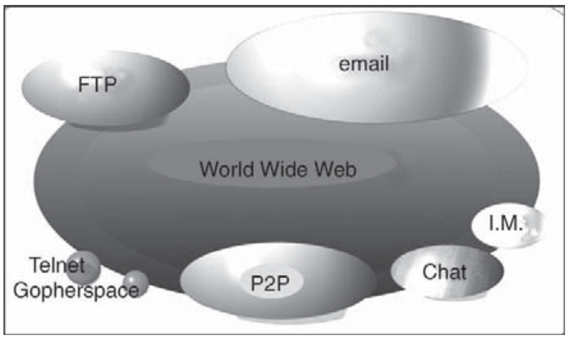
The Internet and the World Wide Web have a whole-to-part relationship. The Internet is the large container, and the Web is a part within the container. It is common in daily conversation to abbreviate them as the "Net" and the "Web", and then swap the words interchangeably. But to be technically precise, the Net is the restaurant, and the Web is the most popular dish on the menu.

Here is the detailed explanation:

The Internet is a Big Collection of Computers and Cables. The Internet is named for "interconnection of computer networks". It is a massive hardware combination of millions of personal, business, and governmental computers, all connected like roads and highways. The Internet started in the 1960's under the original name "ARPAnet". ARPAnet was originally an experiment in how the US military could maintain communications in case of a possible nuclear strike. With time, ARPAnet became a civilian experiment, connecting university mainframe computers for academic purposes.

As personal computers became more mainstream in the 1980's and 1990's, the Internet grew exponentially as more users plugged their computers into the massive network. Today, the Internet has grown into a public spiderweb of millions of personal, government, and commercial computers, all connected by cables and by wireless signals.

No single person owns the Internet. No single government has authority over its operations. Some technical rules and hardware/software standards enforce how people plug into the Internet, but for the most part, the Internet is a free and open broadcast medium of hardware networking.



The Web Is a Big Collection of HTML Pages on the Internet

The World Wide Web, or "Web" for short, is that large software subset of the Internet dedicated to broadcasting HTML pages. The Web is viewed by using free software called web browsers. Born in 1989, the Web is based on hypertext transfer protocol, the language which allows you and me to "jump" (hyperlink) to any other public web page. There are over 40 billion public web pages on the Web today.

The Internet is a worldwide network of computers that use common communication standards and interfaces to provide the physical backbone for a number of interesting applications.

One of the most utilized of these Internet applications is the World Wide Web. What sets the Web apart is an easy-to-use interface to a complex network of computers and data.

3

Process of Software Engineering

Process

Software engineering process and practices are the structures imposed on development of a software product. There are different models of software process (software lifecycle is a synonym) used in different organizations and industries.

RAL has identified three levels of software process for its projects. These levels balance the different needs of different types of projects.

Scaling the process to the project is vital to its success, too much process can be as problematic as too little; too much process can slow down a purely R&D exploration, too little process can slow down a large development project with hard deliverables. The levels are briefly identified as follows:

Level 1: R&D

- No software products delivered, pure research
- Minimal software process

Level 2: Research system

- Larger development team, informal software releases
- Moderate software process

Level 3: Delivered system

- Large software development team, formal software releases
- More formal software process

For example, the Juneau, Alaska Winds Project has evolved from a Level 1 to a Level 3 project over multiple years. It started as a purely R&D effort (Level 1), expanded to a field programme in Juneau (Level 2), and is currently running in the field as a Operational Prototype (Level 3).

The software process and software engineering practices have become more formalized and more structured as the project proceeded through the different levels. RAL has evolved a set of software engineering best practices that implement the three software process levels. These include: source code control, nightly code builds, writing reusable code, using different team models, commitment to deadlines, design and code reviews, risk management, bug tracking, software metrics, software configuration management, requirements management.

Software configuration management (SCM) is a step up in formality and reproducibility from source code control and includes controlling and versioning of software releases.

Source code control is a software engineering best practice used with RAL Level 2 and Level 3 projects. SCM is a best practice used on a number of RAL Level 3 projects.

Description

Software Engineering Process

The elements of a software engineering process are generally enumerated as:

- Marketing Requirements
- System-Level Design
- Detailed Design
- Implementation
- Integration
- Field Testing
- Support

No element of this process ought to commence before the earlier ones are substantially complete, and whenever a change is made to some element, all dependent elements ought to be reviewed or redone in light of that change. It's possible that a given module will be both specified and implemented before its dependent modules are fully specified — this is called advanced development or research.

It is absolutely essential that every element of the software engineering process include several kinds of *review:* peer review, mentor/management review, and cross-disciplinary review. Software engineering elements (whether documents or source code) must have version numbers and auditable histories. "Checking in" a change to an element should require some form of review, and the depth of the review should correspond directly to the scope of the change.

Marketing Requirements

The first step of a software engineering process is to create a document which describes the target customers and their reason for needing this product, and then goes on to list the features of the product which address these customer needs. The Marketing Requirements Document (MRD) is the battleground where the answer to the question "What should we build, and who will use it?" is decided.

In many failed projects, the MRD was handed down like an inscribed stone tablet from marketing to engineering, who would then gripe endlessly about the laws of physics and about how they couldn't actually build that product since they had no ready supply of Kryptonite or whatever. The MRD is a joint effort, with engineering not only reviewing but also writing a lot of the text.

System-Level Design

This is a high-level description of the product, in terms of "modules" (or sometimes "programmes") and of the interaction between these modules. The goals of this document are first, to gain more confidence that the product could work and could be built, and second, to form a basis for estimating the total amount of work it will take to build it. The system-level design document should also outline the system-level testing plan, in terms of customer needs and whether they would be met by the system design being proposed.

Detailed Design

The detailed design is where every module called out in the system-level design document is described in detail. The interface (command line formats, calling API, externally visible data structures) of each module has to be completely determined at this point, as well as dependencies between modules. Two things that will evolve out of the detailed design is a PERT or GANT chart showing what work has to be done and in what order, and more accurate estimates of the time it will take to complete each module.

Every module needs a unit test plan, which tells the implementor what test cases or what kind of test cases they need to generate in their unit testing in order to verify functionality. Note that there are additional, nonfunctional unit tests which will be discussed later.

Implementation

Every module described in the detailed design document has to be implemented. This includes the small act of coding or programming that is the heart and soul of the software engineering process. It's unfortunate that this small act is sometimes the only part of software engineering that is taught (or learned), since it is also the only part of software engineering which can be effectively self-taught.

A module can be considered implemented when it has been created, tested, and successfully used by some other module (or by the system-level testing process). Creating a module is the old edit-compile-repeat cycle. Module testing includes the unit level functional and regression tests called out by the detailed design, and also performance/stress testing, and code coverage analysis.

Integration

When all modules are nominally complete, system-level integration can be done. This is where all of the modules move into a single source pool and are compiled and linked and packaged as a system. Integration can be done incrementally, in parallel with the implementation of the various modules, but it cannot authoritatively approach "doneness" until all modules are substantially complete.

Integration includes the development of a system-level test. If the built package has to be able to install itself (which could mean just unpacking a tarball or copying files from a CD-ROM) then there should be an automated way of doing this, either on dedicated crash and burn systems or in containerized/simulated environments. Sometimes, in the middleware arena, the package is just a built source pool, in which case no installation tools will exist and system testing will be done on the as-built pool. Once the system has been installed (if it is installable), the automated system-level testing process should be able to invoke every public command and call every public entry point, with every possible reasonable combination of arguments.

If the system is capable of creating some kind of database, then the automated system-level testing should create one and then use external (separately written) tools to verify the database's integrity. It's possible that the unit tests will serve some of these needs, and all unit tests should be run in sequence during the integration, build, and packaging process.

Field Testing

Field testing usually begins internally. That means employees of the organization that produced the software package will run it on their own computers. This should ultimately include all "production level" systems — desktops, laptops, and servers.

The statement you want to be able to make at the time you ask customers to run a new software system (or a new version of an existing software system) is “we run it ourselves.” The software developers should be available for direct technical support during internal field testing. Ultimately it will be necessary to run the software externally, meaning on customers’ (or prospective customers’) computers. It’s best to pick “friendly” customers for this exercise since it’s likely that they will find a lot of defects — even some trivial and obvious ones — simply because their usage patterns and habits are likely to be different from those of your internal users.

The software developers should be close to the front of the escalation path during external field testing. Defects encountered during field testing need to be triaged by senior developers and technical marketers, to determine which ones can be fixed in the documentation, which ones need to be fixed before the current version is released, and which ones can be fixed in the next release (or never).

Support

Software defects encountered either during field testing or after the software has been distributed should be recorded in a tracking system. These defects should ultimately be assigned to a software engineer who will propose a change to either the definition and documentation of the system, or the definition of a module, or to the implementation of a module. These changes should include additions to the unit and/or system-level tests, in the form of a regression test to show the defect and therefore show that it has been fixed (and to keep it from recurring later).

Just as the MRD was a joint venture between engineering and marketing, so it is that support is a joint venture between engineering and customer service. The battlegrounds in this venture are the bug list, the categorization of particular bugs, the maximum number of critical defects in a shippable software release, and so on.

Software Quality Attribute

high quality software

Developing high quality software is hard, especially when the interpretation of term "quality" is patchy based on the environment in which it is used. In order to know if quality has been achieved, or degraded, it has to be measured, but determining what to measure and how is the difficult part. Software Quality Attributes are the benchmarks that describe system's intended behaviour within the environment for which it was built.

The quality attributes provide the means for measuring the fitness and suitability of a product. Software architecture has a profound affect on most qualities in one way or another, and software quality attributes affect architecture. Identifying desired system qualities before a system is built allows system designer to mold a solution (starting with its architecture) to match the desired needs of the system within the context of constraints (available resources, interface with legacy systems, etc). When a designer understands the desired qualities before a system is built, then the likelihood of selecting or creating the right architecture is improved.

Statements

Both statements are useless as they provide no tangible way of measuring the behaviour of the system. The quality attributes must be described in terms of scenarios, such as "when 100 users initiate 'complete payment' transition, the payment component, under normal circumstances, will process the requests with an average latency of three seconds." This statement, or scenario, allows an architect to make quantifiable arguments about a system.

A scenario defines the source of stimulus (users), the actual stimulus (initiate transaction), the artifact affected (payment component), the environment in which it exists (normal operation), the effect of the action (transaction processed), and the response measure (within three seconds). Writing such detailed statements is only possible when relevant requirements have been identified and an idea of components has been proposed.

Qualities

Scenarios help describe the qualities of a system, but they don't describe how they will be achieved. Architectural tactics describe how a given quality can be achieved. For each quality there may be a large set of tactics available to an architect. It is the architect's job to select the right tactic in light of the needs of the system and the environment. For example, a performance tactics may include options to develop better processing algorithms, develop a system for parallel processing, or revise event scheduling policy. Whatever tactic is chosen, it must be justified and documented.

Software Qualities

It would be naïve to claim that the list below is as a complete taxonomy of all software qualities – but it's a solid list of general software qualities compiled from respectable sources. Domain specific systems are likely to have an additional set of qualities in addition to the list below. System qualities can be categorized into four parts: runtime qualities, non-runtime qualities, business qualities, and architecture qualities.

Each of the categories and its associated qualities are briefly described below. Other articles on this site provide more information about each of the software quality attributes listed below, their applicable properties, and the conflicts the qualities.

Software Characteristics

Software requirement

- Microsoft Windows 98 SE, Me, NT4 (sp5+), 2000 or XP,
- Word processing software (optional),
- Spell checker (optional),
- Spreadsheet (optional), Microsoft Excel is necessary to generate analysis reports
- Web browser (optional), Internet Explorer 5 or Netscape 6 or above,
- Adobe Acrobat (optional).

Hardware requirement

- PC compatible computer (Pentium II or compatible),
- CD-ROM Drive,
- SVGA or XGA (1024 × 768) graphic screen and card,

- Floppy drive (optional, for Ethnos input transfer),
- Printer port (parallel port RS232).

Minimum advised configuration

Software	Processor	RAM memory (1)	Disk space	System (3)
Tropes basic	200 Mhz	64 (128) Mb	16 Mb	Win 98, 2000,XP
Tropes Zoom (2)	200 Mhz	128 (256) Mb	50 Mb	Win 98, 2000, XP
Ethnos	200 Mhz	128 Mb	50 Mb	Win 98, 2000, XP
Acetic Index	400 Mhz	128 (256) Mb (SP6), 2000, Hardware appliance	2 Gb XP, dedicated	Win NT4

- Under Windows 2000 and XP it is advised to install a minimum of 256 Mb of RAM memory so that system works with optimal performances.
- 512 Mb of memory RAM and 2 Go of disk space of swapfile advised to make decision-making analysis (on consequent documentary bases) with Tropes Zoom.
- For Tropes, Zoom and Index, the conversion of PDF files works only under Windows 2000 or XP.

Maximum volumetry

Software	Text Size	Number of Files	Number of Words	Database Size
Tropes Zoom SE	32 Kb	100	100,000	100 files
Tropes basic	100 Mb	1	10,000,000	-
Tropes Zoom	100 Mb	> 1,000,000	2,147,483,648	20 Gb
Ethnos	N/A	Unlimited	N/A	N/A
Acetic Index	100 Mb	Unlimited	Unlimited	Unlimited

By "unlimited", we understand that the theoretical capacity of these software packages widely exceeds what it is possible to handle on a current computer. Your computer naturally has limited capacities. For Zoom, we indicate the theoretical maximal capacity for a single documentary base. Knowing that you can create an unlimited number of documentary bases. By Database size, we mean the theoretical maximal capacity in terms of pure text indexed by the software. Knowing that it can require terabytes of disc space.

Languages

Language	Tropes Basic (Professional)	Tropes Zoom Special Edition	Tropes Zoom (Professional)	Acetic Index	Ethnos/ Stat'Mania
English	Yes	Yes	Yes	Yes	Yes
French	Yes	Yes	Yes	Yes	Yes
Spanish	Unavailable in France (1)	Contact us	Yes	Yes	Yes
Portuguese	Unavailable in France (1)	Contact us	Yes	Yes	No
German	Unavailable in France (1)	unavailable in France (1)	Yes (2)	Yes (2)	Yes
Italian	Unavailable in France (1)	unavailable in France (1)	Yes (2)	Yes (2)	No

- These languages are not commercialized on this site in France, but are available for certain our European partners.
- Language in the course of finalisation, available under conditions: contact us.

Text Analysis

- Minimum size advised for a text: less than 1 page (1 Kb),
- Maximum size advised for a single text: 5,000 pages (50 Mb),
- Average analysis throughput: from 20,000 words/second (Pentium III 733 MHz) to 80,000 words/second (Pentium IV 3.2 GHz, HT) on local Web pages, for a single processor.

Semantic Search Engine

- Automatic generation of hierarchical keywords,
- Automatic information filtering (based on a pertinence treshold),
- Massive data analysis and information cartography (text-mining),
- Search improvement for the references (nouns, trademarks and proper names),
- Maximum numbers of text databases: unlimited,
- Average indexing throughput: from 1 Gb/hour (Pentium III 733 Mhz) to 4 Gb/hour (Pentium IV 3.2 GHz, HT) on local Web pages, for a single processor.

Optional modules of Market Research and advanced statistics

- Broadcasting of surveys on Internet (Ethnos/Net Survey)
- Optical Character Recognition of questionnaires (Ethnos/ OMR Manager)
- Panel management (Ethnos/Panel Manager)
- Management of phone inquiries (Ethnos/Catiopée)
- Datamining module (Data analysis, Stat'Mania)
- Management of street surveys on PDA (Ethnos/CAPI)

Other features

- File formats converted by our linguistic softwares (Tropes, Zoom and Index): Adobe Acrobat, ASCII, ANSI, HTML, Macromedia Flash, Microsoft Excel, Microsoft Powerpoint, Microsoft Word, Microsoft WordML (Word XML), RTF, XML, SGML and Macintosh texts

- Automatic extraction of Microsoft Outlook messages via an external utility (Zoom Semantic Search Engine)
- Automatic exportation of the results towards other software (Zoom Semantic Search Engine)
- Indexing engine in batch mode (Acetic Index)
- Win32 Application Programming Interface (Acetic Index)
- Real time XML output interface (Acetic Index)
- Distributed fault tolerant and load-balancing Interface (CORBA, Acetic Index)
- Runtime, operation on Intranet, HTML generation (contact us)
- Some features (for example, very large Text Mining) may require the use of an additional statistics software, of data mining software and/or a RDBMS

Software Measurement and Metrics

The measurement information model is a structure linking information needs to the relevant entities and attributes of concern. Entities include processes, products, projects, and resources. The measurement information model describes how the relevant attributes are quantified and converted to indicators that provide a basis for decision-making.

The selection or definition of appropriate measures to address an information need begins with a measurable concept: an idea of which measurable attributes are related to an information need and how they are related. The measurement planner defines measurement constructs that link these attributes to a specifiedinformation need. Each construct may involve several types or levels of measures. This measurement information model (see Figure) identifies the basic terms and concepts with which the measurement analyst must deal. The measurement modelhelps to determine what the measurement planner needs to specify during measurement planning, performance, and evaluation.

Entity

An entity is an object (for example, a process, product, project, or resource) that is to be characterized by measuring its attributes. Typical software engineering objects can be classified as products (e.g., design document, source code, and test case), processes (e.g., design process, testing process, requirements analysis process), projects, and resources (e.g., the programmers and the testers). An entity may have one or more properties that are of interest to meet the information needs. In practice, an entity can be classified into more than one of the above categories.

Measurable attribute

An attribute is a property or characteristic of an entity that can be distinguished quantitatively or qualitatively by human or automated means. An entity may have many attributes, only some of which may be of interest for measurement. The first step in defining a specific instantiation of the measurement information model is to select the attributes that are most relevant to the measurement user's information needs. A given attribute may be incorporated in multiple measurement constructs supporting different information needs.

Base measure

A base measure is an attribute and the method for quantifying it. A base measure is functionally independent of other measures. A base measure captures information about a single attribute. Data collection involves assigning values to base measures. Specifying the expected range and/ or type of values of a base measure helps to verify the quality of the data collected.

Measurement Method

A measurement method is a logical sequence of operations, described generically, used in quantifying an attribute with respect to a specified scale. The operations may involve activities such as counting occurrences or observing the passage of time. The same measurement method may be applied to multiple attributes.

However, each unique combination of an attribute and a method produces a different base measure. Some measurement methods may be implemented in multiple ways. A measurement procedure describes the specific implementation of a measurement method within a given organizational context.

Type of Measurement Method

The type of measurement method depends on the nature of the operations used to quantify an attribute. Two types of method may be distinguished:

1. *Subjective*: Quantification involving human judgment
2. *Objective*: Quantification based on numerical rules such as counting. These rules may be implemented via human or automated means.

Scale

A scale is an ordered set of values, continuous or discrete, or a set of categories to which the attribute is mapped. The measurement method maps the magnitude of the measured attribute to a value on a scale. A unit of measurement often is associated with a scale.

Type of Scale

The type of scale depends on the nature of the relationship between values on the scale.

Four types of scales are commonly defined:

1. *Nominal*: The measurement values are categorical. For example, the classification of defects by their type.
2. *Ordinal*: The measurement values are rankings. For example, the assignment of defects to a severity level.
3. *Interval*: The measurement values have equal distances corresponding to equal quantities of the *attribute*. For example, cyclomatic complexity has the minimum value of one, but each increment represents an additional path.
4. *Ratio:* The measurement values have equal distances corresponding to equal quantities of the *attribute* where the value of zero corresponds to none of the*attribute*. For example, the size of a software component in terms of LOC.

The method of measurement usually affects the type of *scale* that can be used reliably with a given *attribute*.

For example, subjective methods of measurement usually only support ordinal or nominal scales.

Unit of Measurement

A *unit of measurement* is a particular quantity, defined and adopted by convention, with which other quantities of the same kind are compared in order to express their magnitude relative to that quantity. Only quantities expressed in the same units of measurement are directly comparable. Example of units include the hour and the meter.

Derived measure

A derived measure is a measure that is defined as a function of two or more base measures. Derived measures capture information about more than oneattribute. Simple transformations of base measures (for example, taking the square root of a base measure) do not add information, thus do not produce derived measures.

Normalization of data often involves converting base measures into derived measures that can be used to compare different entities.

Measurement Function

A measurement function is an algorithm or calculation performed to combine two or more base measures. The scale and unit of the derived measure depend on the scales and units of the base measures from which it is composed as well as how they are combined by the function.

Indicator

An indicator is an estimate or evaluation of specified attributes derived from a model with respect to defined information needs. Indicators are the basis for analysis and decision-making. These are what should be presented to measurement users.

Measurement is always based on imperfect information, so quantifying the uncertainty, accuracy, or importance of indicators is an essential component of presenting the actual indicator value. Therefore, an interpretation of indicators is performed to provide the desired information product.

Measurement Model

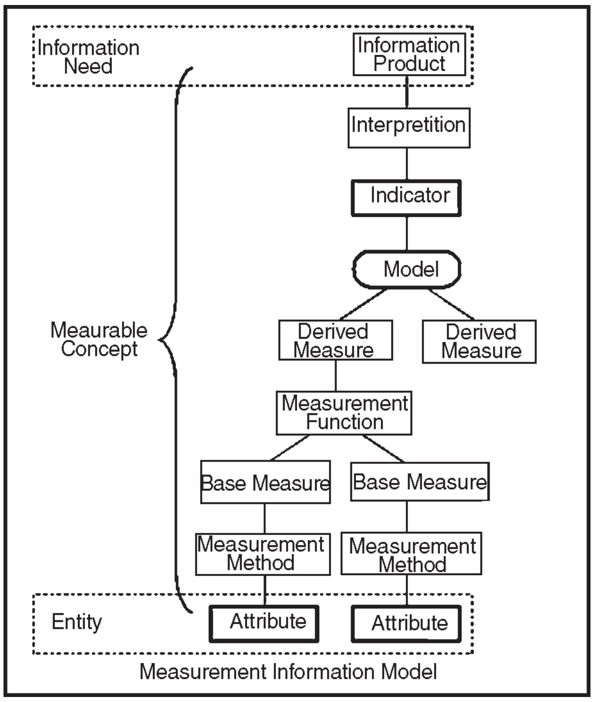
A measurement model is an algorithm or calculation combining one or more base and/or derived measures with associated decision criteria. It is based on an understanding of, or assumptions about, the expected relationship between the component measures and/or their behaviour over time. Models produce estimates or evaluations relevant to defined information needs. The scale and measurement method affect the choice of analysis techniques or models used to produceindicators.

Decision Criteria

Decision criteria are numerical thresholds or targets used to determine the need for action or further investigation, or to describe the level of confidence in a given result. Decision criteria help to interpret the results of measurement. Decision criteria may be calculated or based on a conceptual understanding of expected behaviour. Decision criteria may be derived from historical data, plans, and heuristics, or computed as statistical control limits or statistical confidence limits.

Measurable concept

A measurable concept is an abstract relationship between attributes of entities and information needs. For example, an Information need may be the need to compare the software development productivity of a project group against a target rate. The Measurable Concept in this case is “software development productivity rate”. To evaluate the concept might require measuring the size of the software products and the amount of resource applied to create the products (depending on the chosen model of productivity). Additional examples of Measurable Concepts include quality, risk, performance, capability, maturity, and customer value.



Software Metrics

Effective management of any process requires quantification, measurement, and modeling. Software metrics provide a quantitative basis for the development and validation of models of the software development process. Metrics can be used to improve software productivity and quality. This module introduces the most commonly used software metrics and reviews their use in constructing models of the software development process. Although current metrics and models are certainly inadequate, a number of organizations are achieving promising results through their use. Results should improve further as we gain additional experience with various metrics and oftware metrics are numerical data related to software development. Metrics strongly support software project management activities.

They relate to the four functions of management as follows:

1. *Planning*: Metrics serve as a basis of cost estimating, training planning, resource planning, scheduling, and budgeting.
2. *Organizing*: Size and schedule metrics influence a project's organization.
3. *Controlling*: Metrics are used to status and track software development activities for compliance to plans.
4. *Improving*: Metrics are used as a tool for process improvement and to identify where improvement efforts should be concentrated and measure the effects of process improvement efforts.

A metric quantifies a characteristic of a process or product. Metrics can be directly observable quantities or can be derived from one or more directly observable quantities. Examples of raw metrics include the number of source lines of code, number of documentation pages, number of staff-hours, number of tests, number of requirements, etc. Examples of derived metrics include source lines of code per staff-hour, defects per thousand lines of code, or a cost performance index.

The term *indicator* is used to denote a representation of metric data that provides insight into an ongoing software development project or process improvement activity. Indicators are metrics in a form suitable for assessing project behaviour or process improvement. For example, an indicator may be the behaviour of a metric over time or the ratio of two metrics.

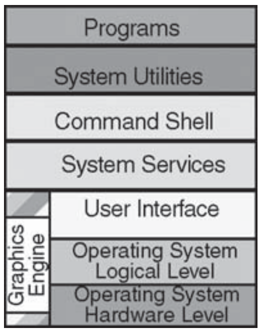
Indicators may include the comparison of actual values versus the plan, project stability metrics, or quality metrics. Examples of indicators used on a project include actual versus planned task completions, actual versus planned staffing, number of trouble reports written and resolved over time, and number of requirements changes over time. Indicators are used in conjunction with one another to provide a more complete picture of project or organization behaviour. For example, a progress indicator is related to requirements and size indicators. All three indicators should be used and interpreted together.

Software Components

A computer system consists of three major components: hardware, software, and humans (users, programmers, administrators, operators, etc.). Software can be further divided into seven layers. Firmware can be categorized as part of hardware, part of software, or both.

The seven layers of software are (top to bottom): Programs; System Utilities; Command Shell; System Services; User Interface; Logical Level; and Hardware Level. A Graphics Engine stradles the bottom three layers.

Strictly speaking, only the bottom two levels are the operating system, although even technical persons will often refer to any level other than programs as part of the operating system (and Microsoft tried to convince the Justice Department that their web browser application is actually a part of their operating system). Because this technical analysis concentrates on servers, Internet Facilities are specifically separated out from the layers.



Examples

The following are examples of each category:

- *Programs*: Examples of Programs include your word processor, spreadsheet, graphics programs, music software, games, etc.
- *System Utilities*: Examples of System Utilities include file copy, hard drive repair, and similar items. On the Macintosh, all the Desk Accessories (calculator, key caps, etc.) and all of the Control Panels are examples of System Utilities.
- *Command Shell*: The Command Shell on the Macintosh is the Finder and was the first commercially available graphic command shell. On Windows, the Command Shell is a poorly integrated comination of the File Manager and the Programme Manager. The command line (C:\ prompt) of MS-DOS or Bourne Shell of UNIX are examples of the older style text-based command shells.

- *System Services*: Examples of System Services are built-in data base query languages on mainframes or the QuickTime media layer of the Macintosh.
- *User Interface*: Until the Macintosh introduced Alan Kay's (inventer of the personal computer, graphic user interfaces, object oriented programming, and software agents) ground breaking ideas on human-computer interfaces, operating systems didn't include support for user interfaces (other than simple text-based shells). The Macintosh user interface is called the Macintosh ToolBox and provides the windows, menus, alert boxes, dialog boxes, scroll bars, buttons, controls, and other user interface elements shared by almost all programs.
- *Logical Level of Operating System*: The Logical Level of the operating system provides high level functions, such as file management, internet and networking facilities, etc.
- *Hardware Level of Operating System*: The Hardware Level of the operating system controls the use of physical system resources, such as the memory manager, process manager, disk drivers, etc.
- *Graphics Engine*: The Graphics Engine includes elements at all three of the lowest levels, from physically displaying things on the monitor to providing high level graphics routines such as fonts and animated sprites.

Human users normally interact with the operating system indirectly, through various programs (application and system) and command shells (text, graphic, etc.), The operating system provides programs with services thrrough system programs and Application Programme Interfaces (APIs).

4

Computer Arithmetic Techniques

Algorithm

To make a computer do anything, you have to write a computer programme. To write a computer programme, you have to tell the computer, step by step, exactly what you want it to do. The computer then “executes” the programme, following each step mechanically, to accomplish the end goal.

When you are telling the computer *what* to do, you also get to choose *how* it’s going to do it. That’s where computer algorithms come in. The algorithm is the basic technique used to get the job done. Let’s follow an example to help get an understanding of the algorithm concept.

Let’s say that you have a friend arriving at the airport, and your friend needs to get from the airport to your house. Here are four different algorithms that you might give your friend for getting to your home:

- The taxi algorithm:
  - Go to the taxi stand.
  - Get in a taxi.
  - Give the driver my address.
- The call-me algorithm:
  - When your plane arrives, call my cell phone.
  - Meet me outside baggage claim.
- The rent-a-car algorithm:
  - Take the shuttle to the rental car place.
  - Rent a car.
  - Follow the directions to get to my house.
- The bus algorithm:
  - Outside baggage claim, catch bus number 70.
  - Transfer to bus 14 on Main Street.
  - Get off on Elm street.
  - Walk two blocks north to my house.

All four of these algorithms accomplish exactly the same goal, but each algorithm does it in completely different way. Each algorithm also has a different cost and a different travel time. Taking a taxi, for example, is probably the fastest way, but also the most expensive. Taking the bus is definitely less expensive, but a whole lot slower. You choose the algorithm based on the circumstances. In computer programming, there are often many different ways — algorithms — to accomplish any given task. Each algorithm has advantages and disadvantages in different situations. Sorting is one place where a lot of research has been done, because computers spend a lot of time sorting lists.

Addition and Subtraction

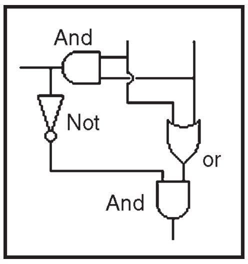
The table for binary addition is:

```
+ 0 1
```

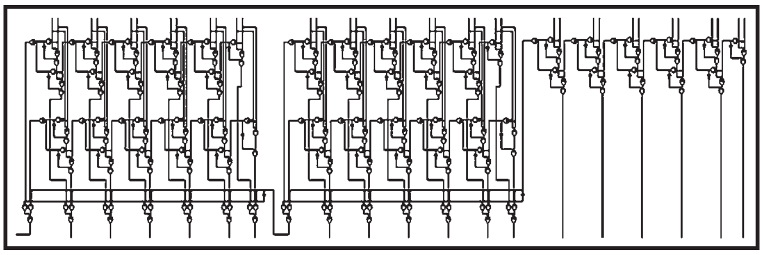
```
0 0 1
1 1 10
```

thus, when two binary bits A and B are added, the bit of the sum in the same place is A XOR B, and the carry is A AND B. The exclusive OR function needs to be built from the fundamental AND, OR, and NOT operations in most forms of logic, and A XOR B is the same as (A OR B) AND (NOT (A AND B)). Of course, A AND B has already been calculated, since it is required as the carry.

Thus, a binary adder has the following logic diagram:

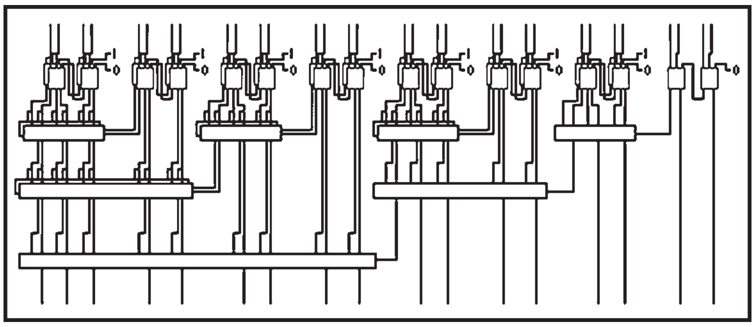


This involves just three gate delays: But addition requires the propagation of carries, and thus the time required to add two long numbers is proportional to their length in the simplest type of adder. This can be speeded up by calculating sums in advance with either input value of the carry, and then selecting the right one, so that carries propagate at a higher level over many bits at a time:



Here, a circuit that adds two binary 18-bit numbers is shown. It is divided into three groups of six bits. Except for the least significant group, each is added together in two ways in parallel, one assuming no carry in, one assuming that a carry in is present; the carry then selects the right result without having to propagate through the six bits of the next group.

If we take this kind of architecture to its ultimate conclusion, the form of carry-select adder we obtain is also known as the Sklansky adder, shown in a more schematic form below:



Newer designs for addition units belonging to the *parallel prefix* and *flagged prefix* classes have since been developed. Parallel-prefix adders include the Brent-Kung adder, the Kogge-Stone adder, and the Han-Carlson adder. Although they do not improve on the speed of the Sklansky adder, they address a limitation of real logic circuits which that design ignores, the fact that logic gates have a limited fan-out.

Binary Multiplication

Binary multiplication does not require much of a multiplication table; all that is required is to make a decision to add or not add a copy of one factor to the result, based on whether the corresponding bit of the other factor is a one or a zero.

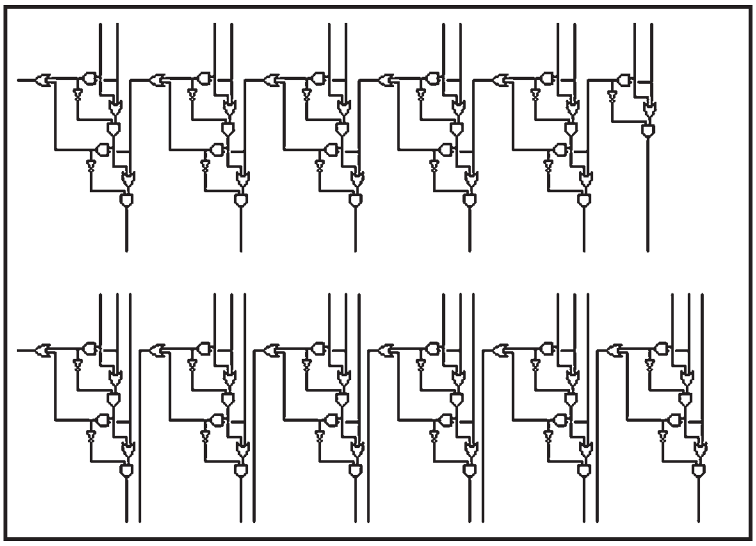
One straightforward, although expensive in terms of the number of gates required, way to speed up multiplication is to perform all the multiplications of one factor by the individual bits of the other in parallel, and then add the results in pairs, and then the sums in pairs, so that the number of addition stages required is proportional to the logarithm to the base 2 of the length of the second factor.

If, at the beginning of a multiplication, one calculated three times the multiplicand, then two bits of the multiplier at a time, which can only have the values 00, 01, 10, or 11, could each be used to create a partial product; this would trim away the top layer of the addition tree, cutting the amount of circuitry required almost in half. (It may be noted that the NORC computer by IBM, an ambitious vacuum-tube computer which used decimal arithmetic, calculated multiples up to nine times the multiplicand in advance to speed multiplication.)

While these basic concepts do play a part in performing high-speed multiplication, they omit another factor which has an even more dramatic effect on the speed of multiplication, but which makes fast multiplication somewhat more difficult to understand.

A special type of adder, called a *carry-save adder*, can be used in multiplication circuits, so that carry propagation can be avoided for all but the last addition in the additions required to perform a multiplication. In order to use this kind of adder, instead of adding numbers in pairs, speed can be increased by adding numbers in groups of three. This is because one can turn three numbers into two numbers, having the same sum, without propagating carries through the number. This is because a full adder, which takes two bits and an incoming carry bit, and outputs a result bit and a carry bit, can also be used as a 3-2 compressor, one type of carry-save adder. The carry bits move to the left, but since these bits are all shifted left independently, the delays of carry propagation are not required.

The diagram below shows the difference between a conventional addition circuit with carry propagation, shown in the top half,



and the 3-2 compressor form of carry-save adder, shown in the bottom half, and involving arguments the same number of bits in length.

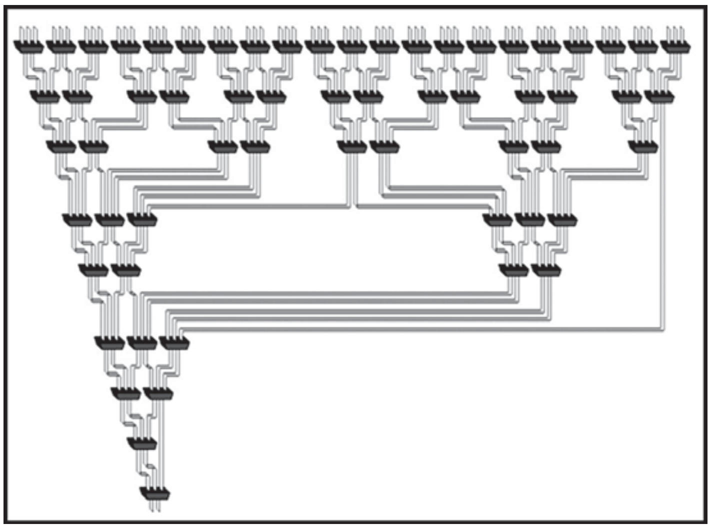
As the diagram attempts to illustrate, although carries do not propagate in the sense of being added in to the previous digit, they do still move one place to the left.

One can then add the two resulting numbers by means of a conventional adder with accelerated carry propagation, such as a carry-select adder as depicted above, or, because carry propagation, even when accelerated, causes such a large delay, one can simply use successive stages of 3-2 compression, performing conventional addition only at the very last stage.

This technique was used to speed multiplication in the IBM 7030 computer known as STRETCH.

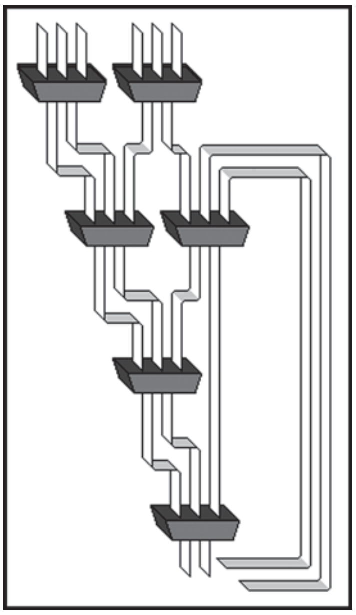
Instead of simply using one carry-save adder over and over to add one partial product after another, or to have a single line of carry-save adders if it is desired to do successive steps from different multiplications at once in a pipelined fashion, if maximum speed is desired and the cost of hardware and the number of transistors used is no object, one could, as shown above, use several carry-save adders in parallel to produce two results from three partial products, and then repeat the process in successive layers until there are only two numbers left to add in a fast carry-propagate adder. This type of multiplication circuit is called a Wallace Tree adder, after its inventor.

The diagram below:



illustrates how, with only two inputs left to go into a carry-propagate adder at the end, successive layers of carry-save adders increase the maximum number of possible partial products that can be included to 3, then 4, and then 6, 9, 13, 19, 28, 42, and 63. The sequence continues: 94, 141, 211...

In the IBM System/360 Model 91, a Wallace Tree of limited size was used, as it was necessary to be somewhat concerned with the number of transistors to use:



Six partial products were handled by this unit, but only twelve bits of these products were handled at a time. Thus, it would add these six numbers together, sending the least significant twelve bits of the final two results to a carry-propagate adder, while the more significant bits were sent back to be combined with the next more significant twelve bits of the partial products being worked on.

The half adder, although it takes two inputs and provides two outputs, has been used to improve the Wallace Tree adder slightly.

One way this has been done is: when there are two left-over values at a given stage to be added, then, they might be put into a half adder. Although this produces two outputs, one of those outputs will consist of carries, and will therefore be shifted one place to the left.

If the number of terms to be added in the next stage is still not an even multiple of three, then, at least at the beginning and end of the part added in the half adders, the number of left-over bits will be reduced by one. This is called a Dadda tree multiplier.

More recently, in U.S. Patent 6,065,033, assigned to the Digital Equipment Corporation, the fact that half the input bits are moved to the left as carry bits was noted explicitly as a way to replace the larger tree generating the middle of the final product by multiple trees of more equal size starting from the middle and proceeding to the left.

As noted above, one could trim away a considerable portion of the addition tree by using two digits of the multiplier at a time to create one partial product. But doing this in the naive way described above requires calculating three times the multiplicand, which requires carry propagation.

Another idea that suggests itself is to note that either less than half of the bits in a multiplier are 1, or less than half of the bits in a multiplier are zero, so that in the former case, one could include only the nonzero partial products, and in the latter case, one would multiply by the one's complement of the multiplier and then make a correction. But this would require an elaborate arrangement for routing the partial products that would more than consume any benefits.

Fortunately, there is a method of achieving the same result without having to wait for carries, known as Booth encoding.

The goal is to make it possible for each pair of bits in the multiplier to introduce only one partial product that needs to be added. Normally, each pair of bits calls for 0, 1, 2, or 3 times the multiplicand, and the last possibility requires carry propagation to calculate.

Knowing that –1 is represented by a string of ones in two's complement arithmetic, the troublesome number 3 presents itself to us as being equal to 4 minus 1. Four times the multiplicand can be instantly generated by a shift. The two's complement of the multiplicand would require carry propagation to produce, but just inverting all the bits of the multiplicand gives us one less than its additive inverse.

It is possible to combine bits representing these errors, since each time this happened, it would be in a different digit, in a single term that could be entered into the addition tree like a partial product. But that, alone, doesn't eliminate carry propagation. If we code 3 as –1 with +1 to carry into the next pair of bits in the multiplier, then, if we find 2 there, once again we will be stuck with 3. As it happens, though, we have one other possible value that can be produced without carry propagation from the multiplicand that is useful. We can also produce –2 times the multiplicand by both inverting it and shifting it; this time, the result will be too small by two, and that, too, can be accounted for in the error term.

Using the fact that –2 is also available to us in our representation of pairs of bits, we can modify the pairs of digits in the multiplier like this:

```
00       0
01  + 1
10  + 1 -2
11  + 1 -1
```

Since only –2, –1, 0, and +1 are initially and directly used to code a pair of bits, a carry of +1 from the pair of bits to the right will result in the possible values -1, 0, +1, and +2; no troublesome value results that requires an additional carry.

A second step is not really needed, since one just has to peek at the first digit of the following pair of bits to see if a carry out would be coming; thus, Booth coding is often simply presented in the manner in which it would be most efficient when implemented, as a coding that goes directly from three input bits to a signed base-4 digit:

```
00.0    0
00.1  + 1
01.0  + 1
01.1  + 2
10.0  - 2
10.1  - 1
11.0  - 1
11.1    0
```

In the table as shown here, I place a binary point between the two bits for which a substitution is being made and the bit that belongs to the preceding group on the right. This serves as a reminder that the last bit encoded is to the right of the place value of the result of the code, and also that the rightmost pair of bits is encoded on the basis of a 0 lying to their right. Of course, the final carry out, not apparent in this form of the table, should not be forgotten; but it can also be found by coding an additional leading two zero bits appended to the multiplier.

And the digits of the multiplier as converted can be used to determine the value of the error term. In addition to the error term, it is also necessary to remember that the partial products can be negative, and so their signs will need to be extended; this can be done by means of extending the various carry-save adders in the Wallace tree only one additional bit to the left with proper organization.

An alternative version of Booth encoding that does require generating three times the multiplicand, in return for converting three bits of the multiplier at a time into a term from the set {–4, –3, –2, –1, 0, +1, +2, +3, +4} has also been used. However, if one is going to allow *hard multiples* of the multiplicand, that is, multiples which require a carry-propagate addition to produce, then, one could also generate five times the multiplicand (5 = 4 + 1), nine times the multiplicand (9 = 8 + 1), and even, through the use of the same principle as underlies Booth encoding, seven times the multiplicand (7 = 8 – 1); three times, of course, gives six and twelve times by a shift, so, if one is willing to build three carry-propagate adders working in parallel, one can produce all the multiples in the set {–8, –7, –6, –5, –4, –3, –2, –1, 0, +1, +2, +3, +4, +5, +6, +7, +8}, and encode four bits of the multiplier at a time.

If the multiples of the multiplicand are generated in advance, and the multiplier is long enough, then one can retire four bits of the multiplier and, using a pre-generated multiple, following the principle used in the NORC, do so with only a single carry-save addition.

If the multiples are not generated in advance, or when a full-width Wallace Tree is used, it would seem that multi-bit Booth encoding could not provide a gain in speed, because an addition requiring carries would take as long as several stages of carry-save addition.

If one is retiring the bits of the multiplier serially, to save hardware, rather than using a full Wallace Tree, in which all of the multiplier is retired at once, and the partial products are then merged into half as many terms at each step, one could retire the last few bits of the multiplier at a slower rate while waiting for the larger multiples of the multiplicand to become available. The Alpha 21264 multiplier unit used this technique, using Booth encoding of bit pairs in the multiplier until a conventional adder produced the value of three times the multiplicand for use in later stages allowing the use of three-bit Booth encoding of the multiplier thereafter.

Arithmetic Operations on Binary and Hexadecimal Numbers

There are more than a few operations we can perform on binary and hexadecimal numbers. For example we can add subtract multiply divide and perform other arithmetic operations.

Although you needn't become an expert at it you should be able to in a pinch perform these operations manually using a piece of paper and a pencil. Having just said that you should be able to perform these operations manually the correct way to perform such arithmetic operations is to have a calculator which does them for you. There are several such calculators on the market; the following table lists some of the manufacturers who produce such devices:

Manufacturers of Hexadecimal Calculators:

- Casio
- Hewlett-Packard

- Sharp
- Texas Instruments.

This list is by no means exhaustive. Other calculator manufacturers probably produce these devices as well. The Hewlett-Packard devices are arguably the best of the bunch. However they are more expensive than the others. Sharp and Casio produce units which sell for well under $50. If you plan on doing any assembly language programming at all owning one of these calculators is essential.

Another alternative to purchasing a hexadecimal calculator is to obtain a TSR (Terminate and Stay Resident) programme such as SideKick which contains a built-in calculator. However unless you already have one of these programmes or you need some of the other features they offer such programmes are not a particularly good value since they cost more than an actual calculator and are not as convenient to use.

To understand why you should spend the money on a calculator consider the following arithmetic problem:

```
  9h
+ 1h
----
```

The tempted to write in the answer "10h" as the solution to this problem. But that is not correct! The correct answer is ten which is "0Ah" not sixteen which is "10h".

A similar problem exists with the arithmetic problem:

```
  10h
-  1h
----
```

The probably tempted to answer "9h" even though the true answer is "0Fh". Remember this problem is asking "what is the difference between sixteen and one?" The answer of course is fifteen which is "0Fh".

Even if the two problems above don't bother you in a stressful situation your brain will switch back into decimal and something else produce the incorrect result. Moral of the story - if you must do an arithmetic computation using hexadecimal numbers by hand take your time and be careful about it. Either that or convert the numbers to decimal perform the operation in decimal and convert them back to hexadecimal.

You should never perform binary arithmetic computations. Since binary numbers usually contain long strings of bits there is too much of an opportunity for you to make a mistake. Always convert binary numbers to hex perform the operation in hex (preferably with a hex calculator) and convert the result back to binary if necessary.

Logical Operations on Bits

There are four main logical operations we'll need to perform on hexadecimal and binary numbers: AND OR XOR (exclusive-or) and NOT. Unlike the arithmetic operations a hexadecimal calculator isn't necessary to perform these operations. It is often easier to do them by hand than to use an electronic device to compute them. The logical AND operation is a dyadic operation (meaning it accepts exactly two operands). These operands are single binary (base 2) bits.

The AND operation is:

```
0 and 0 = 0
0 and 1 = 0
1 and 0 = 0
1 and 1 = 1
```

A compact way to represent the logical AND operation is with a truth table.

A truth table takes the following form:

AND Truth Table		
AND	**0**	**1**
0	**0**	**0**
1	**0**	**1**

This is just like the multiplication tables you encountered in elementary school. The column on the left and the row at the top represent input values to the AND operation.

The value located at the intersection of the row and column (for a particular pair of input values) is the result of logically ANDing those two values together. In English the logical AND operation is "If the first operand is one and the second operand is one the result is one; otherwise the result is zero."

One significant fact to note about the logical AND operation is that you can use it to force a zero result. If one of the operands is zero the result is always zero regardless of the other operand.

In the truth table above for example the row labelled with a zero input contains only zeros and the column labelled with a zero only contains zero results. Conversely if one operand contains a one the result is exactly the value of the second operand. These features of the AND operation are very important particularly when working with bit strings and we want to force individual bits in the string to zero. We will investigate these uses of the logical AND operation in the next segment.

The logical OR operation is also a dyadic operation.

Its definition is:

```
0 or 0 = 0
0 or 1 = 1
1 or 0 = 1
1 or 1 = 1
```

The truth table for the OR operation takes the following form:

OR Truth Table

OR	**0**	**1**
0	**0**	**1**
1	**1**	**1**

Colloquially the logical OR operation is "If the first operand or the second operand (or both) is one the result is one; otherwise the result is zero." This is also known as the inclusive-OR operation.

If one of the operands to the logical-OR operation is a one the result is always one regardless of the second operand's value. If one operand is zero the result is always the value of the second operand. Like the logical AND operation this is an important side-effect of the logical-OR operation that will prove quite useful when working with bit strings.

Note that there is a difference between this form of the inclusive logical OR operation and the standard English meaning. Consider the phrase "I am going to the store or I am going to the park." Such a statement implies that the speaker is going to the store or to the park but not to both places. Therefore the English version of logical OR is slightly different than the inclusive-OR operation; indeed it is closer to the exclusive-OR operation.

The logical XOR (exclusive-or) operation is also a dyadic operation.

It is defined as follows:

```
0 xor 0 = 0
0 xor 1 = 1
1 xor 0 = 1
1 xor 1 = 0
```

The truth table for the XOR operation takes the following form:

XOR Truth Table

XOR	0	1
0	0	1
1	1	0

In English the logical XOR operation is "If the first operand or the second operand but not both is one the result is one; otherwise the result is zero." Note that the exclusive-or operation is closer to the English meaning of the word "or" than is the logical OR operation.

If one of the operands to the logical exclusive-OR operation is a one the result is always the inverse of the other operand; that is if one operand is one the result is zero if the other operand is one and the result is one if the other operand is zero. If the first operand contains a zero then the result is exactly the value of the second operand. This feature lets you selectively invert bits in a bit string.

The logical NOT operation is a monadic operation (meaning it accepts only one operand).

It is:

```
NOT 0 = 1
NOT 1 = 0
```

The truth table for the NOT operation takes the following form:

NOT Truth Table

NOT	0	1
	1	0

Reasonable Operations on Binary Numbers and Bit Strings

The logical functions work only with single bit operands. Since the 80 × 86 uses groups of eight sixteen or thirty-two bits we need to extend the definition of these functions to deal with more than two bits. Logical functions on the 80x86 operate on a bit-by-bit (or bitwise) basis. Given two values these functions operate on bit zero producing bit zero of the result. They operate on bit one of the input values producing bit one of the result etc.

For example if you want to compute the logical AND of the following two eight-bit numbers you would perform the logical AND operation on each column independently of the others:

```
1011 0101
1110 1110
---------
1010 0100
```

This bit-by-bit form of execution can be easily applied to the other logical operations as well.

The logical operations in terms of binary values, it is a great deal easier to perform logical operations on binary values than on values in other bases. Therefore if you want to perform a logical operation on two hexadecimal numbers you should convert them to binary first. This applies to most of the basic logical operations on binary numbers (*e.g.* AND OR XOR etc.).

The ability to force bits to zero or one using the logical AND/OR operations and the ability to invert bits using the logical XOR operation is very important when working with strings of bits (*e.g.* binary numbers).

These operations let you selectively manipulate certain bits within some value while leaving other bits unaffected. For example if you have an eight-bit binary value 'X' and you want to guarantee that bits four through seven contain zeros you could logically AND the value 'X' with the binary value 0000 1111. This bitwise logical AND operation would force the H.O. four bits to zero and pass the L.O. four bits of 'X' through unchanged. Likewise you could force the L.O. bit of 'X' to one and invert bit number two of 'X' by logically ORing 'X' with 0000 0001 and logically exclusive-ORing 'X' with 0000 0100 respectively. Using the logical AND OR and XOR operations to manipulate bit strings in this fashion is know as masking bit strings. We use the term masking because we can use certain values (one for AND zero for OR/XOR) to 'mask out' certain bits from the operation when forcing bits to zero one or their inverse.

Signed and Unsigned Numbers

So far we've treated binary numbers as unsigned values. The binary number...00000 represents zero...00001 represents one...00010 represents two and so on towards infinity. What about negative numbers? Signed values have been tossed around and we've mentioned the two's complement numbering system but we haven't discussed how to represent negative numbers using the binary numbering system. That is what this section is all about!

To represent signed numbers using the binary numbering system we have to place a restriction on our numbers: they must have a finite and fixed number of bits. As far as the 80x86 goes this isn't too much of a restriction after all the 80x86 can only address a finite number of bits. For our purposes we're going to severely limit the number of bits to eight 16 32 or some other small number of bits.

With a fixed number of bits we can only represent a certain number of objects. For example with eight bits we can only represent 256 different objects. Negative values are objects in their own right just like positive numbers. Therefore we'll have to use some of the 256 different values to represent negative numbers. In other words we've got to use up some of the positive numbers to represent negative numbers.

To make things fair we'll assign half of the possible combinations to the negative values and half to the positive values. So we can represent the negative values -128..-1 and the positive values 0..127 with a single eight bit byte. With a 16-bit word we can represent values in the range –32 768..+32 767. With a 32-bit double word we can represent values in the range –2 147 483 648..+2 147 483 647. In general with n bits we can represent the signed values in the range –2**(n–1)to +2**(n–1)–1.

Exactly how do we do it? Well there are many ways but the 80x86 microprocessor uses the two's complement notation. In the two's complement system the H.O. bit of a number is a sign bit. If the H.O. bit is zero the number is positive; if the H.O. bit is one the number is negative. Examples:

For 16-bit numbers:

```
8000h is negative because the H.O. bit is one.
100h is positive because the H.O. bit is zero.
7FFFh is positive.
0FFFFh is negative.
0FFFh is positive.
```

If the H.O. bit is zero then the numeral is positive and is stored as a standard binary value. If the H.O. bit is one then the number is negative and is stored in the two's complement form.

To convert a positive number to its negative two's complement form you use the following algorithm:

- Invert all the bits in the number *i.e.* apply the logical NOT function.
- Add one to the inverted result.

For example to compute the eight bit equivalent of -5:

```
0000 0101 Five (in binary).
1111 1010 Invert all the bits.
1111 1011 Add one to obtain result.
```

If we take minus five and perform the two's complement operation on it we get our original value 00000101 back again just as we expect:

```
1111 1011 Two's complement for -5.
0000 0100 Invert all the bits.
0000 0101 Add one to obtain result (+5).
```

The following examples provide some positive and negative 16-bit signed values:

```
7FFFh: +32767
the largest 16-bit positive number.
8000h: -32768
the smallest 16-bit negative number.
4000h: +16
384.
```

To convert the numbers above to their negative counterpart (i.e. to negate them) do the following:

```
7FFFh: 0111 1111 1111 1111 +32
767t
1000 0000 0000 0000                    Invert  all  the
bits (8000h)
1000 0000 0000 0001 Add one (8001h or -32 767t)
8000h: 1000 0000 0000 0000 -32
768t
0111 1111 1111 1111 Invert all the bits (7FFFh)
1000 0000 0000 0000 Add one (8000h or -32768t)
4000h: 0100 0000 0000 0000 16
384t
1011 1111 1111 1111 Invert all the bits (BFFFh)
1100 0000 0000 0000 Add one (0C000h or -16 384t)
```

8000h inverted becomes 7FFFh. After adding one we obtain 8000h! Wait what's going on here? –(–32 768) is -32 768? Of course not. But the value +32 768 cannot be represented with a 16-bit signed number so we cannot negate the smallest negative value. If you attempt this operation the 80×86 microprocessor will complain about signed arithmetic overflow. Why bother with such a miserable numbering system? Why not use the H.O. bit as a sign flag storing the positive equivalent of the number in the remaining bits? The answer lies in the hardware. As it turns out negating values is the only tedious job. With the two's complement system most other operations are as easy as the binary system. For example suppose you were to perform the addition 5+(–5). The result is zero.

Consider what happens when we add these two values in the two's complement system:

```
                     00000101
                     11111011
                     --------
          1  00000000
```

We end up with a carry into the ninth bit and all other bits are zero. As it turns out if we ignore the carry out of the H.O. bit adding two signed values always produces the correct result when using the two's complement numbering system. This means we can use the same hardware for signed and unsigned addition and subtraction. This wouldn't be the case with some other numbering systems.

Except for the questions at the end of this chapter you will not need to perform the two's complement operation by hand. The 80x86 microprocessor provides an instruction NEG (negate) which performs this operation for you. Furthermore all the hexadecimal calculators will perform this operation by pressing the change sign key (+/– or CHS). Nevertheless performing a two's complement by hand is easy and you should know how to do it.

Once again you should note that the data represented by a set of binary bits depends entirely on the context. The eight bit binary value 11000000b could represent an IBM/ASCII character it could represent the unsigned decimal value 192 or it could represent the signed decimal value –64 etc. As the programmer it is your responsibility to use this data consistently.

Sign and Zero Extension

Since two's complement format integers have a fixed length a small problem develops. What happens if you need to convert an eight bit two's complement value to 16 bits? This problem and its converse (converting a 16 bit value to eight bits) can be accomplished via sign extension and contraction operations. Likewise the 80x86 works with fixed length values even when processing unsigned binary numbers. Zero extension lets you convert small unsigned values to larger unsigned values.

Consider the value "–64". The eight bit two's complement value for this number is 0C0h. The 16-bit equivalent of this number is 0FFC0h. Now consider the value "+64". The eight and 16 bit versions of this value are 40h and 0040h. The difference between the eight and 16 bit numbers can be described by the rule: "If the number is negative the H.O. byte of the 16 bit number contains 0FFh; if the number is positive the H.O. byte of the 16 bit quantity is zero."

To sign extend a value from some number of bits to a greater number of bits is easy just copy the sign bit into all the additional bits in the new format. For example to sign extend an eight bit number to a 16 bit number simply copy bit seven of the eight bit number into bits 8..15 of the 16 bit number. To sign extend a 16 bit number to a double word simply copy bit 15 into bits 16..31 of the double word.

Sign extension is required when manipulating signed values of varying lengths. Often you'll need to add a byte quantity to a word quantity. You must sign extend the byte quantity to a word before the operation takes place. Other operations (multiplication and division in particular) may require a sign extension to 32-bits. You must not sign extend unsigned values.

Examples of sign extension:

Eight Bits	Sixteen Bits	Thirty-two Bits
80h	FF80h	FFFFFF80h
28h	0028h	00000028h
9Ah	FF9Ah	FFFFFF9Ah
7Fh	007Fh	0000007Fh
---	1020h	00001020h
---	8088h	FFFF8088h

To extend an unsigned byte you must zero extend the value. Zero extension is very easy - just store a zero into the H.O. byte(s) of the smaller operand. For example to zero extend the value 82h to 16-bits you simply add a zero to the H.O. byte yielding 0082h.

Eight Bits	Sixteen Bits	Thirty-two Bits
80h	0080h	00000080h
28h	0028h	00000028h
9Ah	009Ah	0000009Ah
7Fh	007Fh	0000007Fh
---	1020h	00001020h
---	8088h	00008088h

Sign contraction converting a value with some number of bits to the identical value with a fewer number of bits is a little more troublesome. Sign extension never fails. Given an m-bit signed value you can always convert it to an n-bit number (where n > m) using sign extension. Unfortunately given an n-bit number you cannot always convert it to an m-bit number if m < n. For example consider the value –448. As a 16-bit hexadecimal number its representation is 0FE40h. Unfortunately the magnitude of this number is too great to fit into an eight bit value so you cannot sign contract it to eight bits. This is an example of an overflow condition that occurs upon conversion. To properly sign contract one value to another you must look at the H.O. byte(s) that you want to discard. The H.O. bytes you wish to remove must all contain either zero or 0FFh. If you encounter any other values you cannot contract it without overflow. Finally the H.O. bit of your resulting value must match every bit you've removed from the number.

Examples (16 bits to eight bits):

FF80h can be sign contracted to 80h

0040h can be sign contracted to 40h

FE40h cannot be sign contracted to 8 bits.

0100h cannot be sign contracted to 8 bits.

Division Arithmetic

Division is the most difficult of the basic arithmetic operations. For a simple computer that uses a single adder circuit for its arithmetic operations, a variant of the conventional long division method used manually, called *nonrestoring division* provides greater simplicity and speed.

This method proceeds as follows, assuming without loss of generality (which means we can fix things by complementing the operands and remembering what we've done, if it isn't so) that both operands are positive:

If the divisor is less than the dividend, then the quotient is zero, the remainder is the dividend, and one is finished.

Otherwise, shift the divisor as many places left as is necessary for its first one bit to be in the same position as the first one bit in the dividend. Also, shift the number one the same number of places left; the result is called the quotient term.

The quotient value starts at zero: Then, do the following until the divisor is shifted right back to its original position:

If the current value in the dividend register is positive, and it has a one bit corresponding to the starting one bit of the value in the divisor register (initially, the divisor as shifted left), subtract the divisor register contents from the dividend register, and add the quotient term to the quotient register. If the current value in the dividend register is negative, and it has a zero bit corresponding to the starting one bit of the value in the divisor register, add the divisor register contents to the dividend reigster, and subtract the quotient term from the quotient register.

Shift the divisor register and the quotient term one place to the right, then repeat until finished (when the quotient term becomes zero at this step, do not repeat).

If, after the final step, the contents of the dividend register are negative, add the original divisor to the dividend register and subtract one from the quotient register. The dividend register will contain the remainder, and the quotient register the quotient.

An example of this is shown below:

Divide 10010100011 by 101101.

```
00010010100011 DD         0000000    Q
10110100000    DR         100000     QT
11111100000011 DD         0100000    Q
1011010000     DR         10000      QT
11111100000011 DD         0100000    Q
```

```
101101000      DR         1000      QT
1111110000011 DD          0100000   Q
10110100       DR         100       QT
1111111011011 1 DD        0011100   Q
1011010        DR         10        QT
0000000001000 1 DD        0011010   Q
101101         DR         1         QT
0000000001000 1 remainder 11010     quotient
```

which produces the correct result; 1187 divided by 45 gives 26 with 17 as the remainder.

Speeding up division further is also possible: One approach would be to begin with the divisor, from which 8, 4, and 2 times the divisor can be immediately derived, and then with one layer of addition stages, derive 3 (and hence 6 and 12) times the divisor, 5 (and hence 10) times the divisor, and 9 times the divisor, and then with a second layer of addition stages, derive the remaining multiples from 1 to 15 of the divisor.

Then an assembly of adders working in parallel to determine the largest multiple that could be subtracted from the dividend or the remaining part of it without causing it to go negative could generate four bits of the quotient in the time a conventional division algorithm could generate one.

The decimal version of this technique was used in the NORC computer, a vacuum tube computer designed for very high-speed calculation.

Another method of division is known as SRT division. In its original form, it was a development of nonrestoring division. Instead of choosing, at each bit position, to add or subtract the divisor, the option of doing nothing, and skipping quickly over several bits in the partial remainder, is also included.

Starting from the same example as given above for nonrestoring division:

Divide 10010100011 by 101101.

```
00010010100011    DD          0000000   Q
10110100000       DR          100000    QT
11111100000011    DD          0100000   Q
10110100          DR          100       QT
11111110110111    DD          0011100   Q
1011010           DR          10        QT
00000000010001    remainder 11010       quotient
```

Thus, when the partial remainder is positive, we align the divisor so that its first 1 bit is under the first one bit of the partial remainder, and subtract; we add a similarly shifted 1 to the quotient.

When the partial remainder is negative, we align the divisor so that its first 1 bit is under the first zero bit of the partial remainder,and add; we subtract a similarly shifted 1 from the quotient.

In the example, an immediate stop is shown when the right answer was achieved; normally, two additional steps would take place; first, a subtraction of the divisor, and then, since the result is negative, an addition in the same digit position; this is the same second chance without shifting as is used to terminate nonrestoring division. The property of shifting at once over multiple zeroes is no longer present in the high-radix forms of SRT division. Thus, in radix 4 SRT division, one might, at each step, either do nothing, add or subtract the divisor at its current shifted position, or add or subtract twice the divisor at its current shifted position. Instead of a simple rule of adding to a zero, and subtracting from a 1, a table of the first few bits of both the partial remainder and the divisor is needed to determine the appropriate action at each step.

To achieve time proportional to the logarithm of the length of the numbers involved, a method is required that attempts to refine an approximation of the reciprocal of the divisor. This basic method, which uses the recurrence relation

```
r' = r * (2 - r*x)
```

where x is the number whose reciprocal is to be found, and r and r' are two successive approximations to its reciprocal, is known as Goldschmidt division, and is described in U.S. patent 3,508,038 with inventors Goldschmidt and Powers, assigned to IBM, and was developed for the IBM System/360 Model 91 computer.

In the Model 91 computer, the results of floating-point divisions were rounded instead of truncated; this was an improvement on the behaviour of the other computers in the System/360 line, but it was still an incompatibility; one page claims that this was corrected in the Model 195 by prescaling, but refers to the characteristic of the Model 91 as though it were a serious bug, producing genuinely wrong answers, which it was not.

The recurrence relation can be made more understandable by splitting it into two parts.

Given that r is an approximation to the reciprocal of x, then we can consider r*x to be 1+e, where e is a small error term reflecting the proportion by which r differs from the real reciprocal. Thus, we can have as our first equation, containing one multiplication,

```
e = r*x - 1
```

If r is (1+e) times the real reciprocal, (1-e) times q will be a much closer approximation to the reciprocal, since the result will be (1-e^2) times the real reciprocal. Thus, the second multiplication can be part of the equation:

```
r' = (1 - e)*r
```

An advantage of leaving the recurrence relation in the form from which it was originally derived is that e is a small quantity compared to 1+e, and so significance can be preserved. We will see how this can be used below.

One difficulty with the use of iterative methods for division is that they do not naturally lend themselves to producing the most accurate result in every case, as required by the IEEE 754 standard.

The obvious way of dealing with this is as follows: If one is calculating a/b, one begins by approximating the reciprocal of b.

Because the accuracy of that approximation doubles with each iteration, there will be some excess precision available at the final iteration.

Obtain an approximation to a/b by multiplying this result by a at the full working precision of the calculation.

If the part of that result that needs to be rounded off to fit the quotient into the desired format is close enough to.4999... or.5000... to create concern (thinking of the mantissa as being scaled so as to become an integer in its ultimate destination), then, first replace that part by an exact.5. Multiply it by b, the divisor. If the result is greater than a, then.5 is too high, so round down. If the result is less than a.5 is too low, so round up.

Because allowing a division to take a variable amount of time could interfere with pipelining in some computer designs, work has been done on finding improved algorithms for IEEE 754 compliant Goldschmidt division.

A table-driven method of division for arguments of limited width, described in a paper by Hung, Fahmy, Mencer, and

Flynn, can also be used to obtain an excellent initial approximation to the reciprocal in the time required for two multiplications; conventional techniques can then double the precision of the approximation in each additional multiplication time used.

To divide A by B: B will be normalized before starting, so that its value is between 1 and 2.

The first few bits of B will be used to obtain two values from a table; one value will be an approximation to 1/B, and the other will be the derivative of the function 1/B with respect to B at the point indicated by the first few bits of B.

In the first multiplication time, multiply, in parallel, both A and B by the approximation to 1/B found in the table, and also multiply 1 plus the remaining bits of B, not used to find a table entry, by the second table entry.

In the second multiplication time, multiply, in parallel, both the modified A and the modified B by the product involving the remaining bits of B.

In subsequent steps, where the modified B is 1 + epsilon, multiply both the modified A and the modified B by 1 – epsilon. Since 1.001 times.999 is.999999, this doubles the precision in each step.

Basically, what is being done here is: as the reciprocal of the divisor is being determined by repeatedly multiplying the divisor by numbers that will bring it closer to 1, the dividend is being multiplied, in parallel, by these numbers, so that it does not need to be multiplied by the reciprocal of the divisor separately, at the end of the computation. Is one exchanging one multiplier for several? Not really, as the reciprocal of the divisor would have to be constructed from several individual multiplications if it were used explicitly. The need for several multipliers, instead of just two, comes from the desire to allow full pipelining.

It is the high-precision starting approximation obtained in the first two steps that was the unique contribution of this chapter. Producing the quotient in parallel with the reciprocal as outlined above is also a potential feature of hardware implementations of the methods of producing accurate quotients by the iterative method outlined in the IBM and Intel patents referred to above.

Thus, the procedure given in the IBM patent involves a recurrence relation with four operations; the reciprocal of b, approximated by r, is improved by the two-step recurrence relation:

```
 m = 1 - b*r
r' = r + m*r
```

(note that m stands for minus epsilon) and the quotient is improved in parallel by the operations:

```
s = a - b*q
q = q + s*r
```

By choosing an initial approximation to the reciprocal of b that is guaranteed to be greater than 1/b rather than less than 1/b for a divisor b whose mantissa consists entirely of ones, the final step is guaranteed to produce a result that will round properly. Once enough approximation steps are performed so that the final r has the precision required, the two steps to improve the quotient are performed one extra time, with the second of the two steps also used to set the condition codes for the division operation as a whole. (All the preceding steps must be performed as round to nearest regardless of the rounding mode that is selected.)

In the Intel improvement of this method, it is noted that once epsilon, the error in an approximation to the reciprocal of b, is known, epsilon squared can be computed directly, instead of being discovered as the error in a subsequent step. Thus, instead of approximating 1/(1+e) as 1-e, one can directly calculate an infinite product yielding the reciprocal,

$(1 - e) * (1 + e^2) * (1 + e^4) * (1 + e^8)...$

since,

$(1 + e) * (1 - e) = 1 - e^2$
$(1 - e^2) * (1 + e^2) = 1 - e^4$
$(1 - e^4) * (1 + e^4) = 1 - e^8$
$(1 - e^8) * (1 + e^8) = 1 - e^{16}$

and so on: if one thinks of the first term as 1-e insted of 1+e, then all the corrections are in the same direction.

A method to obtain a further improvement in efficiency in this type of calculation is discussed in a paper by Ercegovac, Imbert, Matula, Muller, and Wei from Inria. It can happen that the initial approximation to the reciprocal of the divisor is such that as it doubles in accuracy in each step, instead of having half the required accuracy just before the final step, it may be short of the required accuracy by only a few bits. In that case, instead of squaring e^n to obtain e^(2n), one could simply look up 1 + e^(2n) in a table using the first few bits of e, and this could be done at an early point in the computation. Although it still takes two multiplies per iteration, it is also still an improvement, because it now takes only one add.

Square Root

Using the formula (a+b)^2 = a^2 + 2ab + b^2, a method for calculating square roots by hand was devised that resembles long division:

```
8. 4 2 6 1 4 9
------------------------
71. 00 00 00 00 00 00   |
64                     |
-----------------------  |
 7.00                   | 1 6
 6.56                   | 4
 ---------------------- |
 44 00                  | 16 8
 33 64                  | 2
 -----------------------|
   10 36 00             | 1 68 4
   10 10 76             | 6
   -------------------- |
      25 24 00          | 16 85 2
      16 85 21          | 1
      ----------------- |
        8 38 79 00      | 1 68 52 2
        6 73 29 44      | 4
        --------------- |
        1 55 49 56 00   | 16 85 22 8
        1 51 67 06 01   | 9
        --------------- |
           3 82 49 99   |
```

To find the square root of 71, first, it is noted that 71 is between 64 and 81. Thus, we subtract 8 squared, or 64, from 71, and note that our square root will be 8 point something.

In subsequent steps, what has gone before will play the role of a in the formula above, and the next digit in the square root will play the role of b. And, as with 8, a^2 has already been subtracted, so, for whatever digit we choose as b, we must be able to subtract 2ab + b^2 from what remains.

Thus, let us choose 4 as the next digit in the square root. We will subtract 2 * 8 *.4 plus.4 squared, which is 6.56, from the number.

On the next line, what we are dividing by (with the allowance that the b^2 term may change what we use) is 2 *

8.4, which is 16.8. This goes into.44 a bit more than.02 times, so we get 33 60 plus 04, or 33 64 to subtract.

8.42 times 2 is 16.84; this goes into.1036.006 times, and so we get 10 10 76 to subtract. Finally, we start with 8.426 * 2, which is 16.852, and divide.002524 by it, to get 1, and so we subtract 16 85 21.

As the last decimal place of a is one position ahead of b, if we treat both terms as integers instead of as real numbers, the formula becomes 20*ab+b^2, which is what is used when this is done in a more mechanical fashion.

Other than noting that as the numbers involved keep growing, this is not useful as a method of pseudorandom number generation, not much more need be said.

The resemblance between this method of calculating square roots and long division is sufficiently close that it is possible to adapt, for example, high-radix SRT division to calculating square roots. If the first part of the square root is determined from a table, the 2ab term will be large enough, compared to the b^2 term, that a modified table, giving the value of b to try from 2a and the current remainder will keep the error small enough so that it is always possible to proceed to the next digit on the next step.

Another method of approximating the square root of a number is available for computers, which is much faster for sufficiently large numbers, or when performed in software, known as Newton-Raphson iteration. In fact, the method of approximating the reciprocal of a number shown above is another example of Newton-Raphson iteration.

If r is our existing approximation, and r' is the improved approximation, for 1/x, the recurrence relation was:

$$r' = r * (2 - r*x)$$

since if r = (1/x)*(1+e), for some small e, 2 - x*r is 1-e, and (1+e)*(1-e) is 1 - e^2, which is much closer to one.

For square root, we get the recurrence relation:

$$r' = \frac{1}{2}\left(\frac{x}{r} + r\right)$$

and, as x/r and r have, to the first order, equal and opposite discrepancies from the square root of x, r' is once again much closer to the right answer than r.

This is the classic Newton-Raphson iteration for square root. Given that division is slower than multiplication, can it be improved upon?

One simple-minded approach might be to start with an approximation to the square root, r, and an approximation to the reciprocal of the square root, q, and also calculate y, the reciprocal of x, at the beginning.

Then, use the pair of recurrence relations:

$$r' = \frac{1}{2}(qx + r)$$

$$q' = \frac{1}{2}(q + ry)$$

If q is initially the reciprocal of r, then r' and q' are better approximations than r and q after the first step. Perhaps they will continue to improve in later steps, a less elaborate iteration which simply improves q that apparently has been used in practice both on Burroughs and Cray machines (my source, which claimed that, had a typographical error, and omitted the factor of 1/2), is the following:

$$q' = \frac{1}{2}\, q * (3 - x * q * q)$$

Let us suppose that q is equal to the true reciprocal of the square root of x, which we will call p, times (1 + e). Since e is small, 1/(1+e) is approximately 1-e.

Then, q * (3 - x * q * q) becomes, approximately, p * (1 + e) * (3 – (1 + 2e)), which is p * (1 + e) * (2 – 2e), which is approximately 2p, and, thus, half of that is a new approximation in which the new error is of the order of the square of the old error.

Log and Trig Functions: The CORDIC Algorithm and Related Methods

A fast method of calculating trigonometric functions using only shifts and adds was described in a paper which described how it was applied to a device bearing the acronym CORDIC as its name.

A vector (x,y) can be rotated through an angle theta by means of the equations:

```
x' = x cos(theta) - y sin(theta)
y' = y cos(theta) + x sin(theta)
```

In the CORDIC algorithm, a table is required whose contents are:

```
arctangent(1/2)
arctangent(1/4)
arctangent(1/8)
arctangent(1/16)
...
```

Beginning with suitable starting values of x and y, one iterates through a fixed number of steps wherein one performs either

```
x' = x - y M
y' = y + x M
```

or

```
x' = x + y M
y' = y - x M
```

M starts as 1/2, and is divided by two at each step, so multiplying x and y by M is simply a shift. This enlarges the vector (x,y) in addition to rotating it by plus or minus the arctangent of M.

If one is calculating the sine or cosine of an angle, one adds or subtracts the table value from the angle to bring the result closer to zero, to decide which operation is to be performed. Then, the starting values of x and y are chosen so that the successive enlargements will lead, at the end, to a vector whose length is exactly 1.

If one is trying to find an inverse trigonometric function, the goal is instead to make y equal to zero, although striving to make x equal to y will also work, provided one adds 45 degrees to the angle one uses as the starting point.

In that case, one wishes to find theta, so the scaling of x and y are irrelevant; therefore, the rule about performing either a clockwise or counterclockwise rotation at each step is no longer required.

To find either the arcsine or arccosine, one has to first modify the input by a calculation involving a square root, as due to the scaling at each step, only the arctangent can be found directly by this method.

For that method, one requires a table containing the values:

```
log(1 1/2)
log(1 1/2)
log(1 1/4)
log(1 1/8)
log(1 1/16)
...
```

and to find the logarithm of a number between 1 and 2, one at each step chooses either to leave it alone, or add it to itself shifted right by the number of places N that is also the number of the step; the latter is done if its result is less than or equal to 2.

Since, in this method, we don't have to worry about always having to perform a clockwise or counterclockwise rotation in each step, so that the scaling factor is not altered, it is easy to adapt this method to decimal arithmetic. One can have a table of log(2), log(1.1), log(1.01), and so on, and just repeat the steps at each level up to nine times. This was indeed how logarithms were calculated on the HP-35 calculator, for example.

Could one just always go through five steps for each digit? Or six steps? That won't quite work, since however many steps one uses, the total number of steps is either even or odd. So one would have to go through ten steps for each digit instead; after using arctan(0.001), the error going into using arctan(0.0001) would be up to a factor of 1.002, not 1.001.

Given that, there is no loss using a simpler method; when it is necessary to worry about the scale factor (it is not for inverse trig functions, or for calculating the tangent, it is only a concern for sine and cosine) one still can simply either perform the calculation with the arctangent of decimal 0.1, 0.01, 0.001, 0.0001 and so on or not in up to five steps in either direction. At the end, just calculate sqrt(x*x+y*y) to obtain the scale factor.

Although these methods are quite rapid compared to the conventional method of using the Taylor series to approximate a function, they are not the fastest methods known, at least asymptotically for numbers with very high precision. It is possible to evaluate these elementary transcendental functions, as it is possible to perform division, in a time comparable to that required for multiplication; that is, in a number of addition times proportional to the logarithm of the length of the number.

Originally, CORDIC methods were applied to small computer systems with limited hardware resources, to which they are very well suited, but despite the possibility of better methods in theory for the case of very high precisions, the CORDIC methods are still useful in large computers as well, if they calculate transcendental functions in hardware instead of software.

They are superseded by other methods, though, for multi-precision arithmetic, particularly in the régimes where one would use Schönhage-Strassen multiplication.

These algorithms, as described here, require comparisons to be made at each step, on the basis of which decisions are made. While it would not be possible to make the exact comparisons noted here without completing the additions, and performing carry propagation, can comparisons of a limited type be performed on numbers in the raw redundant form used with carry-save adders, and, if so, can these algorithms be modified to make use of such comparisons? One encouraging factor is that, since multiplication is not used, we do not necessarily have to contend with the negative values that are required for Booth encoding.

If instead of having one binary number, we have two binary numbers that we wish to avoid adding, and we want to say something about their combined value by looking only at their first few bits, we can indeed say that if both numbers are of the form 000xxxxx, their sum is of the form 00xxxxxx.

But some pairs of numbers not of the form 000xxxxx will also have sums of the form 00xxxxxx.

The modification of the CORDIC algorithms for this situation that suggests itself is something based on using twice as many steps, involving changes to the numbers of half the size at each step.

The hyperbolic functions sinh and cosh are defined as:

```
              x  -  x
              e  -  e
sinh(x)=      ───────
              2
              x  -  x
              e  +  e
cosh(x)=      ───────
              2
```

We know that:

x + y × y

e = e * e

and so we can derive the equations,

```
    sinh(x+y)  =  sinh(x)cosh(y)  +  cosh(x)sinh(y)
    cosh(x+y)  =  sinh(x)sinh(y)  +  cosh(x)cosh(y)
```

Here, since cosh(x) is always greater than sinh(x), we can choose values of y such that 1/(tanh(y)) is equal to 2, 1 1/2, 1 1/4, 1 1/8, and so on to produce sinh(x) and cosh(x) for arbitrary values of x, starting from some fixed value of x.

Thus, for each step, we once again have:

$$\mathbf{K_n * \sinh(x + y_n) = \sinh(x)\ (1 + 2^{-n}) + \cosh(x)}$$

$$\mathbf{K_n * \cosh(x + y_n) = \sinh(x) + \cosh(x)\ (1 + 2^{-n})}$$

where K[n] is 1/sinh(y[n]), where (1 + 2^(–n)) = 1/tanh(y[n]).

As with the trigonometric functions, sinh(–x) = -sinh(x) and cosh(-x) = cosh(x), so, once again, we can “rotate” either clockwise or counterclockwise at each step, and we need to do so to keep the scaling consistent at the end *if it is sinh,* *cosh, or the exponential function we are calculating.*

Since cosh(x) + sinh(x) = e^x, this can be used as an alternative method of calculating the exponential function, or even logarithms. To calculate logarithms, we would start from a value for x based on the number whose logarithm we wish, so that the test performed in each round on the current approximation would not require scaling. Again, if the logarithm is desired, scaling ceases to be an issue.

Logarithms

Since addition is quicker and requires less circuitry than multiplication, some systems convert numbers to their logarithms at the time of input, retaining them in this form only in memory.

Occasionally, though, one has to add and subtract as well as multiply and divide. A paper by S. C. Lee and A. D. Edgar, "The FOCUS number system", described a way to do this using a relatively short look-up table. Given two numbers a and b, and assuming without loss of generality that a is less than b, one can easily calculate r=a/b. If one then uses r to find (r+1)/r in a table, the value from the table can be multiplied by b to give a+b.

This table is shorter than a lookup table for converting to or from a logarithmic representation, because cases where a is negligible compared to b can be omitted. A table of (r-1)/r is also needed for subtraction.

Unfortunately, there does not seem to be a good way to prevent errors from accumulating, so that one could adapt this to multi-precision arithmetic. However, floating-point numbers are used in implementations of Schönhage-Strassen arithmetic, and so there may be possibilities in that direction.

Floating Point Arithmetic

Arithmetic operations on floating point numbers consist of addition, subtraction, multiplication and division the operations are done with algorithms similar to those used on sign magnitude integers (because of the similarity of representation)—example, only add numbers of the same sign. If the numbers are of opposite sign, must do subtraction.

Addition

Example on decimal value given in scientific notation:

```
  3.25 × 10 ** 3
+ 2.63 × 10 ** –1
```

First step: align decimal points

Second step: add

```
   3.25     × 10 ** 3
+  0.000263 × 10 ** 3
---------------------
   3.250263 × 10 ** 3
```

(presumes use of infinite precision, without regard for accuracy)third step: normalize the result (already normalized!) example on fl pt. value given in binary:

.25 = 0 01111101 00000000000000000000000

100 = 0 10000101 10010000000000000000000

to add these fl. pt. representations.

*Step*1: Align radix points shifting the mantissa LEFT by 1 bit DECREASES THE EXPONENT by 1 and shifting the mantissa RIGHT by 1 bit INCREASES THE EXPONENT by 1 we want to shift the mantissa right, because the bits that fall off the end should come from the least significant end of the mantissa -> choose to shift the.25, since we want to increase it’s exponent. -> shift by

```
  10000101
 –01111101
  00001000 (8) places.
```

0 01111101 00000000000000000000000 (original value)

0 01111110 10000000000000000000000 (shifted 1 place)

(note that hidden bit is shifted into msb of mantissa)

0 01111111 01000000000000000000000 (shifted 2 places)

0 10000000 00100000000000000000000 (shifted 3 places)

0 10000001 00010000000000000000000 (shifted 4 places)

0 10000010 00001000000000000000000 (shifted 5 places)

0 10000011 00000100000000000000000 (shifted 6 places)

0 10000100 00000010000000000000000 (shifted 7 places)

0 10000101 00000001000000000000000 (shifted 8 places)

Step: Add (don’t forget the hidden bit for the 100)

```
  0 10000101 1.10010000000000000000000 (100)
 +0 10000101 0.00000001000000000000000 (.25)
  0 10000101 1.10010001000000000000000
```

Step: normalize the result (get the “hidden bit” to be a 1) it already is for this example.result is

0 10000101 10010001000000000000000

Subtraction

like addition as far as alignment of radix points then the algorithm for subtraction of sign mag. numbers takes over before subtracting, compare magnitudes (don’t forget the hidden bit!) change sign bit if order of operands is changed.don’t forget to normalize number afterward.

Multiplication

Example on decimal values given in scientific notation:

```
  3.0 × 10 ** 1
 +0.5 × 10 ** 2
```

algorithm: multiply mantissas add exponents

```
  3.0  ×10 ** 1
+ 0.5  ×10 ** 2
---------------
  1.50× 10 ** 3
```

example in binary: use a mantissa that is only 4 bits so that I don't spend all day just doing the multiplication part.

```
  0 10000100 0100
× 1 00111100 1100
-----------------
```

```
mantissa multiplication:     1.0100
(don't forget hidden bit)×   1.1100
       --------
       00000
       00000
       10100
       10100
       10100
       ----------
   1000110000
   becomes 10.00110000
```

Add exponents: always add true exponents (otherwise the bias gets added in twice)biased:

```
  10000100
+00111100
----------
```

```
  10000100 01111111 (switch the order of the subtraction,
– 01111111 - 00111100 so that we can get a negative value)
-----------------------------------------------------------
  00000101 01000011
```

true exp true exp is 5. is –67

add true exponents 5 + (–67) is–62.

Re-bias exponent:–62 + 127 is 65.

unsigned representation for 65 is 01000001. put the result back together (and add sign bit). 1 01000001 10.00110000 normalize the result: (moving the radix point one place to the left increases the exponent by 1.) 1 01000001 10.00110000 becomes 01000010 1.000110000 this is the value stored (not the hidden bit!): 1 01000010 000110000

Division

Similar to multiplication.

True division: Do unsigned division on the mantissas (don't forget the hidden bit) subtract TRUE exponents The IEEE standard is very specific about how all this is done.

Unfortunately, the hardware to do all this is pretty slow.

Some comparisons of approximate times:

2's complement integer add	1 time unit
fl. pt add	4 time units
fl. pt multiply	6 time units
fl. pt. divide	13time units

There is a faster way to do division. Its called division by reciprocal approximation. It takes about the same time as a fl. pt. multiply. Unfortunately, the results are not always the same as with true division.

Division by reciprocal approximation: Instead of doing a/b they do a × 1/b. figure out a reciprocal for b, and then use the fl. pt. multiplication hardware. example of a result that isn't the same as with true division.

True division: 3/3	= 1 (exactly)
Reciprocal approx:	1/3 =.33333333
3 ×.33333333	=.99999999, not 1

It is not always possible to get a perfectly accurate reciprocal.

Issues in Floating Point

Note: this discussion only touches the surface of some issues that people deal with. Entire courses could probably be taught on each of the issues.

Rounding

Arithmetic operations on fl. pt. values compute results that cannot be represented in the given amount of precision. So, we must round results. There are MANY ways of rounding. They each have "correct" uses, and exist for different reasons. The goal in a computation is to have the computer round such that the end result is as "correct" as possible. There are even arguments as to what is really correct.

Methods of Rounding

Round towards 0—also called truncation. Figure out how many bits (digits) are available. Take that many bits (digits) for the result and throw away the rest. This has the effect of making the value represented closer to 0.

Example:

.7783 if 3 decimal places available.778

if 2 decimal places available.77

round towards + infinity—regardless of the value, round towards +infinity.

Example:

1.23 if 2 decimal places, 1.3

– 2.86 if 2 decimal places, -2.8

round towards-infinity —regardless of the value, round towards -infinity.

Example:

1.23 if 2 decimal places, 1.2

– 2.86 if 2 decimal places, -2.9

in binary—rounding to 2 digits after radix point round towards + infinity:

```
1.1101
   |
1.11 | 10.00
   ───
1.001
   |
1.00 | 1.01
   ──
round towards-infinity —
1.1101
   |
1.11 | 10.00
───
1.001
   |
1.00 | 1.01
──
```

round towards zero (TRUNCATE):

```
1.1101
   |
1.11 | 10.00
───
1.001
     |
1.00 | 1.01
──
-1.1101
   |
-10.00 | -1.11
───
-1.001
   |
-1.01 | -1.00
```

Round Towards Nearest

Odd case: if there is anything other than 1000... to the right of the number of digits to be kept, then rounded in IEEE standard such that the least significant bit (to be kept) is a zero.

```
1.1111
   |
1.11 | 10.00
```

```
———
1.1101
   |
1.11 | 10.00
———
1.001 (ODD CASE)
   |
1.00 | 1.01
——
-1.1101 (1/4 of the way between)
   |
-10.00 | -1.11
———
-1.001 (ODD CASE)
   |
-1.01 | -1.00
——
```

Note: this is a bit different than the "round to nearest" algorithm for the "tie" case.5) learned in elementary school for decimal numbers.

Use of Standards

—> allows all machines following the standard to exchange data and to calculate the exact same results.

—> IEEE fl. pt. standard sets parameters of data representation (# bits for mantissa vs. exponent)

—> Pentium architecture follows the standard

Overflow and Underflow

Just as with integer arithmetic, floating point arithmetic operations can cause overflow. Detection of overflow in fl. pt. comes by checking exponents before/during normalization. Once overflow has occurred, an infinity value can be represented and propagated through a calculation.

Underflow occurs in fl. pt. representations when a number is to small (close to 0) to be represented. (show number line!)

if a fl. pt. value cannot be normalized (getting a 1 just to the left of the radix point would cause the exponent field to be all 0's) then underflow occurs.

HW vs. SW computing

Floating point operations can be done by hardware (circuitry) or by software (program code).

-> a programmer won't know which is occurring, without prior knowledge of the HW.

-> SW is much slower than HW. by approx. 1000 times.

A difficult (but good) exercise for students would be to design a SW algorithm for doing fl. pt. addition using only integer Operations.

SW to do fl. pt. operations is tedious. It takes lots of shifting and masking to get the data in the right form to use integer arithmetic operations to get a result—and then more shifting and masking to put the number back into fl. pt. format. A common thing that manufacturers used to do is to offer 2 versions of the same architecture, one with HW, and the other with SW fl. pt. ops.

Booths Algorithm and Array Multiplier

Definition of an Algorithm

In the introduction, we gave an informal definition of an algorithm as "a set of instructions for solving a problem" and we illustrated this definition with a recipe, directions to a friend's house, and instructions for changing the oil in a car engine. You also created your own algorithm for putting letters and numbers in order. While these simple algorithms are fine for us, they are much too ambiguous for a computer. In order for an algorithm to be applicable to a computer, it must have certain characteristics. We will specify these characteristics in our formal definition of an algorithm.

An algorithm is a well-ordered collection of unambiguous and effectively computable operations that when executed produces a result and halts in a finite amount of time. With this definition, we can identify five important characteristics of algorithms.

- Algorithms are well-ordered.
- Algorithms have unambiguous operations.
- Algorithms have effectively computable operations.
- Algorithms produce a result.
- Algorithms halt in a finite amount of time.

These characteristics need a little more explanation, so we will look at each one in detail.

Algorithms are Well-ordered

Since an algorithm is a collection of operations or instructions, we must know the correct order in which to execute the instructions. If the order is unclear, we may perform the wrong instruction or we may be uncertain which instruction should be performed next. This characteristic is especially important for computers. A computer can only execute an algorithm if it knows the exact order of steps to perform.

Algorithms have Unambiguous Operations

Each operation in an algorithm must be sufficiently clear so that it does not need to be simplified. Given a list of numbers, you can easily order them from largest to smallest with the simple instruction "Sort these numbers." A computer, however, needs more detail to sort numbers. It must be told to search for the smallest number, how to find the smallest number, how to compare numbers together, etc.

The operation "Sort these numbers" is ambiguous to a computer because the computer has no basic operations for sorting.

Basic operations used for writing algorithms are known as primitive operations or primitives.

When an algorithm is written in computer primitives, then the algorithm is unambiguous and the computer can execute it.

Algorithms have Effectively Computable Operations

Each operation in an algorithm must be doable, that is, the operation must be something that is possible to do. Suppose you were given an algorithm for planting a garden where the first step instructed you to remove all large stones from the soil.

This instruction may not be doable if there is a four ton rock buried just below ground level. For computers, many mathematical operations such as division by zero or finding the square root of a negative number are also impossible. These operations are not effectively computable so they cannot be used in writing algorithms.

Algorithms Produce a Result

In our simple definition of an algorithm, we stated that an algorithm is a set of instructions for solving a problem. Unless an algorithm produces some result, we can never be certain whether our solution is correct.

Have you ever given a command to a computer and discovered that nothing changed? What was your response? You probably thought that the computer was malfunctioning because your command did not produce any type of result.

Without some visible change, you have no way of determining the effect of your command. The same is true with algorithms. Only algorithms which produce results can be verified as either right or wrong.

Algorithms Halt in a Finite Amount of Time

Algorithms should be composed of a finite number of operations and they should complete their execution in a finite amount of time. Suppose we wanted to write an algorithm to print all the integers greater than 1.

5

Designing Software

Designing implies a special regard for the requirement of appearance and improvement of marketability by imparting an attractive appearance with current fashions, within the margins of variation imposed by the requirements of use and economy. The importance of consumer appeal in design is shown by the recent expansion of home design software.

Home design software can assist if you are having a problem visualizing what you so desire in your home. It is a compelling tool used in the design planning and building of the ultimate home. Home design software also goes beyond what you see in planning books, by customizing your home design to match your needs!

Before purchasing your home design software research the different products available and their feature capabilities. Today a lot of these programmes come with step by step instructions, wizards, drag and drop down menus with furniture (It allows you to position interior objects such as beds, chairs and tables, appliances and more within your floor plan creating a virtual feel for the size, layout and function of each room), home fixtures, plants to add to plans, decks and patios and helpful videos.

Most of them also include both a 2-D and 3-D view. Some of the home design software successfully allows you to visualize your plan on an actual colour photo. These software programmes can help you create a brand new home or home improvement remodeling to your existing home to your wishes. After you have planned your dream house most home design software will offer you a 3-D walk through you house. You can now view how your design looks before implementing.

And remember changes are easy to make however you feel necessary. If you are happy with what you see it is also possible to print off colour images. Some home design software is so advanced it goes as far as calculating beams and footings etc. As in the case with purchasing all software ensures you have a support/help centre that can assist with any glitches. I think an appealing feature of home design software is that it is fun to utilize, allowing for a gratifying designing adventure.

Retail Software

Retail trading is the sale of goods in relatively small quantities which can be fast-selling foodstuffs, toiletries, clothing or consumer durables. Most retailing is done via shops or stores and includes the vast numbers of retail chains. Current trends in retailing is the optimization of profits and fewer losses, which means it is of the utmost importance to have the correct and necessary retail software.

Therefore depending on the size of the retail store, different retail software packages are designed to suit different sized stores.

Regardless of which retail software you purchase there are a number of common features to look out for:

- *Point of sale products*: Ensure they are easy to learn and use, up to date with up to date retail tools, including bar codes and credit card readers, adaptable and flexible.
- *Customer database*: Accurate in storing your customer's details such as names and a history of previous buying.
- *Inventory control and Kiosk*: This is to avoid creating disappointment in your customer by being under stocked at any stage, or overstocked which can be damaging to the company. Some retail software programmes can print sales, inventory, low-stock or over-stock reports to keep you up to date. Some retail software even comes with features that will remind you when you need to re-order certain stock.
- *Convenience features*: With keeping to your goal of increased sales, convenience features are an added bonus for *e.g.* totally touch screen suited.
- *Security features*: Needs to be incorporated at each and every level, using passwords to avoid unauthorized staff making certain transactions, including the date, time and name of sales person working at specific check out location.
- *Back office functions*: Linked to the back office making their functions easier and more accurate by transporting all necessary information.

If you are able to implement the right retail software you should constantly be able to view a thorough picture of your business and ensure you are continuously on top of things. Thus by keeping you up to date of popular products in your store, and maybe those that are not selling so great with your customers, allowing you to make profitable changes. Stay neck and tie with your competitors and up with the times, lose cash registers and join the computer revolution with retail software, it has been determined that it does save money.

Elements of software design

The design process can be described as the process of choosing a representation of a solution from a set of alternatives, given the constraints towards a set of goals. It can be divided in two phases: diversification and convergence. The diversification is the phase of generating alternatives. Not necessarily documents describing a possible solution, but, at least, on the designer's mind.

These alternatives are the solution candidates and are generated/obtained from knowledge, catalogs, or previous experience. During the convergence phase, the designer chooses the alternative (or the combination of alternatives) that meets the intended goals.

The alternative selected will be the solution, which will meet the constraints imposed by the problem domain. The solution then may be described using some representation. The representation chosen must fit its purpose: describe the solution and the process to build the artifact that reaches the intended goals.

Goals

Design begins with a need. If something is to be designed, though to be built, it is because the outcome of the design process will fulfill this need. In Software Engineering, the necessity starts from a customer who specifies what are her needs and therefore what are the goals to be achieved with the software system to be designed.

So, the goal of the design process is to achieve a solution that will solve the customer's needs. In software design, goals are also referred to as requirements. Software design is mainly concerned on two types of requirements: functional and nonfunctional requirements.

A functional requirement specifies the functionality that a system will exhibit. In other words, *what* the system is to perform according to the customer. For example, a functional requirement for a sorting programme is to provide the ability to sort integers.

Another example could be related to a software system that manages the inventory of a movie rental store. If we were to enumerate the functional requirements of a system like this, among them would be: the ability to search for a movie by its keywords, the ability to perform a movie rental, the ability to perform a movie return, and many others.

On the other hand, a nonfunctional requirement specifies properties or characteristics that a software system must exhibit other than the observable behaviour [10].

Specifically, it is concerned on *how* the software will function. Back to our sorting programme example, a nonfunctional requirement is the customer's concern with the running time of the sorting function (*e.g.*, it is acceptable that the sorting algorithm execution time has the growth rate of $O(n\log n)$, where n is the size of the input). If we are talking about the movie rental store software, one nonfunctional requirement can be described as exposing some system's functions, such as the search for movie by its keywords, to be accessible via browser to its users.

As nonfunctional requirements play an important role on software architecture, we may return to them in the chapter on Nonfunctional Requirements, where they will exemplified, described, and categorized in detail, as well they will be correlated to their imposers, the system's stakeholders.

Constraints

A design product must be feasible. Considering this, a design constraint is the rule, requirement, relation, convention, or principle that define the context of design, in order to achieve a feasible design product. Smith and Browne gave a detailed description of the role of constraints on design

It is important to know that constraints are related to goals, and sometimes they can even be exchanged. However, in order to differentiate them, it is also important to understand that are not only the goals that rule what is to be designed. In other words, a software system may have clear goals, but its design, or some possibilities of it, may not be feasible because of its constraints.

In order to grasp the role of constraints in design, let us consider two examples. In the first example, despite the software system has a clear goal, its design is not feasible due to some of its constraints. In the second, a software system also has a clear goal, but just a clear design possibility is constrained.

First, consider that a customer describes a simple goal to be achieved by a system: it must be able to decide whether its input, a description of another programme, finishes running or not. An inexperienced software designer might even try to find a design possibility for this requirement – but this would be in vain. There is a theoretical constraint in computer science, widely known as the halting problem, which forbids a programme to decide whether or not another programme halts after its execution, thus not allowing achieving a solution. So, a design was not allowed due to its constra-ints even with a clear stated goal.

Alternatives

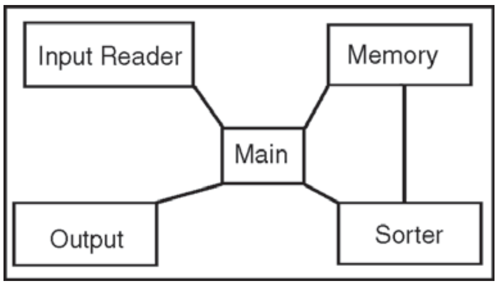
A design alternative is a possibility of solution. Since design problems often have multiple solutions, since design problems often have multiple solutions, the designer is expected to generate multiple design alternatives for a single problem. We must understand that the designer, after understanding the problem's goals and constraints, has two concerns: the alternative generation and the solution election among the alternatives. Alternative generation poses the real challenge for a designer. Unlike decision problems area where decision alternatives are known or can be discovered through search methods, design alternatives must be created. This creation process then must be controlled by design enabling techniques, and designer's experiential knowledge and creative imagination. The solution election is simply the choice of one of the alternatives that, according to the designer, will best solve the problem. This choice must be made employing reasoned analysis and experience. The following subsections better explain solutions and their representations.

Representations

Representations are the language of design. Although the true product of design is a representation for artifact construct, representing the solution is not the only purpose of representation. It also supports the design process. This support happens by allowing communication between stakeholders and by serving as a record of commitments.

A design representation allows communication because it turns alternatives into manipulable products, so they can be described, analysed, and discussed not only by their author but also by others. Please observe that there are multiple dimensions to be represented on a single software design alternative. These dimensions may comprise runtime behaviour, structure, and relation between logical entities to physical entities just to name a few. These dimensions are often exhibited on different types of representations, which later we will name them views.

In order to illustrate design representations, let us present two dimensions of our sorting programme example by using two different representations. The first representation, Figure 1, shows the structure from a design possibility of our example using Unified Modeling Language. Observing this representation, we see how the solution was decomposed, how each class of the structure relates to each other, or even see what points could be replaced by ready-made components, which must implement the same interfaces specified on the representation. One thing that must be noticed is that this representation is not self-contained, since knowledge on UML is needed by the reader to fully understand this representation.



The second representation, Figure, shows the example's runtime behaviour when sorting with high level of detail. Although we cannot see from this representation the same information presented on the previous one, this enables us to analyse how the programme will asymptotically behave according the growth rate of its input. Also, we can have the notion of how much space is needed to perform the algorithm.

Merge Sort Pseudocode:

```
function mergesort (m)
var list left,right,result
  if length (m) ≤ 1
                    return m

var middle = length(m) /
for each × in m up to middle
                    add × to left
for each × in m aftermiddle
                    add × to right
left = mergesort (left)
right = mergesort (right)
result = merge(left right)
return result
function merge left,right
var list result
while length(left) and length(right)
if first(left) ≤first(right)
append first(left) to result
left = first (left)
else
append first(right) to result
right = rest(right)
end file
```

```
  if length(left) > 0
  append rest(left) to result
  if length (right) > 0
  append rest (right) to result
return result
```

Both representations show important aspects of a system's design, nevertheless one involved on its development might be interested on aspects other than its structure or algorithm asymptotic analysis. As a matter of fact, other representations might be wanted for other purposes and it is the role of the design process to provide them.

Solutions

A design solution is nothing more than a description enabling system construction that uses one or many representations in order to expose sufficient details. Its characteristics are listed in the following paragraphs.

Design solutions mirror the problem complexity, usually having many attributes and interrelationships. We can observe this characteristic, for example, when designing a solution for managing the inventory of a DVD rental store. Whatever the solution is, it must contain attributes such as movies, DVDs, clients, and genres, which will represent the elements inherent to the problem.

However, this is not enough. It must also contain relations such as "a client can rent one or more DVDs", "a movie have one or more genres", or "a DVD must contain one or more movies", which will behave just like the relations on the problem domain. Consequently, when many different attributes have different interrelationships between themselves, complexity emerges.

It is difficult to validate design solutions. The complexity of the solution renders many points of possible validation against design goals.

The very problem resides on how well the design goals are described. Usually, only high-level goals are specified for very complex problems, so validation is hardened.

Modular Software Design

In order to produce programs that are readable, reliable, and can be easily maintained or modified, one must use modular software design. This means that, instead of having a large collection of statements strung together in one partition of in-line code, we segment or divide the statements into logical groups called modules.

Each module performs one or two tasks, then passes control to another module. By breaking up the code into "bite-sized chunks", so to speak, we are able to better control the flow of data and control. This is especially true in large software systems.

Overview of Modular Software Design

We begin with several definitions (*Hint:* These may be useful to learn for a future exam) in support of a brief discussion of software design goals. We then progress to examples of code segmentation.

Observation

In the early days of computer programming, when people coded programs in *machine code* (ones and zeroes), it was quite difficult to determine programme function and structure from looking at the code. Humans tend to look at problems solved on a computer in a linguistic sort of way, i.e., expect some flow of control or data to be expressed in the programming language. Ones and zeros don't tell us much, and they certainly give little indication of programme structure or data/control flow.

Definition

Spaghetti code is the term used for a computer programme that is not well structured and tends to have highly tangled flows of data and control.

Example

Most *assembly language code* and *machine code* are good examples of spaghetti code.

The following sample of machine code is illustrative:

```
110101010010001000111001001
010101001000100001011101001
000111001101110001101101010
001111010010010101011001010
001010101111110100101010001
```

Clearly, there is very little discernable structure in this type of code.

Definition In programming languages, the *semantic gap* is the difference between the language you use to programme the hardware (machine code) and the language you would like to use to programme the computer as a system. We call the latter, more abstract language a *high-level language* or HLL.

Observation

Throughout the history of computing, there have been at least hundreds of attempts to make computer programming languages something like English — easy to read and implicitly easy to understand. PASCAL is the result of one such effort. The co-creator of PASCAL, Nicholas Wirth, wanted to have an HLL that was easy to learn, read, and write.

So, he designed PASCAL around the following concepts:

- PASCAL should close or significantly narrow the semantic gap.
- Every PASCAL statement should be like a *clause* in an English-language sentence.
- The PASCAL programme can be thought of as a *sentence* in English (namely, a concatenation of clauses).
- Names of procedures, data structures, and variables in PASCAL should be easily recognizable.

Remark. PASCAL facilitates modular coding via:

- Encapsulating code in PROCEDUREs and FUNCTIONs that constitute a PROGRAMME;
- The use of BEGIN and END statements to define a functional block of code;
- Strict *variable typing* (i.e., assigning datatypes such as *integer*, *real*, or *string* to variables) in support of parameter passing between procedures; and
- User-friendly syntax that narrows (but does not close) the semantic gap.

In the 1960s and 1970s, software designers were faced with large accumulations of spaghetti code from preceding years. Programs were becoming more complex, and it was more difficult to keep software running correctly. After trying various strategies for organizing this morass of code, the following guidelines for software development emerged:

- *Clarity*: Code must be easily understandable by humans, and variable/function names should have obvious meaning.
- *Modularity*: Programs must be divided into small modules.
- *Concision*: Modules must perform a few tasks only, using compact (but not cryptic) notation.
- *Reliability*: Programs must run correctly, in a repeatable manner.
- *Ease of Maintenance*: Software must be easy to maintain and modify, and must be accompanied by comprehensive documentation.

Clearly written software is often an elusive goal, because technical programmers tend to prefer cryptic variable names (e.g., PR2CD$ instead of clear notation such as PRICE). Furthermore, there are many programmers who do not have good writing skills, and definitely don't enjoy writing documentation. Thus, to be a good programmer, must concentrate on improving the quality of your software not only through careful design and programming, but also through careful documentation. Modular code is easy to produce from a design, but often hard to produce from spaghetti code. We discuss this process below, where we show general examples of code modularization. Modern software development tools facilitate the generation of modular code, and often check syntax of programming statements, with some variable type checking possible. Thus, there exists a variety of evolving techniques for software design in modular form.

Concisely written code is important to ensuring proper programme function. For example, if your code is so tangled that you can't determine what it does, how easy will it be for others to understand your work? It is also important not to create excessively complicated procedures, which are difficult to debug and maintain, and thus tend to be unreliable.

Software reliability follows from rigorous software design, checking one's work, and carefully debugging and testing the software you write in an incremental fashion. By *incremental development*, we mean the construction of a software system and testing of that software on a piece-by-piece basis. For example, after you write the lowest-level routines, you should test them all thoroughly before you write the functions or procedures that call those routines.

Ease of maintenance follows directly from clarity and concision. For example, if code can be clearly understood, then you or others would have no trouble understanding and modifying its functionality.

Additionally, concise code is easier to maintain because:

- There is less code to examine and modify,
- There is less probability of making mistakes in modifying the code,
- Debugging is easier due to limited scope of functionality.
- Schematic illustration of control transfers (arcs) in spaghetti code;
- Procedure segmentation according to locaity of control transfer. A main programme that calls three procedures is produced.

A programme produced by this method could have a pseudocode representation similar to the following example:

```
PROCEDURE P1(< args >):
  < procedure definition >
```

```
END-PROC
PROCEDURE P2(< args >):
  < procedure definition >
END-PROC
PROCEDURE P3(< args >):
  < procedure definition >
END-PROC
PROGRAMME Main:
  < declarations and/or executable code >
  P1(< args >) #execute P1
  < executable code >
  P2(< args >) #execute P2
  < executable code >
  P3(< args >) #execute P3
  < executable code >
END-PROG
```

which portrays the modularity shown in Figure b.

There are other methods that can facilitate conversion of spaghetti code to modularized code, which include:

- Determining the *calling hierarchy*, which is the structure defined by the calling sequence of procedures. For example, in the preceding p-code, we would have the calling hierarchy MAIN > (P1 P2 P3). It is not necessarily easy to determine the calling hierarchy from spaghetti code, but this information can sometimes be available from software called an *execution profiler*. Such programs keep a record of control flow and can be used to determine branching and jump behaviour (transfers of control).
- Segmenting code according to functionality, i.e., finding what function each partition of code performs, then encapsulating the code into procedures, each of which perform one or two tasks.
- As a last resort, spaghetti code can simply be chopped into pieces of more or less equal length. For example, if the p-code contains assignment statements only, the variables on

the left side of assignment statements would specified as the *output variables* of a procedure that contains those statements. The variables on the right side of the assignment statements would be specified as *input variables*.

PASCAL Procedural Organization

PASCAL supports *hierarchical programme structure*, in which there is a high-level procedure, often called the *main programme* or *root procedure*. Other procedures are subordinate to the root procedure, and may call each other, but usually do not call the root procedure. Each procedure is comprised of*statements*, which are lines of code that perform a given function.

The PASCAL language provides three methods for encapsulating code in procedures. First, the FUNCTION statement specifies a function that accepts values from its argument list and returns a value or result through the function name. Second, the PROCEDURE statement specifies a procedure that accepts values from its argument list and returns one or more values through its argument list.

Third, the PROGRAMME statement allows the programmer to specify high-level source code that calls predefined procedures to implement a structured software system. We define these statements as follows:

PROGRAMME Specification Statement

Purpose: The Programme statement specifies the name of a main programme (i.e., the top-level procedure).

Syntax:

```
  PROGRAMME programme-name ( input-file , output-
file ); where
  programme-name denotes the name of the programme
```

input-file denotes the name of the file from which the programme reads input

output-file denotes the filename to which the programme writes output.

Example:

```
PROGRAMME Prog1 (myfile.dat, myfile.rpt);
PROGRAMME Prog1;
```

Notes: The input and output file names and their associated parentheses are optional, and may or may not work with various operating systems (e.g., DOS, UNIX, etc.)

Pascal Programs Have Three Parts

1. Programme and variable specification section;
2. Subordinate procedure declarations; and
3. Main programme executable code.

Pascal procedures and functions are also organized in this way. In this class, it is strongly recommended that you define all subordinate procedures at the same level in the main programme. Do not encapsulate procedures within other procedures, but declare them only in the main programme. In other words:

```
DO THIS
MAIN-PROGRAMME
  Proc #1 specification
     &ltproc-1 code>
  Proc #2 specification
     &ltproc-2 code>
  Proc #3 specification
     &ltproc-3 code>
  &ltmain-programme code>
END.
```

```
NOT THIS
MAIN-PROGRAMME
  Proc #1 specification |
    Proc #2 specification |
    &ltproc-2 code> |
    &ltproc-1 code> |
  Proc #3 specification
    &ltproc-3 code>
  &ltmain-programme code>
END.
```

The preceding pseudocode becomes difficult to interpret visually (and, therefore, difficult to maintain) when Procedure #2 is defined within Procedure #1. Although this is valid from the perspective of PASCAL *syntax* it is *not good programming style*, because it decreases *readability* and, therefore, increases *code maintenance cost*.

Function Specification Statement

Purpose: The Function statement specifies the name of a procedure that inputs values through its argument list and can be thought of as returning a result through its name.

Syntax:

```
  FUNCTION function-name ( argument- 1 ...,    argument-
N ); where
  function-name denotes the name of the function
  argument-i denotes the name of the i-th argument of the
function.
```

Example:

```
  FUNCTION sine(x);
  FUNCTION Distance(x,y);
```

Notes: Do not try to pass output values through the argument list of a function. This can cause problems in some PASCAL implementations.

Procedure Specification Statement

Purpose: The Procedure statement specifies the name of a procedure that can input and output values through its argument list.

Syntax:

```
  PROCEDURE proc-name ( argument- 1 ..., argument-N ); where
```

```
    proc-name denotes the name of the procedure
    argument-i denotes the name of the i-th
argument of the procedure.
```

Example:

```
PROCEDURE sine(x,output);
PROCEDURE Distance(x,y,output);
PROCEDURE Smile;
```

BEGIN...END Block Specification Statement

Purpose: The BEGIN...END statement delimits a block of *compound statements.*

```
Syntax: BEGIN &ltstatements> END where
     statements denotes more than one Pascal
statement.
```

Example:

```
BEGIN
  WRITELN('Hello, world');
  WRITELN('Second statement');
  WRITELN('Last statement');
END;
```

Notes: In the preceding example, each statement ends with a semicolon (;). Since the PASCAL design philosophy views each statement as a*clause*, the semicolon punctuation convention (adopted from English) is employed.

General Comments

Indentation is used to highlight and clarify programme structure. For example, each new level of statements should be indented two or three spaces to the right. When a block of statements is closed (e.g., with an END statement), then the indent shifts two or three spaces to the left. Each statement begins on a new line, except for multiple short assignment statements that initialize values in a programme.

In the following section, we consider several examples of PASCAL procedural code.

Writing Modular Code in PASCAL

It is not difficult to use PASCAL module specifications to write useful programs. Here follows a simple example of the FUNCTION construct:

```
  PROGRAMME TestFun;    {Programme specification}
    VAR x: integer;     {Declare variable x as
               integer}
    FUNCTION Xcubed(x); {Function specification}
  BEGIN           {Function begins here}
Xcubed:= x * x * x;  {Function
            definition}
      END;                  {Function ends here}
    BEGIN                   {Programme begins here}
      x:= 4;                {Assign value to x}
      WRITELN('x3=', Xcubed(x)); {Print value of
                         x^3}
    END.                    {Programme ends here}
```

In the preceding PASCAL code, note that the VAR statement specifies a variable of a given datatype. In this case, the variable *x* is specified as an*integer*.

Additionally, the WRITELN statement outputs the legend *x3*= to the screen, followed by the value returned by the function call *Xcubed(x)*. If we preferred not to put the function call in WRITELN's argument list, we could rewrite the preceding code as:

```
  PROGRAMME TestFun;    {Programme specification}
    VAR x,y: integer;   {Declare variables x,y
               as integer}
    FUNCTION Xcubed(x); {Function specification}
      BEGIN             {Function begins here}
         Xcubed:= x * x * x; {Function
                    definition}
      END;              {Function ends here}
    BEGIN         {Programme begins here}
      x:= 4    {Assign value to x}
      y:= Xcubed(x); {Assign function output
               to y}
      WRITELN('x3=', y);   {Print value of y}
    END. {Programme ends here}
```

Let us replicate the functionality of the preceding code by using the PROCEDURE construct and passing the output through a procedural argument instead of a FUNCTION name, as follows:

```
PROGRAMME TestFun;    {Programme specification}
  VAR x,y: integer;   {Declare variables x,y
             as integer}
PROCEDURE Xcubed(x,y); {Function specification}
  BEGIN               {Function begins here}
    y:= x * x * x; {Function definition –
             y gets x^3}
  END;                {Function ends here}
BEGIN                 {Programme begins here}
  x:= 4               {Assign value to x}
  Xcubed(x,y);        {Procedure call}
  WRITELN('x3=', y); {Print value of y =
             x^3}
END.                  {Programme ends here}
```

In the preceding programs, the variables x and y have *global scope*. That is, their definition as integers held throughout the main programme and called procedures (the function was also a called procedure).

In the following section, we shall see that there is a way to define x and y that makes procedures and functions *reusable*. This also facilitates efficiency and reliability in software development.

PASCAL Variables and Datatypes

Programming languages use abstractions called *variables* to store values. Because there are many different types of values (e.g., integer, real, string, etc.), there exists a method called *datatyping* by which one such type can be assigned to each variable.

PASCAL supports *strict typing*, that is, the datatype is assigned to the variable at compile time and does not change thereafter.

In PASCAL, valid datatypes that we will consider in this class are:

- *Integer*: a whole number, such as 1, 2, etc.;
- *Real*: a decimal number, such as -22.7, 231.8942, etc.;
- *Char*: a single character, such as 'H' or 'i';
- *String* : A list of characters, such as 'Hello'; and
- *Array* : A list of variables, such as (1.1, 2.4..., 3.7) or a two-dimensional array. Higher-dimensional structures are possible.

Most (but not all) compiled languages adopt the strict typing convention, to simplify compiler design and maintenance. However, there are some interpreted languages (e.g., SNOBOL) that allow flexible datatyping. This can produce great difficulty when debugging a programme in which a given variable's value is type-dependent.

In PASCAL, a variable name is any string of valid PASCAL characters. We recommend that you use the characters {A-Z,a-z,0-9,_} for your variable names. The following example is illustrative:

```
VALID NAMES         INVALID NAMES
Cost, Price         $amount, @price
score               score+exam-grade
```

In each case of invalid names, reserved symbols or characters that have multiple meanings are used in the name string.

This is bad practice that can lead to compiler errors (i.e., your programme won't compile), or can lead to confusion when debugging or modifying programs that contain such names.In PASCAL, variables are typed using the VAR statement, which is described as follows:

VAR Specification Statement

Purpose: The VARiable statement specifies the name and datatype of procedure or programme variables.

```
  Syntax: VAR varname-1...,varname N : datatype )
; where
      varname-i denotes the name of the i-th
variable in the list
      datatype denotes a valid PASCAL datatype
```

Example:

```
        VAR x,y,z: integer;
        VAR sum,prod: real;
        VAR name,ssn: string;
```

Notes: It is good programming style to specify only one datatype in each VAR statement. It is also good style not to continue VAR statements on multiple lines. This makes the programme easier to read. We next consider the issue of *scope of variables*. This issue is discussed in detail in Chapter 6 of Koffman, the textbook for t12his class, from which we condense the following discussion. In above, we illustrate the following procedure nesting hierarchy:

```
           (Nest > (Outer > Inner, Too))
```

The statements in each procedure operate only on *local variables*. This is good programming practice, and facilitates modularity. If we were to use*global variables*, which are declared once at the beginning of the main programme and then hold through all procedures, this would be bad software engineering practice, because:

- Global variables lead to confusion in debugging, when trying to trace variable types through many pages of code.
- Global variables are convenient to programmers, but they do not make procedures re-usable, since there is no variable declaration at the top of the procedure. In the absence of proper documentation, one cannot know for sure what

datatype is assigned to a given variable. This adversely impacts the clarity, reliability, and maintainability of software.

In contrast, local variables are easy to trace, since they are defined in (and manipulated by) one module only. Since good software engineering practice dictates that modules be kept small, it is much easier to trace the flow of data and control in these small modules. And, the modules can be re-used, because all variable definitions or declarations are local to the module. Pascal has two rules for determining the scope of variables (area of influence of a given variable), which are:

- *Rule* 1: The *scope* of a variable is the block in which the variable is declared.

Example: A variable declared as type T in some procedure P is available within P and all its subordinate procedures as a variable of type T.

- *Rule* 2: The redeclaration of a variable v as having type T within a procedure P holds for P and all its subordinate procedures, but not for higher-level procedures.

Example: Suppose we have the procedural definition hierarchy Main > (P1 > (P1a,P1b), P2)). That is, P1 and P2 are defined within Main and P1a and P1b are defined within P1. If a variable v is declared within Main as a *string* but within P1 as *real*, then v has the type *real* in P1, P1a, and P1b. However, v retains the type *string* in Main and P2. Good software engineering practice dictates that all variables be specified locally in PASCAL, except for Main Programme variables, which are global by default. As noted above, this facilitates modularity and portability of PASCAL code, and makes debugging much easier.

Software Development in Life Cycle Models

The Systems Development Life Cycle (SDLC) is a conceptual model used in project management that describes the stages involved in an information system development project from an initial feasibility study through maintenance of the completed application. Various SDLC methodologies have been developed to guide the processes involved including the waterfall model (the original SDLC method), rapid application development (RAD), joint application development (JAD), the fountain model and the spiral model. Mostly, several models are combined into some sort of hybrid methodology. Documentation is crucial regardless of the type of model chosen or devised for any application, and is usually done in parallel with the development process.

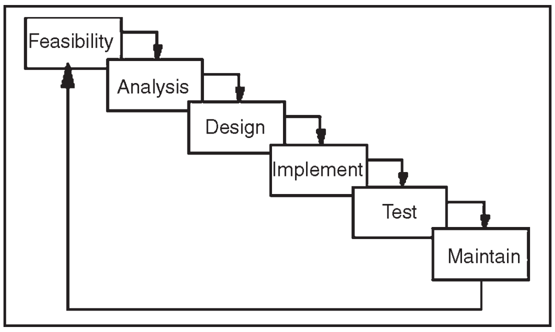


Fig. Briefly on different Phases.

Some methods work better for specific types of projects, but in the final analysis, the most important factor for the success of a project may be how closely particular plan was followed. The image above is the classic Waterfall model methodology, which is the first SDLC method and it describes the various phases involved in development.

Feasibility

The feasibility study is used to determine if the project should get the go-ahead. If the project is to proceed, the feasibility study will produce a project plan and budget estimates for the future stages of development.

Requirement Analysis and Design

Analysis gathers the requirements for the system. This stage includes a detailed study of the business needs of the organization. Options for changing the business process may be considered. Design focuses on high level design like, what programs are needed and how are they going to interact, low-level design (how the individual programs are going to work), interface design (what are the interfaces going to look like) and data design (what data will be required). During these phases, the software's overall structure is defined. Analysis and Design are very crucial in the whole development cycle. Any glitch in the design phase could be very expensive to solve in the later stage of the software development. Much care is taken during this phase. The logical system of the product is developed in this phase.

Implementation

In this phase the designs are translated into code. Computer programs are written using a conventional programming language or an application generator. Programming tools like Compilers, Interpreters, Debuggers are used to generate the code. Different high level programming languages like C, C++, Pascal, Java are used for coding. With respect to the type of application, the right programming language is chosen.

Testing

In this phase the system is tested. Normally programs are written as a series of individual modules, these subject to separate and detailed test. The system is then tested as a whole. The separate modules are brought together and tested as a complete system. The system is tested to ensure that interfaces between modules work (integration testing), the system works on the intended platform and with the expected volume of data (volume testing) and that the system does what the user requires (acceptance/beta testing).

Maintenance

Inevitably the system will need maintenance. Software will definitely undergo change once it is delivered to the customer. There are many reasons for the change. Change could happen because of some unexpected input values into the system. In addition, the changes in the system could directly affect the software operations. The software should be developed to accommodate changes that could happen during the post implementation period.

Description

Curtain Raiser

Like any other set of engineering products, software products are also oriented towards the customer. It is either market driven or it drives the market. Customer Satisfaction was the buzzword of the 80's. Customer Delight is today's buzzword and Customer Ecstasy is the buzzword of the new millennium.

Products that are not customer or user friendly have no place in the market although they are engineered using the best technology. The interface of the product is as crucial as the internal technology of the product.

Market Research

A market study is made to identify a potential customer's need. This process is also known as market research. Here, the already existing need and the possible and potential needs that are available in a segment of the society are studied carefully. The market study is done based on a lot of assumptions.

Assumptions are the crucial factors in the development or inception of a product's development. Unrealistic assumptions can cause a nosedive in the entire venture.

Though assumptions are abstract, there should be a move to develop tangible assumptions to come up with a successful product.

Research and Development

Once the Market Research is carried out, the customer's need is given to the Research & Development division (R&D) to conceptualize a cost-effective system that could potentially solve the customer's needs in a manner that is better than the one adopted by the competitors at present. Once the conceptual system is developed and tested in a hypothetical environment, the development team takes control of it. The development team adopts one of the software development methodologies that is given below, develops the proposed system, and gives it to the customer.

The Sales & Marketing division starts selling the software to the available customers and simultaneously works to develop a niche segment that could potentially buy the software. In addition, the division also passes the feedback from the customers to the developers and the R&D division to make possible value additions to the product.

While developing a software, the company outsources the non-core activities to other companies who specialize in those activities. This accelerates the software development process largely. Some companies work on tie-ups to bring out a highly matured product in a short period.

Software Development Models

The following are some basic popular models that are adopted by many software development firms

- System Development Life Cycle (SDLC) Model
- Prototyping Model
- Rapid Application Development Model
- Component Assembly Model

System Development Life Cycle Model

A software life cycle model depicts the significant phases or activities of a software project from conception until the product is retired. It specifies the relationships between project phases, including transition criteria, feedback mechanisms, milestones, baselines, reviews, and deliverables. Typically, a life cycle model addresses the phases of a software project: requirements phase, design phase, implementation, integration, testing, operations and maintenance. Much of the motivation behind utilizing a life cycle model is to provide structure to avoid the problems of the "undisciplined hacker" or corporate IT bureaucrat (which is probably ten times dangerous then undisciplined hacker). As always, it's a matter of picking the right tool for the job, rather than picking up your hammer and treating everything as a nail.

System/Information Engineering and Modeling

As software is always of a large system (or business), work begins by establishing the requirements for all system elements and then allocating some subset of these requirements to software. This system view is essential when the software must interface with other elements such as hardware, people and other resources. System is the basic and very critical requirement for the existence of software in any entity. So if the system is not in place, the system should be engineered and put in place. In some cases, to extract the maximum output, the system should be re-engineered and spruced up. Once the ideal system is engineered or tuned, the development team studies the software requirement for the system.

Software Requirement Analysis

This process is also known as feasibility study. In this phase, the development team visits the customer and studies their system. They investigate the need for possible software automation in the given system.

By the end of the feasibility study, the team furnishes a document that holds the different specific recommendations for the candidate system. It also includes the personnel assignments, costs, project schedule, target dates etc.... The requirement gathering process is intensified and focussed specially on software. To understand the nature of the programme(s) to be built, the system engineer or "Analyst" must understand the information domain for the software, as well as required function, behaviour, performance and interfacing. The essential purpose of this phase is to find the need and to define the problem that needs to be solved.

System Analysis and Design

In this phase, the software development process, the software's overall structure and its nuances are defined. In terms of the client/server technology, the number of tiers needed for the package architecture, the database design, the data structure design etc... are all defined in this phase.

A software development model is thus created. Analysis and Design are very crucial in the whole development cycle. Any glitch in the design phase could be very expensive to solve in the later stage of the software development. Much care is taken during this phase. The logical system of the product is developed in this phase.

Code Generation

The design must be translated into a machine-readable form. The code generation step performs this task. If the design is performed in a detailed manner, code generation can be accomplished without much complication. Programming tools like compilers, interpreters, debuggers etc... are used to generate the code.

Different high level programming languages like C, C++, Pascal, Java are used for coding. With respect to the type of application, the right programming language is chosen.

Testing

Once the code is generated, the software programme testing begins. Different testing methodologies are available to unravel the bugs that were committed during the previous phases. Different testing tools and methodologies are already available. Some companies build their own testing tools that are tailor made for their own development operations.

Maintenance

The software will definitely undergo change once it is delivered to the customer. There can be many reasons for this change to occur. Change could happen because of some unexpected input values into the system. In addition, the changes in the system could directly affect the software operations. The software should be developed to accommodate changes that could happen during the post implementation period.

Prototyping Model

This is a cyclic version of the linear model. In this model, once the requirement analysis is done and the design for a prototype is made, the development process gets started.

Once the prototype is created, it is given to the customer for evaluation. The customer tests the package and gives his/her feed back to the developer who refines the product according to the customer's exact expectation. After a finite number of iterations, the final software package is given to the customer.

In this methodology, the software is evolved as a result of periodic shuttling of information between the customer and developer. This is the most popular development model in the contemporary IT industry. Most of the successful software products have been developed using this model - as it is very difficult (even for a whiz kid!) to comprehend all the requirements of a customer in one shot.

There are many variations of this model skewed with respect to the project management styles of the companies. New versions of a software product evolve as a result of prototyping.

The goal of prototyping based development is to counter the first two limitations of the waterfall model discussed earlier.

The basic idea here is that instead of freezing the requirements before a design or coding can proceed, a throwaway prototype is built to understand the requirements. This prototype is developed based on the currently known requirements. Development of the prototype obviously undergoes design, coding and testing.

But each of these phases is not done very formally or thoroughly. By using this prototype, the client can get an "actual feel" of the system, since the interactions with prototype can enable the client to better understand the requirements of the desired system.

Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help determining the requirements.

In such situations letting the client "plan" with the prototype provides invaluable and intangible inputs which helps in determining the requirements for the system. It is also an effective method to demonstrate the feasibility of a certain approach.

This might be needed for novel systems where it is not clear those constraints can be met or that algorithms can be developed to implement the requirements. The process model of the prototyping approach is shown in the figure below.

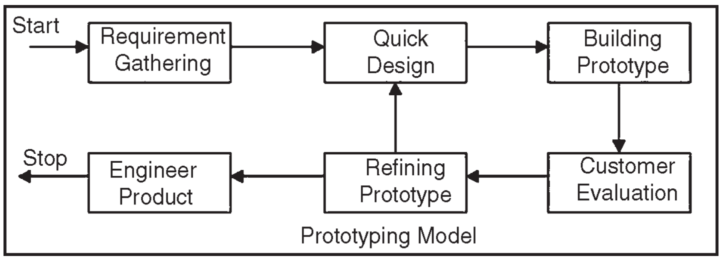


Fig. Prototyping Model.

The basic reason for little common use of prototyping is the cost involved in this built-it-twice approach. However, some argue that prototyping need not be very costly and can actually reduce the overall development cost. The prototype are usually not complete systems and many of the details are not built in the prototype. The goal is to provide a system with overall functionality.

In addition, the cost of testing and writing detailed documents are reduced. These factors helps to reduce the cost of developing the prototype. On the other hand, the experience of developing the prototype will very useful for developers when developing the final system. This experience helps to reduce the cost of development of the final system and results in a more reliable and better designed system.

Advantages of Prototyping

Creating software using the prototype model also has its benefits. One of the key advantages a prototype modeled software has is the time frame of development. Instead of concentrating on documentation, more effort is placed in creating the actual software. This way, the actual software could be released in advance. The work on prototype models could also be spread to others since there are practically no stages of work in thismodel. Everyone has to work on the same thing and at the same time, reducing man hours in creating a software. The work will even be faster and efficient if developers will collaborate more regarding the status of a specific function and develop the necessary adjustments in time for the integration.

Another advantage of having a prototype modeled software is that the software is created using lots of user feedbacks. In every prototype created, users could give their honest opinion about the software. If something is unfavorable, it can be changed. Slowly the programme is created with the customer in mind.

- Users are actively involved in the development
- It provides a better system to users, as users have natural tendency to change their mind in specifying requirements and this method of developing systems supports this user tendency.
- Since in this methodology a working model of the system is provided, the users get a better understanding of the system being developed.
- Errors can be detected much earlier as the system is mode side by side.
- Quicker user feedback is available leading to better solutions.

Disadvantages

Implementing the prototype model for creating software has disadvantages. Since its being built out of concept, most of the models presented in the early stage are not complete. Usually they lack flaws that developers still need to work on them again and again. Since the prototype changes from time to time, it's a nightmare to create a document for this software. There are many things that are removed, changed and added in a single update of the prototype and documenting each of them has been proven difficult.

There is also a great temptation for most developers to create a prototype and stick to it even though it has flaws. Since prototypes are not yet complete software programs, there is always a possibility of a designer flaw. When flawed software is implemented, it could mean losses of important resources.

Lastly, integration could be very difficult for a prototype model. This often happens when other programs are already stable. The prototype software is released and integrated to the company's suite of software. But if there's something wrong the prototype, changes are required not only with the software. It's also possible that the stable software should be changed in order for them to be integrated properly.

Prototype Models Types

There are four types of Prototype Models based on their development planning: the Patch-Up Prototype, Nonoperational Prototype, First-of-a-Series Prototype and Selected Features Prototype.

Patch Up Prototype

This type of Prototype Model encourages cooperation of different developers. Each developer will work on a specific part of the programme. After everyone has done their part, the programme will be integrated with each other resulting in a whole new programme. Since everyone is working on a different field, Patch Up Prototype is a fast development model. If each developer is highly skilled, there is no need to overlap in a specific function of work.

This type of software development model only needs a strong project manager who can monitor the development of the programme. The manager will control the work flow and ensure there is no overlapping of functions among different developers.

Non-Operational Prototype

A non-operational prototype model is used when only a certain part of the programme should be updated. Although it's not a fully operational programme, the specific part of the programme will work or could be tested as planned. The main software or prototype is not affected at all as the dummy programme is applied with the application.

Each developer who is assigned with different stages will have to work with the dummy prototype. This prototype is usually implemented when certain problems in a specific part of the programme arises. Since the software could be in a prototype mode for a very long time, changing and maintenance of specific parts is very important. Slowly it has become a smart way of creating software by introducing small functions of the software.

First of a Series Prototype

Known as a beta version, this Prototype Model could be very efficient if properly launched. In all beta versions, the software is launched and even introduced to the public for testing. It's fully functional software but the aim of being in beta version is to as for feedbacks, suggestions or even practicing the firewall and security of the software.

It could be very successful if the First of a Series Prototype is properly done. But if the programme is half heartedly done, only aiming for additional concept, it will be susceptible to different hacks, ultimately backfiring and destroying the prototype.

Selected Features Prototype

This is another form of releasing software in beta version. However, instead of giving the public the full version of the software in beta, only selected features or limited access to some important tools in the programme is introduced.

Selected Features Prototype is applied to software that are part of a bigger suite of programs. Those released are independent of the suite but the full version should integrate with other software. This is usually done to test the independent feature of the software.

Rapid Application Development (RAD) Model

The RAD modelis a linear sequential software development process that emphasizes an extremely short development cycle.

The RAD model is a "high speed" adaptation of the linear sequential model in which rapid development is achieved by using a component-based construction approach. Used primarily for information systems applications, the RAD approach encompasses the following phases:

Business Modeling

The information flow among business functions is modeled in a way that answers the following questions:

- What information drives the business process?
- What information is generated?
- Who generates it?
- Where does the information go?
- Who processes it?

Data Modeling

The information flow defined as part of the business modeling phase is refined into a set of data objects that are needed to support the business. The characteristic (called attributes) of each object is identified and the relationships between these objects are defined.

Process Modeling

The data objects defined in the data-modeling phase are transformed to achieve the information flow necessary to implement a business function. Processing the descriptions are created for adding, modifying, deleting, or retrieving a data object.

Application Generation

The RAD model assumes the use of the RAD tools like VB, VC++, Delphi etc... rather than creating software using conventional third generation programming languages. The RAD model works to reuse existing programme components (when possible) or create reusable components (when necessary). In all cases, automated tools are used to facilitate construction of the software.

Testing and Turnover

Since the RAD process emphasizes reuse, many of the programme components have already been tested. This minimizes the testing and development time.

Component Assembly Model

Object technologies provide the technical framework for a component-based process model for software engineering. The object oriented paradigm emphasizes the creation of classes that encapsulate both data and the algorithm that are used to manipulate the data. If properly designed and implemented, object oriented classes are reusable across different applicationsand computer based system architectures. Component Assembly Model leads to software reusability. The integration/assembly of the already existing software components accelerate the development process. Nowadays many component libraries are available on the Internet. If the right components are chosen, the integration aspect is made much simpler.

All these different software development models have their own advantages and disadvantages. Nevertheless, in the contemporary commercial software evelopment world, the fusion of all these methodologies is incorporated. Timing is very crucial in software development.

If a delay happens in the development phase, the market could be taken over by the competitor. Also if a 'bug' filled product is launched in a short period of time (quicker than the competitors), it may affect the reputation of the company. So, there should be a tradeoff between the development time and the quality of the product. Customers don't expect a bug free product but they expect a user-friendly product.

Software life cycle models

Waterfall model

The least flexible of the life cycle models. Still it is well suited to projects which have a well defined architecture and established user interface and performance requirements.

The waterfall model *does* work for certain problem domains, notably those where the requirements are well understood in advance and unlikely to change significantly over the course of development.

Software products are oriented towards customers like any other engineering products. It is either driver by market or it drives the market. Customer Satisfaction was the main aim in the 1980's. Customer Delight is today's logo and Customer Ecstasy is the new buzzword of the new millennium. Products which are not customer oriented have no place in the market although they are designed using the best technology. The front end of the product is as crucial as the internal technology of the product.

A market study is necessary to identify a potential customer's need. This process is also called as market research. The already existing need and the possible future needs that are combined together for study.

A lot of assumptions are made during market study. Assumptions are the very important factors in the development or start of a product's development. The assumptions which are not realistic can cause a nosedive in the entire venture.

Although assumptions are conceptual, there should be a move to develop tangible assumptions to move towards a successful product. Once the Market study is done, the customer's need is given to the Research and Development Department to develop a cost-effective system that could potentially solve customer's needs better than the competitors.

Once the system is developed and tested in a hypothetical environment, the development team takes control of it. The development team adopts one of the software development models to develop the proposed system and gives it to the customers.

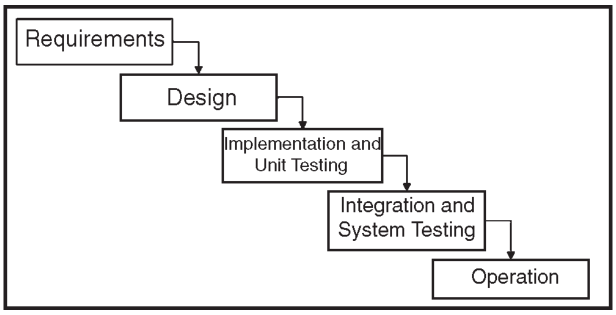

Requirements
Design
Implementation and Unit Testing
Integration and System Testing
Operation

Fig. Waterfall Life Cycle Model.

Advantages

- Simple and easy to use.
- Easy to manage due to the rigidity of the model – each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.

Disadvantages

- Adjusting scope during the life cycle can kill a project
- No working software is produced until late during the life cycle.

- High amounts of risk and uncertainty.
- Poor model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Poor model where requirements are at a moderate to high risk of changing.

Extreme programming (XP)

Is the latest incarnation of Waterfall model and is the most recent software fad. Most postulates of Extreme programming are pure fantasy. It has been well known for a long time that *big bang* or waterfall models don't work well on projects with complex or shifting requirements.

The same is true for XP. Too many shops implement XP as an excuse for not understanding the user requirements. XP try improve classic waterfall model by trying to start coding as early as possible but without creating a full-fledged prototype as the first stage. In this sense it can be considered to be variant of evolutionary prototyping (see below). Often catch phase "Emergent design" is used instead of evolutionary prototyping. It also introduces a very questionable idea of pair programming as an attempt to improve extremely poor communication between developers typical for large projects. While communication in large projects is really critical and attempts to improve it usually pay well, "pair programming" is a questionable strategy.

There are two main problems here:

1. In a way it can be classified as a hidden attempt to create one good programmer out of two mediocre. But in reality it is creating one mediocre programmer from two or one good. No senior developer is going to put up with some

jerk sitting on his lap asking questions about every line. It just prevents the level of concentration needed for high quality coding. Microsoft's idea of having a tester for each programmer is more realistic: one developer writes tests.

2. The actual code to be tested. This forces each of them to communicate and because tester has different priorities then developer such communication brings the developer a new and different perspective on his code, which really improves quality. This combination of different perspectives is a really neat idea as you can see from the stream of Microsoft Office products and operating systems.

Throwaway prototyping model

Typical implementation language is scripting language and Unix shell (due to availability huge amount of components that can be used for construction of the prototype).

Spiral model

The spiral model is a variant of "dialectical spiral" and as such provides useful insights into the life cycle of the system. Can be considered as a generalization of the prototyping model.

That why it is usually implemented as a variant of prototyping model with the first iteration being a prototype. The spiral model is similar to the incremental model, with more emphases placed on risk analysis.

The spiral model has four phases: Planning, Risk Analysis, Engineering and Evaluation. A software project repeatedly passes through these phases in iterations (called Spirals in this model). The baseline spiral, starting in the planning phase, requirements are gathered and risk is assessed.

Each subsequent spirals builds on the baseline spiral. Requirements are gathered during the planning phase. In the risk analysis phase, a process is undertaken to identify risk and alternate solutions.

A prototype is produced at the end of the risk analysis phase. Software is produced in the engineering phase, along with testing at the end of the phase. The evaluation phase allows the customer to evaluate the output of the project to date before the project continues to the next spiral. In the spiral model, the angular component represents progress, and the radius of the spiral represents cost.

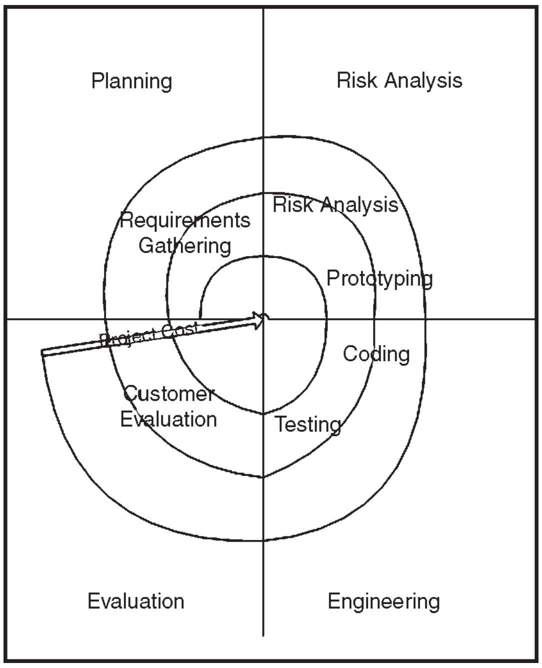


Fig. Spiral Life Cycle Model.

Advantages

- High amount of risk analysis
- Good for large and mission-critical projects.
- Software is produced early in the software life cycle.

Disadvantages

- Can be a costly model to use.
- Risk analysis requires highly specific expertise.
- Project's success is highly dependent on the risk analysis phase.
- Doesn't work well for smaller projects.

Evolutionary prototyping model

This is kind of mix of Waterfall model and prototyping. Presuppose gradual refinement of the prototype until a usable product emerge. Might be suitable in projects where the main problem is user interface requirements, but internal architecture is relatively well established and static. Can help to cope with organizational sclerosis. One variant involves so called "binary" software implementation model using a scripting language plus statically typed language.

In this case system first is coded in a scripting language and then gradually critical components are rewritten in the lower language.

OSS development model

It is the latest variant of evolutionary prototype model. The "waterfall model" was probably the first published model and as a specific model for military it was not as naive as some proponents of other models suggest.

The model was developed to help cope with the increasing complexity of aerospace products. The waterfall model followed a documentation driven paradigm.

Prototyping model was probably the first realistic of early models because many aspects of the syst4m are unclear until a working prototype is developed. A better model, the "spiral model" was suggested by Boehm in 1985. The spiral model is a variant of "dialectical spiral" and as such provides useful insights into the life cycle of the system.

But it also presuppose unlimited resources for the project. No organization can perform more then a couple iterations during the initial development of the system. the first iteration is usually called prototype.

Prototype based development requires more talented managers and good planning while waterfall model works (or does not work) with bad or stupid managers works just fine as the success in this model is more determined by the nature of the task in hand then any organizational circumstances.

Like always humans are flexible and programmer in waterfall model can use guerilla methods of enforcing a sound architecture as manager is actually a hostage of the model and cannot afford to look back and re-implement anything substantial.

Because the life cycle steps are described in very general terms, the models are adaptable and their implementation details will vary among different organizations.

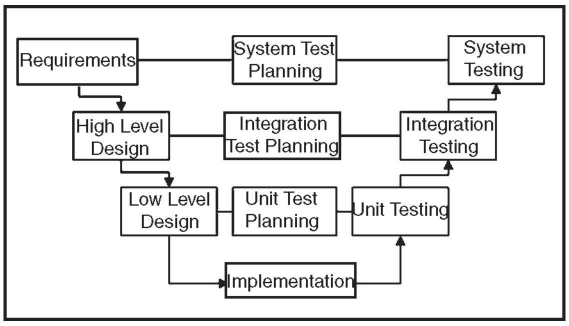
The spiral model is the most general. Most life cycle models can in fact be derived as special instances of the spiral model. Organizations may mix and match different life cycle models to develop a model more tailored to their products and capabilities.

There is nothing wrong about using waterfall model for some components of the complex project that are relatively well understood and straightforward. But mixing and matching definitely needs a certain level of software management talent.

V-Shaped Model

Just like the waterfall model, the V-Shaped life cycle is a sequential path of execution of processes. Each phase must be completed before the next phase begins. Testing is emphasized in this model more so than the waterfall model though.

The testing procedures are developed early in the life cycle before any coding is done, during each of the phases preceding implementation. Requirements begin the life cycle model just like the waterfall model. Before development is started, a system test plan is created. The test plan focuses on meeting the functionality specified in the requirements gathering. The high-level design phase focuses on system architecture and design. An integration test plan is created in this phase as well in order to test the pieces of the software systems ability to work together. The low-level design phase is where the actual software components are designed, and unit tests are created in this phase as well. The implementation phase is, again, where all coding takes place. Once coding is complete, the path of execution continues up the right side of the V where the test plans developed earlier are now put to use.



Advantages

- Simple and easy to use.
- Each phase has specific deliverables.
- Higher chance of success over the waterfall model due to the development of test plans early on during the life cycle.
- Works well for small projects where requirements are easily understood.

Disadvantages

- Very rigid, like the waterfall model.
- Little flexibility and adjusting scope is difficult and expensive.
- Software is developed during the implementation phase, so no early prototypes of the software are produced.
- Model doesn't provide a clear path for problems found during testing phases.

Incremental Model

The incremental model is an intuitive approach to the waterfall model. Multiple development cycles take place here, making the life cycle a "multi-waterfall" cycle. Cycles are divided up into smaller, more easily managed iterations.

Each iteration passes through the requirements, design, implementation and testing phases. A working version of software is produced during the first iteration, so you have working software early on during the software life cycle. Subsequent iterations build on the initial software produced during the first iteration.

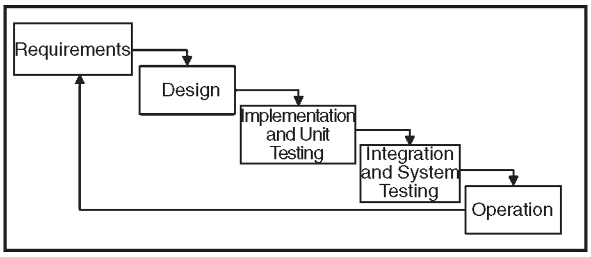

Requirements
Design
Implementation and Unit Testing
Integration and System Testing
Operation

Fig. Incremental Life Cycle Model.

Advantages

- Generates working software quickly and early during the software life cycle.
- *More flexible*: Less costly to change scope and requirements.
- Easier to test and debug during a smaller iteration.
- Easier to manage risk because risky pieces are identified and handled during its iteration.
- Each iteration is an easily managed milestone.

Disadvantages

- Each phase of an iteration is rigid and do not overlap each other.
- Problems may arise pertaining to system architecture because not all requirements are gathered up front for the entire software life cycle.

6

Systems Architecture Engineering

Systems engineering is an interdisciplinary field of engineering that focuses on how complex engineering projects should be designed and managed over the life cycle of the project. Issues such as logistics, the coordination of different teams, and automatic control of machinery become more difficult when dealing with large, complex projects. Systems engineering deals with work-processes and tools to handle such projects, and it overlaps with both technical and human-centered disciplines such as control engineering, industrial engineering, organizational studies, and project management.

History

The term *systems engineering* can be traced back to Bell Telephone Laboratories in the 1940s. The need to identify and manipulate the properties of a system as a whole, which in complex engineering projects may greatly differ from the sum of the parts' properties, motivated the Department of

Defense, NASA, and other industries to apply the discipline. When it was no longer possible to rely on design evolution to improve upon a system and the existing tools were not sufficient to meet growing demands, new methods began to be developed that addressed the complexity directly. The evolution of systems engineering, which continues to this day, comprises the development and identification of new methods and modeling techniques. These methods aid in better comprehension of engineering systems as they grow more complex. Popular tools that are often used in the systems engineering context were developed during these times, including USL, UML, QFD, and IDEF0. In 1990, a professional society for systems engineering, the *National Council on Systems Engineering* (NCOSE), was founded by representatives from a number of U.S. corporations and organizations. NCOSE was created to address the need for improvements in systems engineering practices and education.

As a result of growing involvement from systems engineers outside of the U.S., the name of the organization was changed to the International Council on Systems Engineering (INCOSE) in 1995. Schools in several countries offer graduate programmes in systems engineering, and continuing education options are also available for practicing engineers.

Concept

Systems engineering signifies both an approach and, more recently, a discipline in engineering. The aim of education in systems engineering is to simply formalize the approach and in doing so, identify new methods and research opportunities similar to the way it occurs in other fields of engineering. As an approach, systems engineering is holistic and interdisciplinary in flavour.

Origins and Traditional Scope

The traditional scope of engineering embraces the design, development, production and operation of physical systems, and systems engineering, as originally conceived, falls within this scope. "Systems engineering", in this sense of the term, refers to the distinctive set of concepts, methodologies, organizational structures (and so on) that have been developed to meet the challenges of engineering functional physical systems of unprecedented complexity. The Apollo programme is a leading example of a systems engineering project.

The use of the term " system engineer " has evolved over time to embrace a wider, more holistic concept of "systems" and of engineering processes. This evolution of the definition has been a subject of ongoing controversy [9], and the term continues to be applied to both the narrower and broader scope.

Holistic View

Systems engineering focuses on analyzing and eliciting customer needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation while considering the complete problem, the system lifecycle. Oliver *et al.* claim that the systems engineering process can be decomposed into

- a *Systems Engineering Technical Process*, and
- a *Systems Engineering Management Process.*

Within Oliver's model, the goal of the Management Process is to organize the technical effort in the lifecycle, while the Technical Process includes *assessing available information*, *defining effectiveness measures*, to *create a behaviour model*, *create a structure model*, *perform trade-off analysis*, and *create sequential build & test plan*. Depending on their application, although there are several models that are used in the industry, all of them aim to identify the relation between the various stages mentioned above and incorporate feedback. Examples of such models include the Waterfall model and the VEE model.

Interdisciplinary Field

System development often requires contribution from diverse technical disciplines. By providing a systems (holistic) view of the development effort, systems engineering helps mold all the technical contributors into a unified team effort, forming a structured development process that proceeds from concept to production to operation and, in some cases, to termination and disposal. This perspective is often replicated in educational programmes in that systems engineering courses are taught by faculty from other engineering departments which, in effect, helps create an interdisciplinary environment.

Managing Complexity

The need for systems engineering arose with the increase in complexity of systems and projects, in turn exponentially increasing the possibility of component friction, and therefore the reliability of the design. When speaking in this context, complexity incorporates not only engineering systems, but also the logical human organization of data. At the same time, a system can become more complex due to an increase in size as well as with an increase in the amount of data, variables, or the number of fields that are involved in the design. The International Space Station is an example of such a system. The development of smarter control algorithms, microprocessor design, and analysis of environmental systems also come within the purview of systems engineering. Systems engineering encourages the use of tools and methods to better comprehend and manage complexity in systems. Some examples of these tools can be seen here:

- *System model, Modeling, and Simulation*,
- System architecture,
- *Optimization*,
- *System dynamics*,
- *Systems analysis*,
- *Statistical analysis*,
- *Reliability analysis*, and
- *Decision making*

Taking an interdisciplinary approach to engineering systems is inherently complex since the behaviour of and interaction among system components is not always immediately well defined or understood.

Defining and characterizing such systems and subsystems and the interactions among them is one of the goals of systems engineering. In doing so, the gap that exists between informal requirements from users, operators, marketing organizations, and technical specifications is successfully bridged.

Scope

One way to understand the motivation behind systems engineering is to see it as a method, or practice, to identify and improve common rules that exist within a wide variety of systems. Keeping this in mind, the principles of systems engineering — holism, emergent behaviour, boundary, et al. — can be applied to any system, complex or otherwise, provided systems thinking is employed at all levels. Besides defense and aerospace, many information and technology based companies, software development firms, and industries in the field of electronics & communications require systems engineers as part of their team. An analysis by the INCOSE Systems Engineering center of excellence (SECOE) indicates that optimal effort spent on systems engineering is about 15-20% of the total project effort. At the same time, studies have shown that systems engineering essentially leads to reduction in costs among other benefits. However, no quantitative survey at a larger scale encompassing a wide variety of industries has been conducted until recently. Such studies are underway to determine the effectiveness and quantify the benefits of systems engineering. Systems engineering encourages the use of modeling and simulation to validate assumptions or theories on systems and the interactions within them.

Use of methods that allow early detection of possible failures, in safety engineering, are integrated into the design process. At the same time, decisions made at the beginning of a project whose consequences are not clearly understood can have enormous implications later in the life of a system, and it is the task of the modern systems engineer to explore these issues and make critical decisions. There is no method which guarantees that decisions made today will still be valid when a system goes into service years or decades after it is first conceived but there are techniques to support the process of systems engineering. Examples include the use of soft systems methodology, Jay Wright Forrester's System dynamics method and the Unified Modeling Language (UML), each of which are currently being explored, evaluated and developed to support the engineering decision making process.

Education

Education in systems engineering is often seen as an extension to the regular engineering courses, reflecting the industry attitude that engineering students need a foundational background in one of the traditional engineering disciplines (e.g. automotive engineering, mechanical engineering, industrial engineering, computer engineering, electrical engineering) plus practical, real-world experience in order to be effective as systems engineers. Undergraduate university programmes in systems engineering are rare. INCOSE maintains a continuously updated Directory of Systems Engineering Academic Programmes worldwide. As of 2006, there are about 75 institutions in United States that offer 130 undergraduate and graduate programmes in systems engineering. Education in systems engineering can be taken as *SE-centric* or *Domain-centric*.

- *SE-centric* programmes treat systems engineering as a separate discipline and all the courses are taught focusing on systems engineering practice and techniques.

- *Domain-centric* programmes offer systems engineering as an option that can be exercised with another major field in engineering.

Both these patterns cater to educate the systems engineer who is able to oversee interdisciplinary projects with the depth required of a core-engineer.

Systems Engineering Topics

Systems engineering tools are strategies, procedures, and techniques that aid in performing systems engineering on a project or product. The purpose of these tools vary from database management, graphical browsing, simulation, and reasoning, to document production, neutral import/export and more.

System

There are many definitions of what a system is in the field of systems engineering. Below are a few authoritative definitions:

- ANSI/EIA-632-1999: "An aggregation of end products and enabling products to achieve a given purpose."
- IEEE Std 1220-1998: "A set or arrangement of elements and processes that are related and whose behaviour satisfies customer/operational needs and provides for life cycle sustainment of the products."
- ISO/IEC 15288:2008: "A combination of interacting elements organized to achieve one or more stated purposes."
- NASA Systems Engineering Handbook: "(1) The combination of elements that function together to

produce the capability to meet a need. The elements include all hardware, software, equipment, facilities, personnel, processes, and procedures needed for this purpose. (2) The end product (which performs operational functions) and enabling products (which provide life-cycle support services to the operational end products) that make up a system."

- INCOSE Systems Engineering Handbook: "homogeneous entity that exhibits predefined behaviour in the real world and is composed of heterogeneous parts that do not individually exhibit that behaviour and an integrated configuration of components and/or subsystems."
- INCOSE: "A system is a construct or collection of different elements that together produce results not obtainable by the elements alone. The elements, or parts, can include people, hardware, software, facilities, policies, and documents; that is, all things required to produce systems-level results. The results include system level qualities, properties, characteristics, functions, behaviour and performance. The value added by the system as a whole, beyond that contributed independently by the parts, is primarily created by the relationship among the parts; that is, how they are interconnected."

The Systems Engineering Process

Depending on their application, tools are used for various stages of the systems engineering process:

Using Models

Models play important and diverse roles in systems engineering. A model can be defined in several ways, including:

- An abstraction of reality designed to answer specific questions about the real world
- An imitation, analogue, or representation of a real world process or structure; or
- A conceptual, mathematical, or physical tool to assist a decision maker.

Together, these definitions are broad enough to encompass physical engineering models used in the verification of a system design, as well as schematic models like a functional flow block diagram and mathematical (i.e., quantitative) models used in the trade study process. This section focuses on the last. The main reason for using mathematical models and diagrams in trade studies is to provide estimates of system effectiveness, performance or technical attributes, and cost from a set of known or estimable quantities. Typically, a collection of separate models is needed to provide all of these outcome variables. The heart of any mathematical model is a set of meaningful quantitative relationships among its inputs and outputs. These relationships can be as simple as adding up constituent quantities to obtain a total, or as complex as a set of differential equations describing the trajectory of a spacecraft in a gravitational field. Ideally, the relationships express causality, not just correlation.

Tools for Graphic Representations

Initially, when the primary purpose of a systems engineer is to comprehend a complex problem, graphic representations of a system are used to communicate a system's functional and data requirements. Common graphical representations include:

- Functional Flow Block Diagram (FFBD)
- VisSim
- Data Flow Diagram (DFD)
- N2 (N-Squared) Chart
- IDEF0 Diagram
- UML Use case diagram
- UML Sequence diagram
- USL Function Maps and Type Maps.
- Enterprize Architecture frameworks, like TOGAF, MODAF, Zachman Frameworks etc.

A graphical representation relates the various subsystems or parts of a system through functions, data, or interfaces. Any or each of the above methods are used in an industry based on its requirements. For instance, the N2 chart may be used where interfaces between systems is important. Part of the design phase is to create structural and behavioural models of the system. Once the requirements are understood, it is now the responsibility of a systems engineer to refine them, and to determine, along with other engineers, the best technology for a job.

At this point starting with a trade study, systems engineering encourages the use of weighted choices to determine the best option. A decision matrix, or Pugh method, is one way (QFD is another) to make this choice while considering all criteria that are important. The trade study in turn informs the design which again affects the graphic representations of the system (without changing the requirements). In an SE process, this stage represents the iterative step that is carried out until a feasible solution is found. A decision matrix is often populated using techniques such as statistical analysis, reliability analysis, system dynamics (feedback control), and optimization methods. At times a systems engineer must assess the existence of feasible solutions, and rarely will customer inputs arrive at only one. Some customer requirements will produce no feasible solution. Constraints must be traded to find one or more feasible solutions.

The customers' wants become the most valuable input to such a trade and cannot be assumed. Those wants/desires may only be discovered by the customer once the customer finds that he has overconstrained the problem. Most commonly, many feasible solutions can be found, and a sufficient set of constraints must be defined to produce an optimal solution.

This situation is at times advantageous because one can present an opportunity to improve the design towards one or many ends, such as cost or schedule. Various modeling methods can be used to solve the problem including constraints and a cost function. Systems Modeling Language (SysML), a modeling language used for systems engineering applications, supports the specification, analysis, design, verification and validation of a broad range of complex systems. Universal Systems Language (USL) is a systems oriented object modeling language with executable (computer independent) semantics for defining complex systems, including software.

Related Fields and Sub-fields

Many related fields may be considered tightly coupled to systems engineering. These areas have contributed to the development of systems engineering as a distinct entity.

Cognitive Systems Engineering

Cognitive systems engineering (CSE) is a specific approach to the description and analysis of human-machine systems or sociotechnical systems. The three main themes of CSE are how humans cope with complexity, how work is accomplished by the use of artefacts, and how human-machine systems and socio-technical systems can be described as joint cognitive systems.

CSE has since its beginning become a recognised scientific discipline, sometimes also referred to as Cognitive Engineering. The concept of a Joint Cognitive System (JCS) has in particular become widely used as a way of understanding how complex socio-technical systems can be described with varying degrees of resolution. The more than 20 years of experience with CSE has been described extensively.

Configuration Management

Like systems engineering, Configuration Management as practiced in the defence and aerospace industry is a broad systems-level practice. The field parallels the taskings of systems engineering; where systems engineering deals with requirements development, allocation to development items and verification, Configuration Management deals with requirements capture, traceability to the development item, and audit of development item to ensure that it has achieved the desired functionality that systems engineering and/or Test and Verification Engineering have proven out through objective testing.

Control Engineering

Control engineering and its design and implementation of control systems, used extensively in nearly every industry, is a large sub-field of systems engineering. The cruise control on an automobile and the guidance system for a ballistic missile are two examples. Control systems theory is an active field of applied mathematics involving the investigation of solution spaces and the development of new methods for the analysis of the control process.

Industrial Engineering

Industrial engineering is a branch of engineering that concerns the development, improvement, implementation and evaluation of integrated systems of people, money, knowledge, information, equipment, energy, material and process. Industrial engineering draws upon the principles and methods of engineering analysis and synthesis, as well as mathematical, physical and social sciences together with the principles and methods of engineering analysis and design to specify, predict and evaluate the results to be obtained from such systems.

Interface Design

Interface design and its specification are concerned with assuring that the pieces of a system connect and inter-operate with other parts of the system and with external systems as necessary. Interface design also includes assuring that system interfaces be able to accept new features, including mechanical, electrical and logical interfaces, including reserved wires, plug-space, command codes and bits in communication protocols. This is known as extensibility. Human-Computer Interaction (HCI) or Human-Machine Interface (HMI) is another aspect of interface design, and is a critical aspect of modern systems engineering. Systems engineering principles are applied in the design of network protocols for local-area networks and wide-area networks.

Mechatronic Engineering

Mechatronic engineering, like Systems engineering, is a multidisciplinary field of engineering that uses dynamical systems modeling to express tangible constructs. In that regards it is almost indistinguishable from Systems Engineering, but what sets it apart is the focus on smaller details rather than larger generalizations and relationships. As such, both fields are distinguished by the scope of their projects rather than the methodology of their practice.

Operations Research

Operations research supports systems engineering. The tools of operations research are used in systems analysis, decision making, and trade studies. Several schools teach SE courses within the operations research or industrial engineering department, highlighting the role systems engineering plays in complex projects. Operations research, briefly, is concerned with the optimization of a process under multiple constraints.

Performance Engineering

Performance engineering is the discipline of ensuring a system will meet the customer's expectations for performance throughout its life. Performance is usually defined as the speed with which a certain operation is executed or the capability of executing a number of such operations in a unit of time. Performance may be degraded when an operations queue to be executed is throttled when the capacity is of the system is limited. For example, the performance of a packet-switched network would be characterised by the end-to-end packet transit delay or the number of packets switched within an hour. The design of high-performance systems makes use of analytical or simulation modeling, whereas the delivery of high-performance implementation involves thorough performance testing. Performance engineering relies heavily on statistics, queueing theory and probability theory for its tools and processes.

Programme Management and Project Management

Programme management (or programme management) has many similarities with systems engineering, but has broader-based origins than the engineering ones of systems engineering. Project management is also closely related to both programme management and systems engineering.

Proposal Engineering

Proposal engineering is the application of scientific and mathematical principles to design, construct, and operate a cost-effective proposal development system. Basically, proposal engineering uses the "systems engineering process" to create a cost effective proposal and increase the odds of a successful proposal.

Reliability Engineering

Reliability engineering is the discipline of ensuring a system will meet the customer's expectations for reliability throughout its life; i.e. it will not fail more frequently than expected. Reliability engineering applies to all aspects of the system.

It is closely associated with maintainability, availability and logistics engineering. Reliability engineering is always a critical component of safety engineering, as in failure modes and effects analysis (FMEA) and hazard fault tree analysis, and of security engineering. Reliability engineering relies heavily onstatistics, probability theory and reliability theory for its tools and processes.

Safety Engineering

The techniques of safety engineering may be applied by non-specialist engineers in designing complex systems to minimize the probability of safety-critical failures. The "System Safety Engineering" function helps to identify "safety hazards" in emerging designs, and may assist with techniques to "mitigate" the effects of (potentially) hazardous conditions that cannot be designed out of systems.

Security Engineering

Security engineering can be viewed as an interdisciplinary field that integrates the community of practice for control systems design, reliability, safety and systems engineering.

It may involve such sub-specialties as authentication of system users, system targets and others: people, objects and processes.

Software Engineering

From its beginnings, software engineering has helped shape modern systems engineering practice. The techniques used in the handling of complexes of large software-intensive systems has had a major effect on the shaping and reshaping of the tools, methods and processes of SE.

Systems Architecture

A system architecture or systems architecture is the conceptual model that defines the structure, behaviour, and more views of a system. An architecture description is a formal description and representation of a system, organized in a way that supports reasoning about the structure of the system which comprises system components, the externally visible properties of those components, the relationships (e.g. the behaviour) between them, and provides a plan from which products can be procured, and systems developed, that will work together to implement the overall system. The language for architecture description is called the architecture description language (ADL).

Overview

There is no universally agreed definition of which aspects constitute a system architecture, and various organizations define it in different ways, including:

- The fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

- The composite of the design architectures for products and their life cycle processes.
- A representation of a system in which there is a mapping of functionality onto hardware and software components, a mapping of the software architecture onto the hardware architecture, and human interaction with these components.
- An allocated arrangement of physical elements which provides the design solution for a consumer product or life-cycle process intended to satisfy the requirements of the functional architecture and the requirements baseline.
- An architecture is the most important, pervasive, top-level, strategic inventions, decisions, and their associated rationales about the overall structure (i.e., essential elements and their relationships) and associated characteristics and behaviour.
- A description of the design and contents of a computer system. If documented, it may include information such as a detailed inventory of current hardware, software and networking capabilities; a description of long-range plans and priorities for future purchases, and a plan for upgrading and/or replacing dated equipment and software.
- A formal description of a system, or a detailed plan of the system at component level to guide its implementation.

- The structure of components, their interrelationships, and the principles and guidelines governing their design and evolution over time.

A system architecture can best be thought of as a set of representations of an existing (or To Be Created) system. It is used to convey the informational content of the elements comprising a system, the relationships among those elements, and the rules governing those relationships. The architectural components and set of relationships between these components that an architecture describes may consist of hardware, software, documentation, facilities, manual procedures, or roles played by organizations or people.

A system architecture is primarily concerned with the internal interfaces among the system's components or subsystems, and the interface between the system and its external environment, especially the user. (In the specific case of computer systems, this latter, special interface, is known as the computer human interface, *AKA* human computer interface, or CHI; formerly called the man-machine interface.) A system architecture can be contrasted with system architecture engineering, which is the method and discipline for effectively implementing the architecture of a system:

- It is a *method* because a sequence of steps is prescribed to produce or change the architecture of a system within a set of constraints.
- It is a *discipline* because a body of knowledge is used to inform practitioners as to the most effective way to architect the system within a set of constraints.

History

It is important to keep in mind that the modern systems architecture did not appear out of nowhere. Systems architecture depends heavily on practices and techniques which were developed over thousands of years in many other fields most importantly being, perhaps, civil architecture. Prior to the advent of digital computers, the electronics and other engineering disciplines used the term system as it is still commonly used today. However, with the arrival of digital computers and the development of software engineering as a separate discipline, it was often necessary to distinguish among engineered hardware artifacts, software artifacts, and the combined artifacts. A programmable hardware artifact, or computing machine, that lacks its software programme is impotent; even as a software artifact, or programme, is equally impotent unless it can be used to alter the sequential states of a suitable (hardware) machine. However, a hardware machine and its software programme can be designed to perform an almost illimitable number of abstract and physical tasks. Within the computer and software engineering disciplines (and, often, other engineering disciplines, such as communications), then, the term system came to be defined as containing all of the elements necessary (which generally includes both hardware and software) to perform a useful function.

Consequently, within these engineering disciplines, a system generally refers to a programmable hardware machine and its included programme. And a systems engineer is defined as one concerned with the complete device, both hardware and software and, more particularly, all of the interfaces of the device, including that between hardware and software, and especially between the complete device and its user (the CHI). The hardware engineer deals (more or less) exclusively with the hardware device; the software engineer deals (more or less) exclusively with the software programme; and the systems engineer is responsible for seeing that the software programme is capable of properly running within the hardware device, and that the system composed of the two entities is capable of properly interacting with its external environment, especially the user, and performing its intended function. By analogy, then, a systems architecture makes use of elements of both software and hardware and is used to enable design of such a composite system. A good architecture may be viewed as a 'partitioning scheme,' or algorithm, which partitions all of the system's present and foreseeable requirements into a workable set of cleanly bounded subsystems with nothing left over. That is, it is a partitioning scheme which is exclusive, inclusive, and exhaustive.

A major purpose of the partitioning is to arrange the elements in the sub systems so that there is a minimum of communications needed among them. In both software and hardware, a good sub system tends to be seen to be a meaningful "object". Moreover, a good architecture provides for an easy mapping to the user's requirements and the validation tests of the user's requirements. Ideally, a mapping also exists from every least element to every requirement and test. A *robust architecture* is said to be one that exhibits an optimal degree of fault-tolerance, backward compatibility, forward compatibility, extensibility, reliability, maintainability, availability, serviceability, usability, and such other quality attributes as necessary and/or desirable.

Types of Systems Architectures

Several types of systems architectures (underlain by the same fundamental principles) have been identified as follows:

- Collaborative Systems (such as the Internet, intelligent transportation systems, and joint air defense systems)
- Manufacturing Systems
- Social Systems
- Software and Information Technology Systems
- Strategic Systems Architecture

Systems Architect

In systems engineering, the systems architect is the high-level designer of a system to be implemented. The systems architect establishes the basic structure of the system, defining the essential core design features and elements that provide the framework for all that follows, and are the hardest to change later. The systems architect provides the engineering view of the users' vision for what the system needs to be and do, and the paths along which it must be able to evolve, and strives to maintain the integrity of that vision as it evolves during detailed design and implementation.

Overview

In systems engineering, the systems architect is responsible for:

- Interfacing with the user(s) and sponsor(s) and all other stakeholders in order to determine their (evolving) needs.

- Generating the highest level of system requirements, based on the user's needs and other constraints such as cost and schedule.
- Ensuring that this set of high level requirements is consistent, complete, correct, and operationally defined.
- Performing cost-benefit analyses to determine whether requirements are best met by manual, software, or hardware functions; making maximum use of commercial off-the-shelf or already developed components.
- Developing partitioning algorithms (and other processes) to allocate all present and foreseeable requirements into discrete partitions such that a minimum of communications is needed among partitions, and between the user and the system.
- Partitioning large systems into (successive layers of) subsystems and components each of which can be handled by a single engineer or team of engineers or subordinate architect.
- Interfacing with the design and implementation engineers, or subordinate architects, so that any problems arising during design or implementation can be resolved in accordance with the fundamental architectural concepts, and user needs and constraints.
- Ensuring that a maximally robust architecture is developed.
- Generating a set of acceptance test requirements, together with the designers, test engineers, and the

user, which determine that all of the high level requirements have been met, especially for the computer-human-interface.

- Generating products such as sketches, models, an early user guide, and prototypes to keep the user and the engineers constantly up to date and in agreement on the system to be provided as it is evolving.
- Ensuring that all architectural products and products with architectural input are maintained in the most current state and never allowed to become obsolete.

Main Topics of Systems Architect

Large systems architecture was developed as a way to handle systems too large for one person to conceive of, let alone design. Systems of this size are rapidly becoming the norm, so architectural approaches and architects are increasingly needed to solve the problems of large systems.

Users and Sponsors

Engineers as a group do not have a reputation for understanding and responding to human needs comfortably or for developing humanly functional and aesthetically pleasing products. Architects *are* expected to understand human needs and develop humanly functional and aesthetically pleasing products. A good architect is a translator between the user/sponsor and the engineers— and even among just engineers of different specialities. A good architect is also the principal keeper of the user's vision of the end product— and of the process of deriving requirements from and implementing that vision.

Determining what the users/sponsors actually need, rather than what they say they want, is not engineering. An architect does not follow an exact procedure. S/he communicates with users/sponsors in a highly interactive way— together they extract the true *requirements* necessary for the engineered system. The architect must remain constantly in communication with the end users. Therefore, the architect must be intimately familiar with the user's environment and problem. (The engineer need only be very knowledgeable of the potential engineering solution space.)

High Level Requirements

The user/sponsor should view the architect as the user's representative and provide *all input through the architect.* Direct interaction with project engineers is generally discouraged as the chance of mutual misunderstanding is very high. The user requirements' specification should be a joint product of the user and architect: the user brings his needs and wish list, the architect brings knowledge of what is likely to prove doable within cost and time constraints. When the user needs are translated into a set of high level requirements is also the best time to write the first version of the acceptance test, which should, thereafter, be religiously kept up to date with the requirements. That way, the user will be absolutely clear about what s/he is getting. It is also a safeguard against untestable requirements, misunderstandings, and requirements creep. The development of the first level of engineering requirements is not a purely analytical exercise and should also involve both the architect and engineer. If any compromises are to be made— to meet constraints like cost, schedule, power, or space, the architect must ensure that the final product and overall look and feel do not stray very far from the user's intent. The engineer should focus on developing a design that optimizes the constraints but ensures a workable and reliable product.

The architect is primarily concerned with the comfort and usability of the product; the engineer is primarily concerned with the producibility and utility of the product. The provision of needed services to the user is the true function of an engineered system. However, as systems become ever larger and more complex, and as their emphases move away from simple hardware and software components, the narrow application of traditional systems development principles is found to be insufficient— the application of the more general principles of systems, hardware, and software architecture to the design of (sub)systems is seen to be needed. An architecture is also a simplified model of the finished end product— its primary function is to define the parts and their relationships to each other so that the whole can be seen to be a consistent, complete, and correct representation of what the user had in mind— especially for the computer-human-interface. It is also used to ensure that the parts fit together and relate in the desired way.

It is necessary to distinguish between the architecture of the user's world and the engineered systems architecture. The former represents and addresses problems and solutions in the *user's* world. It is principally captured in the computer-human-interfaces (CHI) of the engineered system. The engineered system represents the *engineering* solutions— how the *engineer* proposes to develop and/or select and combine the components of the technical infrastructure to support the CHI. In the absence of an experienced architect, there is an unfortunate tendency to confuse the two architectures. But— the engineer thinks in terms of hardware and software and the technical solution space, whereas the user may be thinking in terms of solving a problem of getting people from point A to point B in a reasonable amount of time and with a reasonable expenditure of energy, or of getting needed information to customers and staff. A systems architect is expected to combine knowledge of both the architecture of the user's world and of (all potentially useful) engineering systems architectures. The former is a joint activity with the user; the latter is a joint activity with the engineers. The product is a set of high level requirements reflecting the user's requirements which can be used by the engineers to develop systems design requirements. Because requirements evolve over the course of a project, especially a long one, an architect is needed until the system is accepted by the user: the architect is the best insurance that all changes and interpretations made during the course of development do not compromise the user's viewpoint.

Cost/Benefit Analyses

Most engineers are specialists. They know the applications of one field of engineering science intimately, apply their knowledge to practical situations— that is, solve real world problems, evaluate the cost/benefits of various solutions within their specialty, and ensure the correct operation of whatever they design. Architects are generalists. They are not expected to be experts in any one technology but are expected to be knowledgeable of many technologies and able to judge their applicability to specific situations. They also apply their knowledge to practical situations, but evaluate the cost/benefits of various solutions using different technologies, for example, hardware versus software versus manual, and assure that the system as a whole performs according to the user's expectations. Many commercial-off-the-shelf or already developed hardware and software components may be selected independently according to constraints such as cost, response, throughput, etc. In some cases, the architect can already assemble the end system unaided. Or, s/he may still need the help of a hardware or software engineer to select components and to design and build any special purpose function. The architects (or engineers) may also enlist the aid of specialists— in safety, security, communications, special purpose hardware, graphics, human factors, test and evaluation, quality control, RMA, interface management, etc. An effective systems architectural team must have immediate access to specialists in critical specialties.,

Partitioning and Layering

An architect planning a building works on the overall design, making sure it will be pleasing and useful to its inhabitants. While a single architect by himself may be enough to build a single-family house, many engineers may be needed, in addition, to solve the detailed problems that arise when a novel high-rise building is designed. If the job is large and complex enough, parts of the architecture may be designed as independent components. That is, if we are building a housing complex, we may have one architect for the complex, and one for each type of building, as part of an *architectural team.* Large automation systems also require an architect and much engineering talent. If the engineered system is large and complex enough, the systems architect may defer to a hardware architect and a software architect for parts of the job, although they all may be members of a joint architectural team. The architect should sub-allocate the system requirements to major components or subsystems that are within the scope of a single hardware or software engineer, or engineering manager and team. *But the architect must never be viewed as an engineering supervisor.* (If the item is sufficiently large and/or complex, the chief architect will sub-allocate portions to more specialized architects.) Ideally, each such component/subsystem is a sufficiently stand-alone object that it can be tested as a complete component, separate from the whole, using only a simple testbed to supply simulated inputs and record outputs. That is, it is not necessary to know how an air traffic control system works in order to design and build a data management subsystem for it.

It is only necessary to know the constraints under which the subsystem will be expected to operate. A good architect ensures that the system, however complex, is built upon relatively simple and "clean" concepts for each (sub)system or layer and is easily understandable by everyone, especially the user, without special training. The architect will use a minimum of heuristics to ensure that each partition is well defined and clean of kludges, work-arounds, short-cuts, or confusing detail and exceptions. As user needs evolve, (once the system is fielded and in use), it is a lot easier subsequently to evolve a simple concept than one laden with exceptions, special cases, and lots of "fine print." *Layering* the architecture is important for keeping the architecture sufficiently simple at each *layer* so that it remains comprehensible to a single mind. As layers are ascended, whole systems at *lower layers* become simple *components* at the *higher layers,* and may disappear altogether at the *highest layers.*

Acceptance Test

The acceptance test is a principal responsibility of the systems architect. It is the chief means by which the architect will prove to the user that the system is as originally planned and that all subordinate architects and engineers have met their objectives.

Communications with Users and Engineers

A building architect uses sketches, models, and drawings. An automation systems (or software or hardware) architect should use sketches, models, and prototypes to discuss different solutions and results with users, engineers, and other architects. An early, draft version of the user's manual is invaluable, especially in conjunction with a prototype. A set of (engineering) *requirements* as a sole, or even principal, means of communicating with the users is explicitly to be avoided.

Nevertheless, it is important that a workable, well written *set of requirements,* or specification, be created which is understandable to the customer (so that they can properly sign off on it). But it must use precise and unambiguous language so that designers and other implementers are left in no doubt as to meanings or intentions. In particular, all requirements must be testable, and the initial draft of the test plan should be developed contemporaneously with the requirements. All stakeholders should sign off on the acceptance test descriptions, or equivalent, as the sole determinant of the satisfaction of the requirements, at the outset of the programme.