Introduction to Operating System (OS)

By Vinod Sencha
Core Faculty(IS), RTI Jaipur

Course Content:

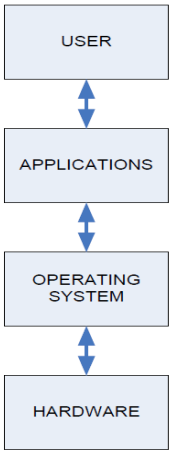
- What is an OS.
- What are its key functions.
- The evaluation of OS.
- What are the popular types of OS.
- Basics of UNIX and Windows.
- Advantages of open source OS like Linux.
- Networks OS.

What is an Operating System?

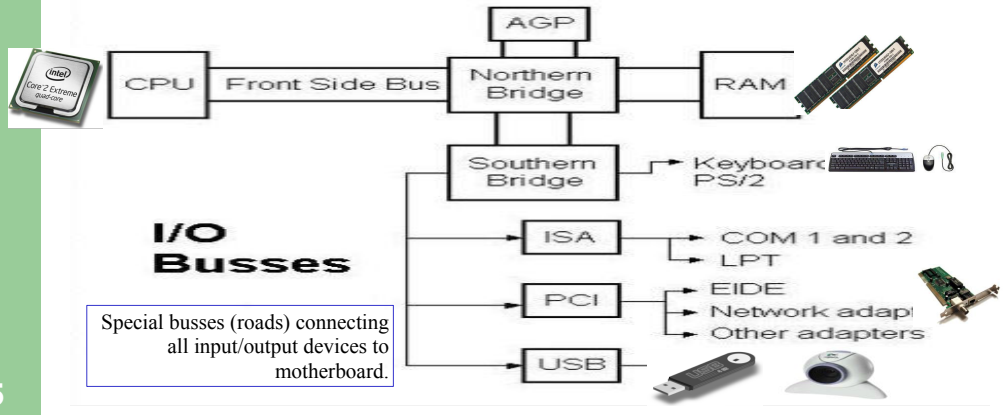
- Computer System = Hardware + Software
- Software = Application Software + System Software(OS)
- An Operating System is a system Software that acts as an intermediary/ interface between a user of a computer and the computer hardware.
- Operating system goals:
  - ➢ Execute user programs and make solving user problems easier
  - ➢ Make the computer system convenient to use
  - ➢ Use the computer hardware in an efficient manner

The Structure of Computer Systems

- ➢ Accessing computer resources is divided into *layers*.
- ➢ Each layer is isolated and only interacts directly with the layer below or above it.
- ➢ If we install a new hardware device
  - ✓ No need to change anything about the user/applications.
  - ✓ However, you do need to make changes to the operating system.
  - ✓ You need to install the device drivers that the operating system will use to control the new device.
- ➢ If we install a new software application
  - ✓ No need to make any changes to your hardware.
  - ✓ But we need to make sure the application is supported by the operating system
  - ✓ user will need to learn how to use the new application.
- ➢ If we change the operating system
  - ✓ Need to make sure that both applications and hardware will compatible with the new operating system.



Computer Architecture



Special busses (roads) connecting all input/output devices to motherboard.

CPU – Central Processing Unit

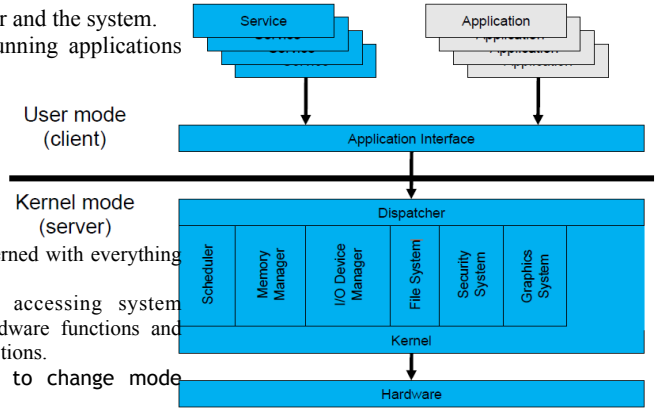
- This is the brain of your computer.
- It performs all of the calculations.
- In order to do its job, the CPU needs commands to perform, and data to work with.
- The instructions and data travel to and from the CPU on the system bus.
- The operating system provides rules for how that information gets back and forth, and how it will be used by the CPU.

- This is like a desk, or a workspace, where your computer temporarily stores all of the information (data) and instructions (software or program code) that it is currently using.
- Each RAM chip contains millions of address spaces.
- Each address space is the same size, and has its own unique identifying number (address).
- The operating system provides the rules for using these memory spaces, and controls storage and retrieval of information from RAM.
- Device drivers for RAM chips are included with the operating system.

Problem: *If RAM needs an operating system to work, and an operating system needs RAM in order to work, how does your computer activate its RAM to load the operating system?*

Operating System Mode

- The *User Mode* is concerned with the actual interface between the user and the system.
- It controls things like running applications and accessing files.



- The *Kernel Mode* is concerned with everything running in the background.
- It controls things like accessing system resources, controlling hardware functions and processing program instructions.
- System calls are used to change mode from User to Kernel.

Kernel

- Kernel is a software code that reside in central core of OS. It has complete control over system.
- When operation system boots, kernel is first part of OS to load in main memory.
- Kernel remains in main memory for entire duration of computer session. The kernel code is usually loaded in to protected area of memory.
- Kernel performs it's task like executing processes and handling interrupts in kernel space.
- User performs it's task in user area of memory.
- This memory separation is made in order to prevent user data and kernel data from interfering with each other.
- Kernel does not interact directly with user, but it interacts using SHELL and other programs and hardware.

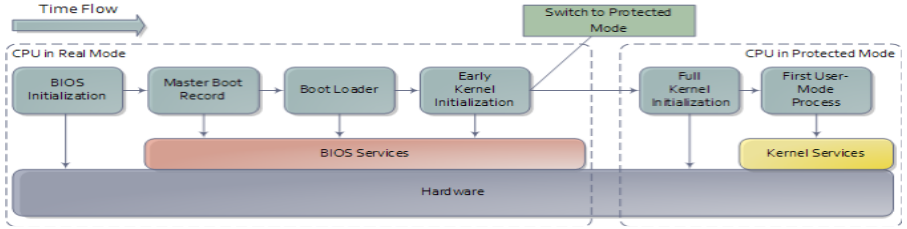
Kernel cont…

- Kernel includes:-
  1. Scheduler: It allocates the Kernel's processing time to various processes.
  2. Supervisor: It grants permission to use computer system resources to each process.
  3. Interrupt handler : It handles all requests from the various hardware devices which compete for kernel services.
  4. Memory manager : allocates space in memory for all users of kernel service.
- kernel provides services for process management, file management, I/O management, memory management.
- System calls are used to provide these type of services.

System Call

- **System call** is the programmatic way in which a computer program/user application requests a service from the kernel of the operating system on which it is executed.
- Application program is just a user-process. Due to security reasons , user applications are not given access to privileged resources(the ones controlled by OS).
- When they need to **do any I/O** or have **some more memory** or **spawn a process** or wait for **signal/interrupt,** it requests operating system to facilitate all these. This **request is made through System Call.**
- System calls are also called **software-interrupts.**

Starting an Operating System(Booting)



- ✓ Power On Switch sends electricity to the motherboard on a wire called the *Voltage Good* line.
- ✓ If the power supply is good, then the BIOS (Basic Input/Output System) chip takes over.
- ✓ In Real Mode, CPU is only capable of using approximately 1 MB of memory built into the motherboard.
- ✓ The BIOS will do a Power-On Self Test (POST) to make sure that all hardware are working.
- ✓ BIOS will then look for a small sector at the very beginning of your primary hard disk called MBR.
- ✓ The MBR contains a list, or map, of all of the partitions on your computer's hard disk (or disks).
- ✓ After the MBR is found the Bootstrap Loader follows basic instructions for starting up the rest of the computer, including the operating system.
- ✓ In Early Kernel Initialization stage, a smaller core of the Kernel is activated.
- ✓ This core includes the device drivers needed to use computer's RAM chips.

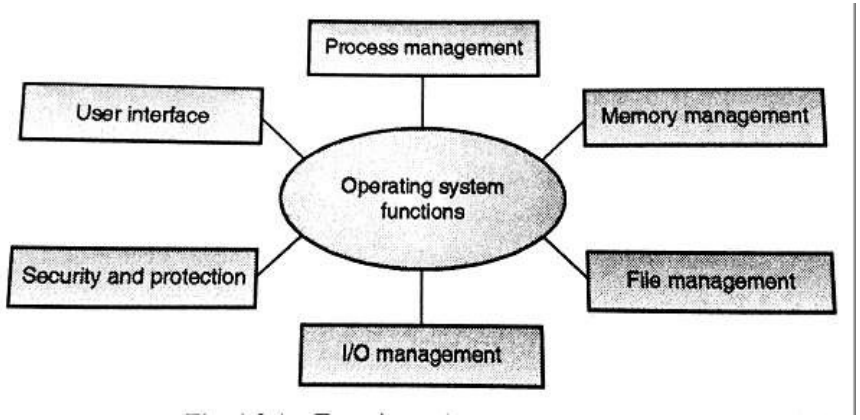
BIOS

- BIOS firmware was stored in a ROM/EPROM (Erasable Programmable Read-Only Memory) chip known as **firmware** on the PC motherboard.
- BIOS can be accessed during the initial phases of the boot procedure by pressing del, F2 or F10.
- Finally, the firmware code cycles through all storage devices and looks for a boot-loader. (usually located in first sector of a disk which is 512 bytes)
- If the boot-loader is found, then the firmware hands over control of the computer to it.

UEFI

- UEFI stands for Unified Extensible Firmware Interface. It does the same job as a BIOS, but with one basic difference: it stores all data about initialization and startup in an .efi file, instead of storing it on the firmware.
- This .efi file is stored on a special partition called EFI System Partition (ESP) on the hard disk. This ESP partition also contains the bootloader.
- UEFI was designed to overcome many limitations of the old BIOS, including:
  - ➢ UEFI supports drive sizes upto 9 zettabytes, whereas BIOS only supports 2.2 terabytes.
  - ➢ UEFI provides faster boot time.
  - ➢ UEFI has discrete driver support, while BIOS has drive support stored in its ROM, so updating BIOS firmware is a bit difficult.
  - ➢ UEFI offers security like "Secure Boot", which prevents the computer from booting from unauthorized/unsigned applications. This helps in preventing rootkits.
  - ➢ UEFI runs in 32bit or 64bit mode, whereas BIOS runs in 16bit mode. So UEFI is able to provide a GUI (navigation with mouse) as opposed to BIOS which allows navigation only using the keyboard.

Functions of Operating System



1. Process Management

- **A *process* is a program in execution.**
- A process needs certain resources, including CPU time, memory, files, and I/O devices to accomplish its task.
- Simultaneous execution leads to multiple processes. Hence creation, execution and termination of a process are the most basic functionality of an OS
- If processes are dependent, than they may try to share same resources. thus task of process synchronization comes to the picture.
- If processes are independent, than a due care needs to be taken to avoid their overlapping in memory area.
- Based on priority, it is important to allow more important processes to execute first than others.

- Memory is a large array of words or bytes, each with its own address.
- It is a repository of quickly accessible data shared by the CPU and I/O devices.
- Main memory is a **volatile** storage device. When the computer made turn off everything stored in RAM will be erased automatically.
- In addition to the physical RAM installed in your computer, most modern operating systems allow your computer to use a *virtual memory system. Virtual memory allows your computer to use part of a permanent storage device (such as a hard disk) as extra memory.*
- The operating system is responsible for the following activities in connections with memory management:
  - ➢ Keep track of which parts of memory are currently being used and by whom.
  - ➢ Decide which processes to load when memory space becomes available.
  - ➢ Allocate and de-allocate memory space as needed.

- A file is a collection of related information defined by its creator.
- *File systems provide the conventions for the encoding, storage and management of data on a storage device such as a hard disk.*
  - ➢ FAT12 (floppy disks)
  - ➢ FAT16 (DOS and older versions of Windows)
  - ➢ FAT32 (older versions of Windows)
  - ➢ NTFS (newer versions of Windows)
  - ➢ EXT3 (Unix/Linux)
  - ➢ HFS+ (Max OS X)
- The operating system is responsible for the following activities in connections with file management:
  - ✦ File creation and deletion.
  - ✦ Directory creation and deletion.
  - ✦ Support of primitives for manipulating files and directories.
  - ✦ Mapping files onto secondary storage.
  - ✦ File backup on stable (nonvolatile) storage media.

4. Device Management or I/O Management

- *Device controllers are components on the motherboard (or on expansion cards) that act as an interface between the CPU and the actual device.*
- *Device drivers, which are the operating system software components that interact with the devices controllers.*
- A special device (inside CPU) called the Interrupt Controller handles the task of receiving interrupt requests and prioritizes them to be forwarded to the processor.
- Deadlocks can occur when two (or more) processes have control of different I/O resources that are needed by the other processes, and they are unwilling to give up control of the device.
- It performs the following activities for device management.
  - ➢ Keeps tracks of all devices connected to system.
  - ➢ Designates a program responsible for every device known as Input/output controller.
  - ➢ Decides which process gets access to a certain device and for how long.
  - ➢ Allocates devices in an effective and efficient way.
  - ➢ Deallocates devices when they are no longer required.

5. Security & Protection

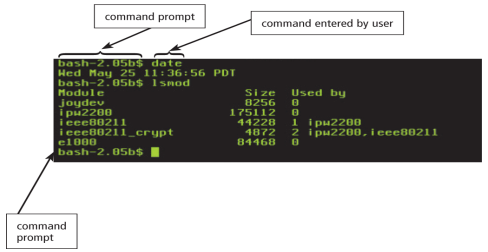
- The operating system uses password protection to protect user data and similar other techniques.
- It also prevents unauthorized access to programs and user data by assigning access right permission to files and directories.
- The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other.

6. User Interface Mechanism

- A **user interface** (**UI**) controls how you enter data and instructions and how information is displayed on the screen
- There are two types of user interfaces
  1. Command Line Interface
  2. Graphical user Interface

1. Command-line interface

- In a command-line interface, a user types commands represented by short keywords or abbreviations or presses special keys on the keyboard to enter data and instructions



2. Graphical User Interface

- With a graphical user interface (GUI), you interact with menus and visual images

History of Operating System

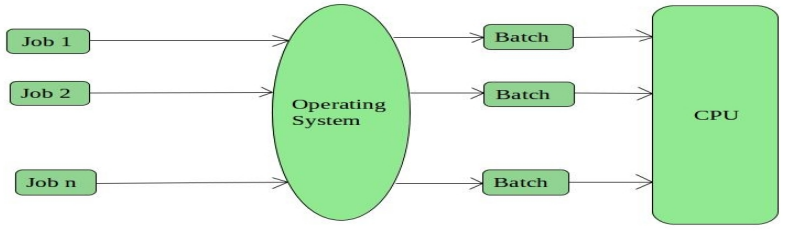
- **The First Generation (1940's to early 1950's)**
  - No Operating System
  - All programming was done in absolute machine language, often by wiring up plug-boards to control the machine's basic functions.
- **The Second Generation (1955-1965)**
  - First operating system was introduced in the early 1950's.It was called GMOS
  - Created by General Motors for IBM's machine the 701.
  - Single-stream batch processing systems
- **The Third Generation (1965-1980)**
  - Introduction of multiprogramming
  - Development of Minicomputer
- **The Fourth Generation (1980-Present Day)**
  - Development of PCs
  - Birth of Windows/MaC OS

Types of Operating Systems

1. Batch Operating System
2. Multiprogramming Operating System
3. Time-Sharing OS
4. Multiprocessing OS
5. Distributed OS
6. Network OS
7. Real Time OS
8. Embedded OS

1. Batch Operating System

- The users of this type of operating system does not interact with the computer directly.
- Each user prepares his job on an off-line device like punch cards and submits it to the computer operator
- There is an operator which takes similar jobs having the same requirement and group them into batches.



1. Batch Operating System cont..

Advantages of Batch Operating System:

- ➢ Processors of the batch systems know how long the job would be when it is in queue
- ➢ Multiple users can share the batch systems
- ➢ The idle time for the batch system is very less
- ➢ It is easy to manage large work repeatedly in batch systems

Disadvantages of Batch Operating System:

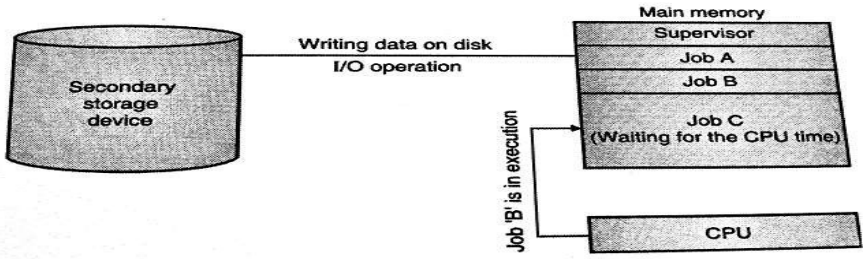
- ➢ The computer operators should be well known with batch systems
- ➢ Batch systems are hard to debug
- ➢ It is sometimes costly
- ➢ The other jobs will have to wait for an unknown time if any job fails

Examples of Batch based Operating System:

IBM's MVS

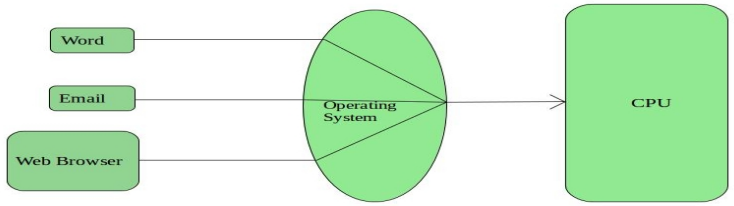
2. Multiprogramming Operating System:

- This type of OS is used to execute more than one jobs simultaneously by a single processor.
- It increases CPU utilization by organizing jobs so that the CPU always has one job to execute.
- Multiprogramming operating systems use the mechanism of job scheduling and CPU scheduling.



3. Time-Sharing Operating Systems

- Each task is given some time to execute so that all the tasks work smoothly.
- These systems are also known as **Multi-tasking Systems.**
- The task can be from a single user or different users also.
- The time that each task gets to execute is called quantum.
- After this time interval is over OS switches over to the next task.



3. Time-Sharing Operating Systems cont..

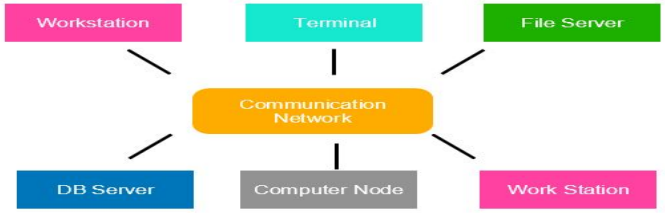
- **Advantages of Time-Sharing OS:**
  - ➢ Each task gets an equal opportunity
  - ➢ Fewer chances of duplication of software
  - ➢ CPU idle time can be reduced
- **Disadvantages of Time-Sharing OS:**
  - ➢ Reliability problem
  - ➢ One must have to take care of the security and integrity of user programs and data
  - ➢ Data communication problem
- **Examples of Time-Sharing Oss**
  Multics, Unix, etc.

4. Multiprocessor operating systems

- Multiprocessor operating systems are also known as parallel OS or tightly coupled OS.
- Such operating systems have more than one processor in close communication that sharing the computer bus, the clock and sometimes memory and peripheral devices.
- It executes multiple jobs at the same time and makes the processing faster.
- It supports large physical address space and larger virtual address space.
- If one processor fails then other processor should retrieve the interrupted process state so execution of process can continue.
- Inter-processes communication mechanism is provided and implemented in hardware.

5. Distributed Operating System

- Various autonomous interconnected computers communicate with each other using a shared communication network.
- Independent systems possess their own memory unit and CPU.
- These are referred to as **loosely coupled systems**.
- Examples:- Locus, DYSEAC



- These systems run on a server and provide the capability to manage data, users, groups, security, applications, and other networking functions.
- These types of operating systems allow shared access of files, printers, security, applications, and other networking functions over a small private network.
- The “ other" computers arc called client computers, and each computer that connects to a network server must be running client software designed to request a specific service.
- popularly known as **tightly coupled systems**.

6. Network Operating System

Advantages of Network Operating System:

➢Highly stable centralized servers

➢Security concerns are handled through servers

➢New technologies and hardware up-gradation are easily integrated into the system

➢Server access is possible remotely from different locations and types of systems

Disadvantages of Network Operating System:

➢Servers are costly

➢User has to depend on a central location for most operations

➢Maintenance and updates are required regularly

Examples of Network Operating System are:

Microsoft Windows Server 2003/2008/2012, UNIX, Linux, Mac OS X, Novell NetWare, and BSD, etc.

7. Real-Time Operating System

- These types of OSs serve real-time systems.
- The time interval required to process and respond to inputs is very small.
- This time interval is called **response time**.
- **Real-time systems** are used when there are time requirements that are very strict like
  - ➢ missile systems,
  - ➢ air traffic control systems,
  - ➢ robots, etc.

8. Embaded Operating System

- An embedded operating system is one that is built into the circuitry of an electronic device.
- Embedded operating systems are now found in automobiles, bar-code scanners, cell phones, medical equipment, and personal digital assistants.
- The most popular embedded operating systems for consumer products, such as PDAs, include the following:
  - ➢ Windows XP Embedded
  - ➢ Windows CE .NET:- it supports wireless communications, multimedia and Web browsing. It also allows for the use of smaller versions of Microsoft Word, Excel, and Outlook.
  - ➢ Palm OS:- It is the standard operating system for Palm-brand PDAs as well as other proprietary handheld devices.
  - ➢ Symbian:- OS found in “ smart” cell phones from Nokia and Sony Ericsson

Popular types of OS

- Desktop Class
  - Windows
  - OS X
  - Unix/Linux
  - Chrome OS
- Server Class
  - Windows Server
  - Mac OS X Server
  - Unix/Linux
- Mobile Class
  - Android
  - iOS
  - Windows Phone

Desktop Class Operating Systems:-

- **Platform:** the hardware required to run a particular operating system
  - Intel platform (IBM-compatible)
    - Windows
    - DOS
    - UNIX
    - Linux
  - Macintosh platform
    - Mac OS
  - iPad and iPhone platform
    - iOS

Ms-DOS

- Single User Single Tasking OS.
- It had no built-in support for networking, and users had to manually install drivers any time they added a new hardware component to their PC.
- DOS supports only 16-bit programs.
- Command line user interface.
- So, why is DOS still in use? Two reasons are its size and simplicity. It does not require much memory or storage space for the system, and it docs not require a powerful computer.

Microsoft Windo Windows 10

- The graphical Microsoft operating system designed for Intel-platform desktop and notebook computers.
- Best known, greatest selection of applications available.
- Current editions include Windows 7, 8, 8.1 and 10.

Mac OS

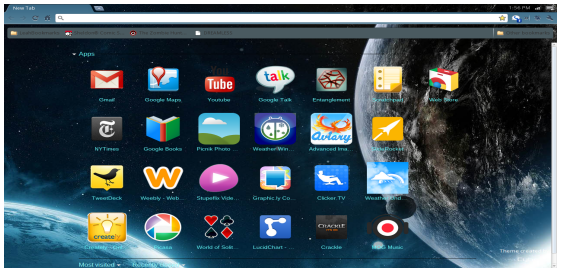
- User-friendly, runs on Mac hardware. Many applications available.
- Current editions include: Sierra, High Sierra, Mojave, Catalina & Big Sur—Version XI(Released in Nov 2020)

Linux

- **Linux:** An open-source, cross-platform operating system that runs on desktops, notebooks, tablets, and smartphones.
  - The name *Linux* is a combination *Linus* (the first name of the first developer) and *UNIX (another operating system*.
- Users are free to modify the code, improve it, and redistribute it,
- Developers are not allowed to charge money for the Linux kernel itself (the main part of the operating system), but they can charge money for **distributions** (**distros** for short).

Google Chrome OS

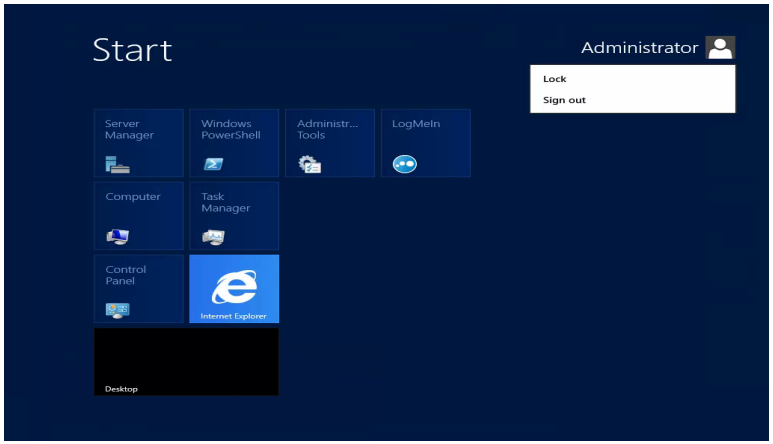
- **Chrome OS**. Is a popular thin client operating system.
- **Thin client** A computer with minimal hardware, designed for a specific task. For example, a thin web client is designed for using the Internet.



Server Operating Systems

- Windows Server
  - Familiar GUI interface for those experienced with Windows
- UNIX
  - Very mature server capabilities, time-tested, large user community, stable
- Linux
  - Free, customizable, many free services and utilities available

Windows Server



UNIX

```
mars@marsmain /usr/portage/app-shells/bash $ sudo /etc/init.d/bluetooth status
Password:
* status: started
mars@marsmain /usr/portage/app-shells/bash $ ping -q -c1 en.wikipedia.org
PING rr.esams.wikimedia.org (91.198.174.2) 56(84) bytes of data.

--- rr.esams.wikimedia.org ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 2ms
rtt min/avg/max/mdev = 49.820/49.820/49.820/0.000 ms
mars@marsmain /usr/portage/app-shells/bash $ grep -i /dev/sda /etc/fstab | cut --fields=-3
/dev/sda1               /boot
/dev/sda2               none
/dev/sda3               /
mars@marsmain /usr/portage/app-shells/bash $ date
Sat Aug  8 02:42:24 MSD 2009
mars@marsmain /usr/portage/app-shells/bash $ lsmod
Module                  Size  Used by
rndis_wlan             23424  0
rndis_host              8696  1 rndis_wlan
cdc_ether               5672  1 rndis_host
usbnet                 18688  3 rndis_wlan,rndis_host,cdc_ether
parport_pc             38424  0
fglrx                2388128  20
parport                39648  1 parport_pc
iTCO_wdt               12272  0
i2c_i801                9380  0
mars@marsmain /usr/portage/app-shells/bash $
```


Tablet and Phone Operating Systems

- **System-on-chip (SoC):** An operating system that comes preinstalled on a chip on a portable device such as a smartphone.
- Popular SoC operating systems:
  - iOS: for iPad, iPhone
  - Android: for a variety of tablets and phones
- Downloadable applications (apps) from an App store, for example:
  - Apple App Store
  - Google Play Store

iOS on the iPhone and iPad

- The Apple-created operating system for Apple tablets and phones.
- The current stable version, iOS 14, was released to the public on September 16, 2020.

Android

- Android, a popular OS for smartphones and tablets, is based on Linux Kernel.
  - Developed by Google
- Current versions include:
  - Android 8 Oreo
  - Android 9 Pie
  - Android 10
  - Android 11 (released on Sep, 2020)

Advantage of Linux Operating System

1. Open Source

As it is open-source, its source code is easily available.
Anyone having programming knowledge can customize the operating system.
One can contribute, modify, distribute, and enhance the code for any purpose.

2. Security

The Linux security feature is the main reason that it is the most favourable option for developers.
It is not completely safe, but it is less vulnerable than others.
Each application needs to authorize by the admin user.
Linux systems do not require any antivirus program.

3. Free

Certainly, the biggest advantage of the Linux system is that it is free to use.
We can easily download it, and there is no need to buy the license for it.
It is distributed under GPL (General Public License).
Comparatively, we have to pay a huge amount for the license of the other OS

Advantage of Linux Operating System

4. Lightweight

The requirements for running Linux are much less than other operating system
In Linux, the memory footprint and disk space are also lower.
Generally, most of the Linux distributions required as little as 128MB of RAM around the same amount for disk space.

5. Stability

Linux is more stable than other operating systems.
Linux does not require to reboot the system to maintain performance levels.
It rarely hangs up or slow down. It has big up-times.

Advantage of Linux Operating System

6. Performance

Linux system provides high performance over different networks.
It is capable of handling a large number of users simultaneously.

7. Flexibility

Linux operating system is very flexible.
It can be used for desktop applications, embedded systems, and server applications too.
It also provides various restriction options for specific computers.
We can install only necessary components for a system.

8. Software Updates

In Linux, the software updates are in user control.
We can select the required updates.
There a large number of system updates are available.
These updates are much faster than other operating systems.
So, the system updates can be installed easily without facing any issue.

Advantage of Linux Operating System

9. Distributions/ Distros

There are many Linux distributions available in the market.

It provides various options and flavors of Linux to the users.

We can choose any distros according to our needs.

Some popular distros are **Ubuntu, Fedora, Debian, Linux Mint, Arch Linux,**

For the beginners, Ubuntu and Linux Mint would be useful.

Debian and Fedora would be good choices for proficient programmers.

10. Live CD/USB

Almost all Linux distributions have **a Live CD/USB** option.

It allows us to try or run the Linux operating system without installing it.

11. Graphical User Interface

Linux is a command-line based OS but it provides an interactive user interface like Windows.

Advantage of Linux Operating System

12. Suitable for programmers

It supports almost all of the most used programming languages such as C/C++, Java, Python, Ruby, and more.

Further, it offers a vast range of useful applications for development.

The programmers prefer the Linux terminal over the Windows command line.

The package manager on Linux system helps programmers to understand how things are done.

Bash scripting is also a functional feature for the programmers.

It also provides support for SSH, which helps in managing the servers quickly.

13. Community Support

Linux provides large community support.

We can find support from various sources.

There are many forums available on the web to assist users.

Further, developers from the various open source communities are ready to help us.

Advantage of Linux Operating System

14. Privacy

Linux always takes care of user privacy as it never takes much private data from the user. Comparatively, other operating systems ask for the user's private data.

15. Networking

Linux facilitates with powerful support for networking. The client-server systems can be easily set to a Linux system. It provides various command-line tools such as ssh, ip, mail, telnet, and more for connectivity with the other systems and servers. Tasks such as network backup are much faster than others.

16. Compatibility

Linux is compatible with a large number of file formats as it supports almost all file formats.

17. Installation

Linux installation process takes less time than other operating systems such as Windows. Further, its installation process is much easy as it requires less user input. It does not require much more system configuration even it can be easily installed on old machines having less configuration.

Advantage of Linux Operating System

18. Multiple Desktop Support

Linux system provides multiple desktop environment support for its enhanced use. The desktop environment option can be selected during installation. We can select any desktop environment such as **GNOME (GNU Network Object Model Environment)** or **KDE (K Desktop Environment)** as both have their specific environment.

19. Multitasking

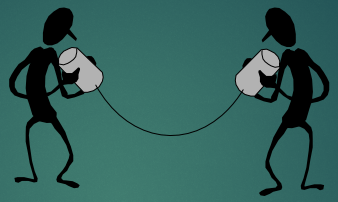
It is a multitasking operating system as it can run multiple tasks simultaneously without affecting the system speed.

20. Heavily Documented for beginners

There are many command-line options that provide documentation on commands, libraries, standards such as manual pages and info pages. Also, there are plenty of documents available on the internet in different formats, such as Linux tutorials, Linux documentation project, Serverfault, and more. To help the beginners, several communities are available such as **Ask Ubuntu**, Reddit, and **StackOverflow.**

Communication Protocol

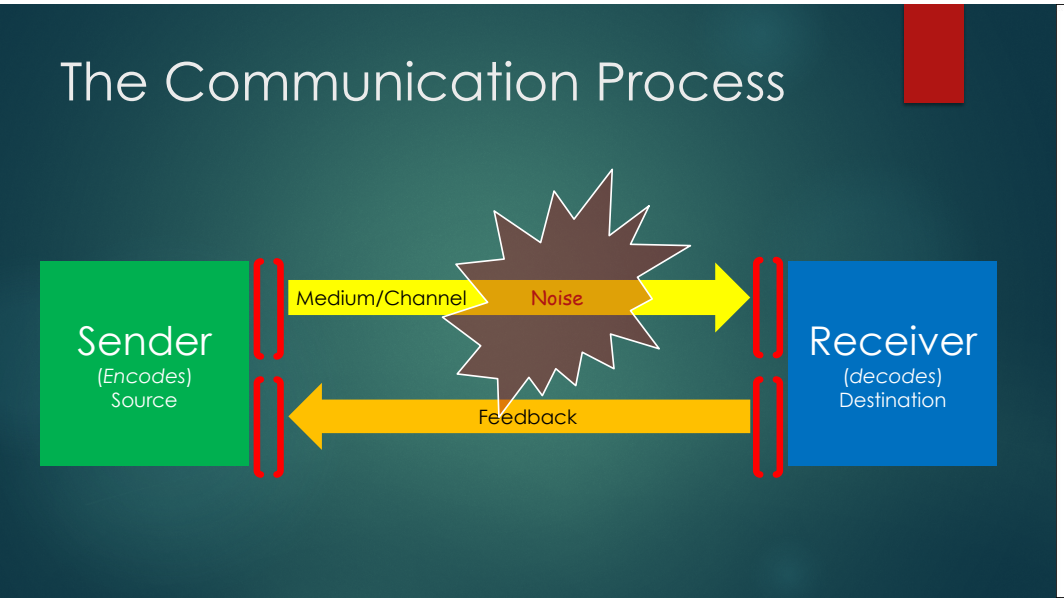
SHIVRAJ DHARNE



Communication…

What is Communication?

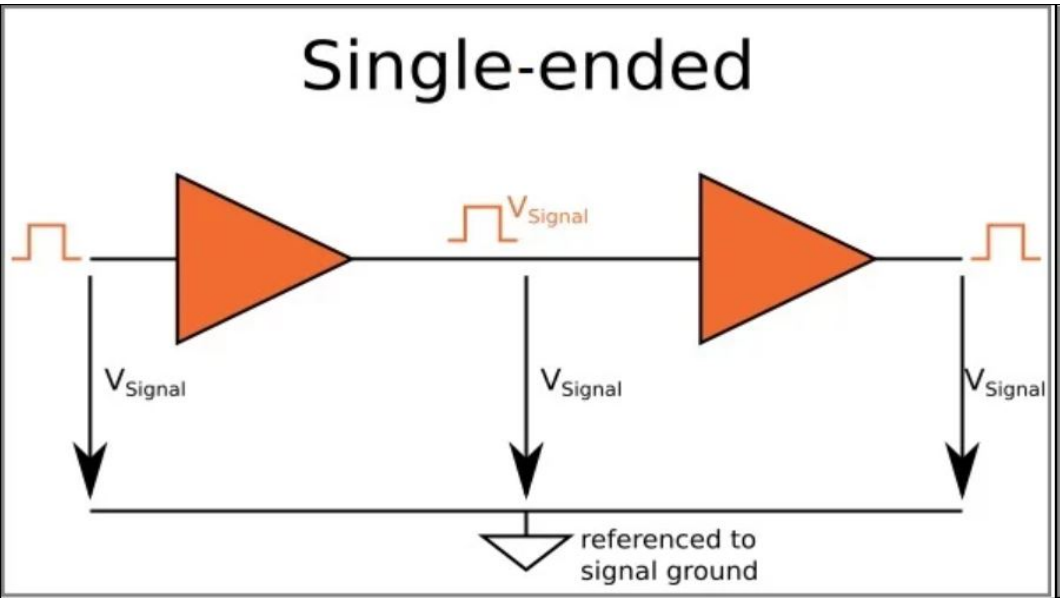
- Dictionary definition
  - The act of transmitting
  - Verbal or written message
  - a process by which information is exchanged between individuals through a common system of symbols, signs, or behavior
- Key words:
  - process
  - information
  - exchanged
  - common system

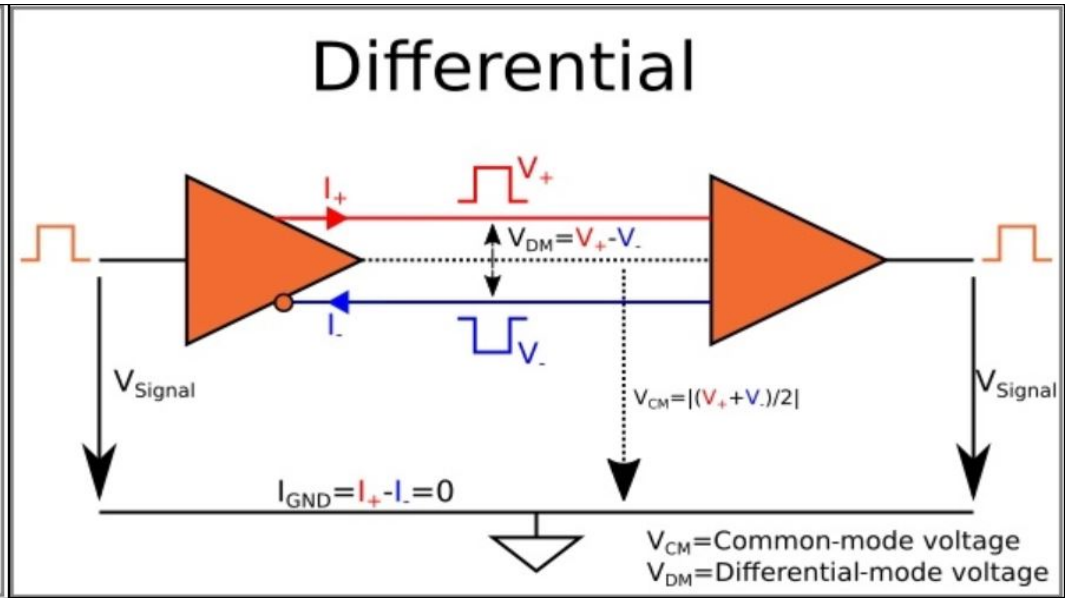


What are Communication Protocols?

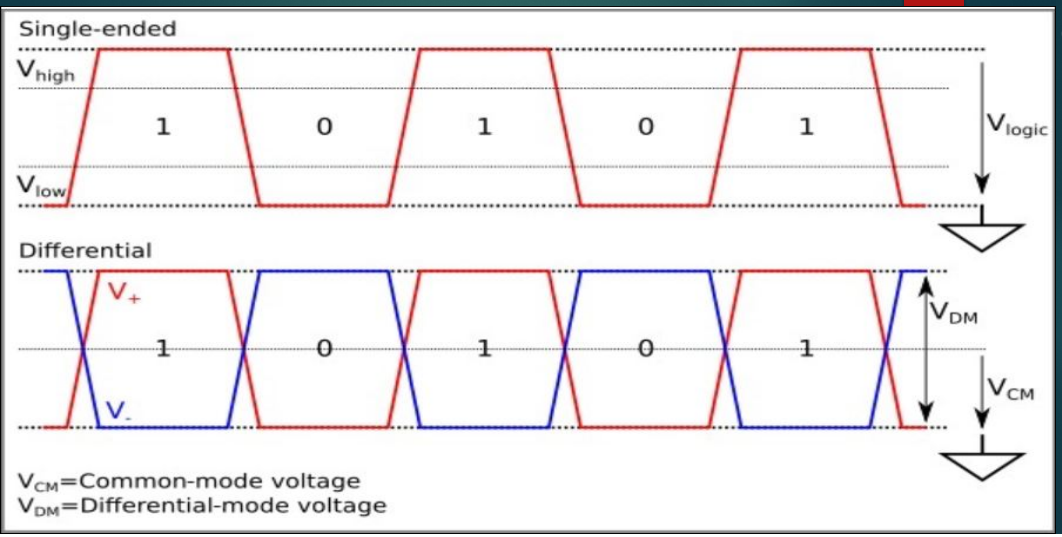
The proper descriptions of digital message formats as well as rules are known communication protocols. The main function of these protocols is to exchange messages from one computer system to another. These are significant in telecommunications systems as they consistently send and receive messages. These protocols cover error detection & correction, signaling, and authentication. They can also explain the semantics, syntax & brings analog & digital communications together.

Communication Protocols





Single ended and Differential signal



Advantages and Disadvantages of differential signaling

Advantages :

- No return current
- Resistance to Incoming EMI and Crosstalk
- Reduction of Outgoing EMI and Crosstalk
- Lower-Voltage Operation
- High or Low State and Precise Timing
- higher data rates

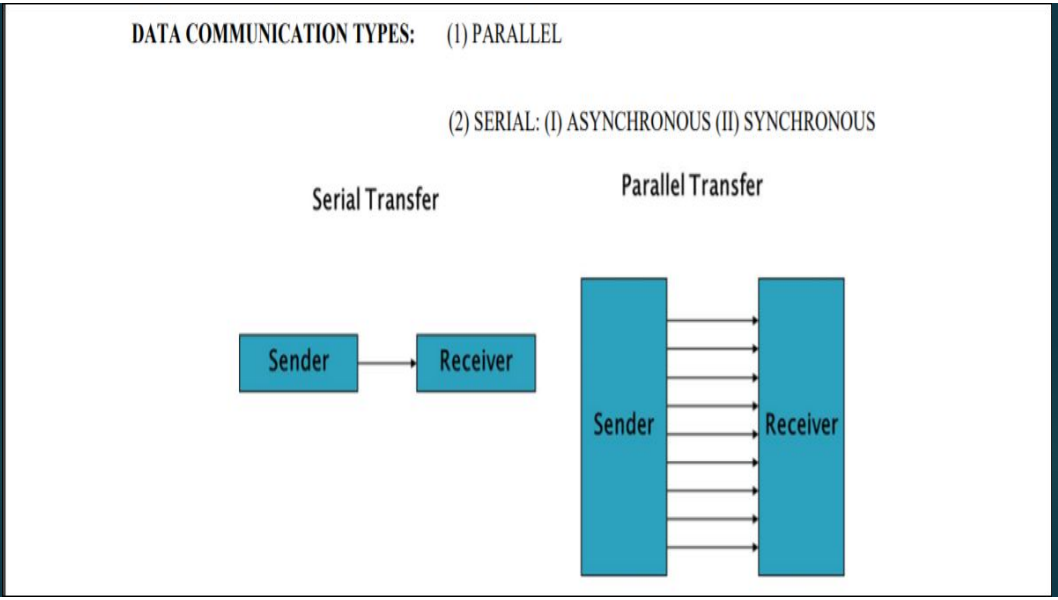
Disadvantages :

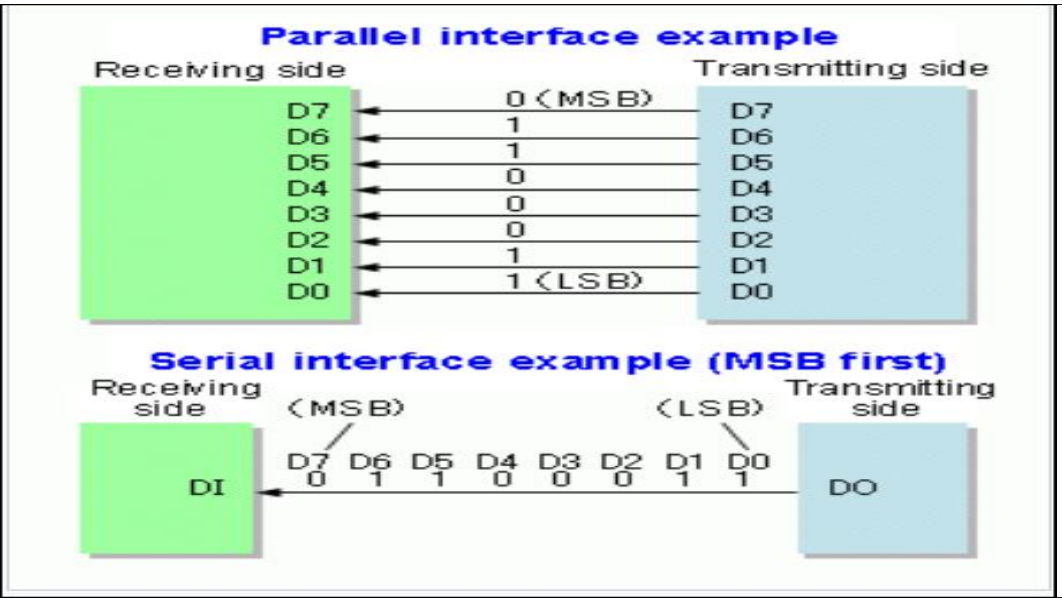
- conductor count increases
- system will need specialized transmitters and receivers instead of standard digital ICs.

Serial Communication Protocols : I2C, SPI, CAN and USB

Single ended signaling – I2C and SPI

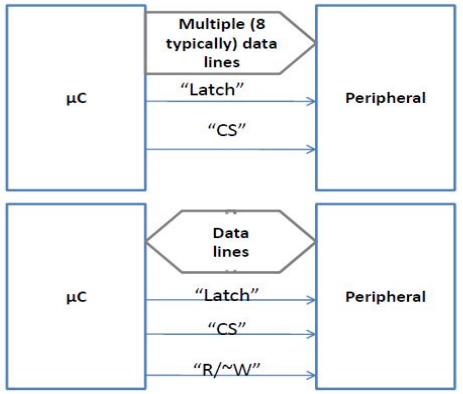
Differential signaling – CAN and USB





Communications

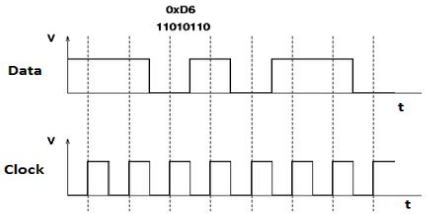
- The simplest is parallel
  - One way
    - There may be mechanism for peripheral to get attention of µC (i.e., interrupt, or poll)
  - Two way

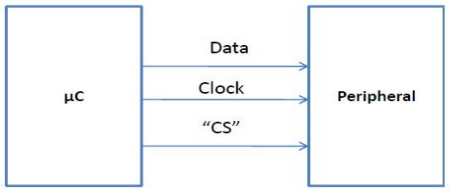


- This is resource expensive (pins, real-estate…) in terms of hardware, but easy to implement

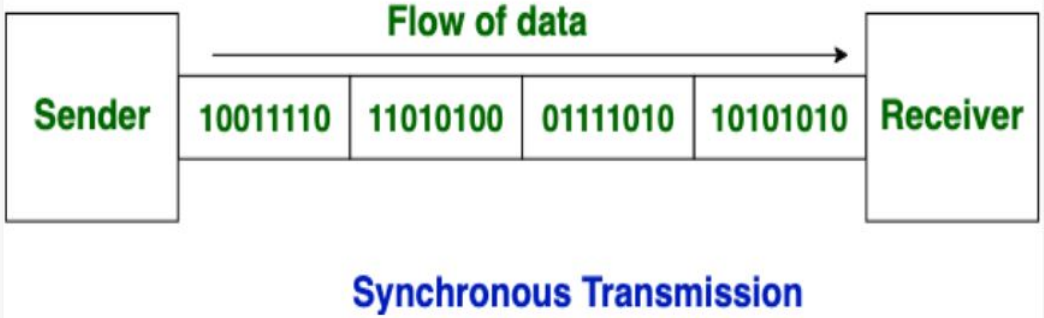
Serial Communications

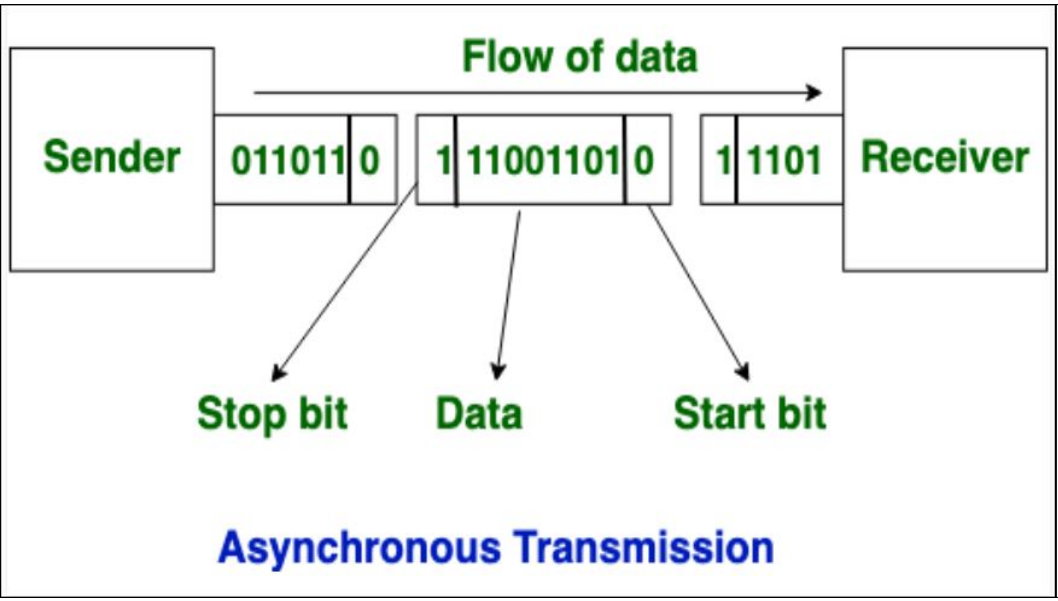
- Many fewer lines are required to transmit data. This is requires fewer pins, but adds complexity.





- Synchronous communications requires clock. Whoever controls the clock controls communication speed.
- Asynchronous has no clock, but speed must be agreed upon beforehand (baud rate).

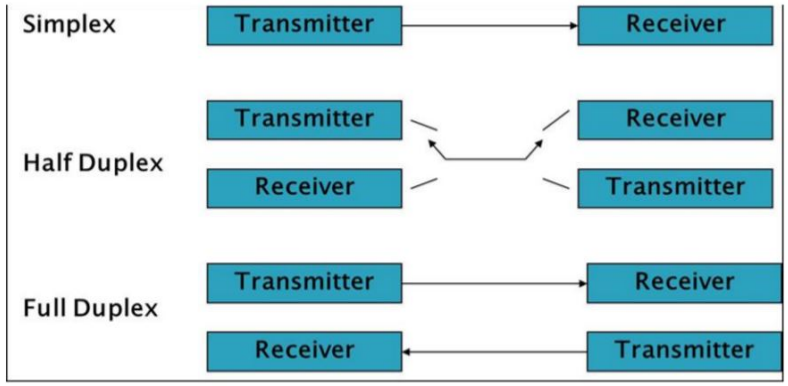




Synchronous:

- Transfers a block of data (characters) at a time.
- Requires clock signal

Example: SPI (serial peripheral interface), I2C (inter integrated circuit).



Data Transmission: In data transmission if the data can be transmitted and received, it is a duplex transmission.

Simplex: Data is transmitted in only one direction i.e. from TX to RX only one TX and one RX only

Half duplex: Data is transmitted in two directions but only one way at a time i.e. two TX's, two RX's and one line

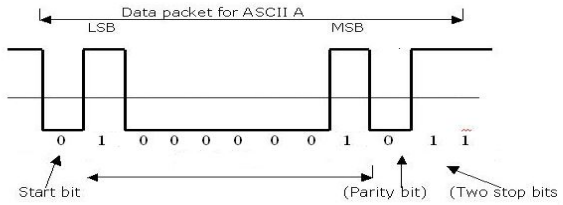
Full duplex: Data is transmitted both ways at the same time i.e. two TX's, two RX's and two lines

A **Protocol** is a set of rules agreed by both the sender and receiver on

- How the data is packed
- How many bits constitute a character
- When the data begins and ends

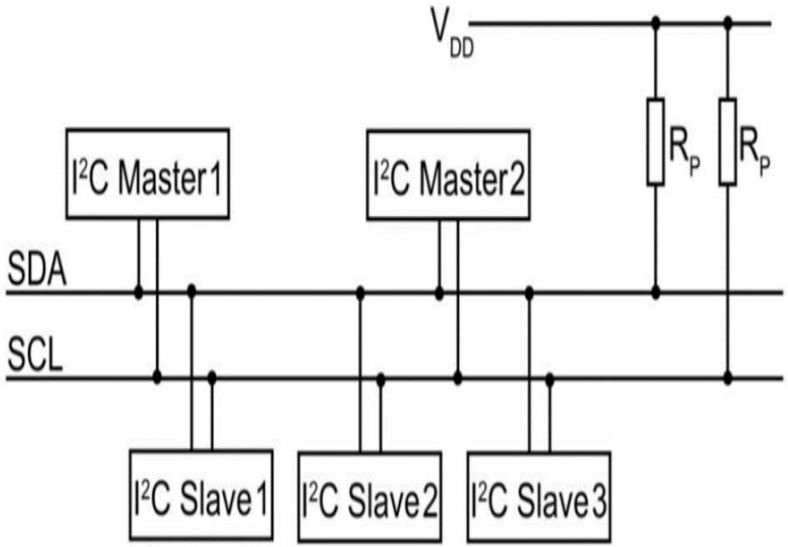
Asynchronous Serial (RS-232)

- Commonly used for one-to-one communication.
- There are many variants, the simplest uses just two lines, TX (transmit) and RX (receive).
- Transmission process (9600 baud, 1 bit=1/9600=0.104 mS)
  - Transmit idles high (when no communication).
  - It goes low for 1 bit (0.104 mS)
  - It sends out data, LSB first (7 or 8 bits)
  - There may be a parity bit (even or odd – error detection)
  - There may be a stop bit (or two)



Clock Synchronisation

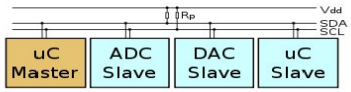
- All masters generate their own clocks on the SCL line to transmit messages on the I2C bus.
- Data is only valid during the high period of the clock.
- Clock synchronization is performed by connecting the I2C interface to the SCL line where the switch goes from high to low. Once the device's clock goes low, it keeps the SCL line in this state until it reaches the high level of the clock.
- If another clock is still in a low period, the low-to-high switch does not change the state of the SCL line. The SCL line is always held low by the device with the longest low period. At this time, the device with a short and low period will enter a high and waiting state.
- When all relevant devices have completed their low period, the clock line goes high.
- After that, there is no difference in the state of the device clock and the SCL line, and all devices begin to count their high period. The device that first completes the high period will pull the SCL line low again.
- The low period of the synchronous SCL clock is determined by the device with the longest low clock period, while the high period is determined by the device with the shortest high clock period.



I2C or I²C

(Inter-Integrated Circuit – Philips)

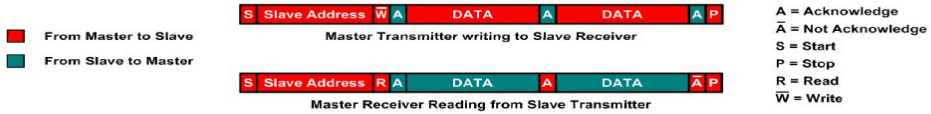
- As with SPI a master-slave system.



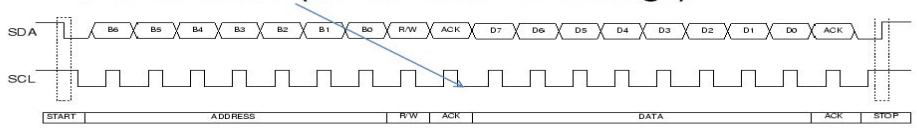
- Also called a 2-wire bus. It Has only clock and data, with pull-up resistors (Rp in diagram).
- Lines can be pulled low by any device, and are high when all devices release them.
- There are no "slave-select" lines – instead the devices have "addresses" that are sent as part of the transmission protocol.
- Four max speeds (100 kbS (*standard*), 400 kbS (*fast*), 1 MbS (*fast plus*), and 3.4 MbS (*high-speed*)

Other Features

- You can transfer multiple bytes in a row

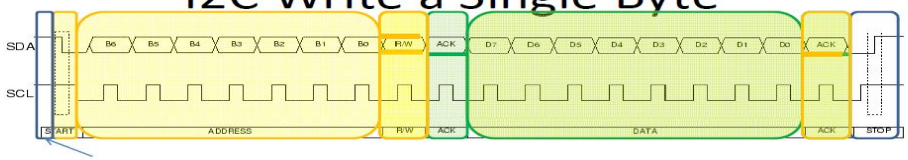


- At end of transfer, slave can hold SCL low to slow transfer down (called "clock-stretching")



- Any device that malfunctions can disable bus.

I2C Write a Single Byte

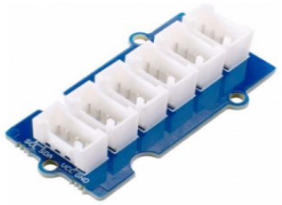


1. All: allow SDA, SCL start high
2. Master: SDA low to signal start
3. Master: Send out SCL, and 7 bit address followed by 0 (~W) on SDA
4. Slave: Pull SDA low to signify ACKnowledge
5. Master: Send out 8 data bits on SDA
6. Slave: Ack
7. All: allow SDA to go high when SCL is high (stop)

- **For "Read",**
  3. Master: Address following by 1 (R) on SDA
  5. Slave: Send out 8 data bits on SDA
  6. Master: Ack

Examples of I2C in Microcontrollers

Grove – I2C Hub (6 Port)



- I2C is a very popular communication protocol. In the Grove system, I2C is used by 80+ sensors for communication, 19 of which are related to environmental monitoring.
- Today more and more MCUs uses 3.3V communication levels, but the traditional ArduinoUno still uses 5V, which leads to many modules, especially sensor modules, needing to be levelled when using them.
- We actually worked on this area, and now most of the Grove sensor modules have a level shifting function, and users do not need to consider the use of 3.3V or 5V MCU when using it. This is in line with Grove's motto; plugin, and use it, it's that simple. For a more detailed sensor review compatibility, you can view our Grove Selection Guide.

How does it work?

- It has 2 Lines which are SCL (serial clock line) and SDA (serial data line acceptance port)
- CL is the clock line for synchronizing transmission. SDA is the data line through which bits of data are sent or received.
- The master device initiates the bus transfer of data and generates a clock to open the transferred device and any addressed device is considered a slave device.
- The relationship between master and slave devices, transmitting and receiving on the bus is not constant. It depends on the direction of data transfer at the time.
- If the master wants to send data to the slave, the master must first address the slave before sending any data.
- The master will then terminate the data transfer. If the master wants to receive data from the slave, the master must again address the slave first.
- The host then receives the data sent by the slave and finally, the receiver terminates the receiving process. The host is also responsible for generating the timing clock and terminating the data transfer.
- It is also necessary to connect the power supply through a pull-up resistor. When the bus is idle, both lines operate on a high power level.
- The capacitance in the line will affect the bus transmission speed. As the current power on the bus is small, when the capacitance is too large, it may cause transmission errors. Thus, its load capacity must be 400pF, so the allowable length of the bus and the number of connected devices can be estimated.

I2C Working Protocol

Data Transmission Method

- The master sends the transmitting signal to every connected slave by switching the SDA line from a high voltage level to a low voltage level and SCL line from high to low after switching the SDA line.
- The master sends each slave the 7 or 10-bit address of the slave and a read/write bit to the slave it wants to communicate with.
- The slave will then compare the address with its own. If the address matches, the slave returns an ACK bit which switches the SDA line low for one bit. If the address does not match its address, the slave leaves the SDA line high
- The master will then send or receive the data frame. After each data frame has been transferred, the receiving device returns another ACK bit to the sender to acknowledge successful transmission.
- To stop the data transmission, the master sends a stop signal to the slave by switching SCL high before switching SDA high

Transmission Modes

Quick Mode:

- Fast mode devices can receive and transmit at 400kbit/s. They have to be able to synchronize with a 400kbit/s transmission and extend the low period of the SCL signal to slow down the transmission.
- Fast mode devices are backwards compatible and can communicate with standard mode devices from 0 to 100 kbit/s I2C bus systems. However, as standard mode devices are not upward compatible, they cannot operate in a fast I2C bus system. The fast mode I2C bus specification has the following characteristics compared to the standard mode:
  - The maximum bit rate is increased to 400 kbit/s;
  - Adjusted the timing of the serial data (SDA) and serial clock (SCL) signals.
  - Has the function of suppressing glitch and the SDA and SCL inputs have Schmitt triggers.
  - The output buffer has a slope control function for the falling edges of the SDA and SCL signals
  - Once the power supply of the fast mode device is turned off, the I/O pins of SDA and SCL must be left idle and cannot block the bus.
  - The external pull-up device connected to the bus must be tuned to accommodate the shortest maximum allowable rise time of the fast mode I2C bus. For buses with a maximum load of 200pF, the pull-up device of each bus can be a resistor. For a bus with a load between 200pF and 400pF, the pull-up device can be a current source (maximum 3mA) or a switched resistor circuit.

High-Speed Mode:

- Hs mode devices can transmit information at bit rates up to 3.4 Mbit/s and remain fully backwards compatible with fast mode or standard mode (F/S mode) devices that can communicate bi-directionally in a speed mixed bus system.
- The Hs mode transmission has the same serial bus principle and data format as the F/S mode system except for arbitration and clock synchronization which is not performed.
- The I2C bus specification in high-speed mode is as follows:
  - In high speed (Hs) mode, the master device has an open-drain output buffer for the high-speed (SDAH) signal and an open-drain pull-down and current source pull-up circuit at the high-speed serial clock (SCLH) output. This shortens the rise time of the SCLH signal and at any time, only one host current source is active;
  - In the Hs mode of a multi-master system, arbitration and clock synchronization are not performed in order to speed up the bit processing capability. The arbitration process normally ends after the host code is transmitted in the F/S mode.
  - The Hs mode master device generates a high and low serial clock signal with a ratio of 1:2 which removes the timing requirements for setup and hold time.
  - The Hs mode device can have a built-in bridge. During Hs mode transmission, the SDAH and SCLH lines of the Hs mode device are separated from the SDA and SCL lines which reduces the capacitive loading of the SDAH and SCLH lines and make rise and fall faster.
  - The difference between Hs mode slave devices and F/S slave devices is the speed at which they operate.
  - The Hs mode device can suppress glitches, and the SDAH and SCLH outputs also have a Schmitt trigger:

Advantages of using I2C

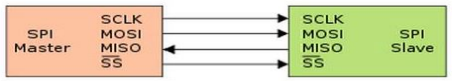
- Has a low pin/signal count even with numerous devices on the bus
- Flexible, as it supports multi-master and multi slave communication.
- Simple as it only uses 2 bidirectional wires to establish communication among multiple devices.
- Adaptable as it can adapt to the needs of various slave devices.
- Support multiple masters.

Disadvantages of using I2C

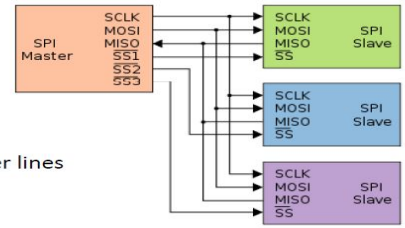
- Slower speed as it requires pull-up resistors rather than push-pull resistors used by SPI. It also has an open-drain design = limited speed.
- Requires more space as the resistors consume valuable PCB real estate.
- May become complex as the number of devices increases.

SPI

(Serial Peripheral Interface - Motorola)

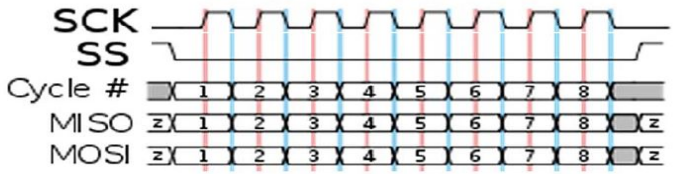


- Two types of devices, masters and slaves.
- We'll consider only one master, but multiple slaves.
- Signals
  - SCLK: Serial CLocK, set by Master
  - MOSI: Master Out, Slave In
  - MISO: Master In, Slave Out
  - ~SS: Slave Select
    - Each slave gets its own slave select (other lines are shared)
    - Pulling line low selects slave



SPI and the clock
(intro)

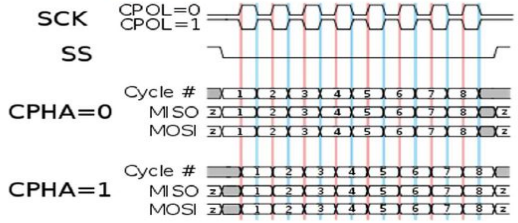
- Pull slave select line low to select device.
- First bit of data gets put on MISO and MOSI (so a byte goes both ways)
- Data gets shifted out (typically 8 bits, but not necessarily)
  - The data gets put on bus on falling edge of clock.
  - The data gets read on the rising edge of clock.



SPI and the clock
(the hard truth)

Unfortunately, clock can be set many ways as determined by clock polarity and phase.

- CPOL=0: Base value of the clock is 0
  - CPHA=0: Data read on rising edge, put on bus on falling edge of SCLK. (i.e., clock is low). (Case from previous slide)
  - CPHA=1: Data read on falling edge, put on bus on rising edge (i.e., clock is high).
- CPOL=1: Base value of the clock is 1
  - CPHA=0: Data read on falling edge, put on bus on rising edge (i.e., clock is high).
  - CPHA=1: Data read on rising edge, put on bus on falling edge (i.e., clock is low).



SPI Interface

What is SPI?

- Stands for **Serial Peripheral Interface** (SPI)
- It is similar to I2C and it is a different form of serial-communications protocol specially designed for microcontrollers to connect.
- Operates at full-duplex where data can be sent and received simultaneously.
- Operate at faster data transmission rates = 8Mbits or more
- It is typically faster than I2C due to the simple protocol. Even if data/clock lines are shared between devices, each device will require a unique address wire.
- Used in places where speed is important. (eg. SD cards, display modules or when info updates and changes quickly like thermometers)

SPI Working Protocol

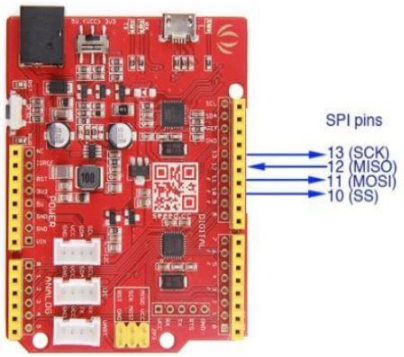
- The SPI communicates via 4 ports which are:
  - MOSI – Master Data Output, Slave Data Input
  - MISO – master data input, slave data output
  - SCLK – clock signal, generated by the master device, up to fPCLK/2, slave mode frequency up to fCPU/2
  - NSS – Slave enabled signal, controlled by the master device, some ICs will be labelled as CS (Chip select)
- In a multi-slave system, each slave requires a separate enable signal, which is slightly more complicated on hardware than the I2C system.
- The SPI interface is actually two simple shift registers in the internal hardware. The transmitted data is 8 bits. It is transmitted bit by bit under the slave enable signal and shift pulse generated by the master device. The high bit is in the front and the low bit is in the back.
- The SPI interface is synchronous serial data transmission between the CPU and the peripheral low-speed device. Under the shift pulse of the master device, the data is transmitted bit by bit. The high bit is in the front and the low bit is in the back. It is full-duplex communication, and the data transmission speed is overall faster than the I2C bus and can reach speeds of a few Mbps.

SPI

How does it work?

- Communicate with 2 ways:
  1. Selecting each device with a Chip Select line. A separate Chip Select line is required for each device. This is the most common way RPi's currently use SPI.
  2. Daisy chaining where each device is connected to the other through its data out to the data in line of the next.
- There is no limit to the number of SPI device that can be connected. However, there are practical limits due to the number of hardware select lines available on the main device with the chip select method or the complexity of passing data through devices in the daisy-chaining method.
- In point-to-point communication, the SPI interface does not require addressing operations and is full-duplex communication, which is simple and efficient.

SPI Seeeduino V4.2



- SPI serial communication can be used with Arduino for communication between two Arduinos where one Arduino will act as master and another one will act as a slave.
- Used to communicate over short distances at high speed.

Advantages of using SPI

- The protocol is simple as there is no complicated slave addressing system like I2C.
- It is the fastest protocol compared to UART and I2C.
- No start and stop bits unlike UART which means data can be transmitted continuously without interruption
- Separate MISO and MOSI lines which means data can be transmitted and received at the same time

Disadvantages of using SPI

- More Pin ports are occupied, the practical limit to a number of devices.
- There is no flow control specified, and no acknowledgement mechanism confirms whether data is received unlike I2C
- Uses four lines – MOSI, MISO, NCLK, NSS
- No form of error check unlike in UART (using parity bit)
- Only 1 master

Comparison of I2C and SPI

Protocol	I2C	SPI
Complexity	Easy to chain multiple devices	Complex as device increases
Speed	Faster than UART	Fastest
Number of devices	Up to 127, but gets complex	Many, but gets complex
Number of wires	2	4
Duplex	Half Duplex	Full Duplex
No. of masters and slaves	Multiple slaves and masters	1 master, multiple slaves

Which is best , I2C or SPI ?

Unfortunately, there is no "best" communication peripheral. Each communication peripheral has its own advantages and disadvantages.

Thus, a user should pick a communication peripheral that suits your project the best. For example, you want the fastest communication peripheral, SPI would be the ideal pick. On another hand, if a user wants to connect many devices without it being too complex, I2C will be the ideal pick as it can connect up to 127 devices and it is simple to manage.

Thank you !!!!

NTU CSIE Open Source System Software
2016.03.29

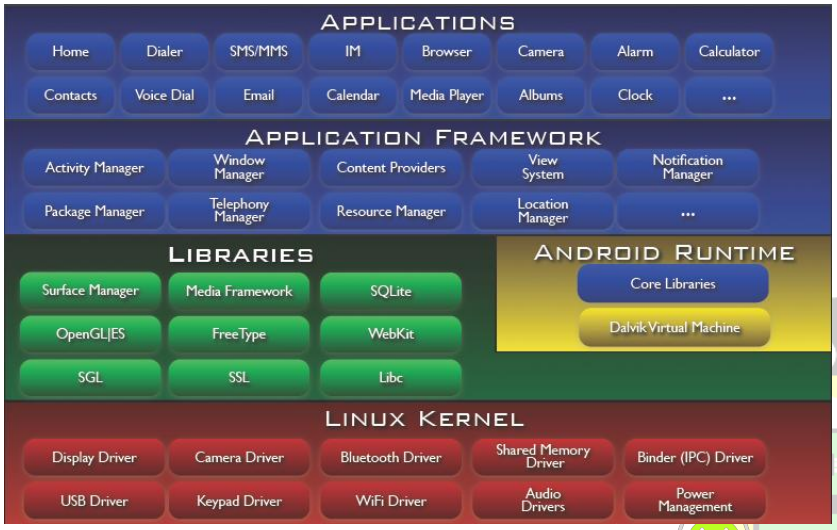
An Introduction to the Android Framework

-- a core architecture view from apps to the kernel --

William W.-Y. Liang (梁文耀), Ph. D.
http://www.ntut.edu.tw/~wyliang
for 台大資工系開源系統軟體課程
hosted by Prof. Shih-Hao Hung

Note: **The Copyrights of the referenced materials and figures go to their original authors.** As a result, the slides are for non-commercial reference only.
For the contents created in this document, the Copyright belongs to William W.-Y. Liang. © 2005-2016 All Rights Reserved.

Android Architecture



Source from http://developer.android.com/

Think: user-space Android?

© 2016 William W.-Y. Liang, All Rights Reserved.

Advantage of the Android Operating System

- High application portability
- Consistent user experience
- Component-based design
- Suitable for resource-limited handheld devices
- Consider the performance issue
- Acceptable effort to port to different hardware

Think: Architecture design and considerations
- Multi-layer design
- Programming languages
- License issues

© 2016 William W.-Y. Liang, All Rights Reserved.

Android/Linux

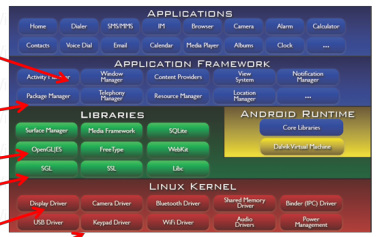
- Android differs from the typical GNU/Linux in that it adopts only the Linux kernel, not everything.
- The first process 'init' transfers to Android's own middleware and application environment.
- BSD libc is used instead of glibc or uClibc.
- As a result, it is called **Android/Linux**, not GNU/Linux.

Think: how different they are?

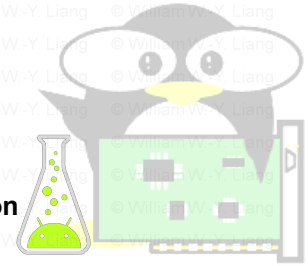
© 2016 William W.-Y. Liang, All Rights Reserved.

Android/Linux System Integration

- Android Application
- Android Application Framework
- Android Java/C/C++ Interface
- Android/Linux Native Code
- Linux System Calls
- Linux Device Driver
- Hardware and Control

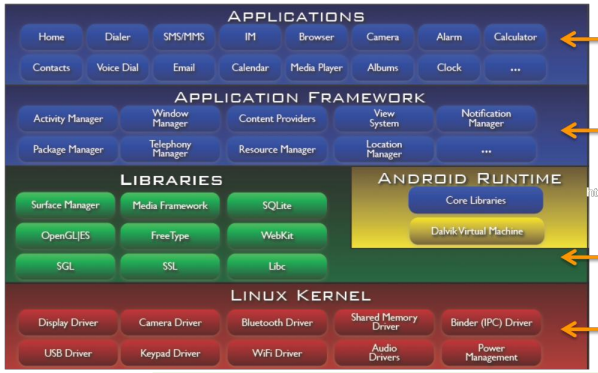


Think: typical efforts for Android system integration



© 2016 William W.-Y. Liang, All Rights Reserved.

Interface between Layers

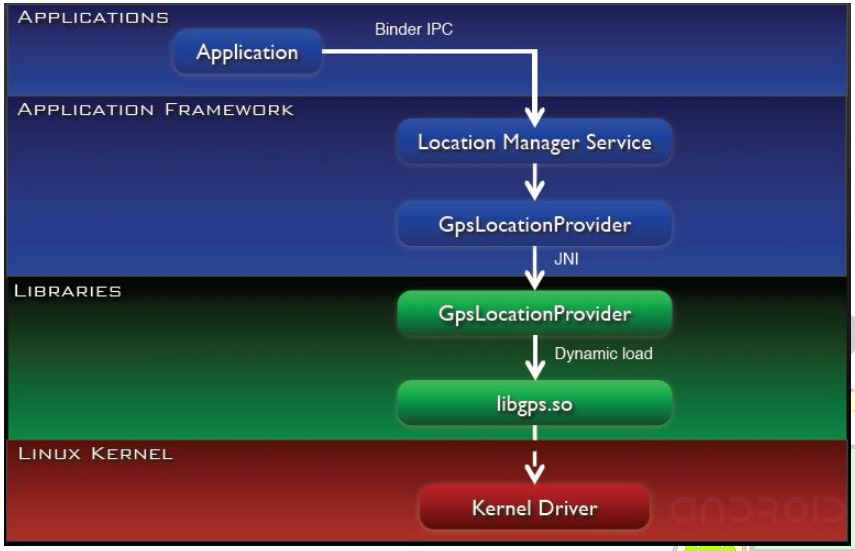


Develop Apps
Using Framework Components
Add Framework
Over the JNI interface
Add Native Control Code
Over the system call
Add Driver

Android System Integration Procedure
1. Add driver
2. Add user-space JNI C/C++ native library
3. Add the Java Middleware
   - Service
   - Framework component
4. Develop the application

© 2016 William W.-Y. Liang, All Rights Reserved.

Example

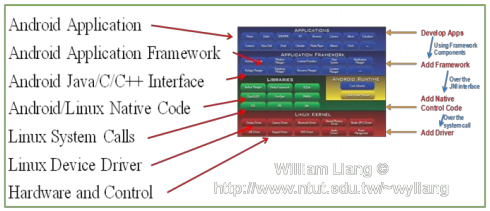


Slides from "Android Anatomy and Physiology," Patrick Brady ©

© 2016 William W.-Y. Liang, All Rights Reserved.

Important Subjects for the Core Architecture

- Android task and process
- Activity and service
- The binder IPC
- Thread operations
- Concurrency control and thread safety
- Android framework components
- User-space device control
- Android Java native interface for C/C++
- Hardware abstraction layer
- Linux device driver and device manipulation



Think: Android boot flow, system services, and subsystems

© 2016 William W.-Y. Liang, All Rights Reserved.

Activity -- the Primary Framework Component

- Displays a user interface component and responds to system/user.
- When an application has a user interface, it contains one or more "Activities".
- Each "Activity" can be invoked by the same or other apps.
- The new Java class must extend the framework "Activity" class.
- Created "Activity" must be defined into the application's Manifest.xml.
- An existing "Activity" can be replaced with a new one that fulfill the same contract (intent).

Source from "Deep inside Android," Gilles Printemps, Esmertec ©

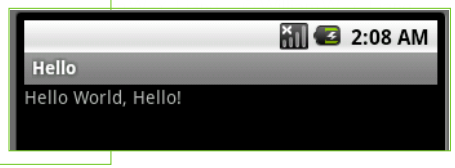
Think: activity stack and process

© 2016 William W.-Y. Liang, All Rights Reserved.

UI in XML

main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />

</LinearLayout>
```




```
package hello.test;

import android.app.Activity;

public class Hello extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        /* pass resource id to attach to screen */
        setContentView(R.layout.main);
    }
}
```

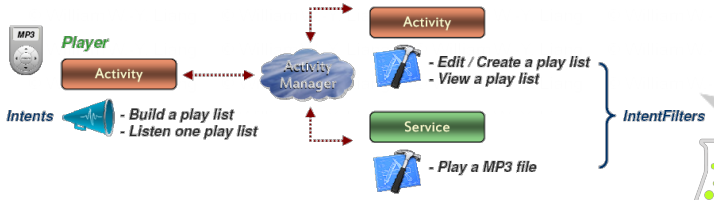
Think: how UI XML gets deployed

Source from http://developer.android.com/

© 2016 William W.-Y. Liang, All Rights Reserved.

Intents and Intent Filters

- Provide a **late runtime (asynchronous) binding** between the code in different applications (for activities, services, and broadcast receivers)
- **Intents**: Simple message objects that represent an intention to do something
- **Intent Filters**: A declaration of capacity and interest in offering assistance to those in need

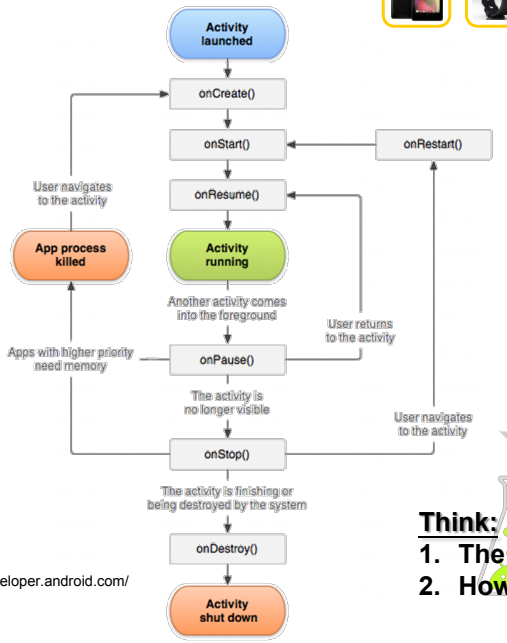


Slide from "Deep inside Android," Gilles Printemps, Esmertec ©

Think: The design philosophy

© 2016 William W.-Y. Liang, All Rights Reserved.

Life Cycle of an Activity



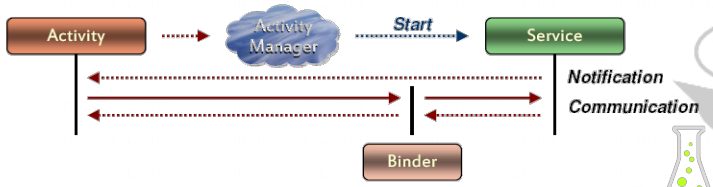
Source from http://developer.android.com/

Think:
1. **The design philosophy**
2. **How to deal with data loss?**

© 2016 William W.-Y. Liang, All Rights Reserved.

The Service Framework Component

- Similar to activities, but without UI
- For long-running background tasks
- Extended from the Service class
- It's possible to connect to (bind to) an ongoing service.
- The connected service can be communicated through the Binder interface.

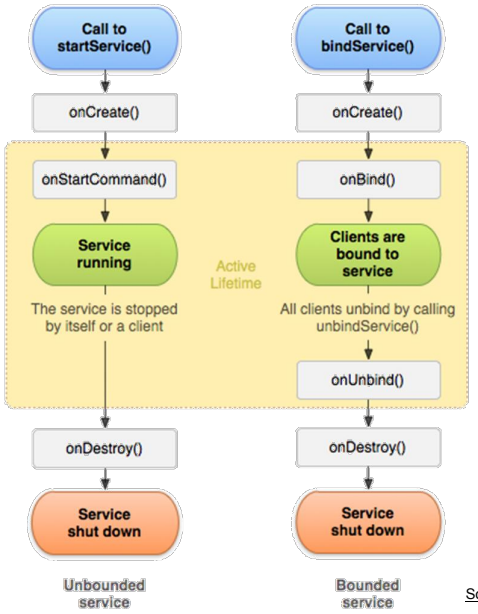


Source from "Deep inside Android," Gilles Printemps, Esmertec ©

Think: service vs. activity

© 2016 William W.-Y. Liang, All Rights Reserved.

Service Lifecycle

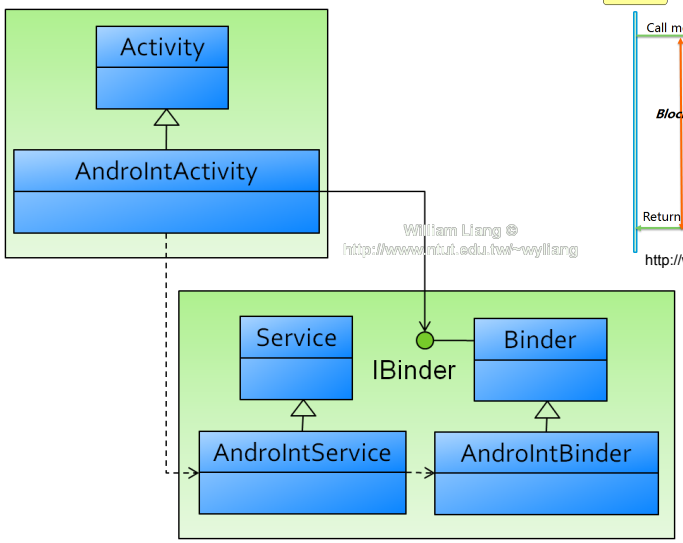


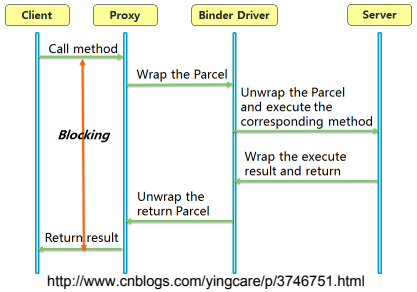
Think: use cases

Source from http://developer.android.com/

© 2016 William W.-Y. Liang, All Rights Reserved.

Example





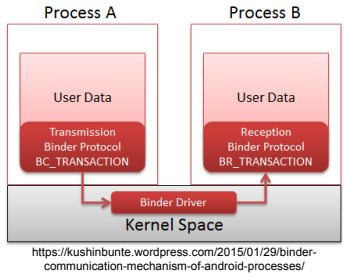
http://www.cnblogs.com/yingcare/p/3746751.html

Think: binder vs. service

© 2016 William W.-Y. Liang, All Rights Reserved.

Binder IPC

- Problems to solve
  - Applications and Services may run in separate processes but must communicate and share data.
  - IPC can introduce significant processing overhead and security holes.
- Solution
  - Driver to facilitate inter-process communication (IPC)
  - High performance through shared memory
  - Reference counting, and mapping of object references across processes
  - Synchronous calls between processes



https://kushinbunte.wordpress.com/2015/01/29/binder-communication-mechanism-of-android-processes/

Think: IPC overheads

Source from "Android Anatomy and Physiology," Patrick Brady ©

© 2016 William W.-Y. Liang, All Rights Reserved.

AIDL – a Wrapper of the Binder IPC

- AIDL: Android Interface Definition Language
  - Used to generate code in a remote procedure call (RPC) form that marshals the Binder IPC communication details.

```
private ServiceConnection mConnection = new ServiceConnection() {
    public void onServiceConnected(ComponentName className, IBinder ibinder) {
        mAddService = IAddService.Stub.asInterface((IBinder) ibinder);
    }

    public void onServiceDiscon
        mAddService = null;
    }
};

private OnClickListener mAddListener = new OnClickListener() {
    public void onClick(View view) {
        int a = Integer.parseInt(mEditText1.getText().toString());
        int b = Integer.parseInt(mEditText2.getText().toString());
        int result = 0;

        try {
            result = mAddService.add(a, b);
        } catch (RemoteException e) {
            e.printStackTrace();
        }

        mButton.setText(String.valueOf("A+B=" + result));
    }
};
```

```
interface IAddService {
    int add(in int a, in int b);
}
```

```
lic class AndroIntService extends Service {
private IBinder mBinder = null;

private static final String LOG_TAG = "AdroIntService";

@Override
public void onCreate() {
    mBinder = new AndroIntBinder();
    Log.i(LOG_TAG, "Service created, binder interface init
}

public int onStartCommand(Intent intent, int flags, int st

@Override
public IBinder onBind(Intent arg0) {
    Log.i(LOG_TAG, "Service bind requestd");
    return mBinder;
}

public void onDestroy() {

private class AndroIntBinder extends IAddService.Stub {
    public int add(int a, int b) throws RemoteExceptin {
        return a+b;
    }
```

Think: how it works?

© 2016 William W.-Y. Liang, All Rights Reserved.

Thread Issues for Android

- The Andoid UI toolkit is not thread-safe.
- Do not manipulate your UI from a worker thread
- Two rules to Android's single UI thread model:
  - Do not access the UI toolkit from outside the UI thread
  - Do not block the UI thread

Problematic Example

```
public void onClick(View v) {
   new Thread(new Runnable() {
      public void run() {
         Bitmap b = loadImageFromNetwork("http://example.com/image.png");
         mImageView.setImageBitmap(b);
      }
   }).start();
}
```

Think:

1. **How to solve the problem**
2. **The ANR issue**

© 2016 William W.-Y. Liang, All Rights Reserved.

The Message Queue

- Message Queue
- Looper
- Handler

```
class LooperThread extends Thread {
    public Handler mHandler;

    public void run() {
       Looper.prepare();

       mHandler = new Handler() {
          public void handleMessage(Message msg) {
             // process incoming messages here
          }
       };

       Looper.loop();
    }
}
```

Think: how the framework works?

© 2016 William W.-Y. Liang, All Rights Reserved.

Process Lifecycle

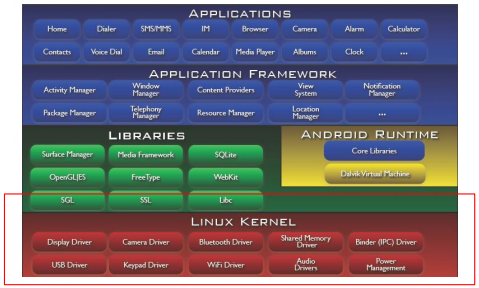
- In Android, the system needs to remove old processes when memory runs low.
- To determine which processes to keep and which to kill
- There are five levels, listed in order of importance:
  - Foreground process
  - Visible process
  - Service process
  - Background process
  - Empty process

Think: when it changes and how it affects?

© 2016 William W.-Y. Liang, All Rights Reserved.

The Linux Kernel

- Android relies on Linux kernel 2.6 and above for core system services, e.g. memory management, process management, network stack, and driver model, security, etc.
- The kernel also acts as an abstraction layer between the hardware and the rest of the software stack.



© 2016 William W.-Y. Liang, All Rights Reserved.

Features that Android added in Kernel

- **Binder**: Binder driver provides high performance inter-process communication (IPC) through shared memory.
- **Ashmem**: The Android shared memory driver, which create a memory region to be shared between processes.
- **Pmem** and **ION**: Used to allocate and manage a physically contiguous memory for user space driver.
- **Power**: The android power management driver
- **Alarm**: It's a driver which provides a real time clock.

Think: Linux mainline kernel vs. Android kernel

© 2016 William W.-Y. Liang, All Rights Reserved.

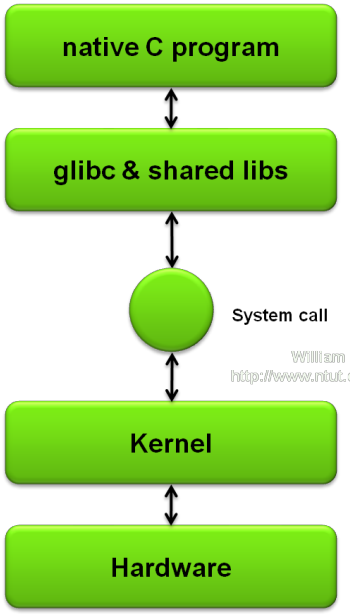
User-space Device Control

- Typical device drivers exist in the Kernel
- However, some of the device control logic can be performed from the user space.
- Two methods to do so
  - Communicate with the kernel-space drivers through the device files or other system interfaces such as sysfs
  - Direct hardware access through memory mapped I/O, by *mmap*

Think: performance issues for the above methods

© 2016 William W.-Y. Liang, All Rights Reserved.

The Traditional Linux Device Control Method

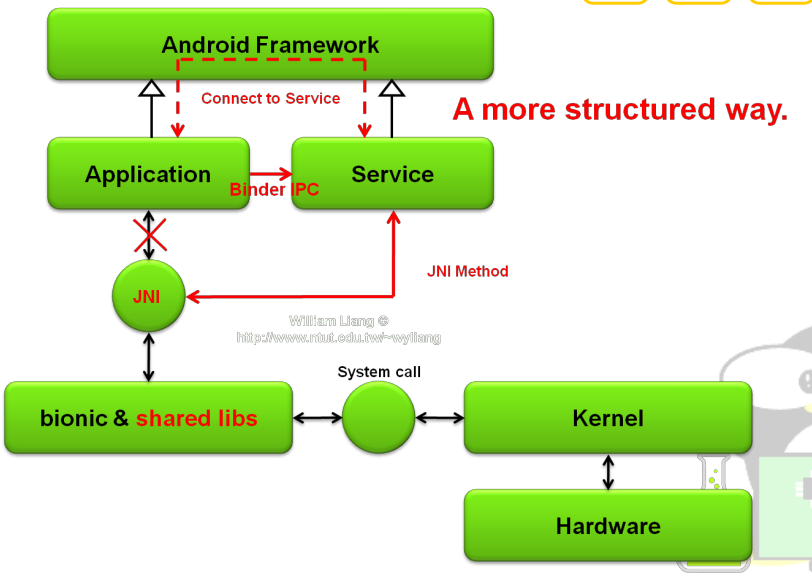



```
#include <stdio.h>
#include <fcntl.h>

#define DEVFILE "/dev/androint"

int main() {
        int fd;
        int in[2] = {10, 20};
        int out;

        printf("AndroInt virtual adder driver test:\n");
        printf("Input A: ");
        scanf("%d", in);
        printf("Input B: ");
        scanf("%d", in+1);
        printf("Input: %d %d\n", in[0], in[1]);

        fd = open(DEVFILE, O_RDWR);
        write(fd, in, sizeof(in));
        read(fd, &out, sizeof(out));
        close(fd);

        printf("Output: %d\n", out);

        return 0;
}
```

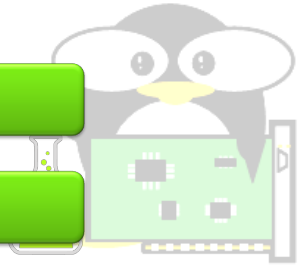
William Liang ©
http://www.ntut.edu.tw/~wyliang

© 2016 William W.-Y. Liang, All Rights Reserved.

The Formal Android Device Control Model



William Liang ©
http://www.ntut.edu.tw/~wyliang



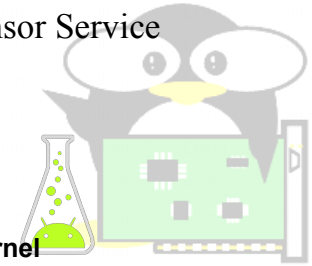
Think: resource management and protection issues

© 2016 William W.-Y. Liang, All Rights Reserved.

Android System Services

- Core Services
  - Activity Manager
  - Package Manager
  - Window Manager
  - Resource Manager
  - Content Providers
  - View System
- Hardware Services
  - Telephony Service
  - Location Service
  - Bluetooth Service
  - WiFi Service
  - USB Service
  - Sensor Service

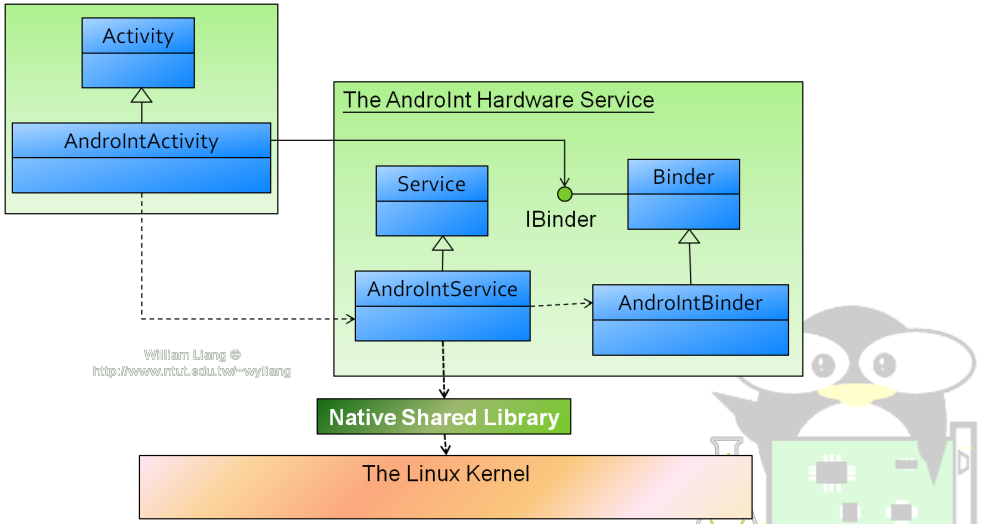
Source from "Deep inside Android," Gilles Printemps, Esmertec ©



Think: compare with the concept used in micro-kernel

© 2016 William W.-Y. Liang, All Rights Reserved.

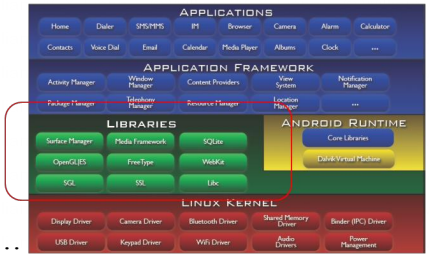
Hardware Service Example



© 2016 William W.-Y. Liang, All Rights Reserved.

Android Native Libraries

- C Library
  - Android includes a set of C/C++ libraries used by various components of the Android system.
  - Bionic C Library: A BSD-derived implementation of the standard C library, tuned for embedded Linux-based devices
- Example Media Libraries
  - SQLite
  - Webview
  - SGL
  - FreeType
  - 3D libraries
  - Many more …



Think:
1. **The Dynamic Shared Library**
2. **Performance Issue**

© 2016 William W.-Y. Liang, All Rights Reserved.

Cross the Language: Java, C/C+, and JNI

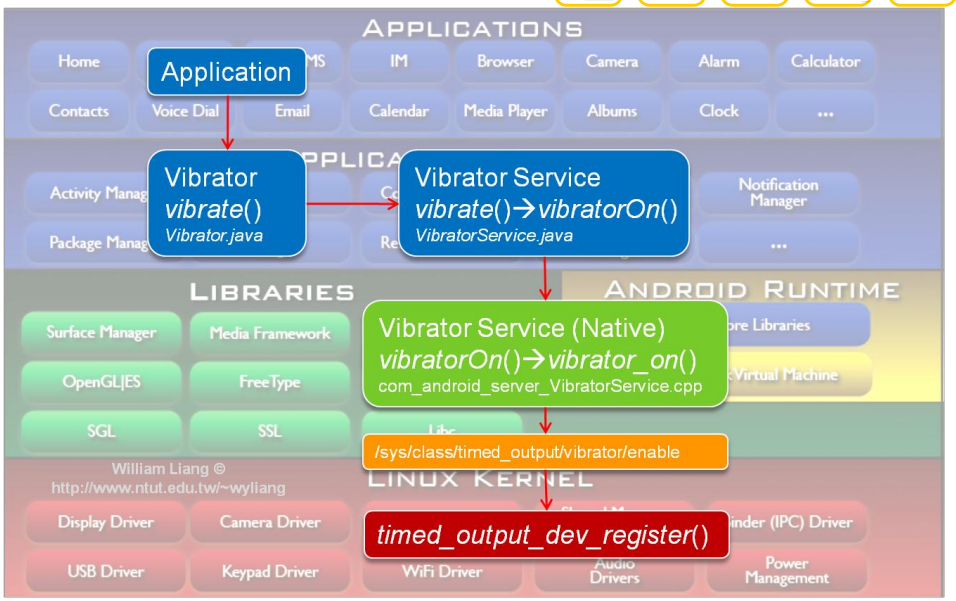
- JNI: Java Native Interface
- Standard interface between Java and C/C++

```
class HelloWorld {
   public native void helloworld();

   static {
      System.loadLibrary("hello");
   }

   public static void main(String[] args) {
      new HelloWorld().helloworld();
   }
}
```

Think:
1. **Bi-directional invocation**
2. **Dalvik VM, JIT, and ART**

© 2016 William W.-Y. Liang, All Rights Reserved.

A Simple Legacy System Service: Vibrator Service



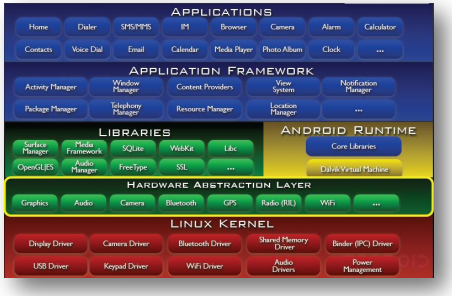
© 2016 William W.-Y. Liang, All Rights Reserved.

The Android Hardware Abstraction Layer

- User space C/C++ library layer
- Defines the interface that Android requires hardware "drivers" to implement
- Separates the Android platform logic from the hardware interface

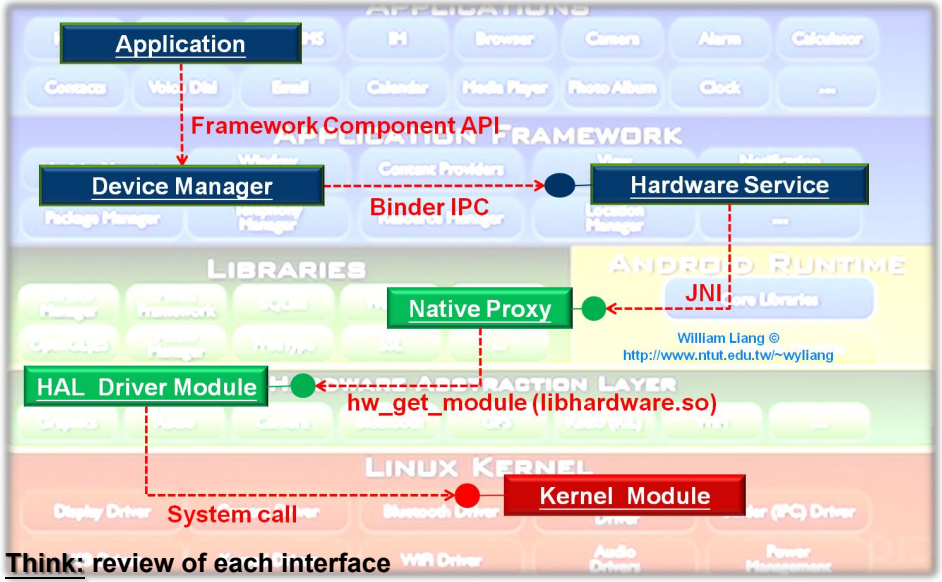
Source from "Android Anatomy and Physiology," Patrick Brady ©

- Kernel drivers are GPL which may expose the proprietary IP.



© 2016 William W.-Y. Liang, All Rights Reserved.

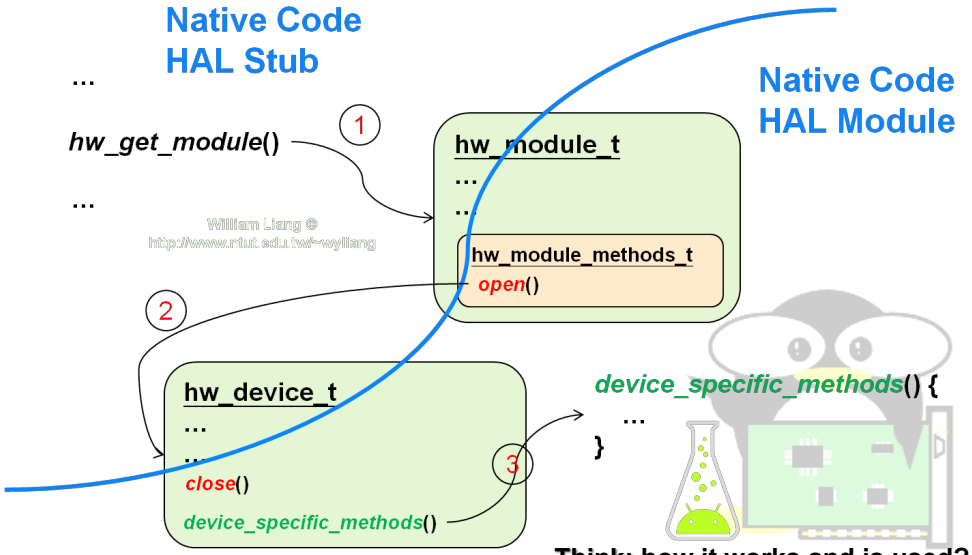
Android Hardware Control Flow Diagram



Think: review of each interface

© 2016 William W.-Y. Liang, All Rights Reserved.

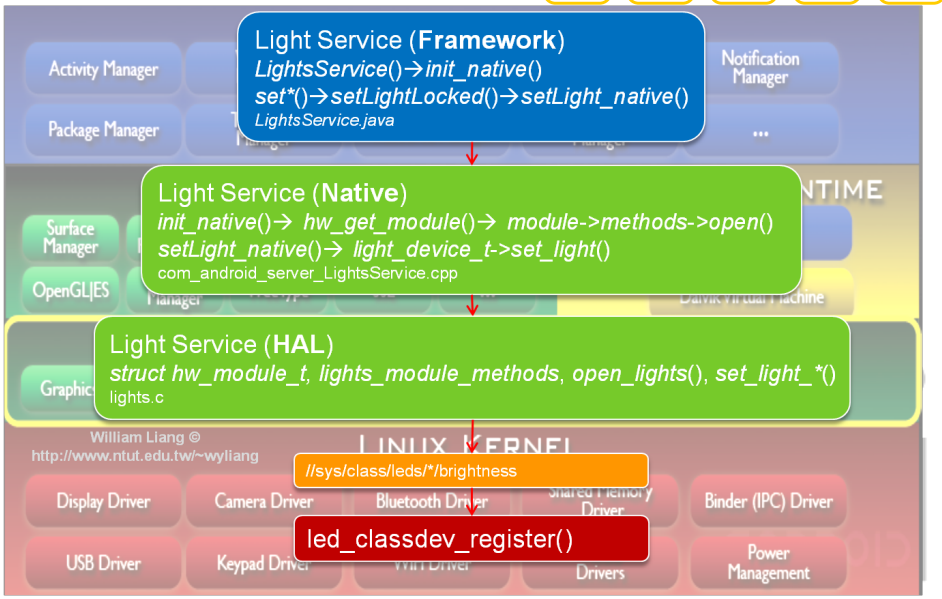
Flow for Retrieving HAL Module and Methods



Think: how it works and is used?

© 2016 William W.-Y. Liang, All Rights Reserved.

HAL Example: Lights Service



© 2016 William W.-Y. Liang, All Rights Reserved.

Q & A

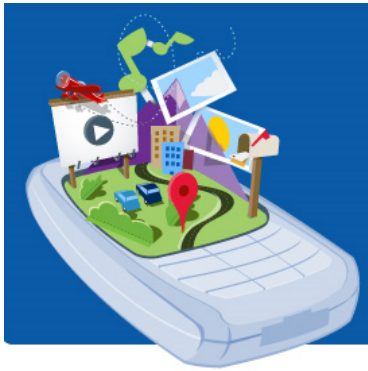
william.wyliang@gmail.com
http://www.ntut.edu.tw/~wyliang
http://www.facebook.com/william.wyliang

Note: **The Copyrights of the referenced materials and figures go to their original authors.** As a result, the slides are for non-commercial reference only.
For the contents created in this document, the Copyright belongs to William W.-Y. Liang. © 2005-2016 All Rights Reserved.

Android Introduction

Platform Overview

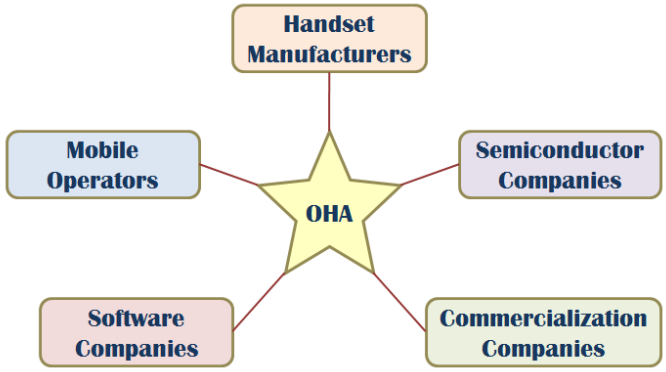
What is Android?



- Android is a software stack for mobile devices that includes an operating system, middleware and key applications.

OHA (Open Handset Alliance)

- A business alliance consisting of 47 companies to develop open standards for mobile devices



Phones

HTC G1, Droid, Tattoo

Motorola Droid (X)

Suno S880

Samsung Galaxy

Sony Ericsson

Tablets

Velocity Micro Cruz

Gome FlyTouch

Acer beTouch

Dawa D7

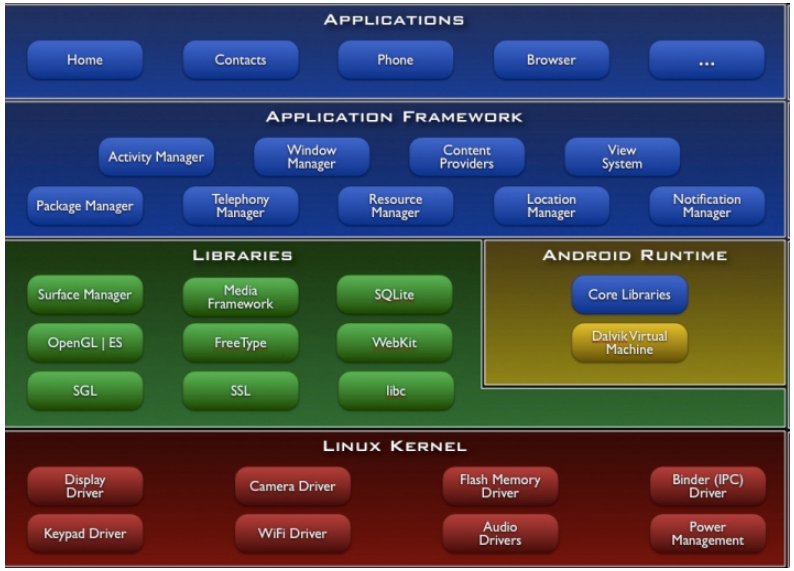
Toshiba Android SmartBook

Cisco Android Tablet

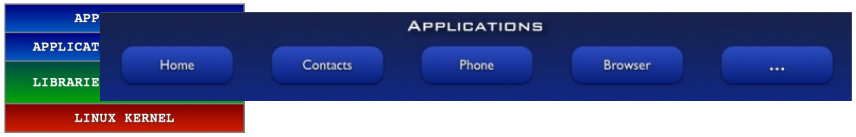
MarketShare

	Feb'10	May'10	Apr'11
RIM	42.1%	41.7%	29%
Apple	25.4%	24.4%	25%
Google	9%	13%	33%
Microsoft	15.1%	13.2%	7.7%
Palm	5.4%	4.8%	2.9%

Architecture

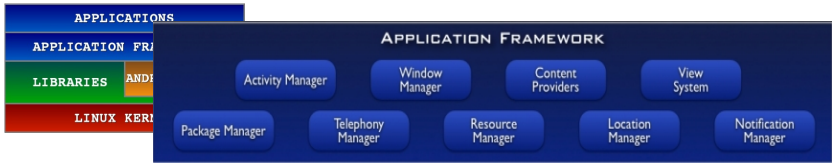


Android S/W Stack - Application



- Android provides a set of core applications:
  - ✓ Email Client
  - ✓ SMS Program
  - ✓ Calendar
  - ✓ Maps
  - ✓ Browser
  - ✓ Contacts
  - ✓ Etc
- All applications are written using the Java language.

Android S/W Stack – App Framework



- Enabling and simplifying the reuse of components
  - ✓ Developers have full access to the same framework APIs used by the core applications.
  - ✓ Users are allowed to replace components.

Android S/W Stack – App Framework (Cont)

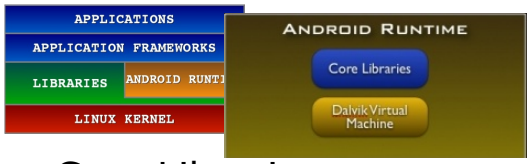
- Features

Feature	Role
View System	Used to build an application, including lists, grids, text boxes, buttons, and embedded web browser
Content Provider	Enabling applications to access data from other applications or to share their own data
Resource Manager	Providing access to non-code resources (localized strings, graphics, and layout files)
Notification Manager	Enabling all applications to display customer alerts in the status bar
Activity Manager	Managing the lifecycle of applications and providing a common navigation backstack

Android S/W Stack - Libraries

- Including a set of C/C++ libraries used by components of the Android system
- Exposed to developers through the Android application framework

Android S/W Stack - Runtime



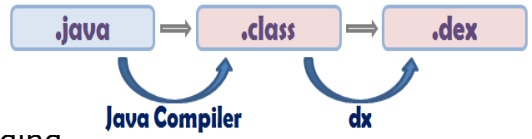
- Core Libraries
  - ✓ Providing most of the functionality available in the core libraries of the Java language
  - ✓ APIs
    - Data Structures
    - Utilities
    - File Access
    - Network Access
    - Graphics
    - Etc

Android S/W Stack – Runtime (Cont)

- Dalvik Virtual Machine
  - ✓ Providing environment on which every Android application runs
    - Each Android application runs in its own process, with its own instance of the Dalvik VM.
    - Dalvik has been written such that a device can run multiple VMs efficiently.
  - ✓ Register-based virtual machine

Android S/W Stack – Runtime (Cont)

- Dalvik Virtual Machine (Cont)
  - ✓ Executing the Dalvik Executable (.dex) format
    - .dex format is optimized for minimal memory footprint.
    - Compilation



  - ✓ Relying
    - Threading
    - Low-level memory management

Android S/W Stack – Linux Kernel

- Relying on Linux Kernel 2.6 for core system services
  - ✓ Memory and Process Management
  - ✓ Network Stack
  - ✓ Driver Model
  - ✓ Security
- Providing an abstraction layer between the H/W and the rest of the S/W stack

S. No	Project Title
1	
2	FPGA based Adaptive Cruise Control System using Ultrasonic sensor
3	
4	
5	Automatic Food Ordering System
6	
7	Implementation of synchronous FIFO in FPGA
8	
9	
10	Object Detection and Recognition using ML in OpenCV on Jetson nano board
11	
12	
13	Safety Sign Identification for autonomous vehicle using Jetson Nano
14	
15	
16	Real time license plate recognition using Image processing techniques on Raspb
17	
18	
19	Smart Home: Automated Security and Monitoring System
20	
21	Implementation of SPI controller on FPGA
22	FFT algorithmn implementation
23	
24	
25	UART VGA SYSTEM
26	
27	Design & Implementation of Cash Controller unit
28	

EEL5881 Software Engineering I
UML Lecture

Yi Luo

Acknowledgements

- Slides material are taken from different sources including:
  - the slides of Mr. Shiyuan Jin's UML class, EEL 4884, Fall 2003.
  - Object-Oriented and Classical Software Engineering, Sixth Edition, WCB/McGraw-Hill, 2005 Stephen R. Schach
  - UML resource page http://www.uml.org/

Outline

- What is UML and why we use UML?
- How to use UML diagrams to design software system?
- What UML Modeling tools we use today?

What is UML and Why we use UML?

- UML → "Unified Modeling Language"
  - Language: express idea, not a methodology
  - Modeling: Describing a software system at a high level of abstraction
  - Unified: UML has become a world standard
    Object Management Group (OMG): www.omg.org

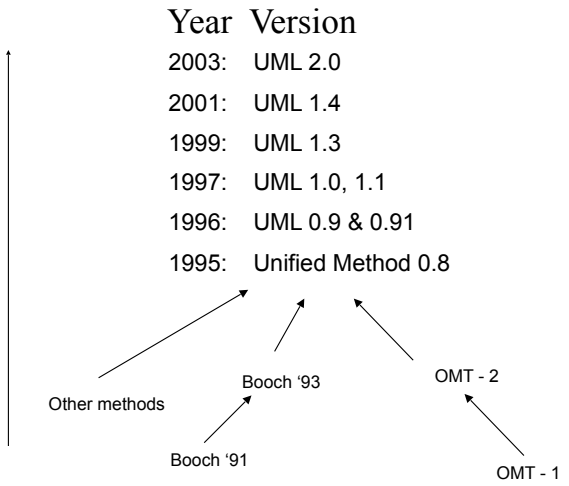
What is UML and Why we use UML?

- More description about UML:
  - It is a industry-standard graphical language for specifying, visualizing, constructing, and documenting the artifacts of software systems
  - The UML uses mostly graphical notations to express the design of software projects.
  - Simplifies the complex process of software design

What is UML and Why we use UML?

- Why we use UML?
  - Use graphical notation: more clearly than natural language (imprecise) and code (too detailed).
  - Help acquire an overall view of a system.
  - UML is *not* dependent on any one language or technology.
  - UML moves us from fragmentation to standardization.

What is UML and Why we use UML?



How to use UML diagrams to design software system?

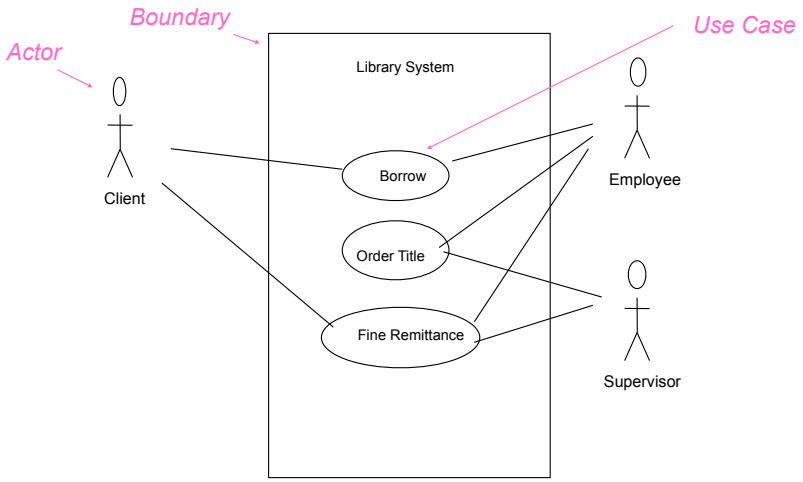
- Types of UML Diagrams:
  - Use Case Diagram
  - Class Diagram
  - Sequence Diagram
  - Collaboration Diagram
  - State Diagram

This is only a subset of diagrams … but are most widely used

Use-Case Diagrams

- A use-case diagram is a set of use cases
- A use case is a model of the interaction between
  - External users of a software product (actors) and
  - The software product itself
  - More precisely, an actor is a user playing a specific role
- describing a set of user **scenarios**
- capturing user requirements
- **contract** between end user and software developers

Use-Case Diagrams



Use-Case Diagrams

- **Actors:** A role that a user plays with respect to the system, including human users and other systems. e.g., inanimate physical objects (e.g. robot); an external system that needs some information from the current system.
- **Use case:** A set of scenarios that describing an interaction between a user and a system, including alternatives.
- **System boundary**: rectangle diagram representing the boundary between the actors and the system.

Use-Case Diagrams

- Association:

communication between an actor and a use case; Represented by a solid line.

- Generalization: relationship between one general use case and a special use case (used for defining special alternatives) Represented by a line with a triangular arrow head toward the parent use case.

Use-Case Diagrams

Include: a dotted line labeled <<include>> beginning at base use case and ending with an arrows pointing to the include use case. The include relationship occurs when a chunk of behavior is similar across more than one use case. Use "include" in stead of copying the description of that behavior.

<<include>>

Extend: a dotted line labeled <<extend>> with an arrow toward the base case. The extending use case may add behavior to the base use case. The base class declares "extension points".

<<extend>>

Use-Case Diagrams

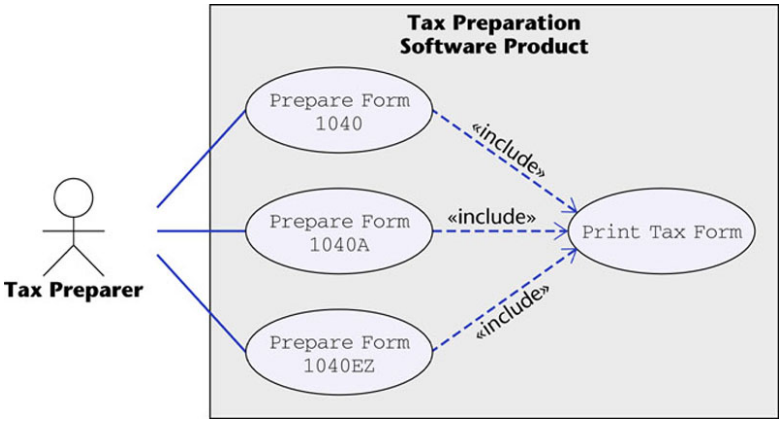
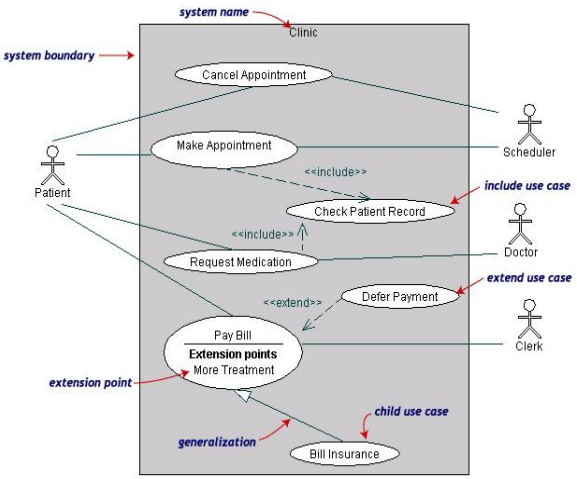


Figure 16.12

The McGraw-Hill Companies, 2005

Use-Case Diagrams

- Both **Make Appointment** and **Request Medication** include **Check Patient Record** as a subtask (include)
- The **extension point** is written inside the base case **Pay bill**; the extending class **Defer payment** adds the behavior of this extension point. (extend)
- **Pay Bill** is a parent use case and **Bill Insurance** is the child use case. (generalization)



(TogetherSoft, Inc)

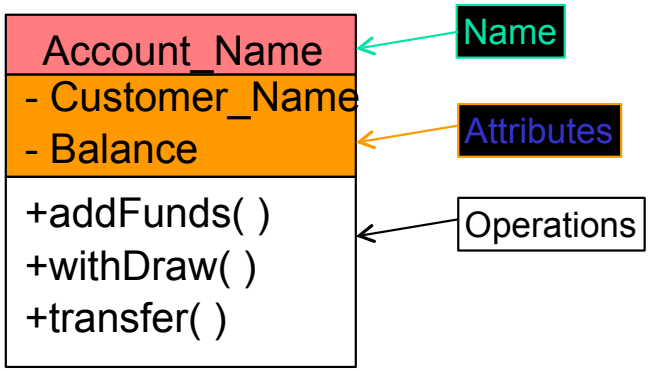
Class diagram

- A class diagram depicts classes and their interrelationships
- Used for describing structure and behavior in the use cases
- Provide a conceptual model of the system in terms of entities and their relationships
- Used for requirement capture, end-user interaction
- Detailed class diagrams are used for developers

Class diagram

- Each class is represented by a rectangle subdivided into three compartments
  - Name
  - Attributes
  - Operations
- Modifiers are used to indicate visibility of attributes and operations.
  - '+' is used to denote Public visibility (everyone)
  - '#' is used to denote Protected visibility (friends and derived)
  - '-' is used to denote Private visibility (no one)
- By default, attributes are hidden and operations are visible.

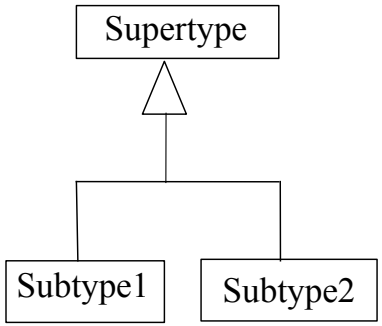
Class diagram

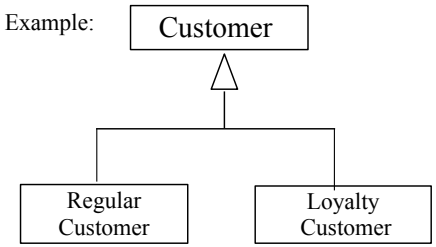


OO Relationships

- There are two kinds of Relationships
  - Generalization (parent-child relationship)
  - Association (student enrolls in course)
- Associations can be further classified as
  - Aggregation
  - Composition

OO Relationships: **Generalization**





-Inheritance is a required feature of object orientation

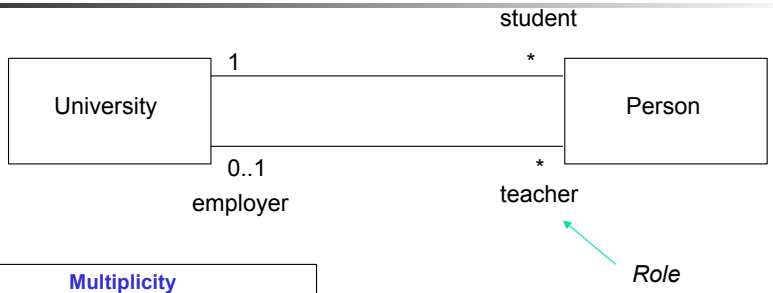
-Generalization expresses a parent/child relationship among related classes.

-Used for abstracting details in several layers

OO Relationships: **Association**

- Represent relationship between instances of classes
  - Student enrolls in a course
  - Courses have students
  - Courses have exams
  - Etc.
- Association has two ends
  - Role names (e.g. enrolls)
  - Multiplicity (e.g. One course can have many students)
  - Navigability (unidirectional, bidirectional)

Association: Multiplicity and Roles

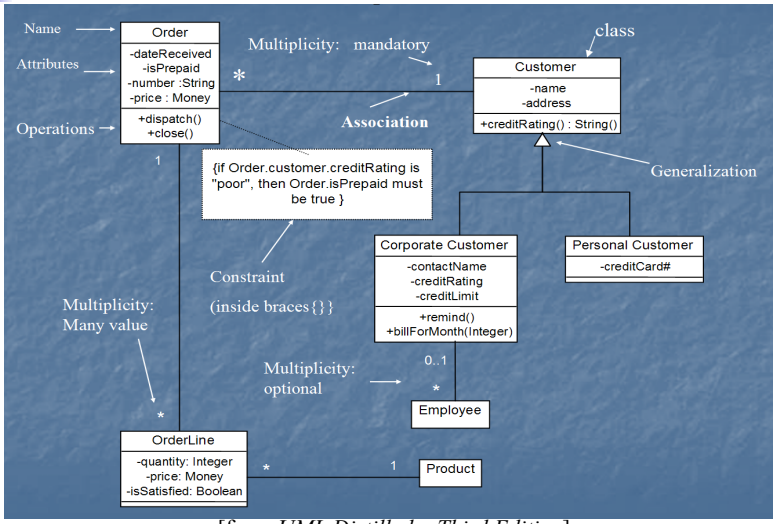


Multiplicity	
Symbol	Meaning
1	One and only one
0..1	Zero or one
M..N	From M to N (natural language)
*	From zero to any positive integer
0..*	From zero to any positive integer
1..*	From one to any positive integer

Role

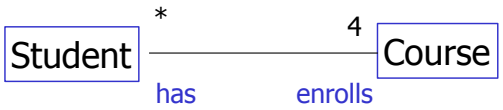
"A given university groups many people; some act as students, others as teachers. A given student belongs to a single university; a given teacher may or may not be working for the university at a particular time."

Class diagram



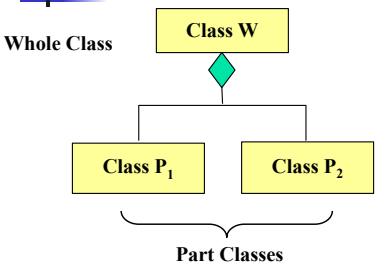
[from *UML Distilled Third Edition*]

Association: Model to Implementation



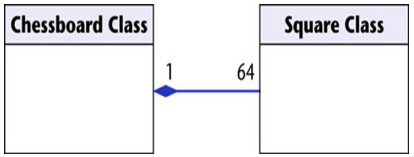

```
Class Student {
    Course enrolls[4];
}

Class Course {
    Student have[];
}
```

Oo Relationships: **Composition**



[From Dr.David A. Workman]

Example



Association

Models the part–whole relationship

Composition

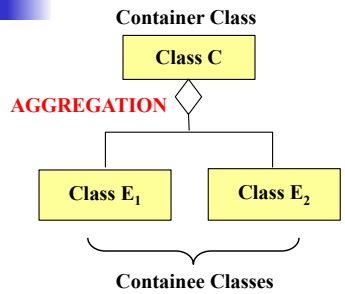
Also models the part–whole relationship but, in addition, Every part may belong to only one whole, and If the whole is deleted, so are the parts

Example:
A number of different chess boards: Each square belongs to only one board. If a chess board is thrown away, all 64 squares on that board go as well.

Figure 16.7

The McGraw-Hill Companies, 2005

Relationships: **Aggregation**

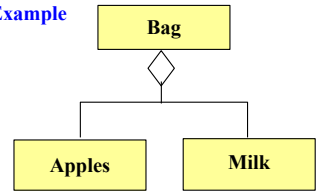


Aggregation:
expresses a relationship among instances of related classes. It is a specific kind of Container-Containee relationship.

express a more informal relationship than composition expresses.

Aggregation is appropriate when Container and Containees have no special access privileges to each other.

Example



[From Dr.David A. Workman]

Aggregation vs. Composition

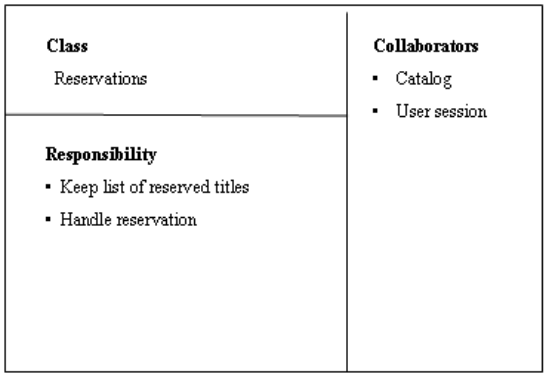
- **Composition** is really a strong form of **association**
  - components have only one owner
  - components cannot exist independent of their owner
  - components live or die with their owner
  - e.g. Each car has an engine that can not be shared with other cars.

- **Aggregations**
may form "part of" the association, but may not be essential to it. They may also exist independent of the aggregate. e.g. Apples may exist independent of the bag.

Good Practice: CRC Card

Class Responsibility Collaborator

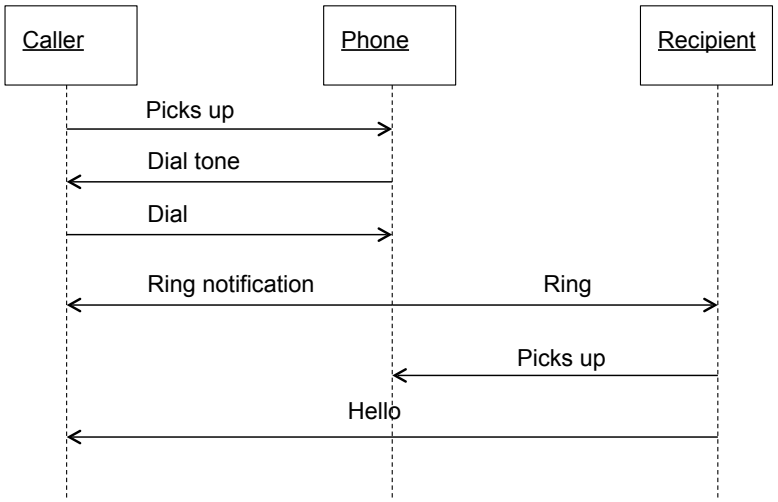
- easy to describe how classes work by moving cards around; allows to quickly consider alternatives.



Interaction Diagrams

- show how objects interact with one another
- UML supports two types of interaction diagrams
  - Sequence diagrams
  - Collaboration diagrams

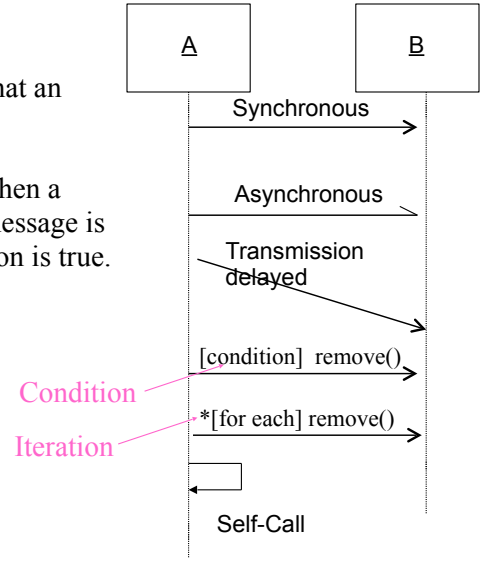
Sequence Diagram(make a phone call)



Sequence Diagram:Object interaction

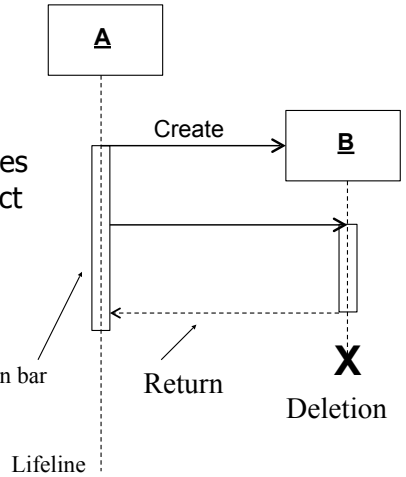
Self-Call: A message that an Object sends to itself.

Condition: indicates when a message is sent. The message is sent only if the condition is true.



Sequence Diagrams – Object Life Spans

- Creation
  - Create message
  - Object life starts at that point
- Activation
  - Symbolized by rectangular stripes
  - Place on the lifeline where object is activated.
  - Rectangle also denotes when object is deactivated.
- Deletion
  - Placing an 'X' on lifeline
  - Object's life ends at that point



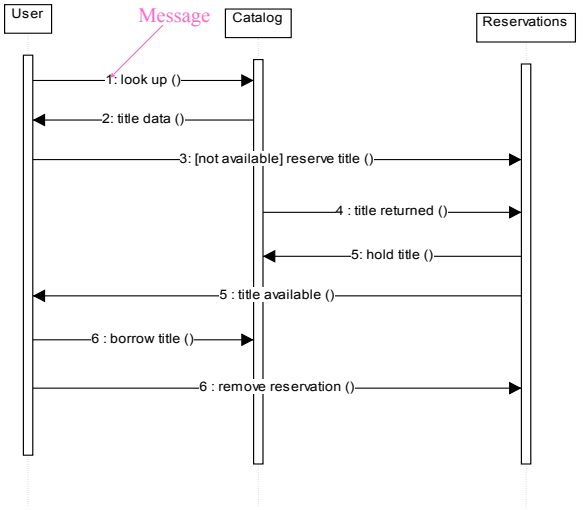
Sequence Diagram

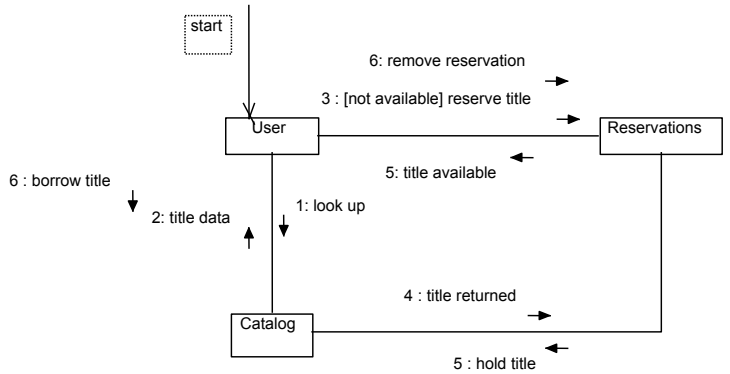
•Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass.

•The horizontal dimension shows the objects participating in the interaction.

•The vertical arrangement of messages indicates their order.

•The labels may contain the seq. # to indicate concurrency.





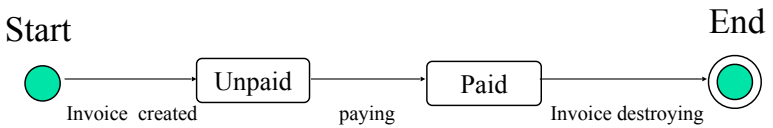
➢Collaboration diagrams are equivalent to sequence diagrams. All the features of sequence diagrams are equally applicable to collaboration diagrams

➢Use a sequence diagram when the transfer of information is the focus of attention

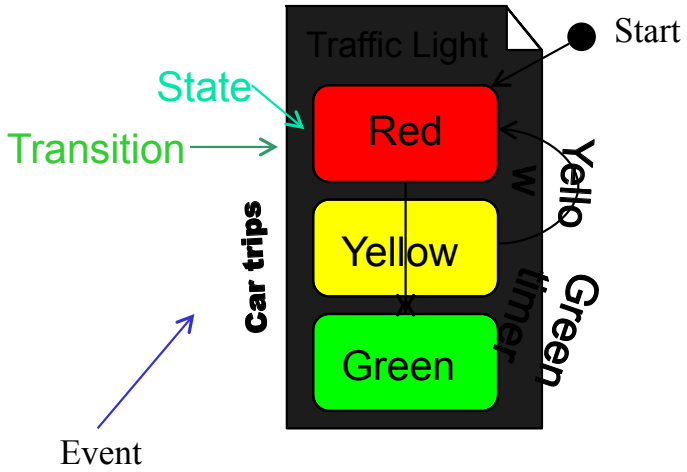
➢Use a collaboration diagram when concentrating on the classes

State Diagrams (Billing Example)

State Diagrams show the sequences of states an object goes through during its life cycle in response to stimuli, together with its responses and actions; an abstraction of all possible behaviors.



State Diagrams (Traffic light example)



What UML Modeling tools we use today?

- List of UML tools http://en.wikipedia.org/wiki/List_of_UML_tools
- ArgoUML: http://argouml.tigris.org/
- Rational Rose (www.rational.com) by IBM
- UML Studio 7.1 (http://www.pragsoft.com/) by Pragsoft Corporation: Capable of handling very large models (tens of thousands of classes). Educational License US$ 125.00; Freeware version.
- TogetherSoft Control Center; TogetherSoft Solo (http://www.borland.com/together/index.html) by Borland

Conclusion

- UML is a standardized specification language for object modeling
- Several UML diagrams:
  - use-case diagram: a number of use cases (use case models the interaction between actors and software)
  - Class diagram: a model of classes showing the static relationships among them including association and generalization.
  - Sequence diagram: shows the way objects interact with one another as messages are passed between them. Dynamic model
  - State diagram: shows states, events that cause transitions between states. Another dynamic model reflecting the behavior of objects and how they react to specific event
- There are several UML tools available

Object Oriented Analysis & Design

Ajit K Nayak
Department of Computer Science & Engineering
Silicon Institute of Technology, Odisha, India

Analysis & Design

- Analysis emphasizes an investigation of the problem and requirements
- i.e. *requirement analysis* is an investigation of the requirements and *object analysis* is an investigation of domain objects
- Design emphasizes a conceptual solution that fulfills requirements, ultimately the design may be implemented.

OO analysis & Design

- During object oriented analysis emphasizes on finding and describing the objects in the problem domain
- Examples: in case of library information system, some of the objects include Book, Library etc.
- During Object Oriented Design emphasizes on defining software objects and how they collaborate to fulfill the requirements
- Example: in the library system, a Book software object may have a title attribute and a getChapter () method
- Finally, during implementation the objects are implemented, such as a Book class in C++

Domain object

Book
title

Visualization of Domain object

```
class Book{
    String title;
    public:
    chapter getChapter(int);
};
```

Implementation

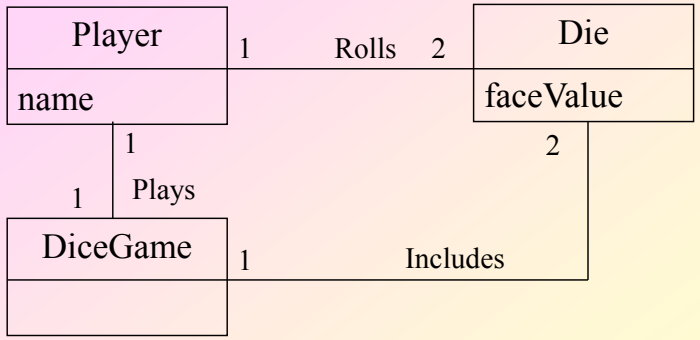
OOAD

- The steps for object oriented analysis and design may be as follows
  - Define Usecases
  - Define domain model
  - Define interaction diagram
  - Define design class diagram

Example

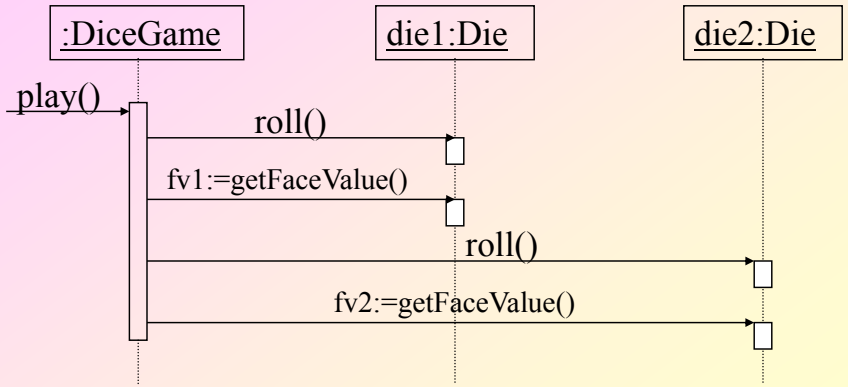
- A *"Dice Game" : A player rolls two die*
- *Usecase: description of related domain process*
  - *Play a Dice Game: A player picks up and rolls the dice. If the dice face value total seven, they win; otherwise, they lose*
- Define domain model: Decomposition of the domain involves identification of objects, attributes and associations.
  - Represented in a diagram

Defining Domain model



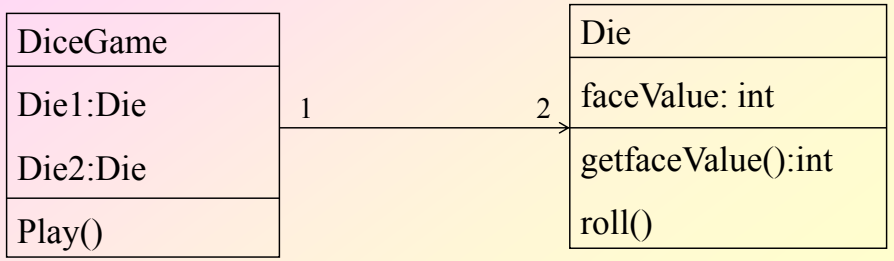
Define Interaction diagram

- Defining software objects and their collaborations.
- This diagram shows the flow of message between software objects and thus invocation of methods



Define design class diagrams

- To create a static view of the class definition with a design class diagram.
- This illustrates the attributes and methods of the classes.



- To represent the analysis and design of object oriented software systems, we use a language called Unified Modeling Language.

UML

- The Unified Modeling Language™ (UML) is the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems
- It simplifies the complex process of software design, making a "blueprint" for construction.
- The Unified Modeling Language, UML, and the UML logos are trademarks of the Object Management Group.
- www.omg.org

Purpose of Modeling?

- Developing a model for an industrial-strength software system prior to its construction or renovation is as essential as having a blueprint for large building.
- Good models are essential for communication among project teams and to assure architectural soundness.
- As the complexity of systems increase, so does the importance of good modeling techniques. There are many additional factors of a project's success, but having a rigorous modeling language standard is one essential factor.

References

- Books
  - The Unified Modeling Language, User Guide
    - Booch, Rumbaugh. Jacobson **(TEXT)**
  - The Unified Modeling Language, Reference Manual
    - Booch, Rumbaugh. Jacobson
  - The Unified Software Development Process
    - Booch, Rumbaugh. Jacobson
  - Using UML
    - Rob Pooley et al.
  - UML Distilled
    - Martin Fowler, Kendall Scott
  - Instant UML
    - Pierre-Alain Muller

Building blocks of UML

- Building blocks of UML constitutes
  - Things
  - Relationships
  - Diagrams

Things

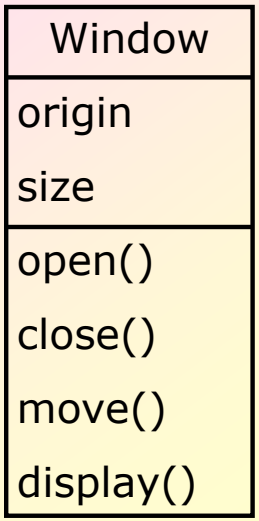
- There are four kind of things defined
  - Structural things
  - Behavioral things
  - Grouping things
  - Annotational things

A. Structural Things

- These are nouns of UML model
- These things represents elements that are either conceptual or physical.
- There are seven kinds of structural things
  - Class
  - Interface
  - Collaboration
  - Use Case
  - Active Class
  - Component
  - Node

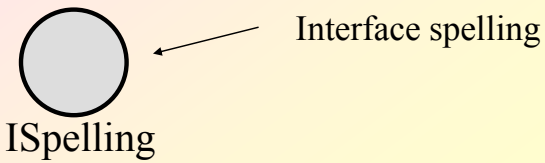
Class

- Is a description of a set of objects that share common attributes, operations, relationships, and semantics.
- A class implements one or more instances
- It is represented by a rectangular box



Interface

- Is a collection of operations that specify a service of a class
- An interface might represent the complete behaviour of a class or only a part of that behaviour
- It is represented by a circle, with its name



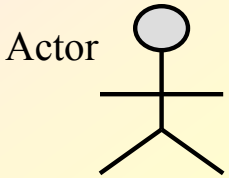
Collaboration

- Defines an interaction and is a society of roles and other elements that work together to provide some cooperative behaviour
- It is represented by an dashed ellipse with its name

Use Case

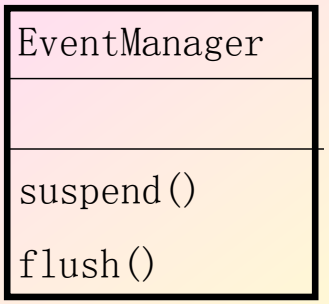
- It is a description of sequence of actions that a system performs that yields an observable result of a value to a particular actor
- It is represented by a solid ellipse with its name

Use case



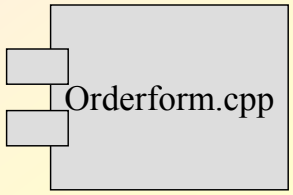
Active Class

- It is a class, whose objects own one or more processes or threads and therefore can initiate control activity
- It is represented by a heavy lined box



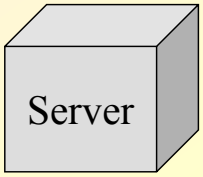
Component

- It is a physical and replaceable part of a system that confirms to and realization of a set of interfaces.
- i.e. physical packaging of classes, interfaces and collaborations



Node

7. It is a physical element that exists in run time and represents a computational resource, generally having some memory and processing capability
8. A set of components may reside as a node and may also migrate from node to node
9. It is represented as a cube



B. Behavioural Things

- These are dynamic parts of UML model
- These are verbs of a model, representing behaviour over time and space
- There are two types of Behavioural things
  - Interaction
  - State Machine

Interaction

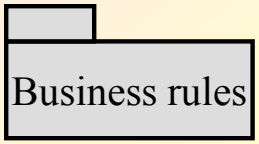
1. It is a behaviour that comprises a set of messages exchanged among a set of objects to accomplish a specific purpose.
2. An interaction involves messages, action sequences and links. (connection between objects)
3. It is represented by a solid arrow

State Machine

- It is a behaviour that specifies the sequence of states of an object or an interaction goes through during its life time in response to events, to gether with its responses to these events.
- A state machine involves
  - States
  - Transitions (flow from one state to another)
  - Events (things that trigger a transition)
  - Activities (response to transitions)
  - It is represented with a rectangular box having rounded corners

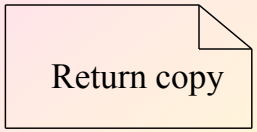
C.Grouping Things

- These are organizational part of UML models
- There is one primary kind of grouping thing called packages
- Package
  - It is a general purpose mechanism for organizing elements in to groups
  - It is rendered as a tabbed folder



D. Annotational Things

- These are explanation parts of UML
- Note is a annotational thing called
- Note
  - It is simply a symbol for rendering constraints and comments attached to an element or a collection of elements.



Some explanatory text

Relationships

- A relationship is a connection among things
- It is rendered as a path, with different kind of lines used to distinguish the kind of relationships

Relationships

- There are 4 kinds of relationships
  - Dependency
  - Association
  - Generalization
  - Realization

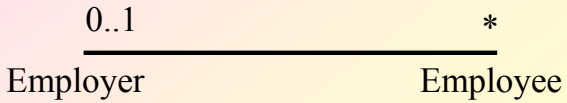
A. Dependency

- It is a semantic relationship between two things in which a change to independent thing may affect the semantic of the other thing
- It is rendered as a directed dashed line

- The form the system displays obviously depends on which form the user selects

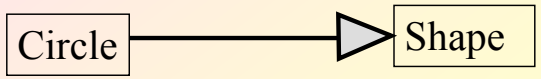
B. Association

- It describes a set of links
  - **Aggregation** is a special kind of association, representing a structural relationship between a whole and its parts
  - It is rendered as a solid line, occasionally including a label and other specifications



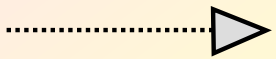
C. Generalization/Specialization

- Child shares the structure and behaviour of the parent
  - ✓ Objects of specialized elements are substitutable for the objects of generalized elements
  - ✓ It is rendered as a solid line with a hollow arrow head pointing to the parent



D. Realization

- It is a relationship between classifiers. (polymorphism)
  - ✓ One classifier specifies a contract that another classifier guarantees to carry out
  - ✓ It is rendered as a dashed line with a hollow arrow head.
  - ✓ The base class provides an interface for which the derived class writes an interface



Diagrams

- A Graphical representation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).

Diagrams

- There are 9 types of Diagrams
  - Class Diagram
    - Shows a set of classes, interfaces, and collaborations and their relationships.
  - Object Diagram
    - Shows a set of objects and their relationships
  - Use Case Diagram
    - Shows a set of use cases and actors
  - Sequence Diagram & Collaboration Diagram
    - These two are interaction kind of diagram. They show a set of objects and their relationships

Diagrams

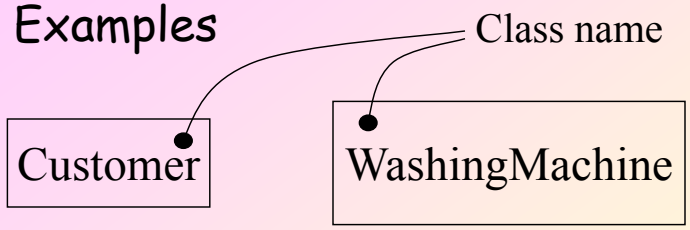
  - Statechart Diagram
    - Shows a state machine, consisting of states, transitions, events and activities
  - Activity Diagram
    - It is a kind of statechart diagram that shows the flow from activity to activity with a system
  - Component Diagram
    - Shows the organisation and dependencies among a set of componets. It addresses the static implementation views of a system
  - Deployment Diagram
    - Shows the configuration of run-time processing nodes and the components that live on them.

Classes

- A class is a descriptions of a set of objects, that share attributes, relationships and semantics.
- Graphically a class is rendered as a rectangle
- Name of a class
  - In practice class names are short nouns or noun phrases drawn from the vocabulary of the system to be modeled
  - Every class has a name that distinguishes it from other classes

Name of a class

  - Name is a textual string, with first letter of every word capitalized
- Examples

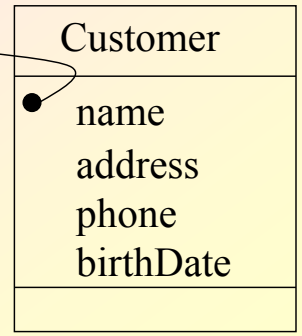


- A path name is the class name prefixed by the name of the package in which that class lives

HouseholdAppliances : : WashingMachine

Attributes of a Class

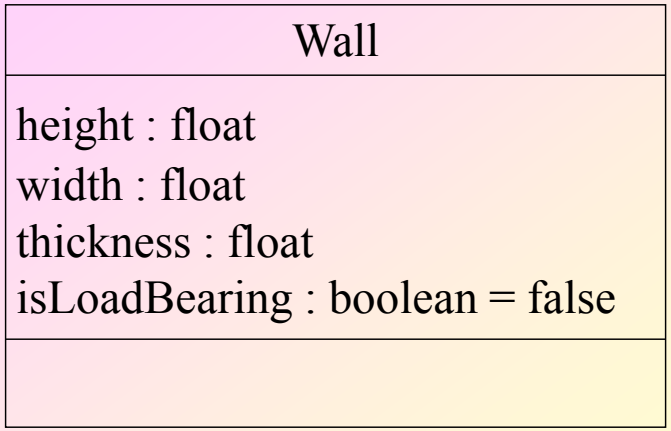
- An attribute is an abstraction of the kind of data or state an object of the class might encompass
- An attribute is a named property of a class
- A class may have any number of attributes or no attributes at all



Attributes of a Class

- Attributes are listed in a compartment just below the class name
- First letter of every word is capital in an attribute name expect the first word
- An attribute may be specified by stating its class and possibly default initial value
- In practice an attribute name is a short noun or noun Phrase that represents some property of its enclosing class

Attribute Example



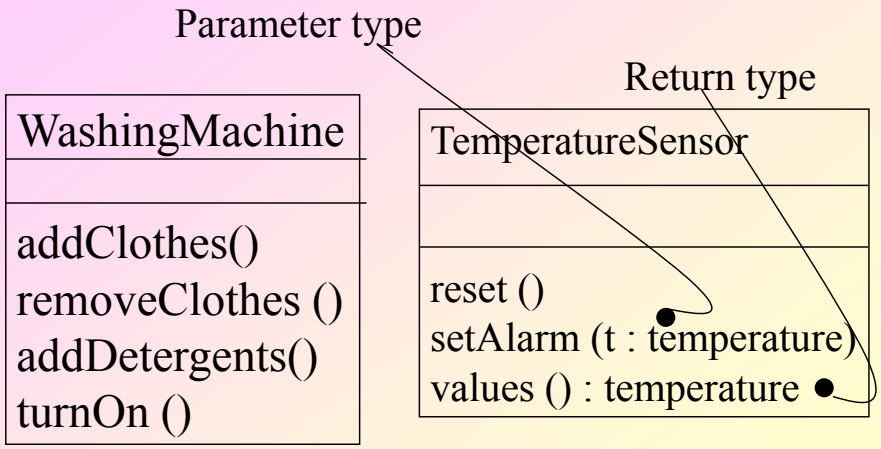
Operations in a Class

- An operation is an abstraction of something, one can do to at an object and that is shared by all objects of that class
- A class may have any no. of operations or no operations at all
- Operation are listed in a Compartment just below The class attribute

Operations in a Class

- The first letter of every word in an operations name is capital, except the first letter of first word
- An operations can be specified By stating its signature, covering the name, type and default values of all parameters and a return type
- In practice, an operation name is a short verb or verb phrase that represents some behavior of its enclosing class

Operations Example

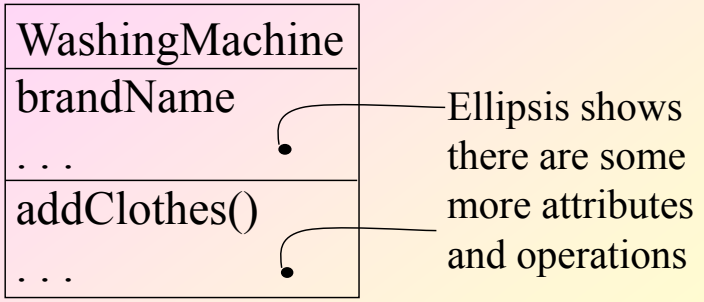


Organising Attributes and Operations

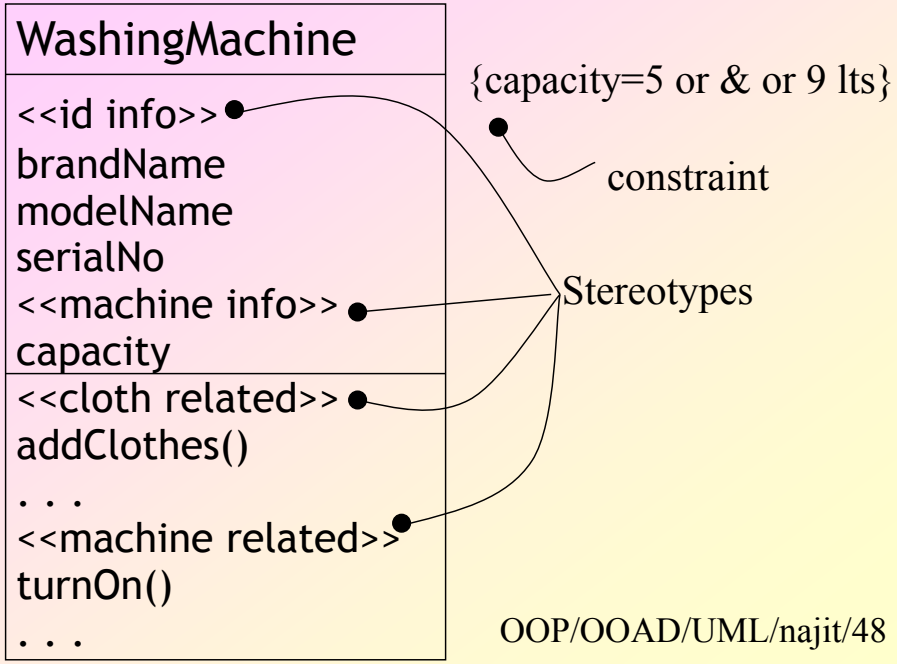
- It may not be possible to list all attributes and operations at once
- Therefore, a class can be elided, I.e. it is possible to show only some or none of attributes and operations
- An empty compartment does not necessarily mean that there are no attributes or operations
- It can be explicitly specified that there are more attributes or operations than shown, by ending each list with an ellipsis.

Attributes & Operations

- To better organise long lists of attributes and operations, those can be grouped.
- Each group is prefixed with a descriptive category by using stereotypes



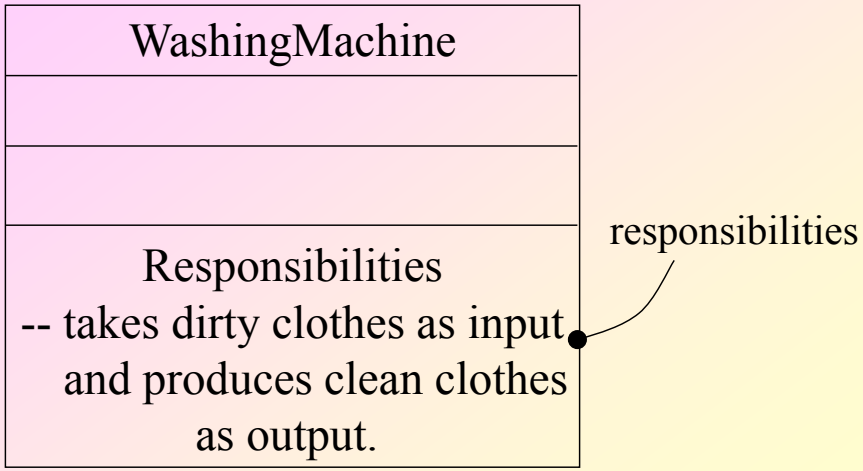
Attributes & Operations



Responsibilities

- It is a contract or an obligation of a class. i.e it is a description of what the class has to do
- Responsibilities can be drawn in a separate compartment at the bottom of the class icon
- A single responsibility is written as a phrase, a sentence, or (at most) a short paragraph (free-form text)

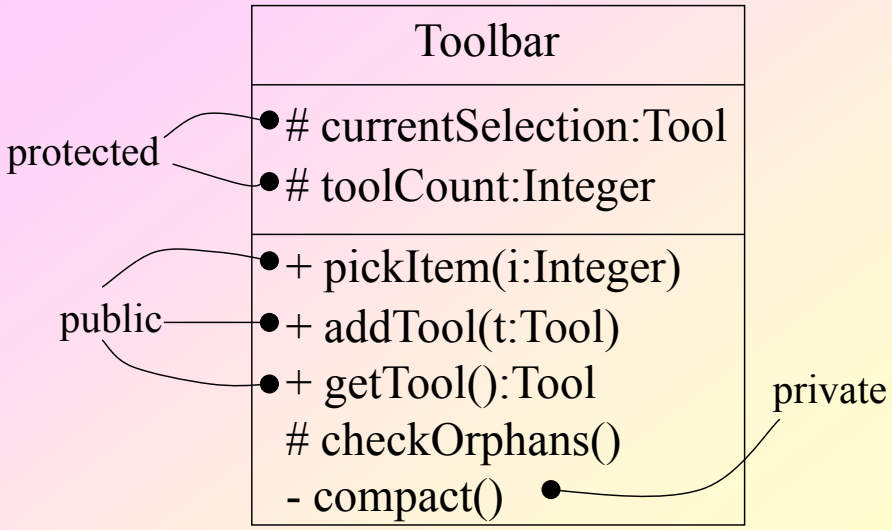
Responsibilities



Visibility of a feature

- It allows to specify the visibility for a classifier's features.
- Three levels of visibility can be specified using UML
  - Public : any outside classifier with visibility to the given classifier can use the feature; specified by prepending the symbol +
  - Protected : Any descendant of the classifier can use the feature; specified by prepending the symbol #
  - Private : only the classifier itself can use the feature; specified by prepending the symbol -

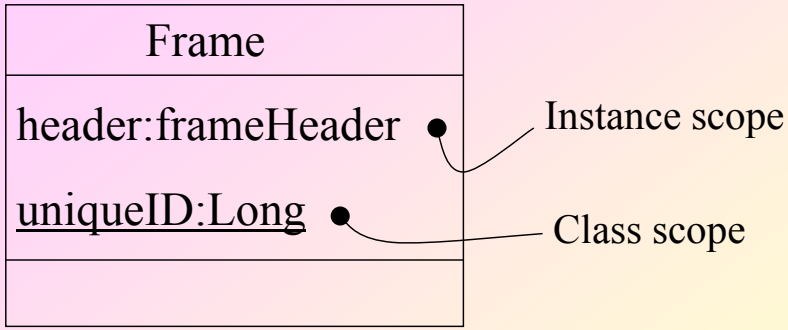
Example Visibility



Scope of a feature

- The owner scope of a feature specifies whether the feature appears in each instance of the classifier or whether there is just a single instance for all instances of the classifier
- UML provides two kinds of scope
  - Instances : Each instance of the classifier holds its own value for the feature
  - Classifier : There is just one value of the feature for all instances of the classifier.

Scope (Contd.)



- Classifier scope is rendered by underlining the feature
- In C++ we call the class scoped as static members of a class

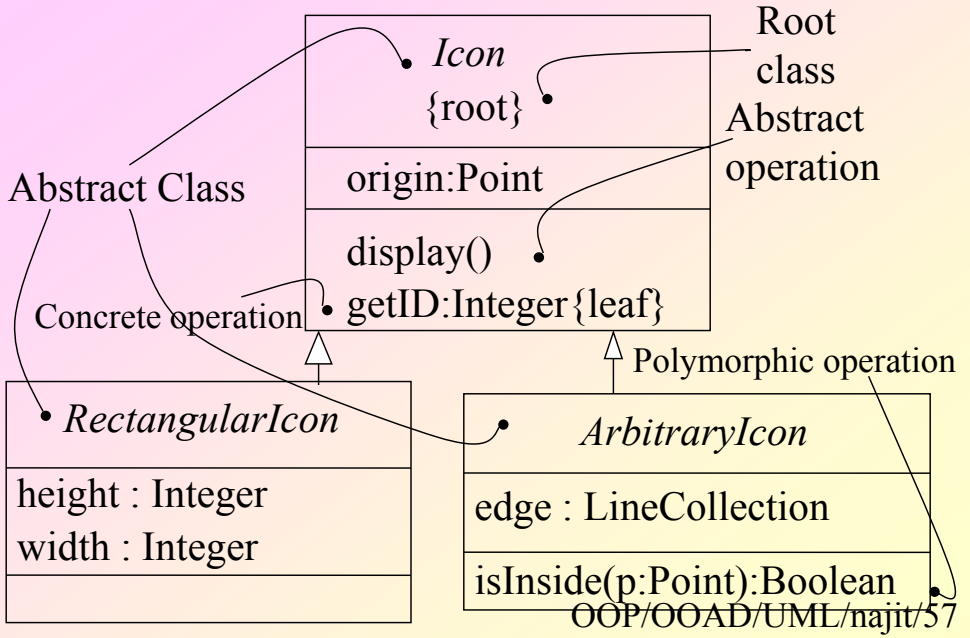
Polymorphic Elements

- Abstarct Class
  - Class having no direct instances (italics)
- Concrete Class
  - Class that may have direct instances
- Root Class
  - Class having no parent but may have children {root}
- Leaf Class
  - Class having parent but no children {leaf}

Polymorphic Elements

- Polymorphic Operations
  - An operation is polymorphic, if it can be specified with the same signature at different points of hierarchy
  - When a message is dispatched at run time, the operation in the hierarchy that is invoked is chosen polymorphycally. i.e. a match is determined at run time according to the type of the object
- Abstract Operation
  - This operation is incomplete and requires a child to supply an implementation. In UML it is specified by writing its name in italics.
- Leaf operation
  - Operation is not polymorphic and may not be overridden

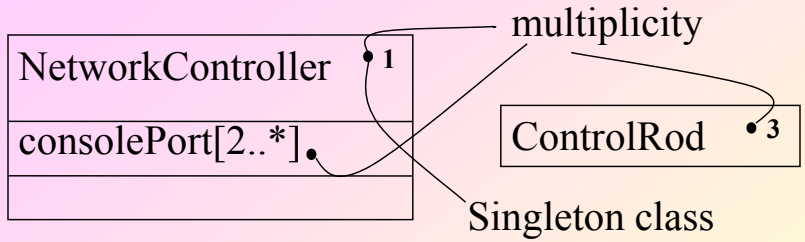
Example Polymorphism



Multiplicity

- It is a specification of the range of allowable cardinalities an entity may assume
- Multiplicity of a class
  - It is used to restrict the no of instances of a class
  - i.e. It tells the no of instances a class may have
- Multiplicity of an attribute
  - It is used to represent a set of elements for an attribute
  - i.e. It maps to representing an array variable in C++

Example Multiplicity



Notes:

- Abstract operations maps to pure virtual functions in C++
- Leaf operations maps to non-virtual functions in C++

Attributes

- In its full form, the syntax of an attribute in UML is

[visibility] name [multiplicity] [: type]
[= initial-value] [{property-string}]

Examples :

origin	Name Only
+origin	Visibility and name
origin : Point	Name and type
head : *Item	Name and complex type
name [0..1] : String	Name, multiplycity, and type

Attributes

- There are three defined properties, that can be used with attributes
  - changeable
    - There are no restrictions on modifying the attribute value
  - addonly
    - For attributes with a multiplicity greater than one, but once created, a value may not be removed or altered
  - frozen
    - The attribute's value may not be changed after the object is initialized(constants)

Operations

- In its full form, the syntax of an operation in UML is

[visibility] name [(parameter-list)] [: return-type]
[{property-string}]

Examples

display	Name Only
+display	Visibility and name
set (n : Name, s : String)	Name and parameters
getID ( ) : Integer	Name and return type
restart () {guarded}	Nameand property

Operation Parameters

- An operation again can be provided with zero or more parameters with following syntax

[direction] name : type [= default-value]

- Direction may be any of the following
  - in : An input parameter, may not be modified
  - out : An output parameter, may be modified to communicate information to the caller
  - inout : An input parameter, may be modified

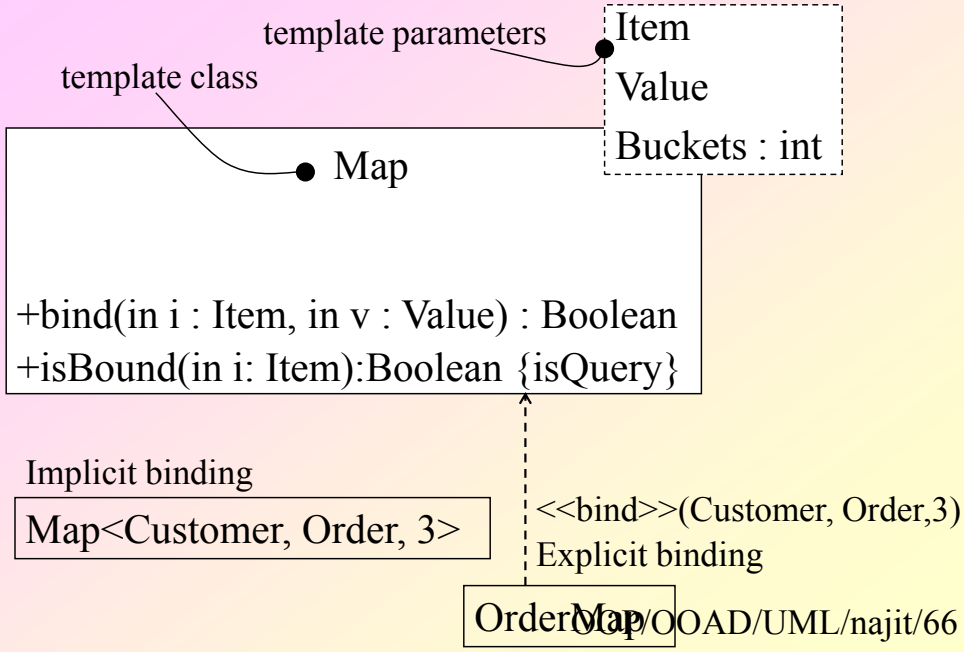
Operation Properties

- In addition to leaf property described earlier there are four other properties defined as follows
  - isQuery : It is a pure function that has no side effects (states unchanged)
  - sequential : callers must coordinate outside the object so that only one flow is in the object at a time
  - guarded : integrity of the object is guaranteed in the multiple flow of control by sequentializing all calls
  - concurrent : integrity of the object is guaranteed in the presence of multiple flows of control by treating the operation as atomic

Template Classes

- Template is a parameterized element
- Template class define a family of classes
- A Template includes slots for classes, objects and values, and these slots serve as the template parameter
- Instance of a template class is a concrete class that can be used just like any ordinary class
- The most use of template class is to specify containers that can be instantiated for specific elements.

Template Classes



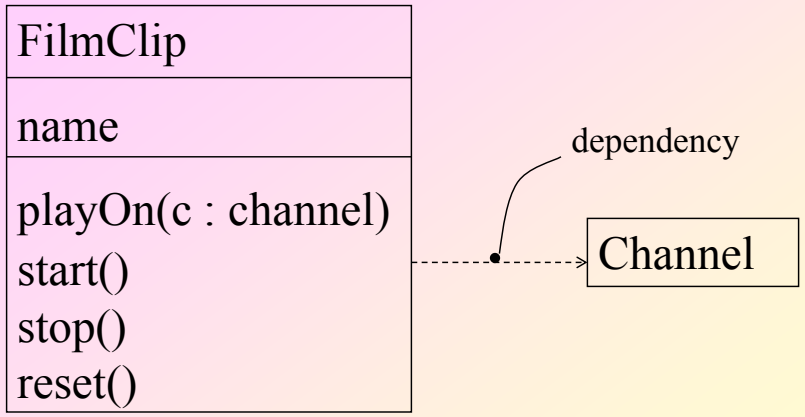
Stereotypes for Class

- The UML defines four standard sterotypes that apply to classes
  - Metaclass : specifies a classifier whose objects are all classes
  - Powertype : specifies a classifier whose objects are the children of a given parent
  - Stereotype : classifier is a stereotype that may be applied to other elements
  - Utility : specifies a class whose attributes and operations are all class scoped

Relationships

- A relationship is a connection among things
- A relationship is rendered as a path, with different kinds lines
- Dependency
  - It is a using relationship that states that a change in specification of one thing may affect another thing that uses it; but not necessarily the reverse
  - It is rendered as a dashed directed line

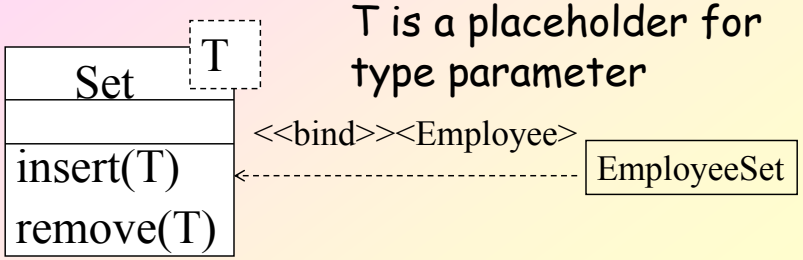
Dependency



➢Dependencies will be used in the context of classes to show that one class uses another class as an argument in the signature of an operation

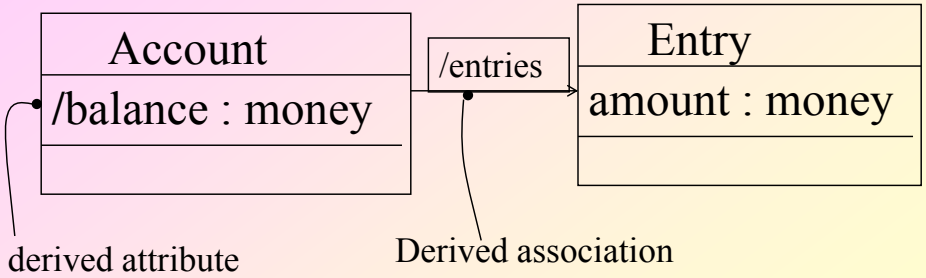
Dependency

- There are 8 stereotypes that apply to dependency relationship
  - 1. Bind : specifies that the source instantiates the target template using the given actual parameters



Dependency

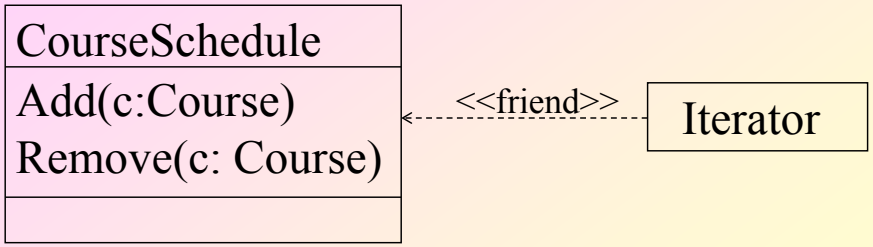
  - 2. Derive : specifies that source may be computed from target



{balance = sum of amount of entries}

Dependency

- 3. friend : specifies that source is given special visibility into the target

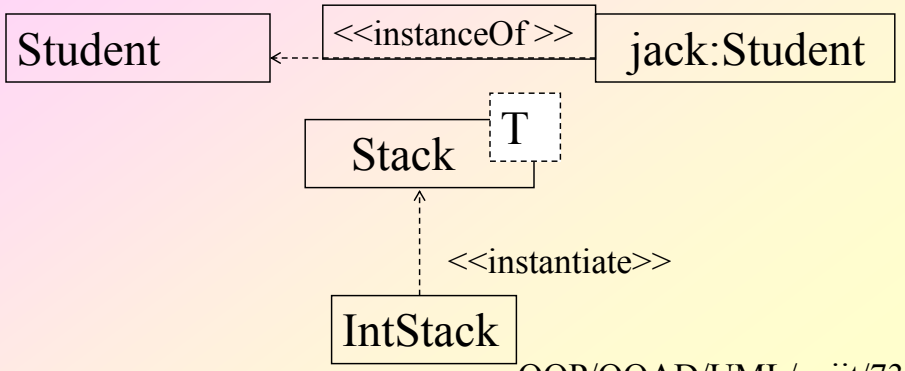


Iterator can visualize CourseSchedule

But the reverse may not be true

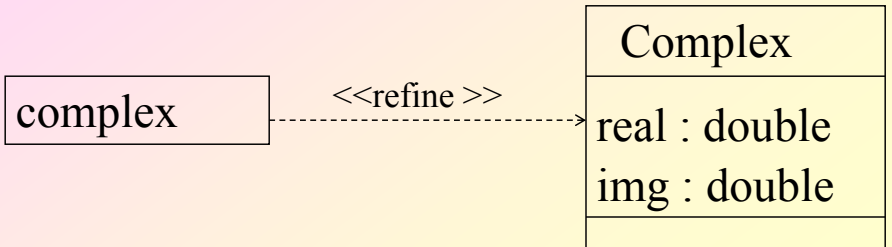
Dependency

- 4. instanceOf : specifies that source object is an instance of target classifier
- 5. Instantiate : specifies that the source creates instances of the target



Dependency

- 6. powertype : specifies that target is a powertype of the source;
  - A powertype is a classifier whose objects are all the children of a given parent
- 7. Refine : specifies that source at a finer degree of abstraction than the target
- 8. Use : specifies that the semantics of the source element depends on the public part of the target



Dependency

- There are two stereotypes that apply to dependency relationship among packages
  - 1. Access : specifies that the source package is granted the right to reference the elements of the target package
  - 2. Import : it is a kind of access that specifies that the public contents of the target package enter the flat namespace of the source, as if they had been declared in the source

Dependency

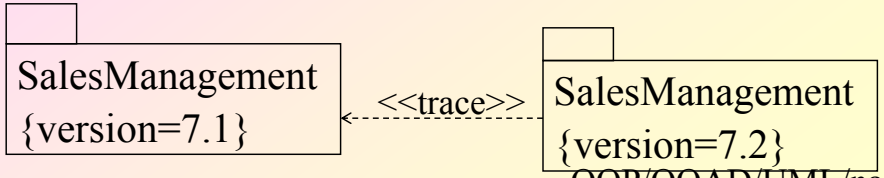
- There are two stereotypes that apply to dependency relationship among use cases
  - 1. external : specifies that the target use case extends the behaviour of the source
  - 2. include : specifies that the source use case explicitly incorporates the behaviour of another use case of a location specified by the source

Dependency

- There are three stereotypes that apply to objects
  - 1. become : specifies that the target is the same object as the source but at a later point in time and with possibly different values, sate or roles
  - 2. call : specifies that the source operation invokes the target operation
  - 3. Copy : specifies that the target object is an exact, but independent copy of the source

Dependency

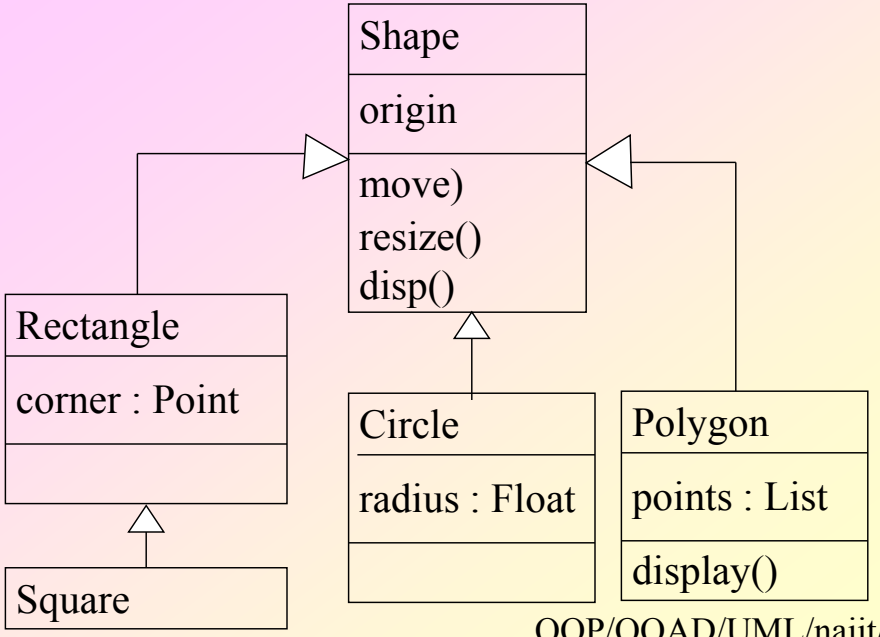
- There is one stereotype that apply to state machines
  - Send : specifies that the source operation sends the target event
- There is one stereotype that apply to subsystems
  - trace : specifies that the target is an historical ancestor of the source



Generalization

- It is a relationship between a general thing (super type) and a more specific kind of that thing (sub type)
- Objects of the child is substitutable for the parent, but not the reverse
- An operation of a child that has the same signature as an operation in a parent overrides the operation of the parent
- It is rendered as a solid directed line with a large open arrow head

Generalization

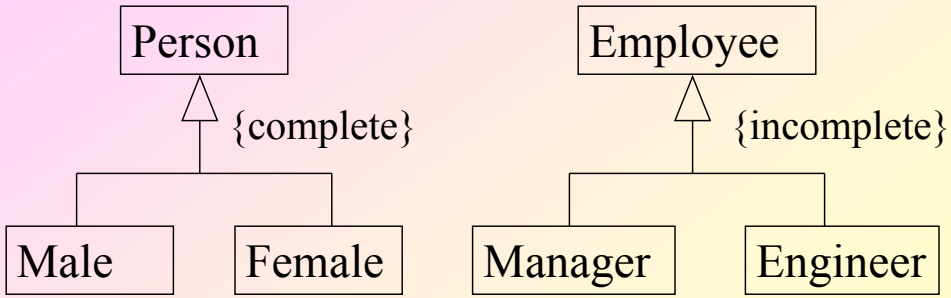


Generalization

- There is one stereotype that apply to generalization
  - implementation : specifies that the child inherits the implementation of the parent, but does not make public nor support its interface, thereby violating substitutability
  - It maps to private inheritance in C++
- Four standard constraints that apply to generalization
  - 1. complete : specifies that all children in the generalization have been specified and no additional children are permitted

Generalization

- incomplete : specifies that not all children in the generalization have been specified and additional children are permitted



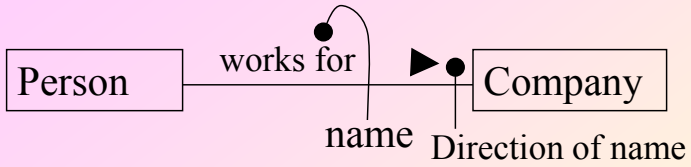
Generalization

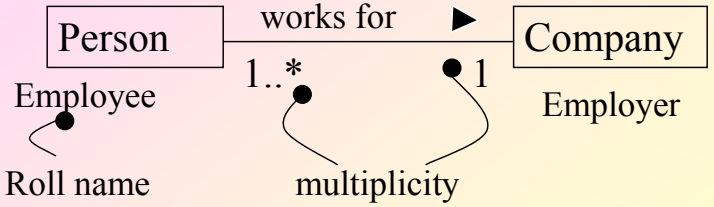
- Rest two applies for multiple inheritance
  - 3. disjoint : specifies that objects of parent may have no more than one of the children as a type
  - 4. Overlapping : specifies that objects of parent may have more than one of the children as a type
  - More than one subtype can be a type for an instance

Association

- It is a structural relationship that specifies that objects of one thing are connected to objects of another thing
- An association has name, role and multiplicity
  - Name : used to describe the relationship
  - Role : Each class plays a role in an association
  - Multiplicity : it tells how many objects may be connected across an instance of an association

Association

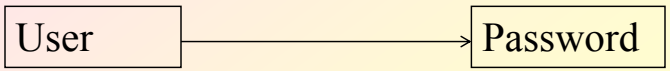




Association

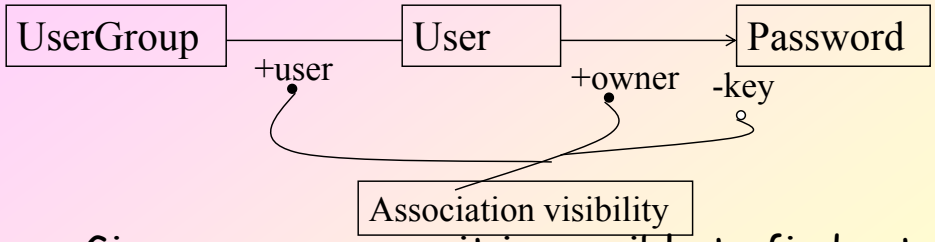
- Properties of Association
  - Navigation : Given an association between two classes, it is possible to navigate from object of one kind to object of other

  - i.e. Given a book you can find out the library and vice-versa
  - If not specified, then an association is bi-directional



Association

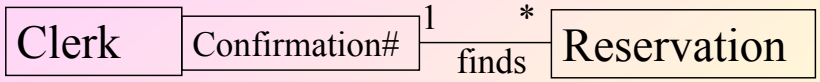
- Visibility : Given an association between two classes, objects of one class can see and navigate to objects of the other.



- Given a user group, it is possible to find out a user but not the password
- This visibility can be limited across association by giving a visibility symbol to a role name

Association

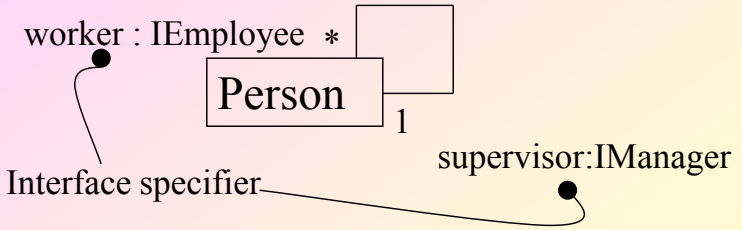
- Qualification : It is required to choose the object at the other end in case of a one-many relation



- I.e. the first class has to rely on a specific attribute to find the target object
- The ID information is called the qualifier

Association

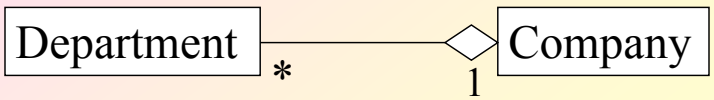
- Interface specifier : In the context of association, as source class may choose to present only one part of its face to the world



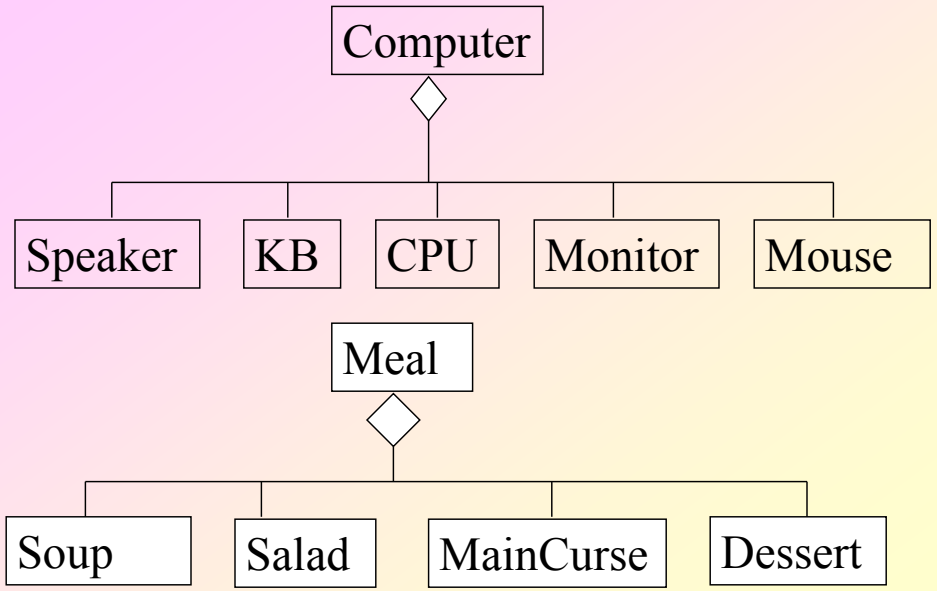
- A person may realize many interfaces like Imanager, Iemployee, and so on …
- Person in the role of supervisor presents only Imanager face to the worker.

Aggregation

- It is a whole/part relationship
- one class represents a larger thing and it consists of smaller things (has a)
- It is a special kind of association and is specified by plane association with an open diamond at the whole end

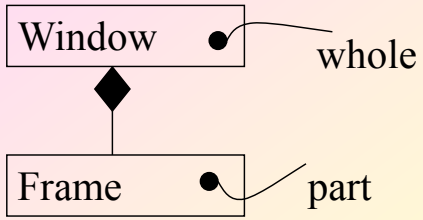


Aggregation



Composition

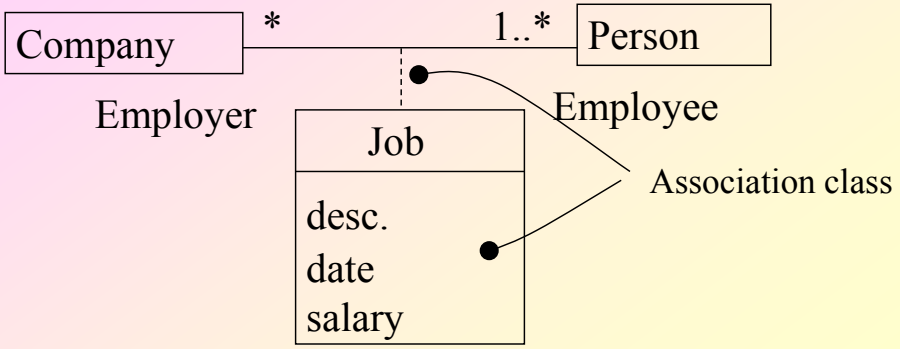
- Composition is a special form of aggregation
- Composite must manage the creation and destruction of its parts



- When an window object is destroyed, the frame objects is also destroyed

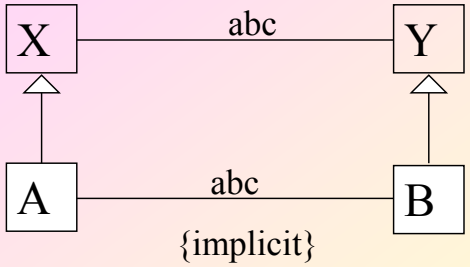
Association classes

- In an association between two classes, the association itself might have some properties



Association

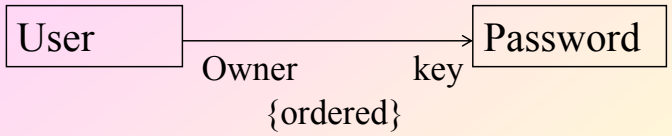
- There are five constraints that apply to association
  - Implicit : specifies that the relationship is not manifest but, rather, is only conceptual



- Association between two base classes specify that same association between children of these classes

Association constraints

- Ordered : specifies that the set of objects at one end of an association are in an explicit order



- Passwords associated with the user might be kept in a least-recently user order
- Changeable : links between objects may be added, removed and changed freely

Association constraints

- Addonly : new links may be added from an object
- Frozen : A link, once added from an object, may not be modified or deleted

UNIFIED MODELING LANGUAGE
Silicon ...beyond teaching

Thank You

Requirements Analysis and Use-Case Diagrams

Original slides by PJ Davies, UBC, for ECE 314, Real-time Software Engineering

With Reference to:

Real-Time UML by BP Douglass.
ISBN 0-201-65784-8

How are design ideas communicated in a team environment?

- If the software is large scale, employing perhaps dozens of developers over several years, it is important that all members of the development team communicate using a common language.
- This isn't meant to imply that they all need to be fluent in English or C++, but it does mean that they need to be able to describe their software's operation and design to another person.
- That is, the ideas in the head of say the analyst have to be conveyed to the designer in some way so that he/she can implement that idea in code.
- Just as mathematicians use algebra and electronics engineers have evolved circuit notation and theory to describe their ideas, software engineers have evolved their own notation for describing the architecture and behaviour of software system.
- That notation is called UML. The Unified Modelling Language. Some might prefer the title Universal Modelling language since it can be used to model many things besides software.

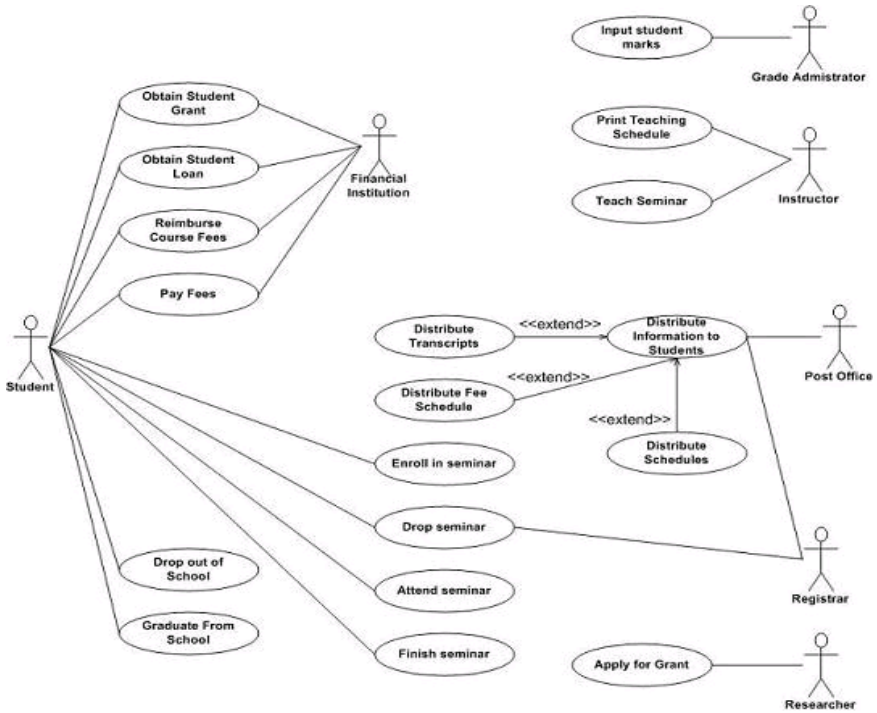
What is UML ?

- UML is not a language in the same way that we view programming languages such as ‘C++’, ‘Java’ or ‘Basic’.
- UML is however a language in the sense that it has syntax and semantics which convey meaning, understanding and constraints (i.e. what is right and wrong and the limitations of those decisions) to the reader and thereby allows two people fluent in that language to communicate and understand the intention of the other.
- UML represents a collection of 13 essentially graphical (i.e. drawing) notations supplemented by textual descriptions designed to capture requirements and design alternatives. You don’t have to use them all, you just chose the ones that capture important information about the system you are working on.
- UML is to software engineers what building plans are to an architect and an electrical circuit diagrams is to an electrician.
- **Note**: UML does not work well for small projects or projects where just a few developers are involved. If you attempt to use it in this environment it will seem more of a burden than an aid, but then it was never intended for that. It works best for very complex systems involving dozens of developers over a long period of time where it is impossible for one or two people to maintain the structure of the software in their head as they develop it.

What are the 7 most important diagram types?

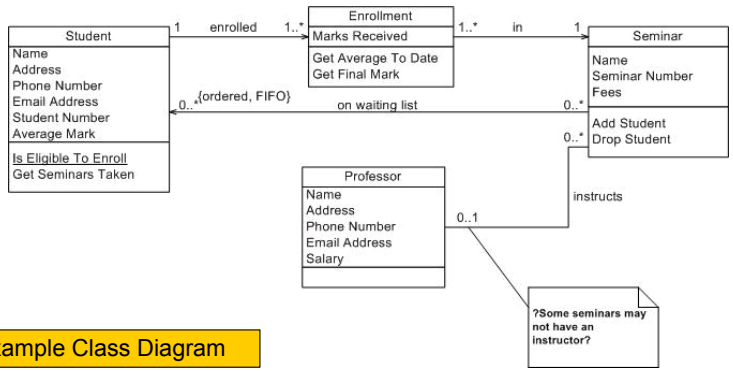
- Use Case Diagrams:
  - A simple but very effective model used during the analysis phase for analysing requirements through the process of exploring user interactions with the system.
  - The process involves documenting
    - Who initiates an interaction,
    - What information goes into the system,
    - What information comes out and
    - What the measurable benefit is to the user who initiates the interaction (i.e. what they get out of it).

  Requirements analysis attempts to uncover and document the services the system provides to the user.



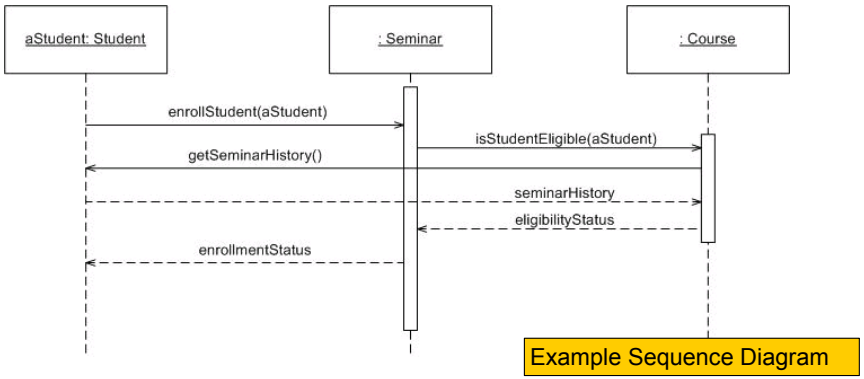
Example Use-case Diagram for a student database

- Class Diagrams:
  - A powerful tool for exploring architecture, functionality and relationships between objects in our system (i.e. instances of classes).



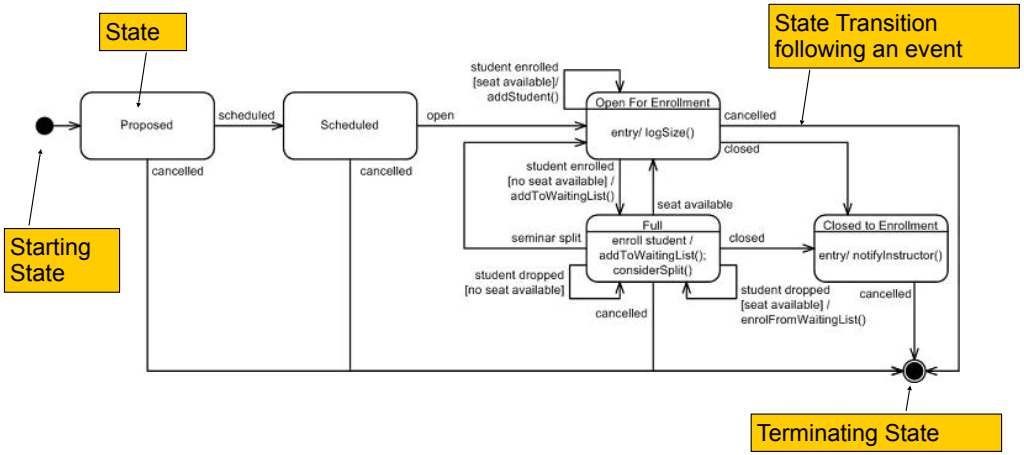
Example Class Diagram

- Sequence and Communication (*Collaboration*) Diagrams:
  - These two diagrams model the interaction of a set of collaborating objects through a process of message passing as they attempt to achieve the functionality expressed in one or more use cases.
  - In essence they model the behaviour of our system in response to inputs from the external world.



Example Sequence Diagram

- State Charts:
  - An extension of state transition diagrams.
  - Model the time dependent behaviour of objects or systems in response to messages sent to it over a period of time.



- Activity Diagrams.
  - Used during analysis to explore areas of parallelism or concurrency in the customer business model.
  - Useful in areas of business systems modelling where many processes or activities within the business are carried out in parallel, e.g. simultaneously raising an invoice while at the same time producing a delivery note and shipping the goods or order.
  - Gives an indication to the designer of things that happen in parallel and things that happen in sequence.
  - A bit like a flowchart but with parallelism and synchronisation built in.
  - Can be useful in modelling concurrent processes/threads

- Deployment Diagrams:
  - Deployment diagrams show how complex software will be deployed across a complex distribution of computers and networks, and probably have the weakest syntax of all the diagram types (i.e. almost anything goes)
  - They are essentially a sketch of the system's physical architecture e.g. computers, disk drives, GUI's, databases, hardware interfaces, networks, programs running on system X and Y.
- (see http://www.agilemodeling.com/essays/umlDiagrams.htm)

Object Oriented Analysis and the Role of Domain Experts

- During the early stages of any project development, much of your time will be spent in the analysis phase, attempting to understand '*what*' the system must do and how it interacts with the real world.
- That is, what interfaces, functionality and behaviour the system should provide for your customer.
- Such analysis is often conducted in the presence of one or more '*domain experts*', i.e. somebody who is intimately familiar with the business process or procedures that you are trying to automate (i.e. the problem domain) and can describe them to somebody else, often in a non-technical way.
- In simple terms, a domain expert is somebody who understands or knows how to do the job before it has been automated.

Example Domain Experts

- An *accountant*, could be considered to be a domain expert in the sense that he/she knows intimately the procedures, forms, rules and regulations to be followed when dealing with the Revenue Canada.
- Likewise, Architects and Electricians are domain experts when it comes to obtaining advice on building and planning regulations and electrical installations.
- In a much simpler vein, Mary sat at a supermarket checkout, or Fred assembling engines for Toyota are also *domain experts* since their experience makes them uniquely qualified to comment on the processes involved.

  In fact Mary and Fred's experience makes them especially useful, since they may well have evolved new tips-and-techniques for 'getting the job done' that may only exist inside their head (as opposed to a procedures manual) and as such, it is very important to get them working with you.

User Interaction Analysis – Use Cases

- There are a number of different techniques that can be used to uncover this expertise or functionality, (*see analysis techniques in SE readings*) but one of the most powerful is to concentrate on *user interactions* that will take place within the system.
- For example, consider the process of automating an antiquated library where all books, membership and loan details are stored on some kind of paper based card indexing system that is maintained manually by a librarian.
- A librarian represents one particular kind of domain expert, as he or she understands the rules and regulations governing how the library does business i.e. rules for loaning/returning books, fines associated with late returns etc.

Example Library user Interaction - Use Cases

- As a user, you expect a library to offer you the following services and the librarian, being a domain expert will know how to perform these tasks.

Library Use-cases

- Checking out a book for loan.
- Checking in a returned book.
- Checking if a book is in stock and where to find it.
- Reserving a book that is currently out on loan.
- Dealing with payment of overdue fines.
- Adding new members to the library.
- Deleting old members from the library.
- Dealing with changes of members details e.g. name address etc.

- In this case the librarian is part of the system that we are trying to automate and he/she may or may not be present in the automated version or at the very least he or she may find their job has changed
- So we need to interview him/her to gain the knowledge inside their head.

User Interaction Analysis – Use Cases

- Likewise a head librarian (*who is a more specialised librarian*) represents another kind of a domain expert but from a different perspective.

  - Adding new copies of a book to the library.
  - Deleting old copies of a book from the library.
  - Issuing 'Book Overdue' letters and fines.

- Each of the previous bulleted points represents in UML terminology, a specific '*Use-Case*'.

Definition of a Use-Case

- A process or procedure, describing a *user's interaction* with the system (e.g. library) for a *specified, identifiable purpose.* (e.g. borrowing a book).

- As such, each Use-Case describes a step-by-step sequence of operations, iterations and events that document

  - The Interaction taking place.
  - The Measurable Benefits to the user interacting with the system.
  - The Effect of that Interaction on the system.

- It is important to document these use-cases as fully as possible as each use-case captures some important functionality that our system will have to provide in the automated version of the library.

Documenting an Example Use-case

- Let's consider one of the library 'Use-Cases' given previously, e.g. borrowing a book, and document the step-by-step interaction that takes place between a user (*the library member or person wishing to borrow the book*) and the System (*in this case the librarian with his/her card indexing system which represents the system in its current manual state – i.e. prior to automation*).

- A simple statement of the overall objective of this use-case is often the place to start and could be obtained by interviewing a domain expert to uncover what he/she knows about the rules and procedures for carrying out that task manually.

- Initially we start off by seeking only to define the objectives of the use-case, from the point of view of the user. i.e. who is the user, what interaction takes place from their perspective and what benefits the user gets from that interaction?

- How much detail you capture in this initial statement of objective is down to the individual, as analysis is an iterative process which we revisit many times before we are happy with the results of it.

General Statement of Objective - Use-Case "Borrow Book"

- *The member identifies him or herself to the librarian and indicates which books they wish to borrow.*

- *If it is acceptable for them to borrow these books, i.e. they are not marked "for reference only", or the number of books on loan to the customer is less than some predetermined maximum, then the books are loaned to the customer for a specified loan period.*

- *The members loan record is updated to reflect the loaned books.*

- *The libraries card index system is updated to show who has borrowed the books.*

- Once a general statement of objective has been documented, we could attempt to 'flesh-out' the detail by re-interviewing all interested parties, and attempt to expand our understanding of these objectives in more detail perhaps including the library's business logic.

More Detailed Description - Use-Case "Borrow Book"

Use-Case: Borrow Book

- The borrower/member identifies himself or herself to the librarian using their membership card.
- The borrower/member presents one or more books to the Librarian.
- The Librarian checks the books to make sure they can be loaned.
- The Librarian checks the membership card to make sure it is valid.
- The Librarian looks up the member's records in his/her card indexing system and checks that the number of loaned books will be less than 6 (the maximum that can be loaned at any time to a library member).
- If acceptable, each book is then stamped with the appropriate return date (2 weeks from today).
- Each book has its identifying card removed from the inside cover.
- The Librarian updates the member's loan details by placing the identifying cards into that members record maintained by the library.

End

- Once we have a good understanding of how users borrow books and the libraries business logic and rules, we can then think about how a machine may be able to automate some of these procedures, e.g. Bar codes to identify members and books, a database to replace the card indexing system and maybe even the librarian (*either completely or in part*).

Exercise:

- See if you can describe overall objectives and detailed use-case descriptions for the other use cases associated with the library system. In particular identify
  - Who are the users that initiate the interaction,
  - What are the benefits to the user from that interaction and
  - What is the effect on the system, i.e. how is it updated or changed by this interaction.
- Some other use cases:
  - Checking in a returned book.
  - Checking if a book is in stock and where to find it.
  - Reserving a book that is currently out on loan.

Example 2. A Cash Dispenser/ATM.

- As another example of use-case analysis, let's attempt to identify a user interaction with a cash dispenser or ATM.
- In essence, this system (a completely automated system, as opposed to a manual one) provides the following functionality.
  - Request Cash
  - Request Balance
  - Request Statement
  - Request Cheque book
- Let's take the 'Request Cash' use-case and identify the interaction that takes place between user (a person with an ID card wishing to borrow money) and the system (the cash dispenser/ATM).

General Statement of Objective - Use-Case "Request Cash"

- *The user identifies him/her self to the system and requests a withdrawal for an amount of cash.*
- *A check is made, to make sure their account would remain in credit after the withdrawal, and if so, they are dispensed the cash and their account is debited accordingly.*

More Detailed Description - Use-Case "Request Cash"

Use Case Request cash

- The user inserts their ID card into the system.
- The system reads the magnetic strip from the card to identify user & account.
- The system contacts the banks central computer to request the PIN number for the card and their account details.
- The system prompts the user to enter their PIN.
- The user enters their PIN.
- If PIN is authenticated the user is prompted for the amount of the withdrawal. If not, the card is returned to the user with an appropriate failed identification message.
- The system prompts for the amount of the cash withdrawal.
- The user enters the amount of the cash withdrawal.
- The system checks with the banks central computer to ensure that the user has sufficient funds to make the cash withdrawal.
- If there are sufficient funds, the cash is dispensed and the customer's account at the Bank Central Computer is debited accordingly, otherwise an appropriate "insufficient funds" message is displayed
- The card is returned to the user and a receipt is printed.

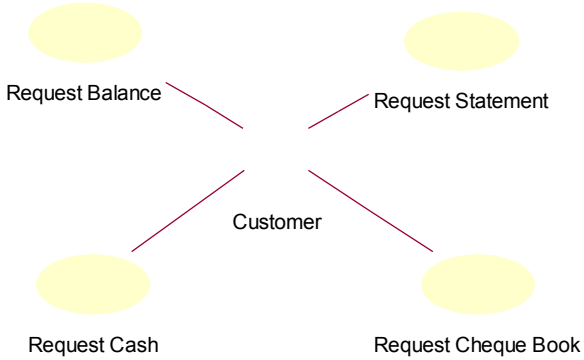
End

Question: How much detail should we include in a use-case? What about descriptions of data, e.g. PIN, Magnetic Strip, Account Details etc, are these important to document?

Exercise: See if you can describe in detail the other use cases for an ATM.

Capturing Use-Cases: The Use Case Diagram

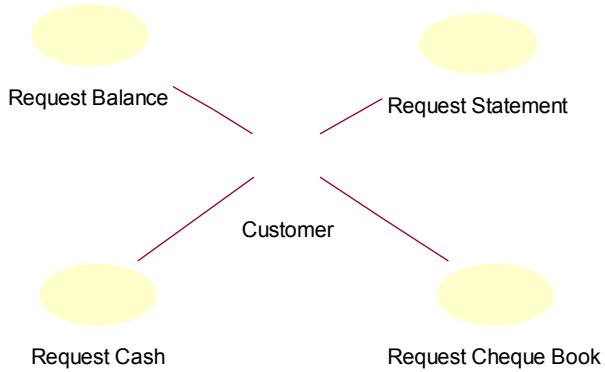
- Use cases are captured in UML on a '*Use-Case Diagram*', as shown below.
- You will notice that it is a ***very simple diagram***, involving just two symbols, a stick figure referred to as an 'actor', and a named oval representing each of the identified use-cases.



ATM Use-Case Diagram

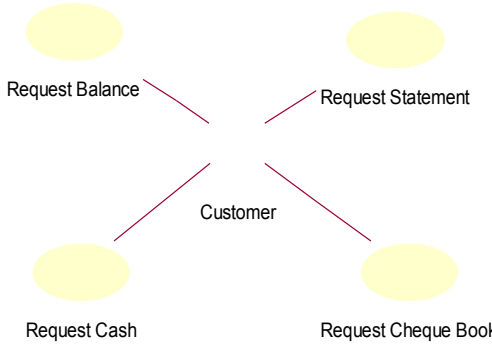
What does the Oval represent?

- The oval represents a unique *use-case* which attempts to capture high level requirements or functionality that our system must provide to its users. As such, all use-cases must be initiated by Actors.
- Each use-case is documented elsewhere with a detailed description of the interaction between user and system and the benefits to the user.



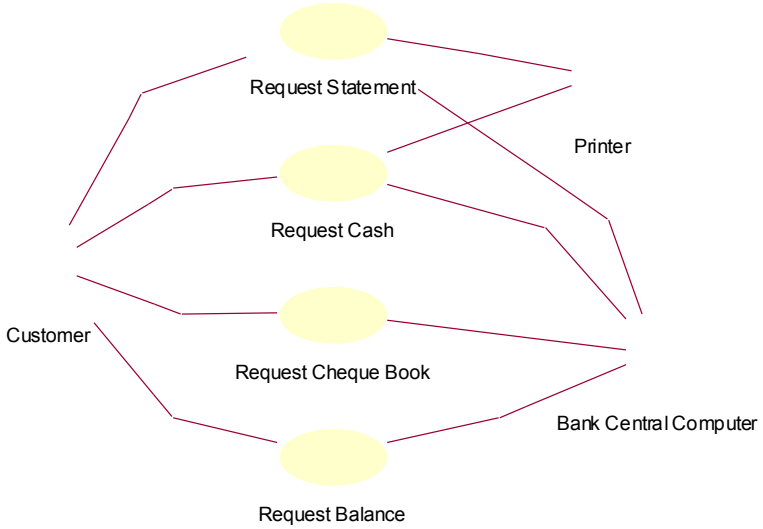
What is an Actor?

- An actor represents an *external entity* outside the domain of the system we are modelling. Most commonly these are the users of the system who initiate one or more use-cases and are typically people, but could be other hardware.
- Note that some people involved in the execution of a use-case are NOT necessarily actors. For example, we would probably not show people that happen to *take part* in the execution of a use case as an actor, (such as a librarian or the staff who refill the ATM with cash).
- The UML is quite clear about this, an Actor must be somebody/something that initiates an interaction with the system and gets some measurable benefit from that interaction.



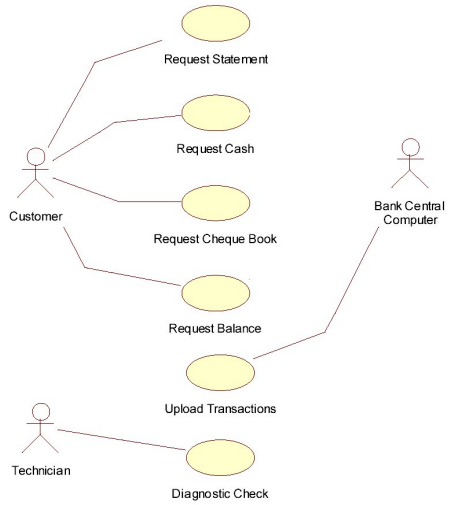
Would we show the Bank Computer and Printer as Actors?

- Possibly, but because they do not *initiate* a use-case, the UML refers to them as "secondary" actors, a bit like a librarian in an automated library system, he or she is *involved* when a user (*the primary actor*) borrows a book, but does not initiate the borrowing.



What if a computer did initiate a use-case?

- Suppose another use-case exists which allows the bank central computer to request the ATM upload details of all the transactions that had taken place that day.
- Suppose also a new use case called '*Diagnostic Check*' which is run by a new actor called Technician, which involves the use of the Bank Central Computer.



A Use case diagram looks so simple, why bother with it?

- Firstly it shows the BIG-PICTURE without getting bogged down in the details of design and implementation (i.e. It's pretty much independent of hardware, OS, GUI, programming language, databases, networks etc) so it focuses on what needs to be done, not on how it will be done.
- Secondly, the customer can easily relate to a use-case diagram and identify the major objectives of their business within it.
- This means that any major or miss-understood functionality required from the system is less likely to be overlooked during analysis (very important) or incorrectly interpreted.
- From a developers point of view they can immediately assess the functionality required and assess the risk involved in implementing each use case. Resources, hiring and training can then be allocated appropriately as part of the project planning.

- The software development manager can also begin to plan delivery schedules, estimate costs, hire new developers, buy the tools and training needed to successfully deliver the product.
- Because modern software development follows an *iterative* approach, each new release of the software can be based around the implementation of one or more use-cases.

  Thus it is easy to identify what functionality is required with each new release.

  In fact, the customer should be actively involved in the planning and release process by prioritising the use cases in terms of must-haves and nice-to-haves, thus they get to chose the functionality of each new release.

Use-Case Scenarios

- When we talk about use-cases, we always end up discussing scenarios.
- In essence, a scenario is a *specific instance* of a use case that is played out between actor and system at run time. What does this mean?
- Well take for example the ATM *Request Cash* use-case. What *could* happen when a particular customer comes to make a cash withdrawal.
- Well ideally, we would like the user to enter the correct PIN and have sufficient money in their account to make the requested withdrawal.
- This is certainly one *particular scenario* that could be played out between user and system and is probably the primary or most commonly acted out scenario for this use case. Certainly it is the one the bank had in mind when it decided to offer the cash withdrawal functionality inside their ATM.

- However, it is easy to envisage a different scenario whereby the customer incorrectly enters their PIN and the transaction is aborted.
- This is a different use-case scenario since the user interaction and outcome are not the same as someone who interacts and successfully obtains money.
- Put simply, you will probably have a scenario for every 'what-if' type question that can be posed during analysis. For example
  - What if the users PIN is incorrectly entered?
  - What if the user has insufficient funds in their account?
  - What if the cash dispenser cannot read the cards magnetic strip?
  - What if the cash dispenser is out of money?
  - What if the bank central computer is off-line?

- It's important to realise that scenarios are NOT Errors since they may well reflect important business logic and rules of operation for the customers business and thus the system must be made aware of the different outcomes and be able to deal with them.
- A scenario then captures the many different possible interactions and outcomes that could occur when executing a specific use-case.
- Put another way, a use-case binds together a set of scenarios that a user could face when interacting with the system for a specific goal, objective or aim.

Documenting Scenarios in a Use Case

- There are many different ways to document scenarios in a use-case.
- The important thing obviously is to document all those 'what-happens-if' type situations that could arise so that the developers implement them. Don't get hung up trying to evolve a clever formal procedure for documenting them, just chose the one that best captures the type of scenario you are faced with documenting.

- One simple way to document a scenario is to use structured pseudo-code in your use-case description as shown (next slide) for the cash dispenser use-case 'Request Cash', although other means are acceptable to for example
  - Formulas (e.g. cost of heating a home based on square feet)
  - Look up tables: A mapping of inputs to outputs (e.g. income tax bands (e.g. $10-25k = 20%, $25-50k = 25% etc.)
  - Algorithms (description of step-by-step method for working something out)
  - Flowcharts (describing business rules etc.)
  - Decision Trees: (a cross between a look up table and flowchart)
- Ask yourself one simple question. Is my description reasonably unambiguous, complete, and understandable by others? If so, then it's good enough (*at least to start with*), it doesn't have to be perfect, (*analysis like most other software development activities, is an iterative technique*).

Documenting Simple Scenarios in the Use-Case Request Cash

Start of Primary scenario/transaction

1. The user inserts their ID card into the system.
2. The system reads the magnetic strip from the card.
3. If the system cannot read the card then **<<Scenario 1>>**
4. The system contacts the banks central computer to request the PIN number for the card and their account details.
5. If bank central computer cannot access users account then **<<Scenario 2>>**
6. The system prompts the user for their PIN.
7. The user enters their PIN.
8. If PIN cannot be authenticated **<<Scenario 3>>**
9. The user is prompted for the amount of the withdrawal.
10. The user enters the amount of withdrawal.
11. The system checks with the banks central computer
12. If the user has insufficient funds **<<Scenario 4>>**
13. The cash is dispensed and the customer's account at the Bank Central Computer is debited with the withdrawal amount.
14. The card is returned to the user and a receipt issued.

End-Of-Transaction

Scenario 1: The users card is returned. End of Transaction

Scenario 2: The users card is returned. End of Transaction

Scenario 3: The user is given two more attempts to enter a correct PIN. If this fails the card is kept and the transaction ends. Otherwise resume primary scenario.

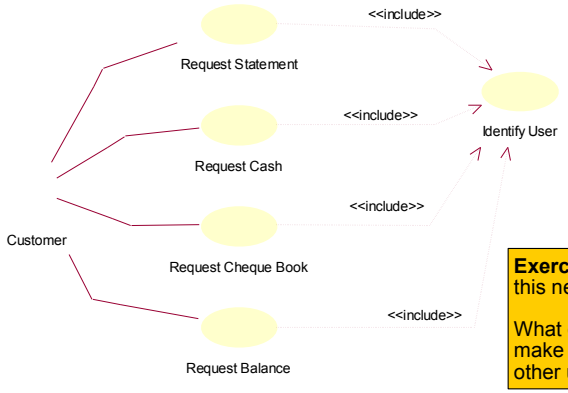
Scenario 4: The user is given the opportunity to enter a lesser amount or cancel the transaction. If cancel is chosen, the card is returned and the transaction ends. If the lesser amount is acceptable then resume primary scenario.

Use Case Relationships – Includes

- When designing use-cases it is sometimes apparent that there exists some commonality or replication between the steps involved in the execution of one or more use cases. For example take the ATM once again.
- In each one of the four use cases below:

  - *Request Cash*
  - *Request Balance*
  - *Request Statement*
  - *Request Cheque Book*

  the user is required to insert their ID card and enter their PIN, which is then verified by the bank central computer.
- Rather than duplicate this common user interaction within each of the above four use-case descriptions, we might extract it and chose to represent it with a mini-use-case called *'identify user'* whose functionality is *included* as part of the other four use-cases.
- Such an '*includes*' relationship is shown in the simplified diagram (next slide). The dashed line indicates a dependency relationship i.e. one use-case depending upon another. The arrow points to the use-case that will be included, thus



Exercise: See if you can document this new mini use-case.

What changes would you now make to the documentation of the other use cases?

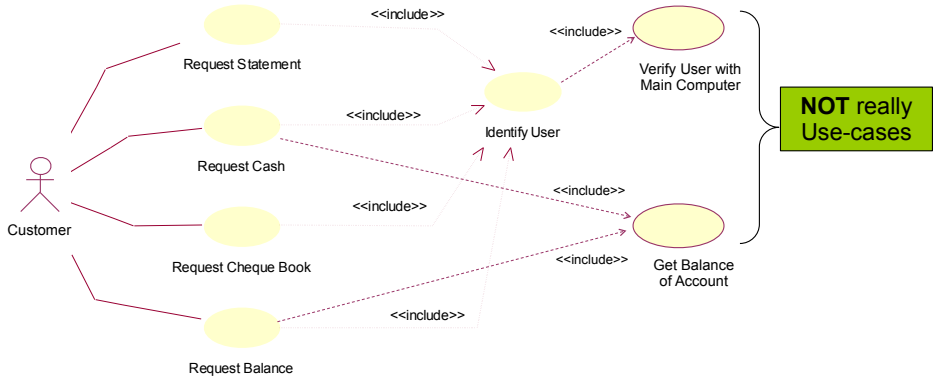
VERY Important Note:

- Don't fall into the trap of treating use-cases like functions in a program where you continually apply hierarchical decomposition to break them down into smaller entities.
- Use-cases, (even mini ones like 'Identify-User'), should always involve some documentable interaction between the actor and the system and be able to describe the measurable benefit to the user and effect on system, otherwise they simply aren't use-cases.
- Although not always a definitive question, try asking yourself if the customer would be happy to pay for a release of the software involving your new use case. If not then it's probably not very interesting to them because it's an implementation detail or functionality that is simple just a part of a bigger use-case and probably not a use-case in its own right.

- For example, The mini-use case below '*Verify User with main computer*' may be something that is ***essential*** to the success of the use-case 'Identify User' and similarly, "Get Balance of Account" may be common to both request cash' and 'request balance',

  but neither of these two should be shown as mini-use cases because

  1. There is no user interaction in either use-case.
  2. There is no direct, immediate, measurable benefit to the user from the execution of that use-case. If the user is not aware of the execution or outcome of the use-case, then it simply isn't a use-case because there **IS NO USER**.



- In summary, if a use-case does not contain any user-interaction OR, does not lead to any direct, measurable benefit for the user, then it is NOT a use-case, it is simple functionality that will eventually be embedded or hidden *within* another larger use-case and should not appear on a use-case diagram.

Documenting 'Includes' relationships in a Use-Case: Request Cash

Start of Primary scenario/transaction

1. Include *Identify User* *(a prerequisite or precondition for the execution of this use-case)*
2. The user is prompted for the amount of the withdrawal. **note the assumption of success**
3. The user enters the amount of withdrawal.
4. The system checks the account balance with the banks central computer
5. If the user has insufficient funds **<<Scenario 1>>**
6. The cash is dispensed and the customer's account at the Bank Central Computer is debited with the withdrawal amount.
7. The card is returned to the user and a receipt issued.

End-Of-Transaction

Scenario 1: The user is given the opportunity to enter a lesser amount or cancel the transaction. If cancel is chosen, the card is returned and the transaction ends. If the lesser amount is acceptable, resume.

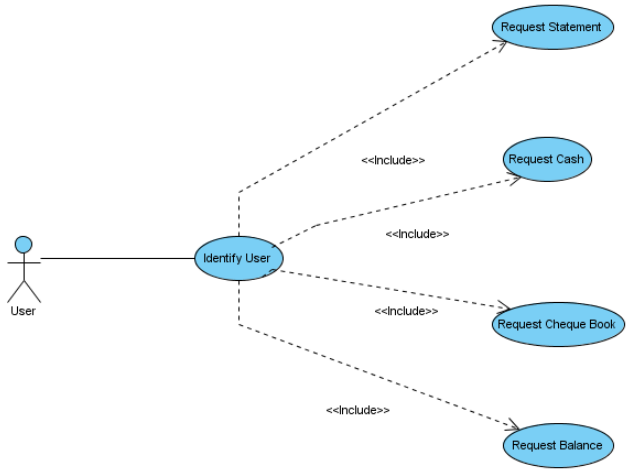
Start of Included scenario/transaction

1. The user inserts their ID card into the system.
2. The system reads the magnetic strip from the card.
3. If identification fails **<< Scenario 1>>**
4. The system contacts the banks central computer to request the PIN number for the card and their account details.
5. If bank central computer cannot access users account **<<Scenario 2>>**
6. If PIN cannot be authenticated **<<Scenario 3>>**

End-Of-Transaction

Scenario 1: The users card is returned. End of Transaction
Scenario 2: The users card is returned. End of Transaction
Scenario 3: The user is given two more attempts to enter a correct PIN. If this fails the card is kept and the transaction ends. Otherwise resume primary scenario.

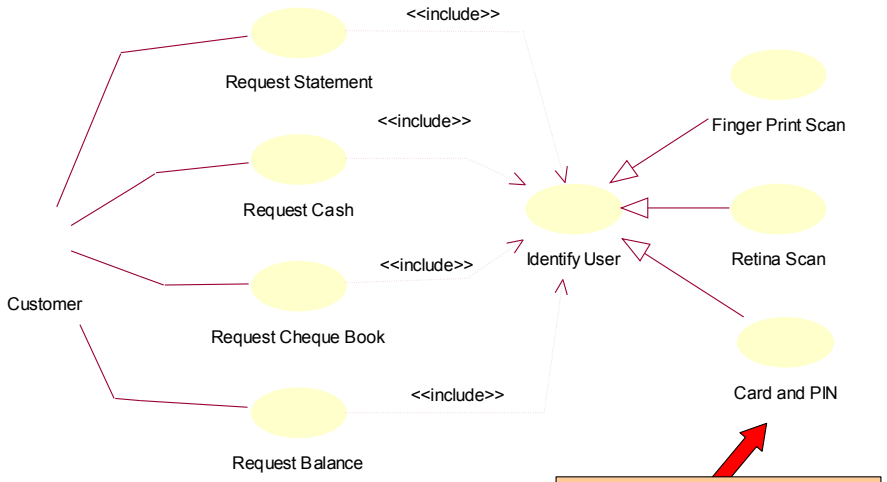
- What's wrong with this use-case diagram ?



Use Case Relationships – Generalisation

- Another useful relationship that can occur between use-cases is one of *generalisation*.
- This occurs when two of more use-cases attempt to achieve the same goal or objective but achieve it via different means.
- For example, suppose an ID card with a magnetic strip were not the only way that you could identify yourself to a ATM.
- Let's suppose that fingerprint and retina scan are also acceptable.
- One could imagine that a user might be offered the choice of which method of identification to use when they arrive at the ATM.
- Alternatively a range of different ATM models might be produced with only one of these 3 means of identification built in, thus their interaction with the user would vary but the outcome/benefits to the user would be the same, namely that the users identification is established.
- It is therefore possible to identify three different use-cases bound together by the common objective of identifying the user.

- We would represent these three use-cases using a generalisation relationship, which is shown below. You can relate this to the previous use case diagrams.



- This diagram should be interpreted to mean that there are three possible methods for identifying a user. Thus the four major use cases

  - *Request Cash*
  - *Request Balance*
  - *Request Statement*
  - *Request Cheque Book*

  will obviously invoke only one of these three means of identification. The choice may be one made by the user, or imposed upon them by the model of ATM they are using.

- When documenting generalisation use cases the *base* or *root* use-case (i.e. Identify User) should be documented in very general terms, i.e. it should just list the objectives of the use-case. For instance the following should suffice.

  "*Identify the user*"
  obtain their account details from the bank central computer

- The three specific or *derived* use-cases can be documented with specific details.

- Generalisation then is about isolating common user objectives and expressing that commonality in the base use-case.

- The various specific details of achieving that can be documented in the derived (real) use-cases.

Use Case Relationships – Extends

- The final use-case relationship is one that allows *extensions* to be made to a use-case.

- This can be used to model *optional behaviour*, particularly interesting and important *scenarios* within a use-case.

- For example, in the ATM, the use-case '*Request Cash*' contained a number of different *scenarios* that could be acted out when a customer requests cash.

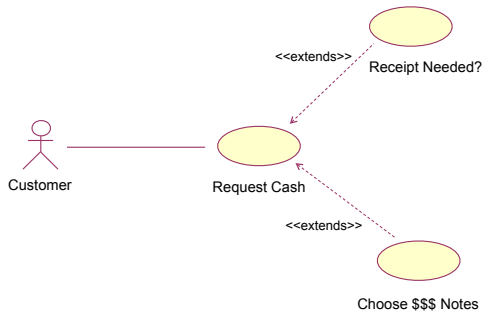
- We could of course document these scenarios with carefully chosen textual descriptions placed within the use-case documentation itself as shown previously.

- However, if the scenarios were particularly interesting (and they involved some user-interaction with a direct measurable benefit to the user) then we could document them in their own mini-use cases and add an extends relationship to their parent or containing use-case (the one documenting the primary use-case)

- **Example**: In the diagram below, we see two separate mini use-cases that extend their containing use-case. These use-cases deal with optional user-interaction that *might (or might not)* occur when their containing use case is executing. (Note the emphasis on 'might occur'. If the mini use-case always gets executed when the parent use-case runs, then the mini use-case should be modelled with an '*includes*' relationship, if it's optional, model with *extends*)

- Note the direction of the arrow from extended use-case back to parent or containing use-case. In other words "Receipt Needed" and "Choose $$$ Notes" both *extend the functionality* contained within the use-case "Request Cash".

- Of course to qualify as a use-case, the user still has to be involved with some significant interaction within the system and there must be some measurable benefit that can be identified. In the case of these two mini-use cases, the benefits to the user are that they get a paper receipt and get to choose the type of '$' notes that they are given.



Documenting a Use-Case with Extends

Use-Case Request Cash
1. . . .
2. . . .
3. If(users wants to chose type of $ notes)
4. Extends Choose $$$ Notes
5. If user chooses to have a receipt
6. Extends Print Receipt
7. . . .
8. . . .
9. The card is returned to the user and a receipt is issued.

End-Of-Transaction

- Notice how the extended use case is called just by referring to its name, this isn't a precise syntax, but its understandable and thus works as well as any other approach.
- Thus in step 5 above, the extended use-case "Print Receipt" is invoked if the user asks for a receipt.
- It's important not to get hung up on a specific syntax and notation for describing use-cases, they are not precise programming statements, they capture requirements in an written manner that both analyst and customer can understand.
- We model them more precisely using sequence diagrams.

Embedded Systems

Chapter 1:Embedded Architecture

BY
AMRUTA CHINTAWAR

Embedded Architecture

- Embedded computers
- Characteristics
- Applications
- Challenges
- Embedded memories
- ES design process
- Requirement specifications
- Architecture
- H/W and S/W components
- System Integration
- Design examples

Definition

- Embedded computing system: any device that includes a programmable computer but is not itself a general-purpose computer.
- System has a S/W embedded into Computer H/W, which makes the system dedicated for an application or specific part of an application or product or part of a larger system.

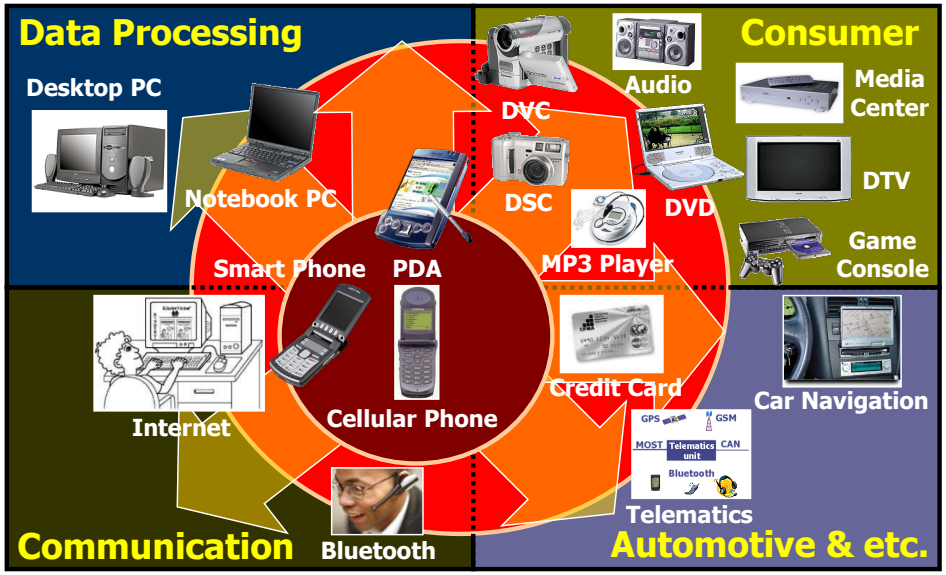
Definition

- Embedded systems (ES) = information processing systems embedded into a larger product
- Main reason for buying is not information processing

Comparison

- **Embedded Systems**
  - Few applications that are known at design-time.
  - Not programmable by end user.
  - Fixed run-time requirements (additional computing power not useful).
  - Criteria:
    - cost
    - power consumption
    - predictability
- **General Purpose Computing**
  - Broad class of applications.
  - Programmable by end user.
  - Faster is better.
  - Criteria:
    - cost
    - power consumption
    - average speed

Various Embedded Mobile Systems



Examples

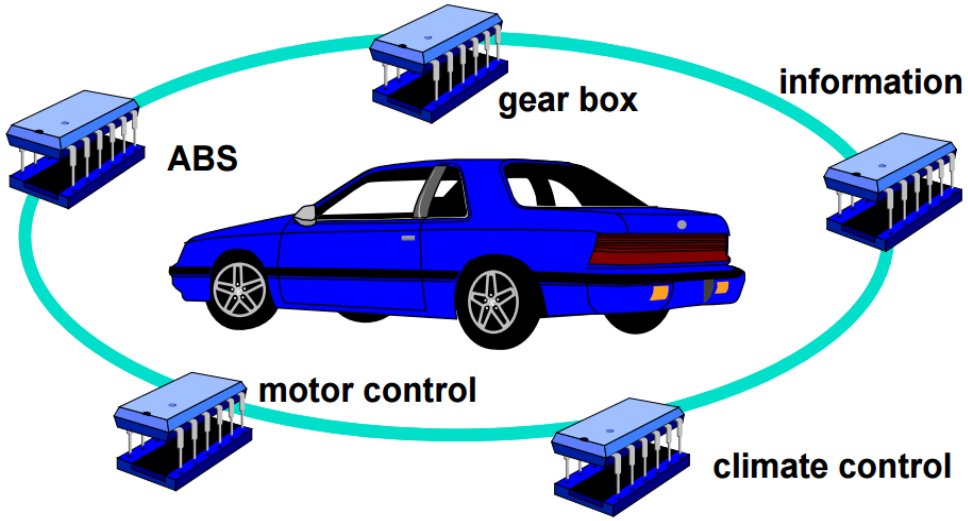
- Biomedical Instrumentation: ECG Recorder, Blood cell recorder, patient monitor system
- Communication system: pagers, cellular phones, cable TV terminals, fax and transreceiver, video games and so on.
- Peripheral controllers of a computer –keyboard controller, DRAM controller, Printer Controller, LAN controller, Disk drive Controller

Examples

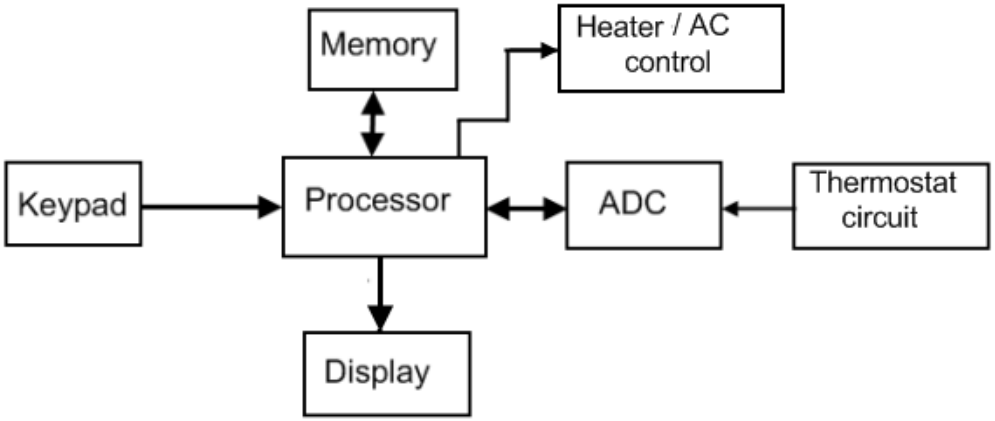
- Industrial Instrumentation- Process controller, DC motor controller, robotic system, CNC machines controller, close loop engine controller, industrial moisture recorder cum controller
- Scientific –digital storage system, CRT display controller, spectrum analyzer.

Example

Car as an integrated control-, communication and information system.

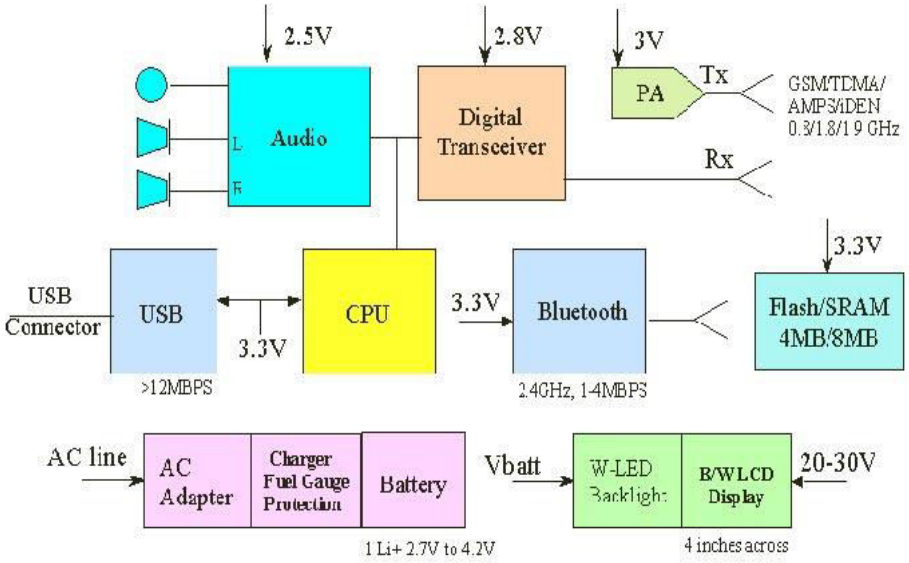


Example: Digital Thermostat



A digital thermostat as an example of embedded system

Example: Mobile phone



Characteristics

- Dedicated and sophisticated functionality
- Low manufacturing cost, low power
- Complex Algorithm
- GUI, other user interface
- Reliability
- Real Time operations
  - -latencies
  - -Deadlines
- Multirate Operations
  - -Audio
  - -Vedio

Design Metrics/Goals

- Reliability
- Power Consumption
- Cost :NRE cost & Manufacturing cost
- Weight/size
- Time: time to prototype & time to market
- Flexibility
- Performance
- Debuggability
- Safty & maintaince
- Maximum usage of resources

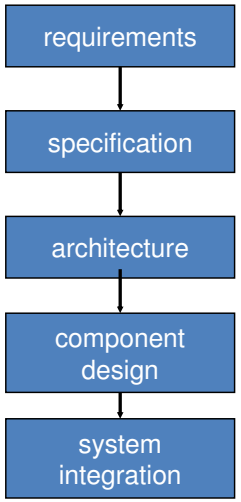
Challenges

- How much hardware do we need?
  - How big is the CPU? Memory?
- How do we meet our deadlines?
  - Faster hardware or cleverer software?
- How do we minimize power?
  - Turn off unnecessary logic? Reduce memory accesses?
- Does it really work?
  - Is the specification correct?
  - Does the implementation meet the spec?
  - Reliability in safety-critical systems

Challenges

- How do we work on the system?
  - Complex testing
    - How do we test for real-time characteristics?
    - How do we test on real data?
  - Limited observability and controllability
  - Restricted development environments
    - What is our development platform?

Level Abstraction



Architecture

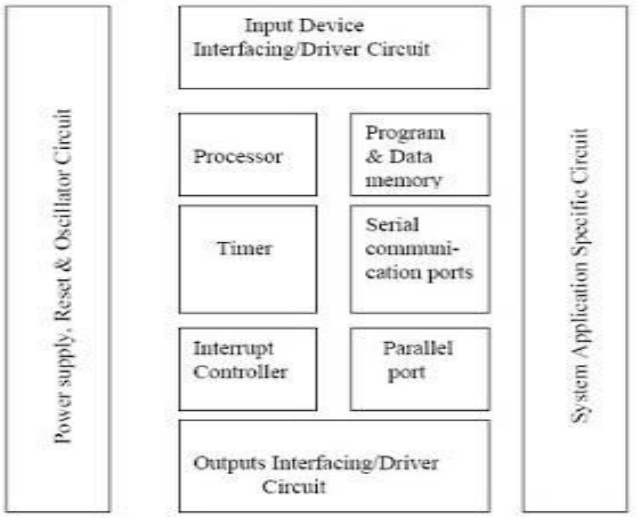


Fig. 6

Components of Embedded system

- H/W
  - **Processor**
  - **Power source and clock**
  - **Reset circuit**
  - **Memory Unit**
  - **Interrupt Handler**
  - **Linking Embedded System H/W**
  - **I/O communication Unit**
- S/W
  - **ROM image/Applications/w**
  - **Programming Languages**
  - **Device Drivers**
  - **Program Models**

Components of Embedded system

- RTOS/EOS
- S/W Tools
  - **Development tools**
  - **Simulator**
  - **Project Manager**
  - **IDE**

Types of Cores

Requirments

- **Able to solve complex Algorithm**
- **Meet Deadlines**
- **No.of Bits to be operated**
- **Bus Width**
- **Clock Frequency**
- **Performance(MIPS/MFLOPS)**

Types of Cores

- GPP: General Purpose Processor
  - -Microprocessors
  - -Embedded Processors
- ASIP: Application Specific Instruction Processor
  - -Micro controller
  - -Embedded Micro controller
  - -DSP and media Processor
  - -Network Processor
- SPP: Single Purpose Processor
  - -Coprocessors **eg:Math-coprocessor**
  - -Accelerators **eg:java acce**
  - -Controllers **eg:DMA**

Types of Cores

- **ASIC/VLSI chip: Application specific Integrated circuit**
  - -GPP/ASIP integrated with analog circuits on VLSI chip using EDA tools.
- **ASSP:Application Specific System Processor**
  - -set top box
  - -Mpeg
  - -HD tv
  - **eg:vedio Processors**
- **Multicore Processors/Multiprocessor using GPP**
  - **eg:Embedded firewall cum Router**

Power Sources,Clock and Reset Circuit

- **Power supply**
  - -Own supply
  - -Supply from System
  - -Charge Pumps
- **Clock**
  - -External Clock supply
  - -Oscillator
  - -RTC
- **Reset**
  - -Power On
  - -External/Internal Reset
  - -WDT
  - -BOR

Memory Unit,Interrupt Handlers

Memory
- ROM/EPROM/FLASH (internal/External)
- RAM
- Caches

Interrupt Handler
- External port interrupt
- I/O,Timer, RTC, interrupts
- S/W Interrupt/Exceptions

Linking ES H/W
- Multiplexers
- Decodes

I/O communication Unit

- I/O,O/P devices
  - -Sensors,
  - -actuators
  - -converters
  - -keypads
  - -displays
- Buses
  - -Parallel Buses
  - -serial buses

S/W-ROM Image, Programming Language

- ROM Image
  - -Final Machine s/w
  - -Can be compressed code or data
  - -Distinct ROM image in distinct ES
- Programming Languages
  - -Machine
  - -Assembly
  - -High Level

S/W-Device Driver, Managers

It connects external H/w with Processor

- Controlling
- Receiving
- Sending

Part of OS

- Manages device
- Initializing
- Testing
- Detecting
- Allocating port addresses

RTOS/OS

It performs functions

- Multitasking
- Scheduling
- Management
- Resource protection
- Interprocess Communication

eg: Ucos-II, Vxworks, windows CE, RT linux,QNX

Embedded Memories

- Internal RAM at µc *–SRAM used as reg, temp data, stack*
- RAM AT SoC or External RAM
- Internal/external caches at µp *-hold copy of system memory pages*
- External RAM chips *-DRAM used to hold extra data*
- Flash EPROM/EEPROM *-result stored in NV memory*
- ROM/PROM/MROM/OTP *-Application S/W, OS*
- Memory addresses at system ports *-RAM buffers*
- Memory Stick *-large storage such as audio, video*

I/O Devices

- DAC using PWM
- ADC
- LCD,LED and Touch Screen
- Keypad/keyboard/T9 keypad
- Pulse dialer
- Modem
- Transceiver
- Interrupt handler *-mechanism to handle various interrupts and also to deal with pending services*

Difference between RISC and CISC

RISC	CISC
Reduced instruction set	Complex Instruction set
Maximum instructions are single cycle(fixed size),thus supports pipelining	Variable size Instructions,so generally do not have pipelining
Orthogonal instruction set	Non-Orthogonal
Operations are performed on registers, so large no of Registers. For memory only Load and Store	Operations are performed on both registers and memory. Limited number of GPRs

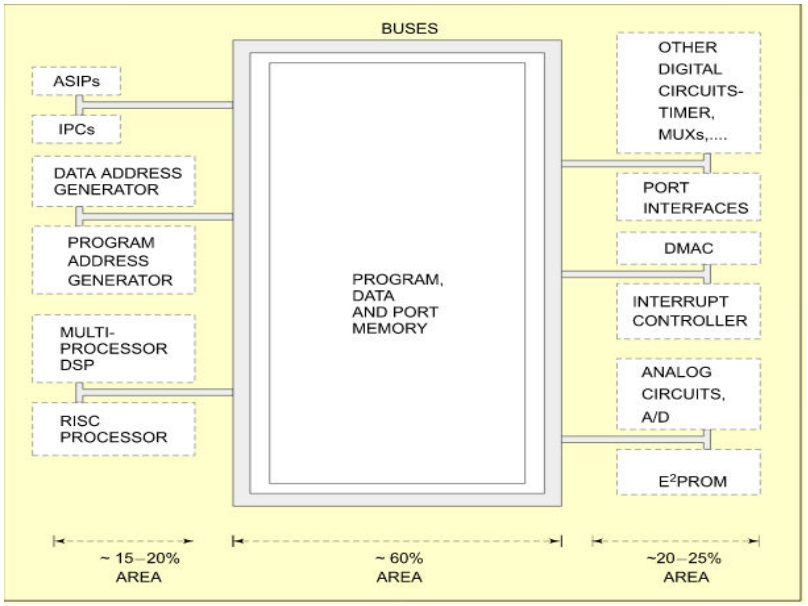
Difference between RISC and CISC

RISC	CISC
Hardwired Control unit	Microcode control Unit
Small in size with resp to die area and No.of pins	Comparatively large in size since more complex instruction needs to be implemented.
Harvard architecture	Harvard or Von-Neuman
Eg:PIC18,ARM	8051,8086

SoC

- System designed on a single chip
- Processor with all analog, digital and S/W build on a single VLSI chip

SoC



SoC

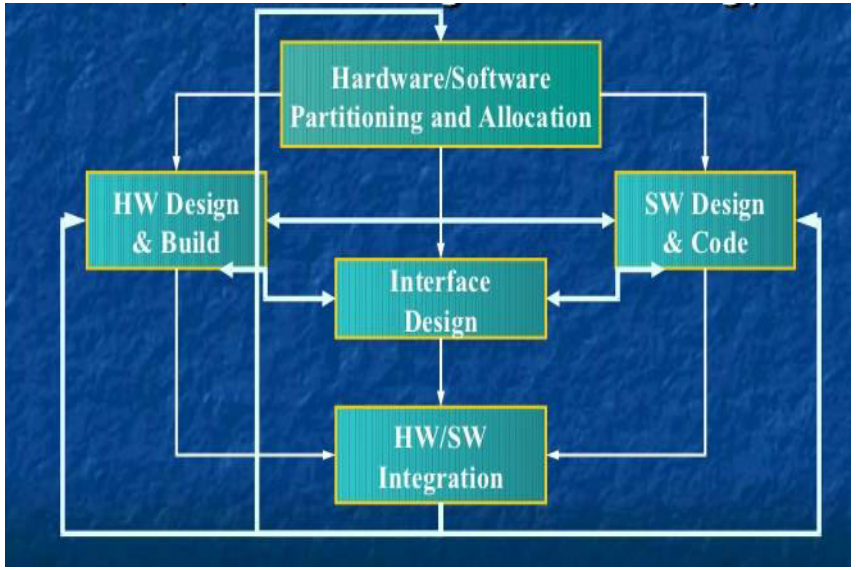
It embeds

- EGPP or ASIP
- SPP or Multiprocessor
- N/W bus protocol
- Encryption and Dycryption
- Signal processing such as FFT,DCT
- Memories
- IP-Intellectual
- PLDs or FPGA
- Accelerators or other logic and analog units

IP-Intellectual Property

- A standard source solution for synthesizing a higher level component by configuring an FPGA core/VLSI core
- Components-gate level sophistication in circuits above that of counter, register, multiplier, FLU and ALU

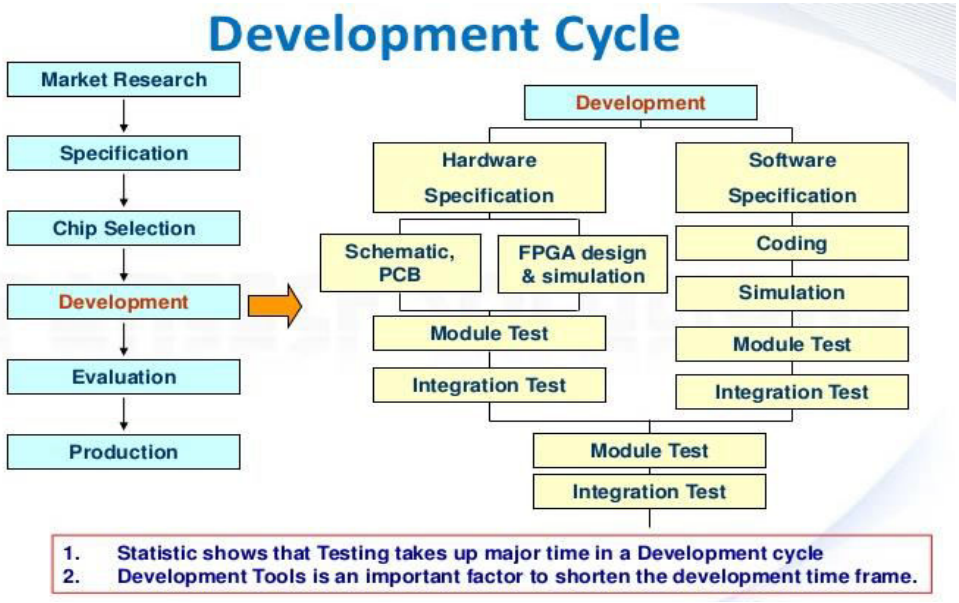
H/W and S/W Co-Design



H/W and S/W Co-Design

- Joint optimization of H/W and S/W to optimize design metrics
- S/W and H/W partitioning at early stage
- Both proceed in parallel with interactions and feedback
- System specification required
- H/W synthesis
- S/W synthesis
- Simulation
- Implementation

Development Process



Levels of Abstraction from Top to Bottom

- Requirements
- Specifications
- Architecture
- Components
- System Integration

Requirement

Complete clarity of:

- Required Purpose
- Inputs
- Outputs
- Functioning
- Design metrics
- Validation requirements for finally developed system specifications
- Consistency in the requirements

Specifications

- Clear specification of customer expectations from the product
- Needs specification for
  - H/W, eg: Peripherals, Divices,Processors and memory specifications
  - Data types and processing specifications
- Expected system behavior specifications
- Constraints of design
- Expected lifecycle specifications of the product

Specifications

- Process specifications analyzed by making list of I/Ps on event list, O/Ps on events, process activated on each event

S/W Architectural Layers

- How the different elements- data structures, data bases, algorithms, control functions, state transition functions, process, data and program flow are to be organized

- Data base and Data structure design- appropriate for given problem

Eg: tree like structure

H/W Components

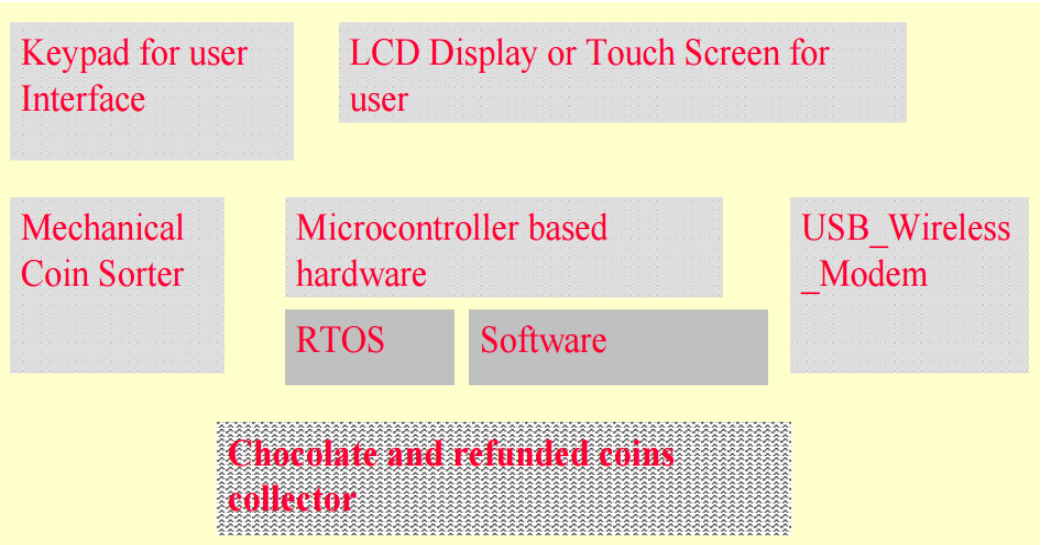
- Processors, ASIP, Single Processors
- All Types of Memory as per requirement
- Internal and External peripherals and devices
- Ports and Buses in the system
- Power sources and battery

Design Examples

- ACVM
- Smart Card
- Digital Camera
- Mobile phones
- Mobile Computer
- Set of Robots

ACVM

ACVM Block diagram



ACVM

- Coin insertion slot
- Keypad on the top of the machine.
- LCD display unit on the top of the machine. It displays menus, text entered into the ACVM and pictograms, welcome, thank and other messages.
- Graphic interactions with the machine.
- Displays time and date.

ACVM

- Delivery slot so that child can collect the chocolate and coins, if refunded.
- Internet connection port so that owner can know status of the ACVM sales from remote.

ACVM H/W

- Microcontroller or ASIP (Application Specific Instruction Set Processor)
- RAM for storing temporary variables and stack
- ROM for application codes and RTOS codes for scheduling the tasks
- Flash memory for storing user preferences, contact data, user address, user date of birth, user identification code, answers of FAQs

ACVM H/W

- Timer and Interrupt controller
- A TCP/IP port (Internet broadband connection) to the ACVM for remote control and for getting ACVM status reports by owner.
- ACVM specific hardware
- Power supply .

ACVM S/W

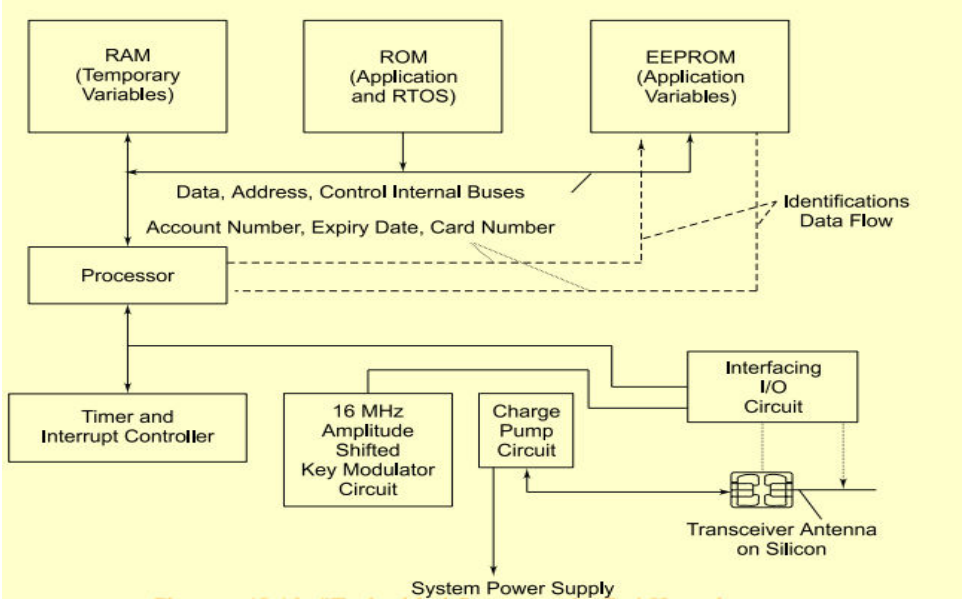
- Keypad input read
- Display
- Read coins
- Deliver chocolate
- TCP/IP stack processing
- TCP/IP stack communication

SMART CARD

SMART CARD

- Smart card– a plastic card in ISO standard dimensions, 85.60 mm x 53.98 x 0.80 mm.
- Embedded system on a card.
- SoC (System-On-Chip).
- ISO recommended standards are ISO7816 (1 to 4) for host-machine contact based cards and ISO14443 (Part A or B) for the contact-less cards.
- Silicon chip is just a few mm in size and is concealed in-between the layers. Its very small size protects the card from bending

SMART CARD



ROM

- Microcontroller or ASIP (Application Specific Instruction Set Processor)
- RAM for temporary variables and stack
- ROM for application codes and RTOS codes for scheduling the tasks
- EEPROM for storing user data, user address, user identification codes, card number and expiry date
- Timer and Interrupt controller
- A carrier frequency ~16 MHz generating circuit and Amplitude Shifted Key (ASK)
- Interfacing circuit for the I/Os
- Charge pump

ROM

- Fabrication key, Personalization key An utilisation lock
- RTOS and application using only the logical addresses.

Embedded S/W

- Boot-up, Initialisation and OS programs
- Smart card secure file system
- Connection establishment and termination
- Communication with host
- Cryptography
- Host authentication
- Card authentication
- Addition parameters or recent new data sent by the host (for example, present balance left)

Smart Card OS Special Features

- Protected environment.
- Every method, class and run time library should be scalable.
- Code-size generated be optimum.
- Memory should not exceed 64 kB memory.
- Limiting uses of specific data types; multidimensional arrays, long 64-bit integer and floating points

Smart Card OS Limitimg Features

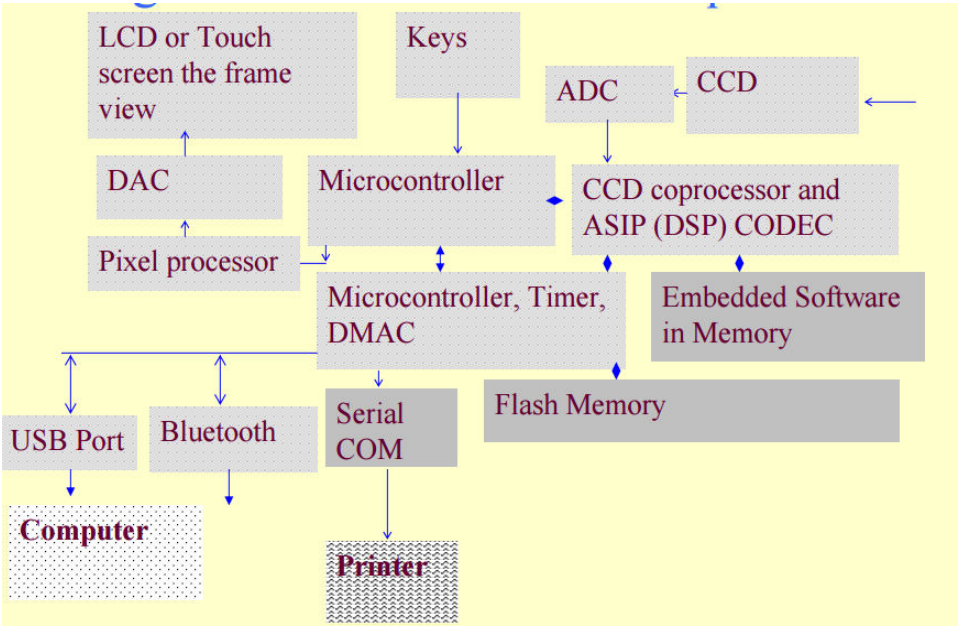
- Limiting uses of the error handlers, exceptions, signals, serialization, debugging and profiling. [Serialization means process of converting an object is converted into a data stream for transferring it to network or from one process to another. At receiver end there is de-serialization.]

Smart Card OS file System

- Three-layered file system for the data.
- Master file to store all file headers.
- Dedicated file to hold a file grouping and headers of the immediate successor elementary files of the group.
- Elementary file to hold the file header and its file data.
- Fixed-length or variable-file length management
- Classes for the network, sockets, connections, data grams, character-input output and streams, security management, digital-certification, symmetric and asymmetric keys-based cryptography and digital signatures.

DIGITAL CAMERA

Digital Camera H/W Components



Digital Camera

- 4 M pixel/6 M pixel still images, clear visual display (ClearVid) CMOS sensor, 7 cm wide LCD photo display screen, enhanced imaging processor, double anti blur solution and high-speed processing engine, 10X optical and 20X digital zooms
- Record high definition video-clips. It therefore has speaker microphone(s) for high quality recorded sound.
- Audio/video Out Port for connecting to a TV/DVD player.

Digital Camera Arrangements

- Keys on the camera.
- Shutter, lens and charge coupled device (CCD) array sensors
- Good resolution photo quality LCD display unit
- Displays text such as image-title, shooting data and time and serial number. It displays messages. It displays the GUI menu when user interacts with the camera.
- Self-timer lamp for flash.

Internal Units

- Internal memory flash to store OS and embedded software and limited number of image files
- Flash memory stick of 2 GB or more for large storage.
- Universal Serial Bus (USB), Bluetooth and serial COM port for connecting it to computer, mobile and printer.

Internal Units

- LCD screen to display frame view.
- Saved images display using the navigation keys.
- Frame light falls on the CCD array, which through an ADC transmits the bits for each pixel in each row in the frame and for the dark area pixels in each row for offset correction in CCD signaled light intensities for each row.
- The CCD bits of each pixel in each row and column are offset corrected by CCD signal processor (CCDSP).

ASIP and Single purpose processors

- For Signals compression using a JPEG CODEC and saved in one jpg file for each frame.
- For DSP for compression using the discrete cosine transformations (DCTs) and decompression.
- For DCT Huffman coding for the JPEG compression.
- For decompression by inverse DCT before the DAC sends input for display unit through pixel processor.
- Pixel processor (for example, image contrast, brightness, rotation, translation, color adjustment)

Digital Camera H/W

- Microcontroller or ASIP (Application Specific Instruction Set Processor)
- Multiple processors (CCDSP, DSP, Pixel Processor and others)
- RAM for storing temporary variables and stack
- ROM for application codes and RTOS codes for scheduling the tasks

Digital Camera S/W

- CCD signal processing for off-set correction
- JPEG coding
- JPEG decoding
- Pixel processing before display
- Memory and file systems
- Light, flash and display device drivers
- LCD, USB and Bluetooth Port device- drivers for port operations for display, printer and computer communication control

Seminar On

Architecture design of a virtualized Embedded System

Guided by :-
Mr. Debi Prasada Mishra
Lecturer
Dept. of Information Technology

Presented by :-
Rajeev Mohanty
6th Sem. IT
Reg. no.: 1001106231

Contents

Cont….

Introduction

- In recent years a technological breakthrough has been witnessed with the classical model of infrastructure of embedded system .
- Classical embedded systems are not much user friendly so the virtualization layer is added to the embedded system.
- With the arrival of virtualization, infrastructure concepts have so profoundly evolved.

What is Embedded System ?

- embedded system is a hybrid of hardware and software.

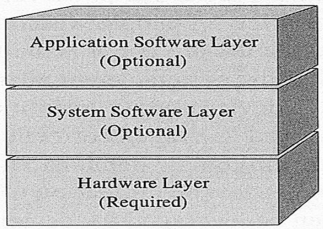


Fig 2.2 . general architecture of an embedded system
Source:-www.google.co.in/system & client

- Embedded Systems is simply the brain of most of the electronics based systems to access, process, store and control the data .

Cont…

- Classical embedded systems are those real time system which are not perform fully by sensing the environment and achieve the dedicated goal.

- Using the concept of virtualization it is possible to design an embedded system enjoying all the benefits and contributions offered by virtualization.

Basic principle & characteristics

- Organizing or performing one or many tasks according to a fixed set of rules, program or plan.

- embedded system is a system is an engineering artifact involving computation that is subject to physical constraints.

- embedded systems design is to obtain desired functionality under both kinds of constraints has software embedded into hardware.

(cont….)

- Embedded systems are application specific & single functioned

- Efficiency is of paramount importance for embedded systems
- Embedded systems are typically designed to meet real time constraints

- They generally have minimal or no user interface

Examples of Embedded Systems

- Point of sales terminals: automatic chocolate vending machine.
- Stepper motor controllers for a robotics system.
- Washing or cooking systems.
- Multitasking toys.
- Microcontroller-based single or multi display digital panel meter.

Cont…

- Keyboard controller
- SD, MMI and network access cards
- The peripheral controllers of a computer
- An antilock braking system monitor
- ECG LCD display cum recorder
- Spectrum analyzer

State of the Art

The given overview of the state of the art in terms of multi-agents systems, embedded systems and virtualizations techniques has been discussed.

- **Multi-agent systems**

A multi-agent system (MAS) is a system composed of multiple interacting intelligent agents within an environment.

(cont…)

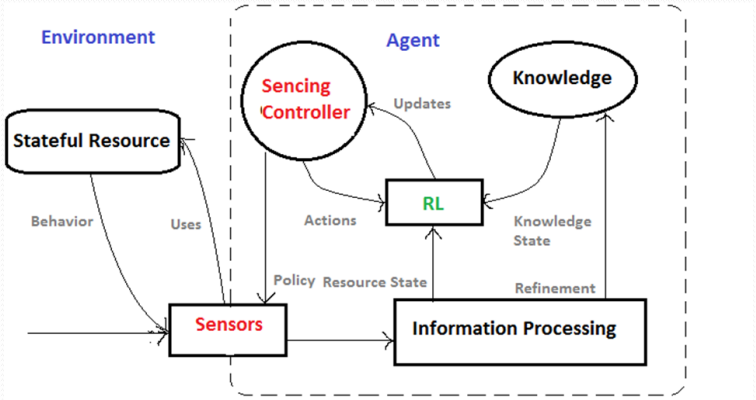


Fig 3.1 . General architecture of an agent in its interaction with its environment
Source:-www.enggjournals.com/ijcse/doc

Embedded System

- An embedded system is a special purpose system in which the computer is completely encapsulated by the device it controls

- Embedded systems bring several advantages by providing traditional systems based on conventional computers.

Cont…

Constraints

•System stability

•Mastery of the security

•The cost of production

•Low energy consumption

•Responsiveness

•Autonomy

Cont…

Design approaches

•Approach based on the CLASSICDESIGN

•Approach based on CODESIGN

Virtualization

- Embedded systems are used in many critical applications .

- achieving a high level of quality and dependability to embedded systems is an ultimate goal So the virtualization is required.

Cont…

Principle of Virtualization

It is a Framework or methodology of dividing the resources of a computer into multiple execution environments

- Partitioning.
- Transparency

Virtualization Techniques or Solution

- Insulation
- Para-virtualization
- Full virtualization

Insulation

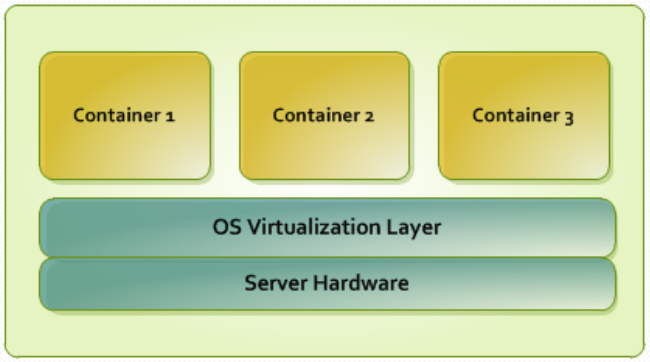


Fig 5.1 Virtualization for isolation
Source:-www.enggjournals.com/ijcse/doc

Para-virtualization

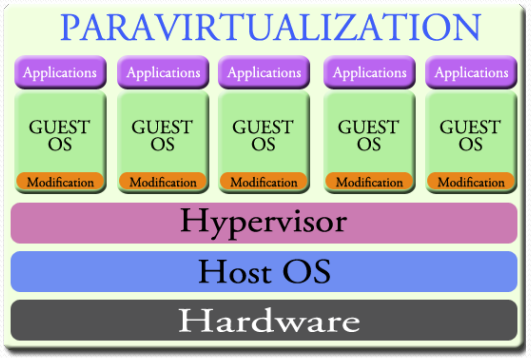


Fig 5.2 para-virtualization architecture
Source:-www.enggjournals.com/ijcse/doc

Full virtualization

- The hypervisor
- Emulation
- Fields of application

Problem related to the conventional embedded system

- Constraint related to the software update of an embedded system
- Hardware related constraint
- Hardware Design

About the proposed solution

Modeling ADMs

- To better understand the problem it has been opted a model of the architecture using multi-agent systems.
- The case of classic and improved architecture of embedded system

Case of classic embedded system

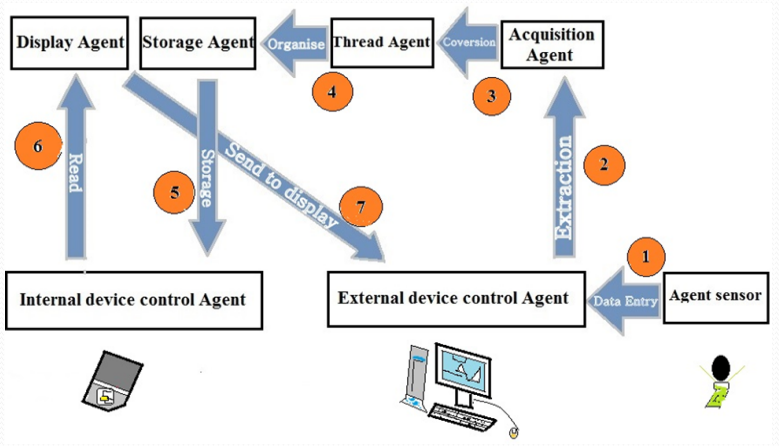


Figure 7.1 Classical architecture of an embedded system
Source:-www.enggjournals.com/ijcse/doc

Case of improved embedded system

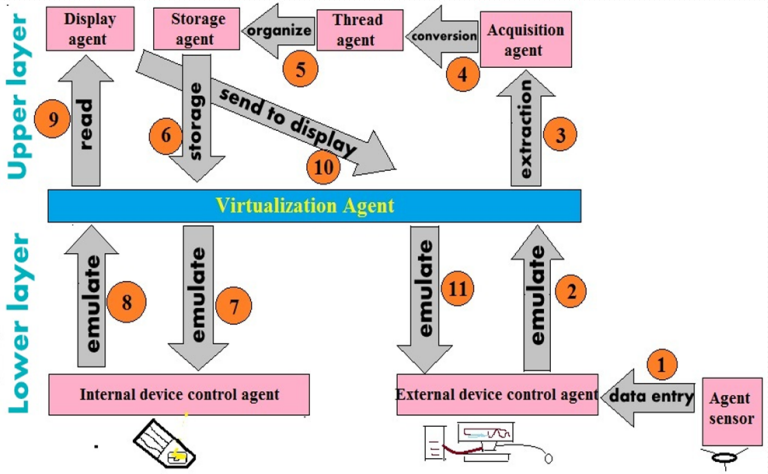


Figure 7.2 improved architecture of an embedded system
Source:-www.enggjournals.com/ijcse/doc

Description

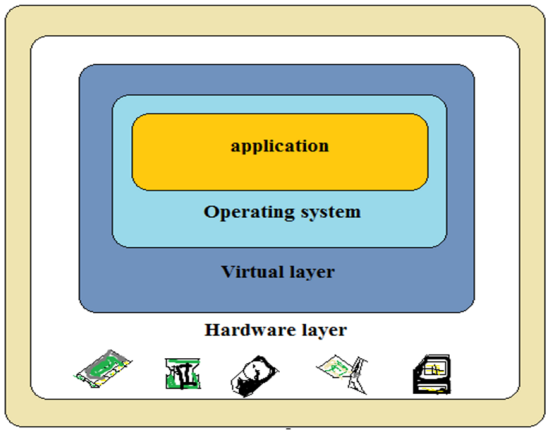


Figure 8.1 Virtual architecture of the embedded system
Source:- www.enggjournals.com/ijcse/doc

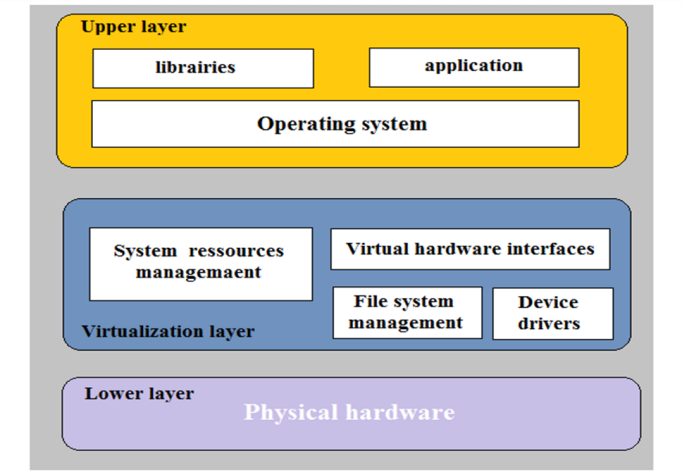


Figure 8.2 Technical architecture of the visualization layer
Source:-www.enggjournals.com/ijcse/doc

Prototype implementation

To illustrate this technical architecture, a prototype of a virtualized embedded system has been build

- Hardware Layer
- Virtualization Layer

Hardware Layer

- 1.6Ghz Intel Atom E6xx single chip processor companion chip with EG20T
- 512Mbyte DDR2-SDRAM, soldered on board
- 8 Mbit BIOS / BOOT Flash
- Internal Low Profile USB socket, bootable
- 2x SATA 3Gbit interfaces with +5 V and +12 V power header
- 4x Intel 82574L Gigabit Ethernet ports, Auto-MDIX RJ-45, protected to 700W/40A Surge

- 4x Intel 82574L Gigabit Ethernet ports, Auto-MDIX RJ- 45, protected to 700W/40A Surge
- 2x Serial ports, DB9 and 10 pins internal header • USB 2.0 interface, 2x internal, 1x external port, bootable

- Power LED, Disk LED, Error LED, Status LED, Network LED's
- 1 Full Mini-PCI Express shared with mSATA socket.
- 1 USB only Mini-PCI Express shared with mSATAsocket
- 2x PCI Express Slots, right angle

Figure9.1. Technical Components used in the visualization layer
Source:-www.enggjournals.com/ijcse/doc

Virtualization layer

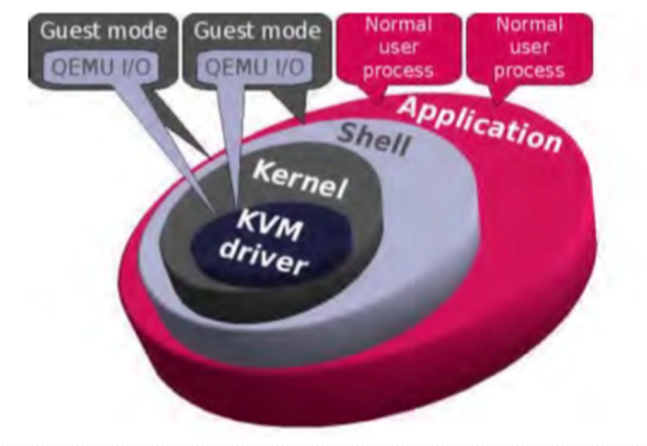


Figure9.2. KVM architecture
Source:-www.enggjournals.com/ijcse/doc

Benefits of this architecture in the design of embedded systems

- In the design of embedded systems, it is possible to add the virtualization layer at the hardware layer
- it is possible to separately design the hardware and the software part of embedded system to finally break with the old model

- This will reduce industrial waste which is a major cause of environmental problems.

Conclusion

- After a long time a drawback to the development of embedded systems has been designed an architecture based on virtualization layers associated with low control of internal and external devices.
- So it has to possible to implement a prototype based on the Linux kernel KVM. This allowed to conclude that it is possible with this architecture to benefit from the contributions of virtualization in embedded systems.
- It is clear that the future of embedded systems must address the implementation layers of virtualization at the hardware level as well as standardization of these layers.

References

1. Doc Searls, The Next Bang: The Explosive Combination of Embedded Linux, XML and Instant Messaging", , September 2000, Linux Journal, http://www.linuxjournal.com/lj-issues/issue77/4195.html
2. D. Kalinsky, R. Kalinsky ; « Introduction to I2C », Embedded.com. 2001. http://embedded.com/story/OEG20010718S0073 [18] M. Khemakhem, A. Belghith, « Agent
Based Architecture for Parallel and Distributed Complex Information processing », January 2007, Vol. 2. n. 1,
3. J. Ferber: Les systèmes multi-agents, vers une intelligence collective, Paris, InterEditions, 1995.
4. Guessoum Z., Un environnement opérationnel de conception et de réalisation de systèmes multi-agents, Thèse de doctorat, Université Paris 6, mai 1996
5. R. El Bejjet, H. Medromi, « A Generic Platform for a Multi-Agent Systems Simulation », September 2010, Vol. 5. n. 5, pp. 505-509.
6. Craigh Hollabaugh, Embedded Linux; Sams 2002

Thank you

Hardware/Software Co-Design

T S PRADEEPKUMAR
SCS, VIT

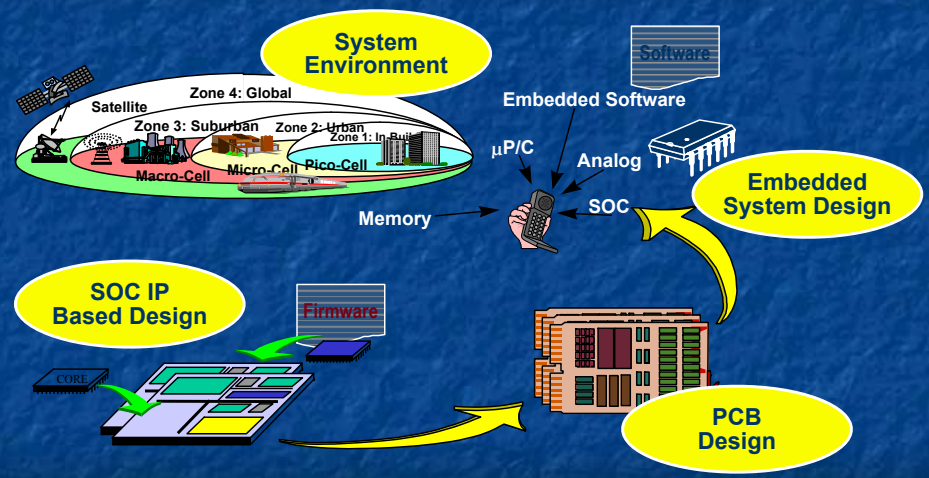
Outline

- Embedded Hardware
- Embedded Software
- Issues in Embedded System Design
- Hardware/ Software Partioning
- Designing Embedded Systems
- Co-Design

Embedded Systems

- An embedded system
  - uses a computer to perform some function, but
  - is not used (nor perceived) as a computer
- Software is used for features and flexibility
- Hardware is used for performance
- Typical characteristics:
  - it performs a single function
  - it is part of a larger (controlled) system
  - cost and reliability are often the most significant aspects

Embedded Systems



Embedded Hardware

- Input
  - Sensors
  - Sample and Hold Circuit
  - A/D Converters
- Communication
  - UART
- Processing Units
  - ASIC
  - Processors
  - Reconfigurable processors

Embedded Hardware

- Memories
  - RAM, ROM, Flash, Cache
- Output
  - D/A Converters,
  - Actuators

Embedded Software

- Real Time Operating Systems
  - General Requirements
- Scheduling in RTOS
  - Aperiodic
  - Periodic
- Real Time Databases
- Other Software Architectures
  - Function Queue Scheduling
  - Round Robin (with Interrupts)

Issues while Designing ES

- Choosing Right platform
- Memory and I/O Requirements
  - WDT, Cache, Flash memory, etc
- Processors Choice
  - PLC
  - Micro Controller or DSP
  - ASIC or FPGA

Issues…

- Hardware Software Tradeoff
- Porting Issues of OS in the Target Board

Hardware /Software Partitioning

- Definition
  - A HW/SW partitioning algorithm implements a **specification** on some sort of **multiprocessor architecture**
- Usually
  - Multiprocessor architecture = one CPU + some ASICs on CPU bus

Hardware /Software Partitioning

- Terminology
  - Allocation
    - Synthesis methods which design the multiprocessor topology along with the Process Elements and SW architecture
  - Scheduling
    - The process of assigning Process Element (CPU and/or ASICs) time to processes to get executed

Hardware /Software Partitioning

- Hw/Sw partitioning can speedup software
- Can reduce energy too
- In most partitioning algorithms
  - Type of CPU is fixed and given
  - ASICs must be synthesized

Designing of Embedded Systems

There are five stages of development of Embedded Systems

- Requirement Analysis
- Design
  - Data Structures, Software Architecture, Interfaces and algorithms
- Coding
- Testing
- Maintenance

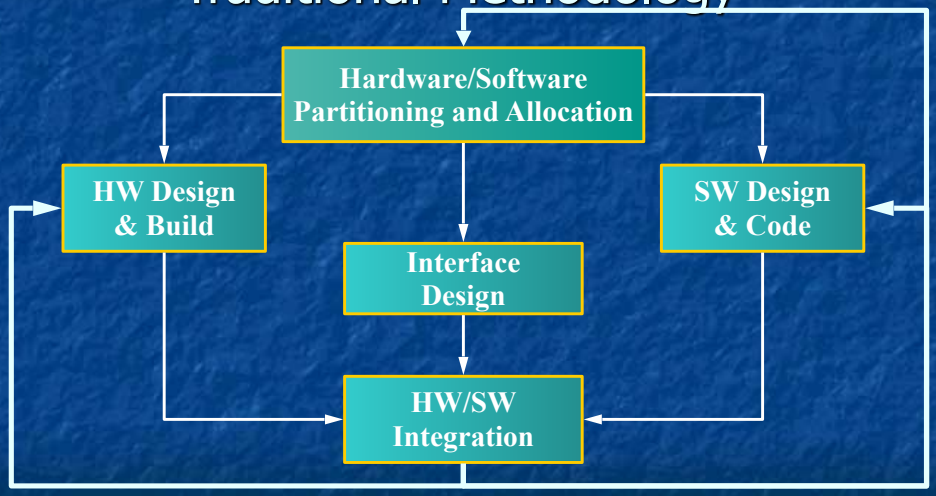
Designing of Embedded Systems

There are three Design Flows

- Waterfall Model
- Spiral Model
- Successive refinement

For Designing, Unified Modeling Language (UML) is used

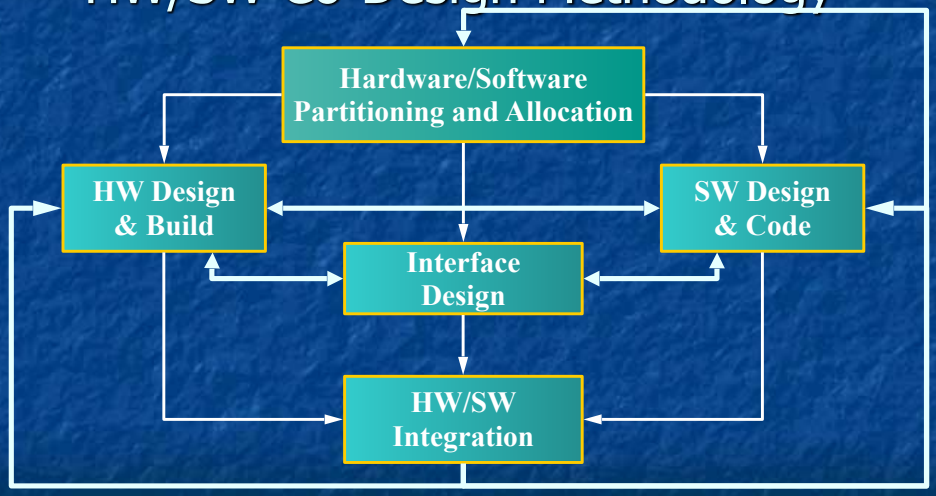
Embedded System Design
Traditional Methodology



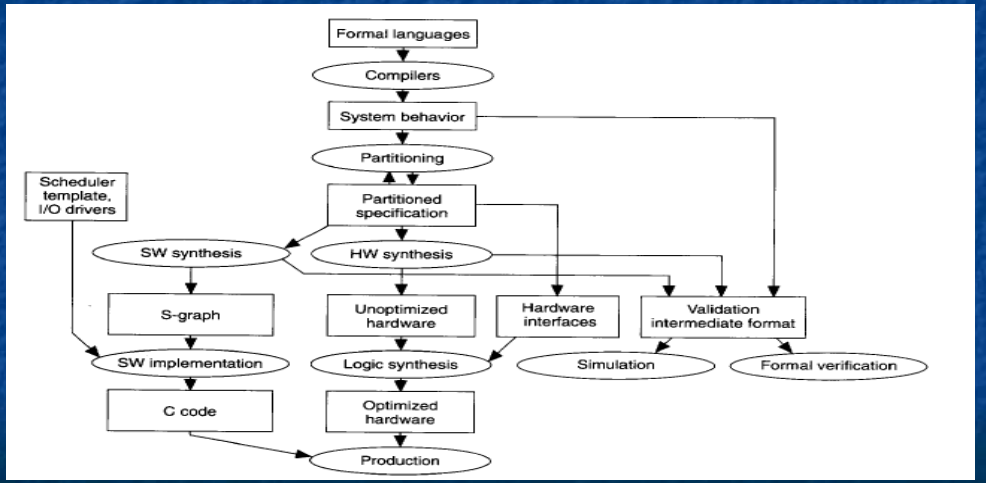
Problems with Past Design Method

- Lack of unified system-level representation
  - Can't verify the entire HW-SW system
  - Hard to find incompatibilities across HW-SW boundary (often found only when prototype is built)
- Architecture is defined a priori, based on expert evaluation of the functionality and constraints
- Lack of well-defined design flow
  - Time-to-market problems
  - Specification revision becomes difficult

Embedded System Design
HW/SW Co-Design Methodology



Co-Design Framework



Co-Design

- The software functionality should be partitioned in such a fashion that processors in the system do not get overloaded when the system is operating at peak capacity.
- This involves simulating the system with the proposed software and hardware architecture.

Co-Design

- The system should be designed for future growth by considering a scalable architecture, i.e. system capacity can be increased by adding new hardware modules. The system will not scale very well if some hardware or software module becomes a bottleneck in increasing system capacity.

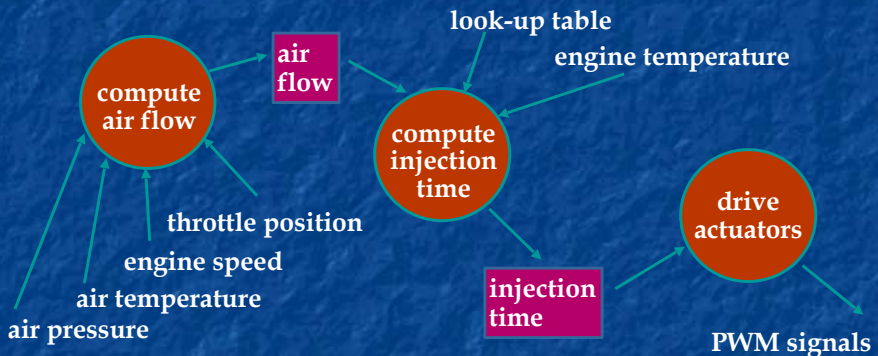
Co-Design

- Software modules that interact very closely with each other should be placed on the same processor, this will reduce delays in the system.
- Higher system performance can be achieved by this approach by inter-processor message communication.

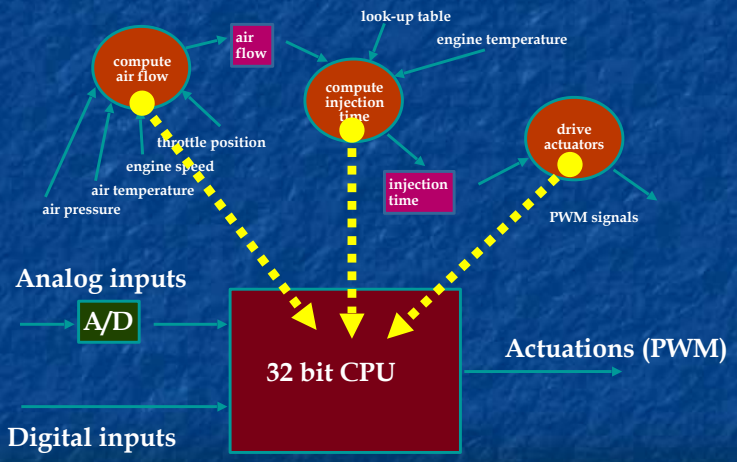
Embedded Controller Example: Engine Control Unit (ECU)

- **Task: control the torque produced by the engine**
- **by timing fuel injection and spark**
  - Major constraints:
    Low fuel consumption
    Low exhaust emission

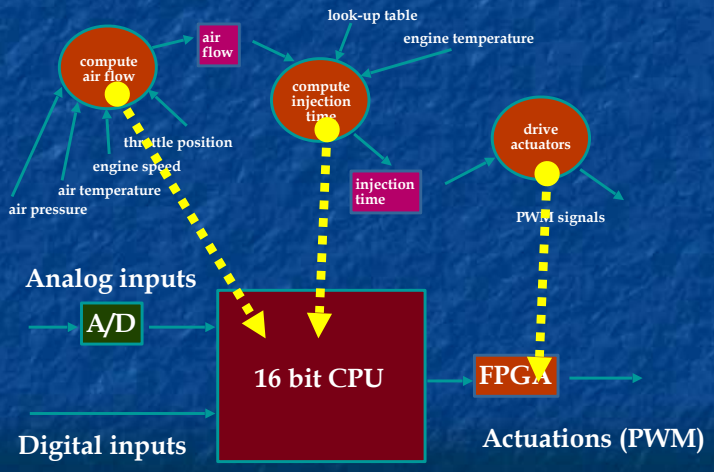
ECU **Task: control injection time (3 sub-tasks)**



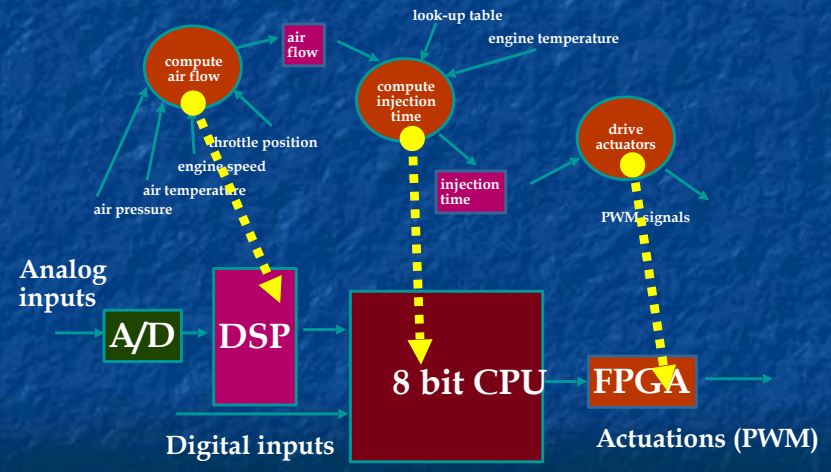
ECU- Option 1



ECU- Option 2



ECU- Option 3



Thank You