Prepared by **EBIN P.M C.Eng. (I), M.Tech**
ASSISTANT PROFESSOR
KOTTAYAM INSTITUTE OF TECHNOLOGY & SCIENCE (KITS)

MODULE-1
FUNDAMENTALS OF EMBEDDED SYSTEMS

SYSTEM

- A system is an arrangement in which all its unit assemble work together according to a set of rules.
- It can also be defined as a way of working, organizing or doing one or many tasks according to a fixed plan. For example, a watch is a time displaying system. Its components follow a set of rules to show time. If one of its parts fails, the watch will stop working. So we can say, in a system, all its subcomponents depend on each other.

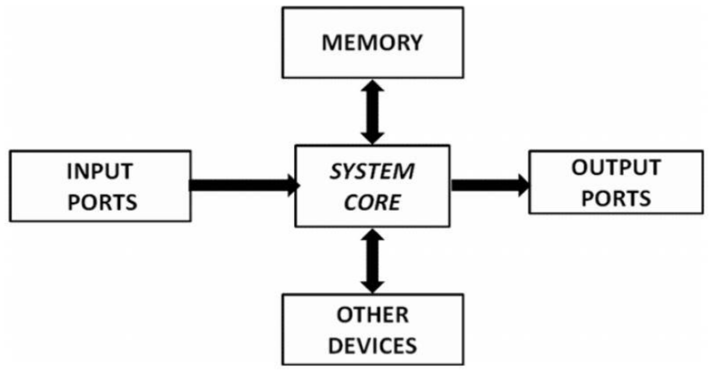
EMBEDDED SYSTEM

- Embedded means something that is attached to another thing.
- An embedded system can be thought of as a computer hardware system having software embedded in it.
- An embedded system can be an independent system or it can be a part of a large system. An embedded system is a microcontroller or microprocessor based system which is designed to perform a specific task.
- For example, a fire alarm is an embedded system; it will sense only smoke.
- An embedded system has three components − It has hardware. It has application software. It has Real Time Operating system that supervises the application software and provide mechanism to let the processor run a process as per scheduling by following a plan to control the latencies.
- RTOS defines the way the system works. It sets the rules during the execution of application program.
- A small scale embedded system may not have RTOS.
- So we can define an embedded system as a Microcontroller based, software driven, and reliable, real-time control system.
- An embedded system is designed to do a specific job only. Example: a washing machine can only wash clothes, an air conditioner can control the temperature in the room in which it is placed.
- The hardware & mechanical components will consist all the physically visible things that are used for input, output, etc.
- An embedded system will always have a chip (either microprocessor or microcontroller) that has the code or software which drives the system.

An embedded system is a combination of 3 things:

- Hardware
- Software
- Mechanical Components

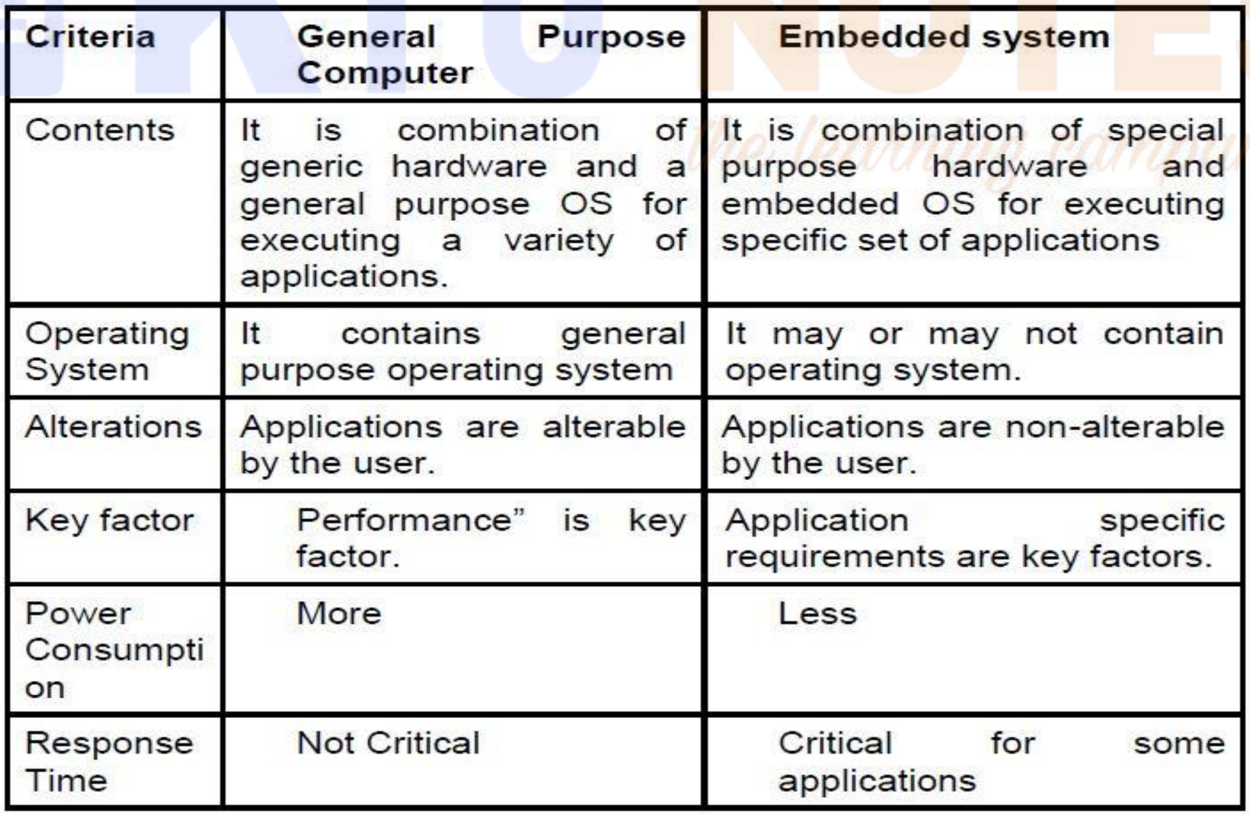
And it is supposed to do one specific task only. Diagrammatically an embedded system can be represented as follows:



EMBEDDED SYSTEM & GENERAL PURPOSE COMPUTER

- The Embedded System and the General purpose computer are at two extremes.
- The embedded system is designed to perform a specific task
- The general purpose computer is meant for general use. It can be used for playing games, watching movies, creating software, work on documents or spreadsheets etc.

Following are certain specific points of difference between embedded systems and general purpose computers:



Criteria	General Purpose Computer	Embedded system
Contents	It is combination of generic hardware and a general purpose OS for executing a variety of applications.	It is combination of special purpose hardware and embedded OS for executing specific set of applications
Operating System	It contains general purpose operating system	It may or may not contain operating system.
Alterations	Applications are alterable by the user.	Applications are non-alterable by the user.
Key factor	Performance" is key factor.	Application specific requirements are key factors.
Power Consumption	More	Less
Response Time	Not Critical	Critical for some applications

Embedded computing systems have to provide sophisticated functionality:

■ ***Complex algorithms:*** The operations performed by the microprocessor may be very sophisticated. For example, the microprocessor that controls an automobile engine must perform complicated filtering functions to optimize the performance of the car while minimizing pollution and fuel utilization.

■ ***User interface:*** Microprocessors are frequently used to control complex user interfaces that may include multiple menus and many options. The moving maps in Global Positioning System (GPS) navigation are good examples of sophisticated user interfaces.

To make things more difficult, embedded computing operations must often be performed to meet deadlines:

■ ***Real time:*** Many embedded computing systems have to perform in real time—if the data is not ready by a certain deadline, the system breaks. In some cases, failure to meet a deadline is unsafe and can even endanger lives. In other cases, missing a deadline does not create safety problems but does create unhappy customers—missed deadlines in printers ,for example, can result in scrambled pages.

■ ***Multirate:*** Not only must operations be completed by deadlines, but many embedded computing systems have several real-time activities going on at the same time. They may simultaneously control some operations that run at slow rates and others that run at high rates. Multimedia applications are prime examples of **multirate** behavior. The audio and video portions of a multimedia stream run at very different rates, but they must remain closely synchronized. Failure to meet a deadline on either the audio or video portions spoils the perception of the entire presentation.

Costs of various sorts are also very important:

■ ***Manufacturing cost:*** The total cost of building the system is very important in many cases. Manufacturing cost is determined by many factors, including the type of microprocessor used, the amount of memory required, and the types of I/O devices.

■ ***Power and energy:*** Power consumption directly affects the cost of the hardware, since a larger power supply may be necessary. Energy consumption affects battery life, which is important in many applications, as well as heat consumption, which can be important even in desktop applications.

CHARACTERISTICS OF AN EMBEDDED SYSTEM

- **Single-functioned** − an embedded system usually performs a specialized operation and does the same repeatedly. For example: A pager always functions as a pager.

- **Tightly constrained** − All computing systems have constraints on design metrics, but those on an embedded system can be especially tight. Design metrics is a measure of an implementation's features such as its cost, size, power, and performance. It must be of a size to fit on a single chip, must perform fast enough to process data in real time and consume minimum power to extend battery life.
- **Reactive and Real time** − Many embedded systems must continually react to changes in the system's environment and must compute certain results in real time without any delay. Consider an example of a car cruise controller; it continually monitors and reacts to speed and brake sensors. It must compute acceleration or de-accelerations repeatedly within a limited time; a delayed computation can result in failure to control of the car.
- **Microprocessors based** − It must be microprocessor or microcontroller based.
- **Memory** − It must have a memory, as its software usually embeds in ROM. It does not need any secondary memories in the computer.
- **Connected** − It must have connected peripherals to connect input and output devices.
- **HW-SW systems** − Software is used for more features and flexibility. Hardware is used for performance and security.

APPLICATION OF EMBEDDED SYSTEM

The application areas and the products in the embedded domain are countless.

1. Consumer Electronics: Camcorders, Cameras.

2. Household appliances: Washing machine, Refrigerator.

3. Automotive industry: Anti-lock breaking system (ABS), engine control.

4. Home automation & security systems: Air conditioners, sprinklers, fire alarms.

5. Telecom: Cellular phones, telephone switches.

6. Computer peripherals: Printers, scanners.

7. Computer networking systems: Network routers and switches.

8. Healthcare: EEG, ECG machines.

9. Banking & Retail: Automatic teller machines, point of sales.

10. Card Readers: Barcode, smart card readers.

CHALLENGES IN EMBEDDED COMPUTING SYSTEM DESIGN

➢ ***How much hardware do we need?***

- Control over the amount of computing power we apply to our problem.
- Select the type of microprocessor ,amount of memory and the peripheral devices
- To meet both performance deadlines and manufacturing cost constraints, the choice of hardware is important
- Too little hardware and the system fail to meet its deadlines, too much hardware and it becomes too expensive.

➢ ***How do we meet deadlines?***

- The brute force way of meeting a deadline is to speed up the hardware so that the program runs faster, that makes the system more expensive.
- It is also entirely possible that increasing the CPU clock rate may not make enough difference to execution time, since the program's speed may be limited by the memory system.

➢ ***How do we minimize power consumption?***

- In battery-powered applications, power consumption is extremely important. Even in non-battery applications, excessive power consumption can increase heat dissipation.
- One way to make a digital system consume less power is to make it run more slowly, but naively slowing down the system can obviously lead to missed deadlines.
- Careful design is required to slow down the noncritical parts of the machine for power consumption while still meeting necessary performance goals.

➢ ***How do we design for upgradability?***

- The hardware platform may be used over several product generations or for several different versions of a product in the same generation, with few or no changes.
- We want to be able to add features by changing software.

➢ ***Does it really work?***

- Reliability is always important when selling products
- Reliability is especially important in some applications, such as safety-critical systems.

Let's consider some ways in which the nature of embedded computing machines makes their design more difficult.

■ ***Complex testing:*** Exercising an embedded system is generally more difficult than typing in some data. We may have to run a real machine in order to generate the proper data. The timing of data is often important, meaning that we cannot separate the testing of an embedded computer from the machine in which it is embedded.

■ ***Limited observability and controllability:*** Embedded computing systems usually do not come with keyboards and screens. This makes it more difficult to see what is going on and to affect the system's operation. We may be forced to watch the values of electrical signals on the microprocessor bus, for example, to know what is going on inside the system. Moreover, in real-time applications we may not be able to easily stop the system to see what is going on inside.

■ ***Restricted development environments:*** The development environments for embedded systems (the tools used to develop software and hardware) are often much more limited than those available for PCs and workstations. We generally compile code on one type of machine, such as a PC, and download it onto the embedded system. To debug the code, we must usually rely on programs that run on the PC or workstation and then look inside the embedded system.

PERFORMANCE IN EMBEDDED COMPUTING

- Embedded system designers have a very clear performance goal in mind—their program must meet its ***deadline***. The heart of embedded computing is ***real-time computing***

- The program receives its input data; the deadline is the time at which a computation must be finished. If the program does not produce the required output by the deadline, then the program does not work, even if the output that it eventually produces is functionally correct. This notion of deadline-driven programming is at once simple and demanding.

- We need tools to help us analyze the real-time performance of embedded systems; we also need to adopt programming disciplines and styles that make it possible to analyze these programs.

- In order to understand the real-time behavior of an embedded computing system, we have to analyze the system at several different levels of abstraction. Those layers include:

- ***CPU:*** The CPU clearly influences the behavior of the program, particularly when the CPU is a pipelined processor with a cache.
- ***Platform:*** The platform includes the bus and I/O devices. The platform components that surround the CPU are responsible for feeding the CPU and can dramatically affect its performance.
- ***Program:*** Programs are very large and the CPU sees only a small window of the program at a time. We must consider the structure of the entire program to determine its overall behavior.
- ***Task:*** We generally run several programs simultaneously on a CPU, creating a ***multitasking system***. The tasks interact with each other in ways that have profound implications for performance.

- ***Multiprocessor:*** Many embedded systems have more than one processor—they may include multiple programmable CPUs as well as accelerators. Once again, the interaction between these processors adds yet more complexity to the analysis of overall system performance.

THE EMBEDDED SYSTEM DESIGN PROCESS

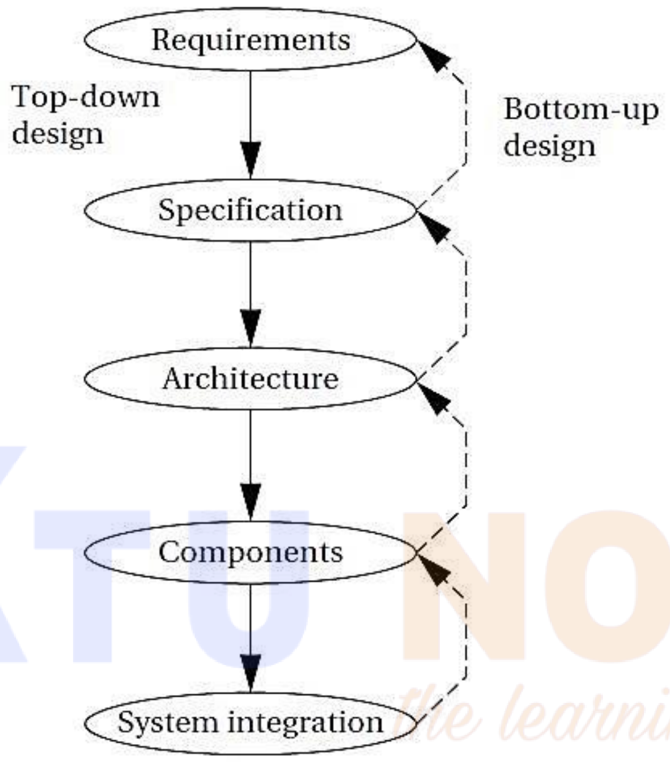


Fig: Major levels of abstraction in the design process

The above figure summarizes the major steps in the embedded system design process. In this top–down view, we start with the system ***requirements***.

In the next step, ***specification***, we create a more detailed description of what we want. The specification states only how the system behaves, not how it is built.

The details of the system's internals begin to take shape when we develop the **architecture**, which gives the system structure in terms of large **components**. Once we know the components we need, we can design those components, including both software modules and any specialized hardware we need. Based on those components, we can finally build a complete system.

We also need to consider the major goals of the design:

■ **manufacturing cost**

■ **performance (both overall speed and deadlines)**

■ **power consumption.**

We must also consider the tasks we need to perform at every step in the design process. At each step in the design, we add detail:

■ We must ***analyze*** the design at each step to determine how we can meet the specifications.

■ We must then ***refine*** the design to add detail.

■ And we must verify the design to ensure that it still meets all system goals, such as cost, speed, and so on.

Specification

- The specification is more precise—it serves as the contract between the customer and the architects.
- The specification must be carefully written so that it accurately reflects the customer's requirements and does so in a way that can be clearly followed during design.
- The specification should be understandable enough so that someone can verify that it meets system requirements and overall expectations of the customer.
- It should also be unambiguous enough that designers know what they need to build.
- If global characteristics of the specification are wrong or incomplete, the overall system architecture derived from the specification may be inadequate to meet the needs of implementation.

Architecture design of embedded system

- The specification does not say how the system does things, only what the system does.
- Describing how the system implements those functions is the purpose of the architecture.
- The architecture is a plan for the overall structure of the system that will be used later to design the components that make up the architecture.
- The creation of the architecture is the first phase of what many designers think of as design.

To understand what an architectural description is, let's look at sample architecture for the moving map. The moving map is a handheld device that displays for the user a map of the terrain around the user's current position; the map display changes as the user and the map device change position. The moving map obtains its position from the GPS, a satellite-based navigation system. Following figure shows sample system architecture in the form of a ***block diagram*** that shows major operations and data flows among them.

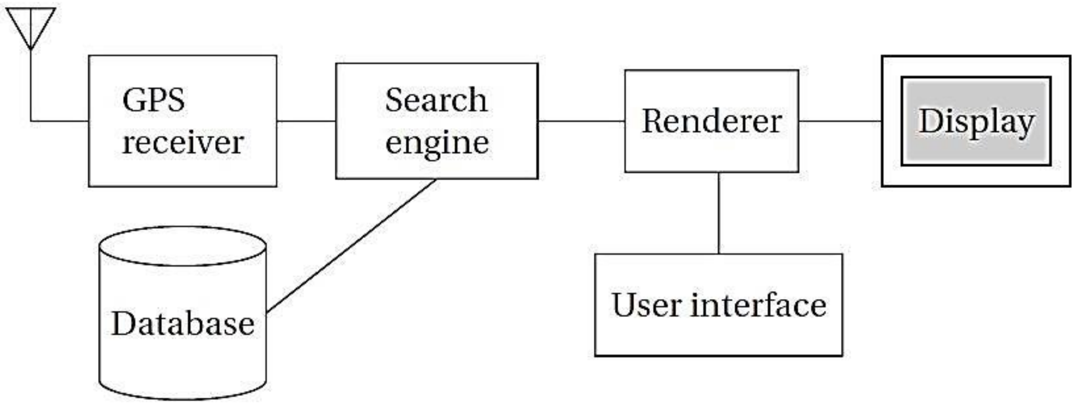


Fig: Block diagram for the moving map.

This block diagram is still quite abstract—we have not yet specified which operations will be performed by software running on a CPU, what will be done by special-purpose hardware, and so on.

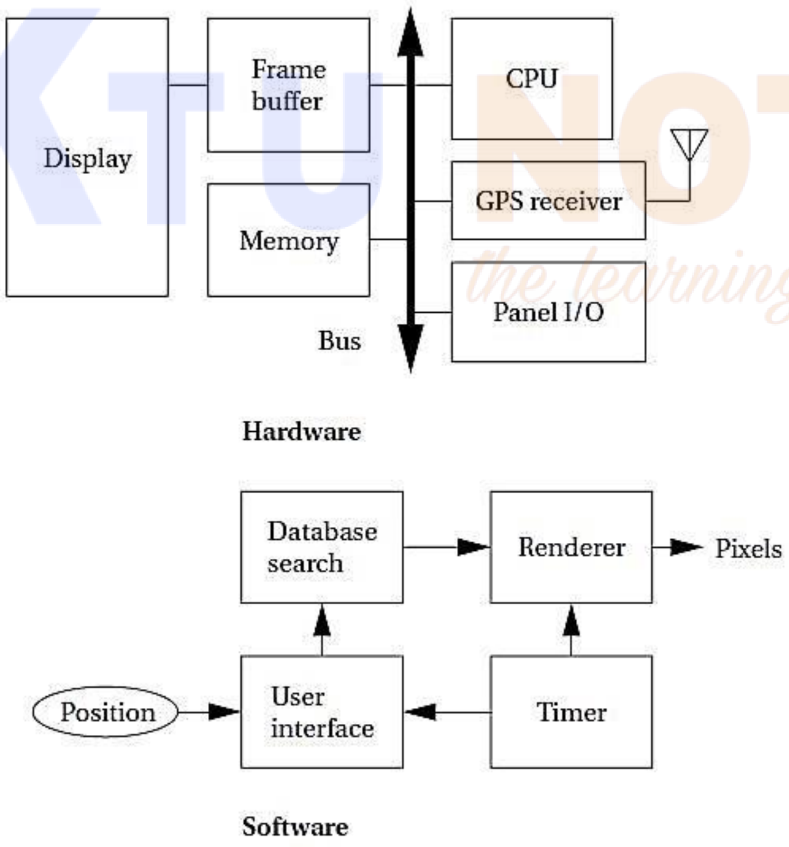


Fig: Hardware and software architectures for the moving map.

Designing Hardware and Software Components

- The architectural description tells us what components we need.
- The components will in general include both hardware—FPGAs, boards, and so on—and software modules.
- Some of the components will be ready-made. The CPU, for example, will be a standard component in almost all cases, as will memory chips and many other components.
- In the moving map, the GPS receiver is a good example of a specialized component that will nonetheless be a predesigned, standard component. We can also make use of standard software modules.

Unified Modeling Language (UML) - UML was designed to be useful at many levels of abstraction in the design process. UML is useful because it encourages design by successive refinement and progressively adding detail to the design, rather than rethinking the design at each new level of abstraction. UML is an ***object-oriented*** modeling language.

Structural Description

- By ***structural description***, we mean the basic components of the system.
- The principal component of an object-oriented design is the ***object.*** An object includes a set of ***attributes*** that define its internal state. When implemented in a programming language, these attributes usually become variables or constants held in a data structure.
- In some cases, we will add the type of the attribute after A class is a form of type definition—all objects derived from the same class have the same characteristics, although their attributes may have different values.
- A class defines the attributes that an object may have. It also defines the ***operations*** that determine how the object interacts with the rest of the world. In a programming language, the operations would become pieces of code used to manipulate the object. The UML description of the ***Display*** class is shown below

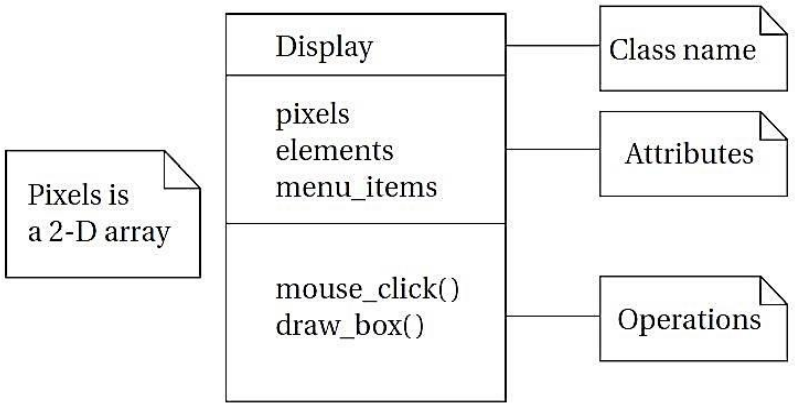


Fig: A class in UML notation.

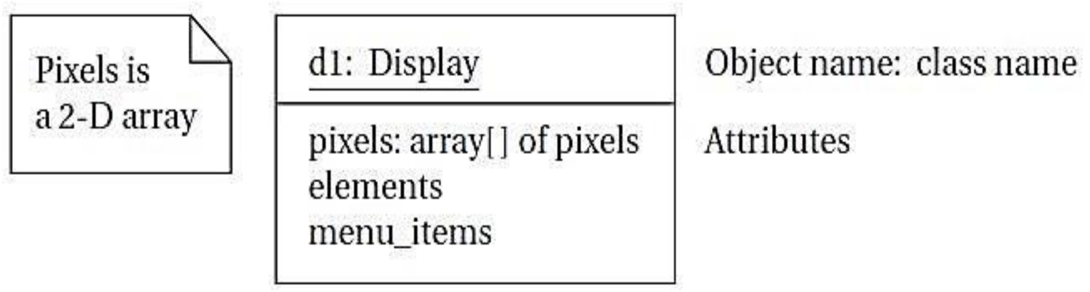


Fig: An object in UML notation.

- The class has the name that we saw used in the *d* 1 object since *d* 1 is an instance of class ***Display***.
- The ***Display*** class defines the *pixels* attribute seen in the object; remember that when we instantiate the class an object, that object will have its own memory so that different objects of the same class have their own values for the attributes.
- Other classes can examine and modify class attributes; if we have to do something more complex than use the attribute directly, we define a behavior to perform that function.
- A class defines both the ***interface*** for a particular type of object and that object's ***implementation***.
- When we use an object, we do not directly manipulate its attributes—we can only read or modify the object's state through the operations that define the interface to the object.

There are several types of *relationships* that can exist between objects and classes:

■ ***Association*** occurs between objects that communicate with each other but have no ownership relationship between them.

■ ***Aggregation*** describes a complex object made of smaller objects.

■ ***Composition*** is a type of aggregation in which the owner does not allow access to the component objects.

■ ***Generalization*** allows us to define one class in terms of another.

Unified Modeling Language allows us to define one class in terms of another. An example is shown below, where we ***derive*** two particular types of displays. The first, ***BW_display***, describes a black- and-white display. This does not require us to add new attributes or operations, but we can specialize both to work on one-bit pixels. The second, ***Color_map_display***, uses a graphic device known as a color map to allow the user to select from behaviors—for example, large number of available colors even with a small number of bits per pixel. This class defines a *color_map* attribute that determines how pixel values are mapped onto display colors.

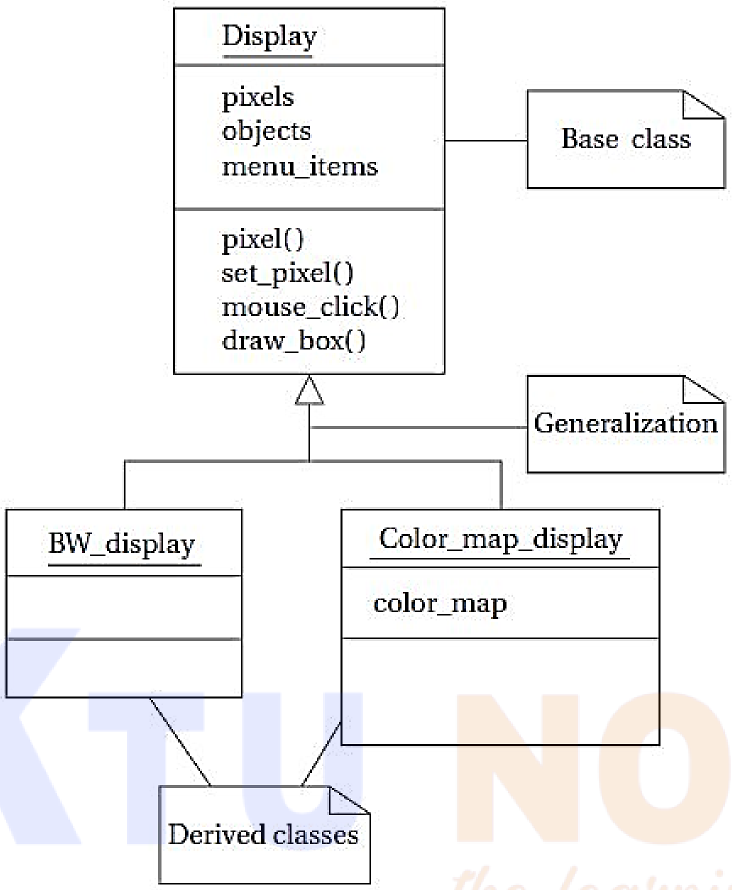


Fig: Derived classes as a form of generalization in UML.

- A ***derived class*** inherits all the attributes and operations from its ***base class***.
- In this class, *Display* is the base class for the two derived classes. A derived class is defined to include all the attributes of its base class.
- This relation is transitive—if *Display* were derived from another class, both ***BW_display*** and ***Color_map_display*** would inherit all the attributes and operations of *Display's* base class as well.
- **Unified Modeling Language** considers inheritance to be one form of generalization. A generalization relationship is shown in a UML diagram as an arrow with an open (unfilled) arrowhead.
- UML also allows us to define ***multiple inheritances***, in which a class is derived from more than one base class.

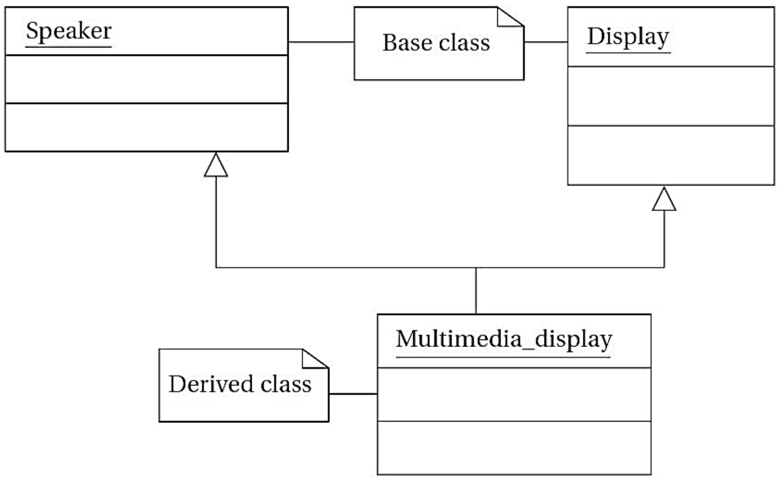
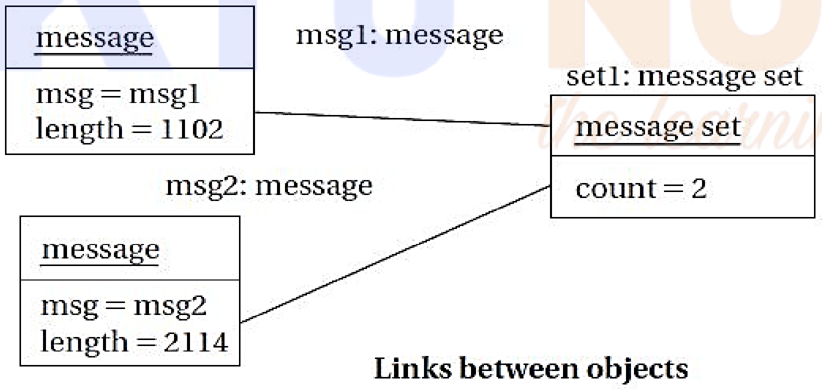
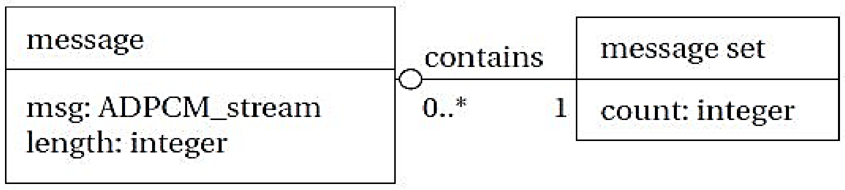


Fig: Multiple inheritances in UML.

- In the above figure we have created a ***Multimedia_display*** class by combining the ***Display*** class with a ***Speaker*** class for sound.
- The derived class inherits all the attributes and operations of both its base classes, *Display* and *Speaker*.
- A ***link*** describes a relationship between objects; association is to link as class is to object. Following figure shows examples of links and an association



Links between objects



Association between classes

- When we consider the actual objects in the system, there is a set of messages that keeps track of the current number of active messages (two in the above example) and points to the active messages. In this case, the link defines the ***contains*** relation.
- The association is drawn as a line between the two labeled with the name of the association, namely, ***contains*.**

Behavioral Description

- We have to specify the behavior of the system as well as its structure. One way to specify the behavior of an operation is a ***state machine***. Following figure shows UML states; the transition between two states is shown by a skeleton arrow.

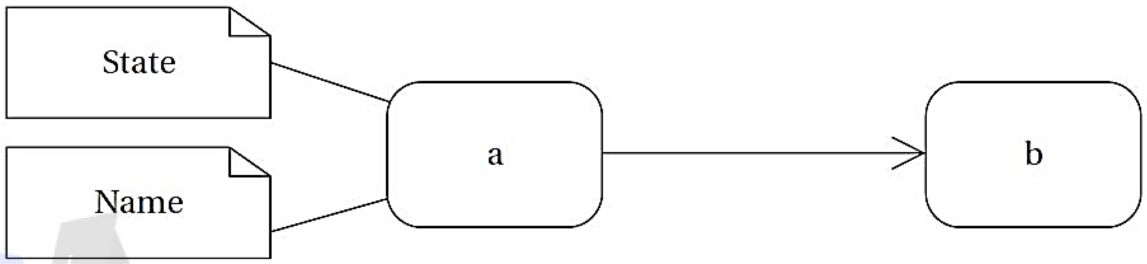


Fig: A state and transition in UML.

- The state machines will not rely on the operation of a clock, as in hardware; rather, changes from one state to another are triggered by the occurrence of ***events***.
- An event is some type of action. The event may originate outside the system, such as a user pressing a button. It may also originate inside, such as when one routine finishes its computation and passes the result on to another routine. The three types of events defined by UML are

- A ***signal*** is an asynchronous occurrence. It is defined in UML by an object that is labeled as a <<*signal*>>. The object in the diagram serves as a declaration of the event's existence. Because it is an object, a signal may have parameters that are passed to the signal's receiver.
- A ***call event*** follows the model of a procedure call in a programming language.
- A ***time-out event*** causes the machine to leave a state after a certain amount of time. The label *tm(time-value)* on the edge gives the amount of time after which the transition occurs. A time-out is generally implemented with an external timer. This notation simplifies the specification and allows us to defer implementation details about the time-out mechanism.

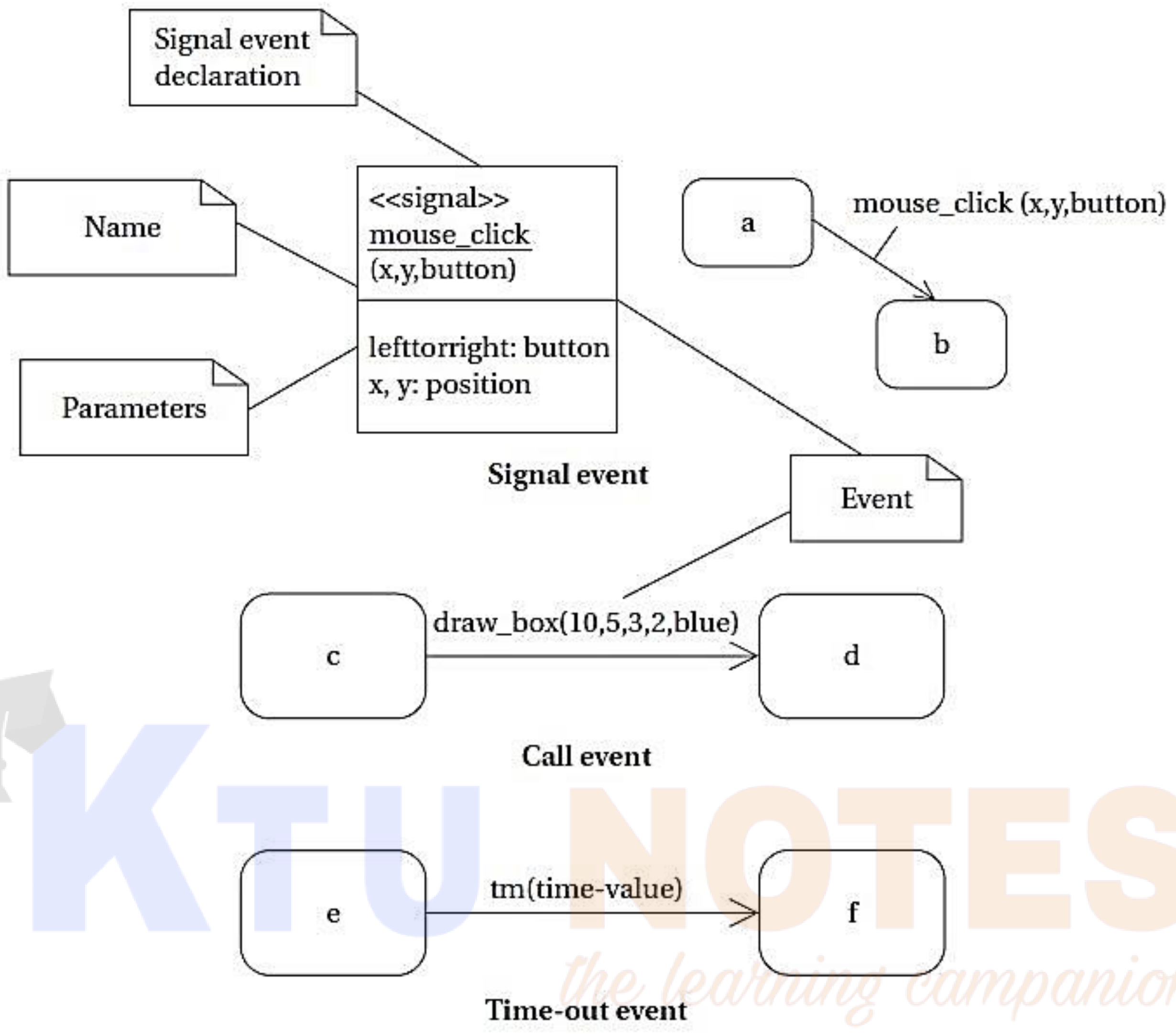


Fig: Signal, call, and time-out events in UML.

- It is sometimes useful to show the sequence of operations over time, particularly when several objects are involved. In this case, we can create a sequence diagram, like the one for a mouse click scenario shown below
- A ***sequence diagram*** is somewhat similar to a hardware timing diagram,
- The time flows vertically in a sequence diagram, whereas time typically flows horizontally in a timing diagram.
- The sequence diagram is designed to show a particular scenario or choice of events
- In the following ***sequence diagram***, the sequence shows what happens when a mouse click is on the menu region. Processing includes three objects shown at the top of the diagram. Extending below each object is its *lifeline*, a dashed line that shows how long the object is alive. In this case, all the objects remain alive for the entire sequence, but in other cases objects may be created or destroyed during processing. The boxes along the

lifelines show the *focus of control* in the sequence, that is, when the object is actively processing. In this case, the mouse object is active only long enough to create the *mouse_click* event. The display object remains in play longer; it in turn uses call events to invoke the menu object twice: once to determine which menu item was selected and again to actually execute the menu call. The find_region( ) call is internal to the display object, so it does not appear as an event in the diagram.

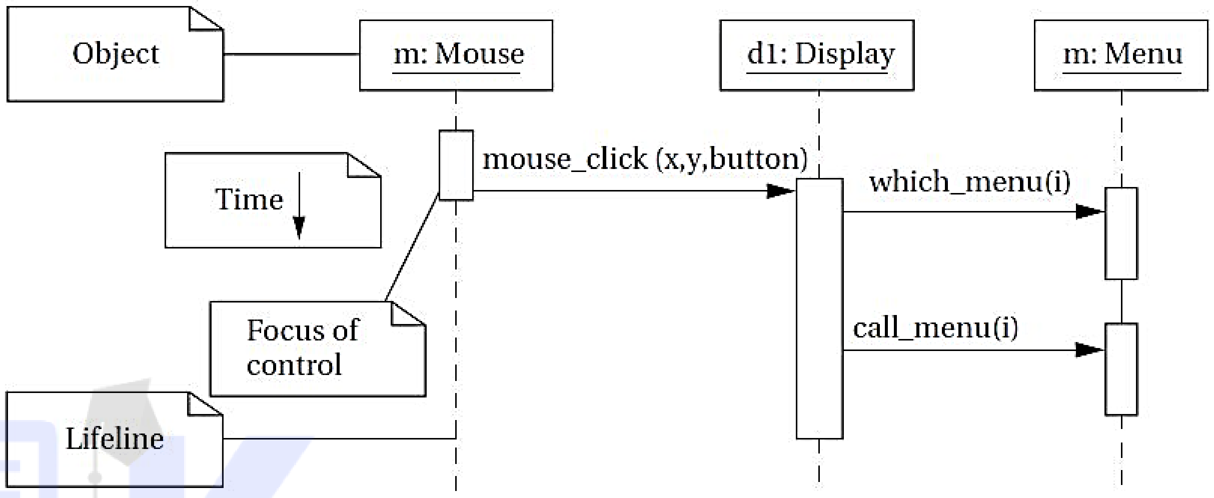


Fig: A sequence diagram in UML.