Day 1

Definitions

Todd D. Morton

Embedded Systems are the electronic systems that contain a microprocessor or a microcontroller, but we do not think of them as computers – the computer is hidden or embedded in the system.

Wayne Wolf

Embedded System is any device that includes a programmable computer but is not itself intended to be a general purpose computer.

Raj Kamal

An embedded system is a system that has software embedded into computer-hardware, which makes a system dedicated for an application(s) or specific part of an application or product or part of a larger system.

Components of Embedded Systems

Three main components of an Embedded System are as follows

1. Embeds hardware to give computer like functionalities

2. Embeds main application software generally into flash or ROM and the application software performs concurrently the number of tasks.

3. Embeds a real time operating system (RTOS), which supervises the application software tasks running on the hardware and organizes the accesses to system resources according to priorities and timing constraints of tasks in the system.

Embedded Computers

Whirlwind

- First computer designed to support ***real-time*** operation
- Conceived as a mechanism for controlling an aircraft simulator.

- Extremely large physically(4000 vacuum tubes)

Intel 4004

- First microprocessor designed for an embedded application, namely, a calculator.
- The calculator was not a general-purpose computer - it merely provided basic arithmetic functions.

HP-35

- First handheld calculator to perform transcendental functions
- Used several chips to implement the CPU, rather than a single-chip microprocessor.
- The ability to write programs to perform math rather than having to design digital circuits to perform operations like trigonometric functions was critical to the successful design of the calculator.

Microprocessors come in many different levels of sophistication.

- An **8-bit *microcontroller*** is designed for low-cost applications and includes on-board memory and I/O devices.
- A **16-bit microcontroller** is often used for more sophisticated applications that may require either longer word lengths or off-chip I/O and memory.
- A **32-bit *RISC* microprocessor** offers very high performance for computation-intensive applications.

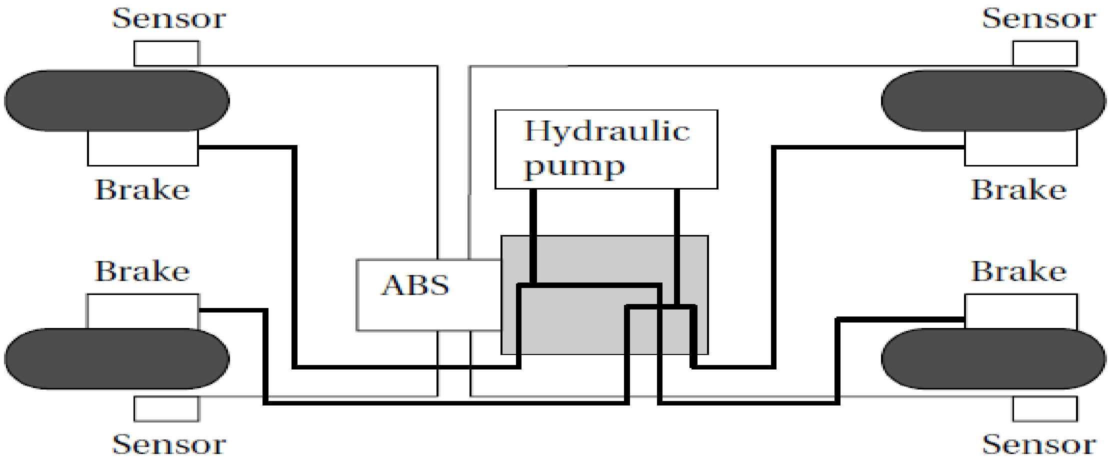
Examples of Embedded Systems

1. PDA or Palmtop (Personal Digital Assistance):- Uses 32-bit microcontrollers.
2. Cell phone: - Uses a 32-bit microcontroller.
3. Household appliance:-

For example, front panel of microwave oven that also uses a microcontroller, but typically it will have its word size much smaller than that of the earlier examples; because the functionality that it handles is much less.

4. Camera: - Uses a 32-bit processor because it handles complex functions. Similarly, in an analog to be a 5. Digital TV: - Uses a 32-bit processor because in an analog TV, the microcontroller handles primarily the problem of tuning and channel selection. But in a digital TV, decompression, disk family and particularly on the set top box, your microcontroller handles a number of complex functions.

5. Antilock Braking System: - reduces skidding by pumping the brakes.



The purpose of an ABS is to temporarily release the brake on a wheel when it rotates too slowly when a wheel stops turning, the car starts skidding and becomes hard to control. It sits between the hydraulic pump, which provides power to the brakes, and the brakes themselves as seen in the following diagram. This hookup allows the ABS system to modulate the brakes in order to keep the wheels from locking. The ABS system uses sensors on each wheel to measure the speed of the wheel. The wheel speeds are used by the ABS system to determine how to vary the hydraulic fluid pressure to prevent the wheels from skidding.

Day 2

Characteristics of Embedded Systems

1. Provide sophisticated (complex) 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.

2. Real Time Operation

- *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.

3. Cost of various sort

- *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.

Challenges in Embedded Computing System Design

1. *How much hardware do we need?*

We cannot only select the type of microprocessor used, but also select the amount of memory, the peripheral devices, and more. Since we often must meet **both performance deadlines and manufacturing cost constraints**, the choice of hardware is important - too little hardware and the system fails to meet its deadlines, too much hardware and it becomes too expensive.

2. *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. Of course, 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.

3. *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 non-critical parts of the machine for power consumption while still meeting necessary performance goals.

4. *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. However, we want to be able to add features by changing software.

5. *Does it really work?*

Reliability is always important when selling products. Customers rightly expect that products they buy will work. If we wait until we have a running system and try to eliminate the bugs, it will be too expensive to fix them, and it will take too long as well.

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. 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***. Embedded computing is ***real-time computing***, which is the science and art of programming to deadlines. 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.

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.

Day 3

THE EMBEDDED SYSTEM DESIGN PROCESS

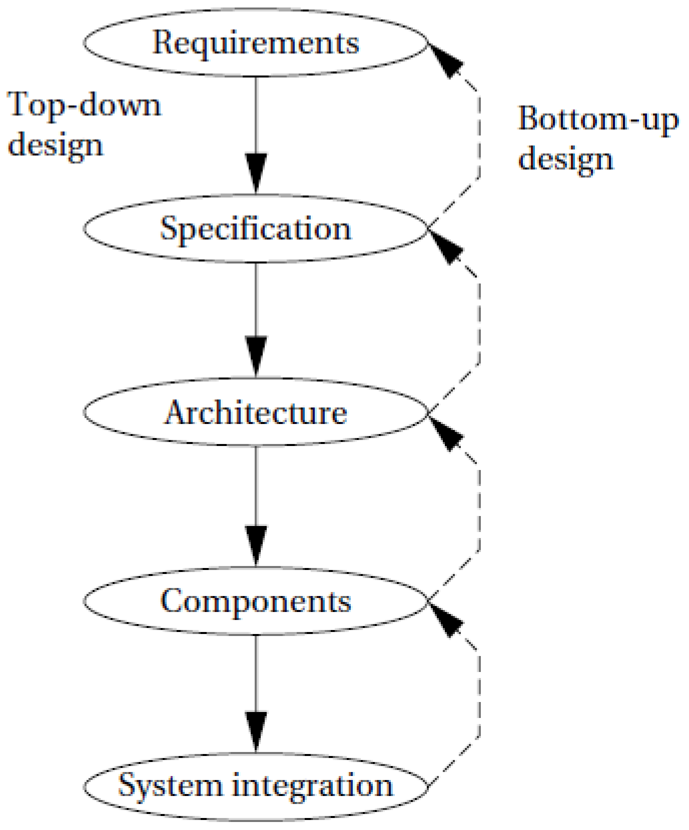


Figure: Major level of abstraction in the design process

In the top–down view, we start with the system ***requirements***. In the next step, ***specification***, we create a more detailed description of what we want. But 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.

The alternative is a **bottom–up** view in which we start with components to build a system. Bottom–up design steps are shown in the figure as dashed-line arrows. We need bottom–up design

because we do not have perfect insight into how later stages of the design process will turn out. Decisions at one stage of design are based upon estimates of what will happen later.
Major goals of the design are as follows:

- Manufacturing cost.
- Performance (both overall speed and deadlines).
- Power consumption.

At each step in the design, we add detail:

- A*nalyze* the design at each step to determine how we can meet the specifications.
- R*efine* the design to add detail.
- *Verify* the design to ensure that it still meets all system goals, such as cost, speed, and so on.

Requirements

The initial stages of the design process capture this information for use in creating the architecture and components. We generally proceed in two phases: First, we gather an informal description from the customers known as requirements, and we refine the requirements into a specification that contains enough information to begin designing the system architecture.

Separating out requirements analysis and specification is often necessary because of the large gap between what the customers can describe about the system they want and what the architects need to design the system.

Capturing a consistent set of requirements from the customer and then massaging those requirements into a more formal specification is a structured way to manage the process of translating from the consumer's language to the designer's.

Requirements may be ***functional*** or ***nonfunctional***. We must of course capture the basic functions of the embedded system, but functional description is often not sufficient. Typical nonfunctional requirements include:

- *Performance:* The speed of the system is often a major consideration both for the usability of the system and for its ultimate cost.
- *Cost:* The target cost or purchase price for the system is almost always a consideration. Cost typically has two major components: ***manufacturing cost*** includes the cost of components and assembly; ***nonrecurring engineering*** **(NRE)** costs include the personnel and other costs of designing the system.

- *Physical size and weight:* The physical aspects of the final system can vary greatly depending upon the application. An industrial control system for an assembly line may be designed to fit into a standard-size rack with no strict limitations on weight
- *Power consumption:* Power is important in battery-powered systems and is often important in other applications as well. Power can be specified in the requirements stage in terms of battery life.

One good way to refine at least the user interface portion of a system's requirements is to build a ***mock-up***. The mock-up may use canned data to simulate functionality in a restricted demonstration, and it may be executed on a PC or a workstation. But it should give the customer a good idea of how the system will be used and how the user can react to it. Physical, nonfunctional models of devices can also give customers a better idea of characteristics such as size and weight.

Requirements form that can be filled out at the start of the project. We can use the form as a checklist in considering the basic characteristics of the system. Some of the entries in the form:

- *Name:* This is simple but helpful. Giving a name to the project not only simplifies talking about it to other people but can also crystallize the purpose of the machine.
- *Purpose:* This should be a brief one- or two-line description of what the system is supposed to do. If you can't describe the essence of your system in one or two lines, chances are that you don't understand it well enough.
- *Inputs and outputs:* These two entries are more complex than they seem. The inputs and outputs to the system encompass a wealth of detail:
  - ➢ *Types of data:* Analog electronic signals? Digital data? Mechanical inputs?
  - ➢ *Data characteristics:* Periodically arriving data, such as digital audio samples? Occasional user inputs? How many bits per data element?
  - ➢ *Types of I/O devices:* Buttons? Analog/digital converters? Video displays?
- *Functions:* This is a more detailed description of what the system does. A good way to approach this is to work from the inputs to the outputs: When the system receives an input, what does it do? How do user interface inputs affect these functions? How do different functions interact?

- *Performance:* It is essential that the performance requirements be identified early since they must be carefully measured during implementation to ensure that the system works properly.
- *Manufacturing cost:* This includes primarily the cost of the hardware components. Even if you don't know exactly how much you can afford to spend on system components, you should have some idea of the eventual cost range. Cost has a substantial influence on architecture.
- *Power:* Similarly, you may have only a rough idea of how much power the system can consume, but a little information can go a long way. Battery-powered machines must be much more careful about how they spend energy.
- *Physical size and weight:* You should give some indication of the physical size of the system to help guide certain architectural decisions. A desktop machine has much more flexibility in the components used than, for example, a lapel mounted voice recorder.

Example of Requirement form for GPS moving map

Name:	GPS moving map
Purpose:	Consumer-grade moving map for driving use
Inputs:	Power button, two control buttons
Outputs:	Back-lit LCD display 400 X 600
Functions:	Uses 5-receiver GPS system; three user-selectable resolutions; Always displays current latitude and longitude
Performance:	Updates screen within 0.25 seconds upon movement
Manufacturing:	cost $30
Power:	100mW
Physical size and weight:	No more than 2" X 6," 12 ounces

Specification

It serves as the contract between the customer and the architects. As such, 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. Designers can run into several different types of problems caused by unclear specifications. If the behavior of some feature in a particular situation is unclear from the specification, the designer may implement the wrong functionality. 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.

A specification of the GPS system would include several components:

- Data received from the GPS satellite constellation.
- Map data.
- User interface.
- Operations that must be performed to satisfy customer requests.
- Background actions required to keep the system running, such as operating the GPS receiver.

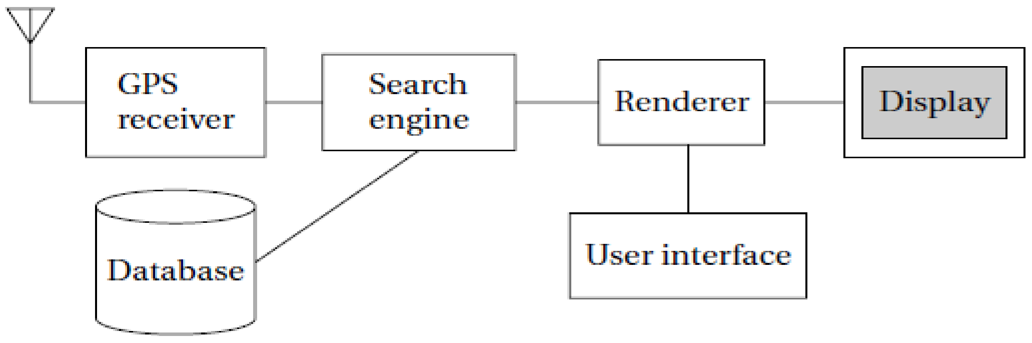
UML, a language for describing specifications and we will use it to write a specification.

Day 4

Architecture Design

The specification does not say how the system does things, only what the system does. 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.

Figure shows a sample system architecture in the form of a ***block diagram*** that shows major operations and data flows among them. This block diagram is still quite abstract. The diagram does, however, go a long way toward describing how to implement the functions described in the specification. For example, that we need to search the topographic database and to render the results for the display.



We refine that system block diagram into two block diagrams:

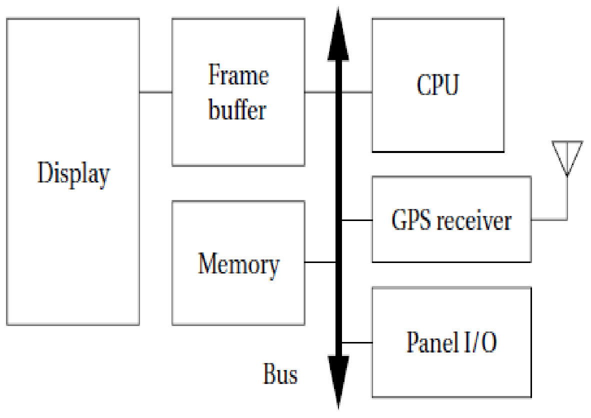
- Hardware block diagram
- Software block diagram

The hardware block diagram clearly shows that we have one central CPU surrounded by memory and I/O devices. In particular, we have chosen to use two memories: a frame buffer for the pixels to be displayed and a separate program/data memory for general use by the CPU.

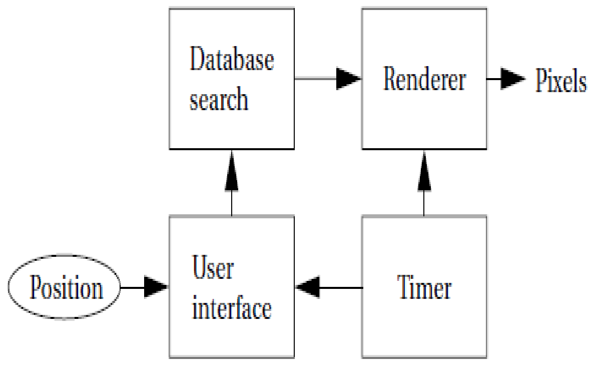
The software block diagram fairly closely follows the system block diagram, but we have added a timer to control when we read the buttons on the user interface and render data onto the screen.

To have a truly complete architectural description, we require more detail, such as where units in the software block diagram will be executed in the hardware block diagram and when operations will be performed in time.

Architectural descriptions must be designed to satisfy both functional and nonfunctional requirements. Not only must all the required functions be present, but we must meet cost, speed, power and other nonfunctional constraints.



Hardware



Software

Designing Hardware and Software components

The architectural description tells us what components we need. The component design effort builds those components in conformance to the architecture and specification. The components will in general include both hardware and software modules.

Some of the components will be ready-made. For example CPU, memory chips and so on. 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. One good example is the topographic database.

You will have to design some components yourself. Even if you are using only standard integrated circuits, you may have to design the printed circuit board that connects them. You will probably have to do a lot of custom programming as well.

When creating these embedded software modules, you must of course make use of your expertise to ensure that the system runs properly in real time and that it does not take up more memory space than is allowed.

The power consumption of the moving map software example is particularly important. You may need to be very careful about how you read and write memory to minimize power.

System Integration

This phase usually consists of a lot more than just plugging everything together and standing back. Bugs are typically found during system integration, and good planning can help us find the bugs quickly. By building up the system in phases and running properly chosen tests, we can often find bugs more easily. If we debug only a few modules at a time, we are more likely to uncover the simple bugs and able to easily recognize them.

System integration is difficult because it usually uncovers problems. Careful attention to inserting appropriate debugging facilities during design can help ease system integration problems, but the nature of embedded computing means that this phase will always be a challenge.

FORMALISMS FOR SYSTEM DESIGN

Unified Modeling Language

UML is an ***object-oriented*** modeling language. 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.

Object-oriented design emphasizes two concepts of importance:

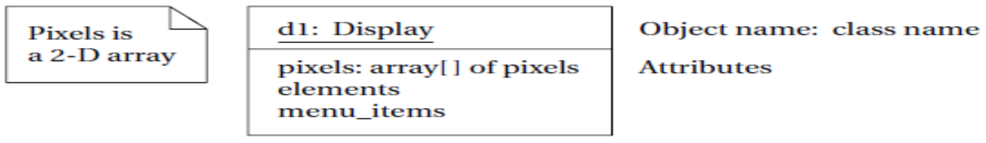
- It encourages the design to be described as a number of interacting objects, rather than a few large monolithic blocks of code.
- At least some of those objects will correspond to real pieces of software or hardware in the system. We can also use UML to model the outside world that interacts with our system, in which case the objects may correspond to people or other machines.

Structural Description

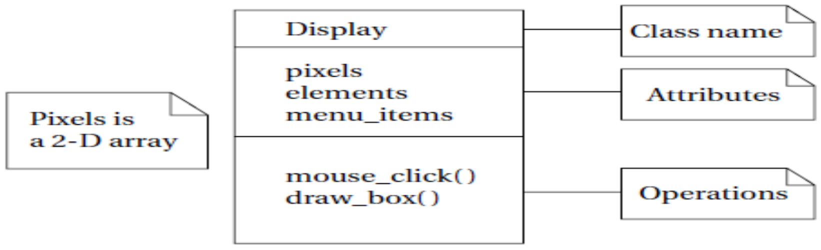
S*tructural description* mean the basic components of the system. The principal component of an object-oriented design is the ***object***. An object includes a set of ***attributes*** that define its internal state. When implemented in a programming language, these attributes usually become variables or constants held in a data structure.

An object describing a display (such as a CRT screen) is shown in UML notation. The text in the folded-corner page icon is a ***note***; it does not correspond to an object in the system and only serves as a comment. The attribute is an array of pixels that holds the contents of the display.

The object is identified in two ways: It has a unique name, and it is a member of a ***class***. The name is underlined to show that this is a description of an object and not of a class. 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. 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.



An object in UML notation.

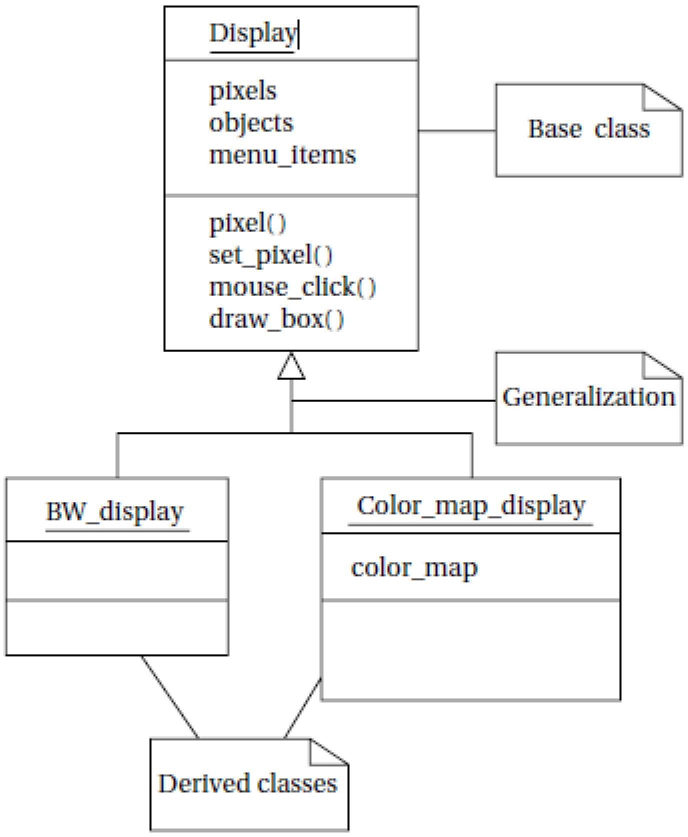


A class in UML notation.

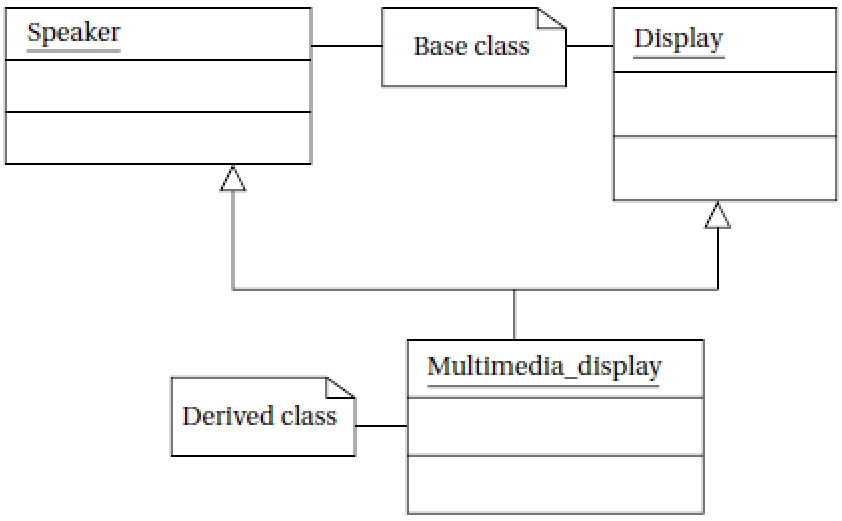
There are several types of ***relationships*** that can exist between objects and classes:

- ***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.

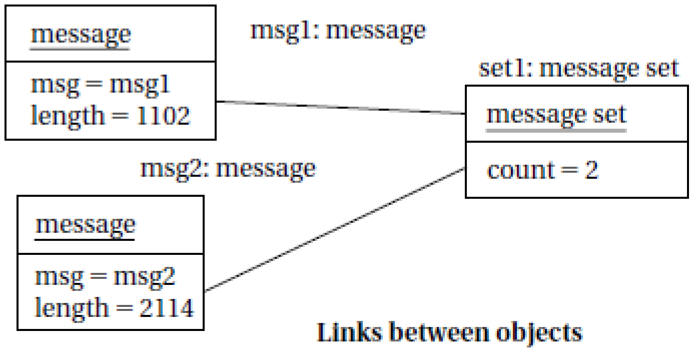
A ***derived class*** inherits all the attributes and operations from its ***base class***. In this class, *Display* is the base class for the two derived classes. A derived class is defined to include all the attributes of its base class



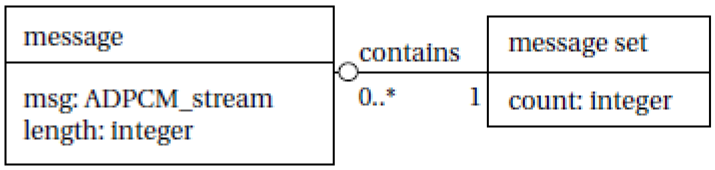
UML also allows us to define ***multiple inheritance***, in which a class is derived from more than one base class. We have created a *Multimedia_display* class by combining the *Display* class with a *Speaker* class for sound. The derived class inherits all the attributes and operations of both its base classes, *Display* and *Speaker*. Because multiple inheritance causes the sizes of the attribute set and operations to expand so quickly, it should be used with care.



A ***link*** describes a relationship between objects; association is to link as class is to object. We need links because objects often do not stand alone; associations let us capture type information about these links.



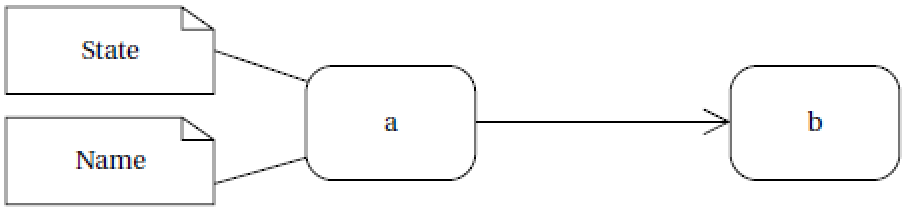
Links between objects



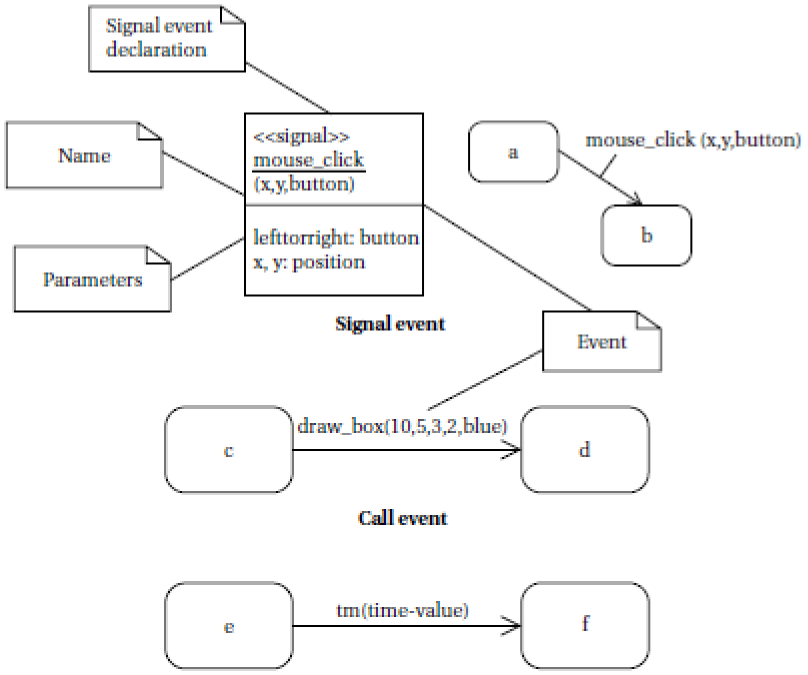
Association between classes

Behavioral Structure

One way to specify the behavior of an operation is a ***state machine***. UML states and transition as show in figure. These 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



We use a certain combination of elements in an object or class many times. We can give these patterns names, which are called ***stereotypes*** in UML. A stereotype name is written in the form <<signal>>.



Signal event

Call event

Time-out event

- 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.