PDHonline Course G368 (2 PDH)

Bridging the Gap between Software and Non-Software Engineers

Instructor: Tracy P. Jong, Esq. and Cheng–Ning Jong, PE

2020

PDH Online | PDH Center

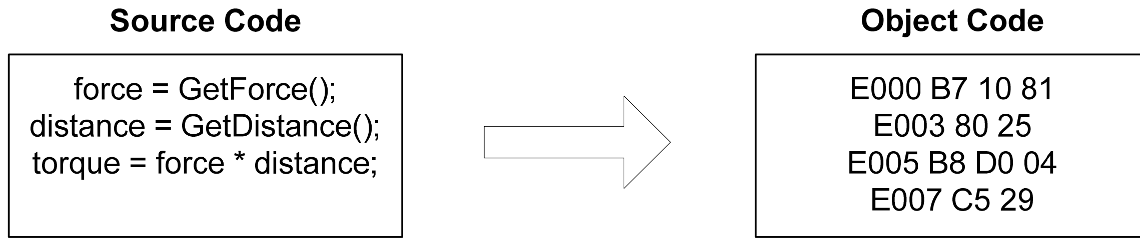
5272 Meadow Estates Drive
Fairfax, VA 22030-6658
Phone: 703-988-0088
www.PDHonline.com

An Approved Continuing Education Provider

Table of Contents

1. How does a line of code turn into machine action? 3
2. Ad Hoc Measures Versus Design 5
3. Issues of Data Acquisition 9
4. Issues of Real Time Control Systems 10
5. Object-oriented Programming 13
6. Unified Modeling Language..17
7. Computer Aided Design (CAD) 20
8. Finite Element Analysis 22

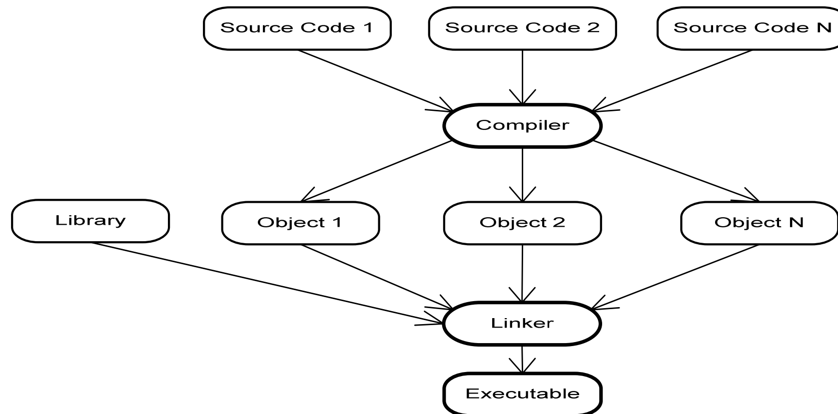
How does a line of code turn into machine action?



A line of code can take on various forms depending on the compiler utilized. A *compiler* is a software program that takes text written in a computer language called *source code* and translates it into another form of computer language called *object code* or simply *object*. A compiler requires the source code to be written in certain syntax so that it can be properly processed.

Even though there are many commercial *Integrated Development Environment (IDE)* tools available to facilitate software development, a simple *text editor* is all that is required to write code. A basic IDE incorporates a source code (text) editor, compiler, linker and debugger. The function of a *source code editor* is very much the same as a normal text editor such as in a typical word processing program. However, a programmer need not be concerned with tab, indentation, margin setting and the like because the compiler's parser is typically designed to handle such details. A *linker's* function is to take one or more objects generated by the compiler and assemble them into an executable program. A *debugger's* function is to facilitate verification and debugging of the source code. Upon debugging, the source code may be relatively free from syntax errors but there is no guarantee that the resulting source code performs according to product requirements.

The following diagram depicts the process of generating an executable file from source code.



How does a text document legible to a human be converted to machine language? The compiler lexically scans the source code and *parses* the text into a symbols table and establishes relationships between functions and variables with their definitions and eventually builds object code. (The "object" code is not to be confused with object oriented programming which will be explained in a later section.) Object code is sometimes referred to as machine code or machine language. In this example, multiple source code files (1, 2…N) are compiled using the "Compiler" to result in multiple object files (1, 2…N). The object files are then linked by the "Linker" to yield an "Executable" file.

Source code may be written in any text editor in high level or low level syntax by a team of software engineers. In preparing source code, a high-level language is not necessarily superior as a tool to a low-level language. Java, C, C++, Fortran, Basic and Pascal are popular high level computer languages used in industry and academia. Low-level languages include Assembly language and machine code. Machine code is native to each microprocessor and it requires no interpretation in order for it to work with the microprocessor. Normally, machine code comes in a series of individual instructions. It is theoretically possible to write machine code directly, however, it is almost never done in practice. Machine code deals directly with hardware memory addressing and even the simplest of software operations may require many machine code instructions.

A high-level language merely describes a higher level of abstraction than a lower-level language. For instance, a high-level language would have human legible syntax and be more compact in its functionality. It frees the software engineer from having to worry about interactions between machine code and computer hardware, such as input/output (I/O) registers, memory management, rudimentary calculations, etc. In comparison, the use of a low-level language such as Assembly requires considerable knowledge of the microprocessor in use and its peripherals (e.g., memory module and I/O registers).

There is another category of high level language which utilizes interpreters instead of compilers. Commercial examples include Perl and Ruby. An *interpreter* fetches lines of code written in a high level language and executes them one line a time without requiring translation to object code first. An interpreted piece of code typically takes longer to run since the interpreted code is not stored and it must be interpreted all over again each time it runs.

In a large software system, each software engineer may be responsible for a discrete portion of code. For example, let's consider an automobile control system. The overall system comprises several discrete control systems, each one managing several components. These control systems might include an engine control system, a transmission control system and a vehicle stability control system. Implementing the entire system may involve several software engineers such that one is responsible for the code for the oxygen sensor and another for the valve controller. It quickly becomes clear why communication and

an integrated approach are fundamental to the process of developing good software applications! Typically, there is at least one software engineer serving as a lead software architect who devises overall connectivity of various functions of the system. Just as a building architect would do, planning ahead for future software extensibility and reuse are among the top priorities of a software architect.

An engine control system may take inputs such as the oxygen content readings of the pre-combusted air/fuel mixture and exhaust gas and uses them to determine outputs in the form of fuel injection valve positions. Each control system component would be represented or modeled in a text file as source code. In a multi-component system, there exists a need to resolve dependencies amongst these components. This is where a linker plays its role. Various software utilities may also be embedded in pre-compiled software libraries which must be linked with user-created object code to produce the executable program. A *loader* is used to put the program in memory. A *Central Processing Unit (CPU)* or a microprocessor then takes each instruction of the executable program and executes it.

Typically, a computer is equipped with I/O registers (data, control and status) and I/O ports. The CPU executes an executable program and works in concert with the I/O registers and I/O ports to control the I/O operations. In the case of motor control, a motor gets its input drive signal from an output port of an I/O port. Most output ports provide only signal level power. Therefore, a separate power supply or amplifier is required for driving the motor. Alternatively, a sensor reading may be obtained by connecting the output signal of a sensor to the input port of the I/O port.

Ad Hoc Measures Versus Design

Most projects start out as small and manageable systems. Overall system design is usually not a critical focus of the project. As an experienced software engineer knows, this can create difficult challenges when requirements change (e.g., new features added to the system or upgraded speeds of existing components) or the system has technical difficulties from unforeseen hardware issues. Occasionally, there are unforeseen software architectural problem which cannot be easily modified.

How many times have you navigated a new city and thought that the street design was nonsensical and anything but user friendly? Without planning at the initial stages decades ago, the many ad hoc additions of streets and bridges over the years have created nothing less than a complicated mess. (in fact, Rochester, NY refers to one such entanglement as its “can of worms!”) Sometimes it seems as though you should wipe the slate clean and start over.

©Tracy P. Jong and Cheng–Ning Jong

You do not want to be part of a project team that has invested eighteen months in a project before you learn that the core design principle it employs has not met the requirements. In the real world, scheduling and cost constraints often force engineers to patch together a system with "temporary" ad hoc measures which do not fit well into the overall engineered system, do not undergo thorough, rigorous design analysis of a well architected design and do not involve well understood engineering practices. Oftentimes, system behavior is not well understood and an ad-hoc measure that works well for one part of the system may cause problems in another part of the system and/or may even cause conflicting responses. For instance, a poorly implemented energy star requirement may inadvertently become active and causes a power supply to turn off while a machine is in use.

In a poorly designed and understood system, engineers may have to resort to empirical tests. Typically, due to cost and time constraints, only a small set of scenarios is used in a test matrix. These selected scenarios set may not represent the realistic range of applications. For example, a test matrix may test only three discrete temperatures. However, temperature variations often affect material properties which can result in system timing changes. To make matters worse, temperature may have a non-linear effect on material properties and system timing. In such cases, critical temperature variations may go undetected until the product is in the field.

A well planned engineering project, on the other hand, uses design principle to provide generic solutions to wide ranging scenarios. While this approach is more costly "on the front end," the same design principle may be applied to other products within a company to provide long term cost efficiencies. An ad-hoc measure is cost effective to implement in the short run but it is not reusable, can create significant field problems, and may be used well past its intended life span.

Designed systems provide an opportunity for some level of standardization in a company's software. Employee turnover occurs over the life span of a particular project. By following some basic design protocols, many employees can work on a single component over time (repairs, upgrades, and the like). With many ad hoc features, a single employee may be the only person familiar with that component and its interplay within the system. The problems this can create hardly need amplification. Additionally, the coordinated project team members can better predict the interplay of various components, effectively planning for them in the implementation of their particular assigned component.

Let's consider a single input single output classical closed loop control system. In this system, a fuser is to be heated to a specified fusing temperature before printing is started and fusing temperature is to be maintained. A thermistor placed in proximity of a fusing surface provides the temperature feedback to the controller.

A set of ill-posed requirements may resemble the following:

(1) Turn on heat lamp at (T_{curr} – 200 degrees C) * 100 % duty cycle until temperature achieves 197 degrees C.
(2) When temperature rises to 200 degrees, start sending sheet to the fuser.
(3) When temperature drops to 195 degrees C, turn on heat lamp until the fuser roll temperature achieves 198 degrees C. Repeat step (2).

This set of requirements resembles a *proportional derivative (PD) controller*. The proportional part arises from the step requiring higher input if the difference between the current temperature and the target temperature is greater. The derivative part stems from the requirement that the proportional control be turned off prior to achieving the target temperature, indicating an overshoot is not desired.

Instead of spelling out all the steps, it would be preferable to develop a requirement such as the following:

Fuser temperature is to be maintained at 200 degrees C with allowable overshoot within 1%.

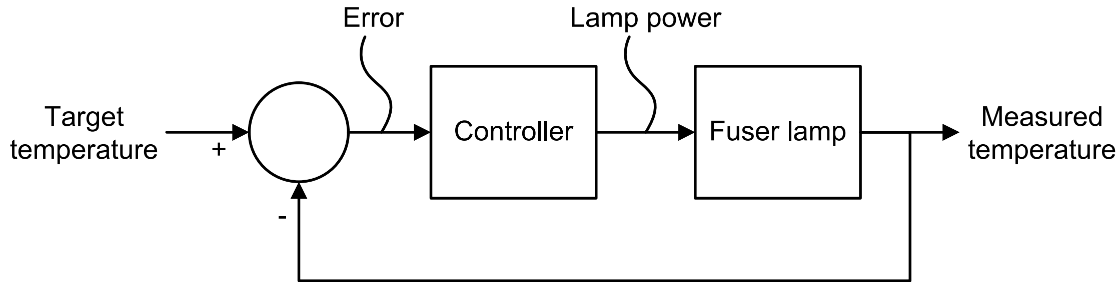
Translation: Target fusing surface temperature = 200 degrees C. Use a Proportional Derivative (PD) or Proportional Integral Derivative (PID) controller. We know the Derivative component is important since we need to keep the overshoot within 1%. There is no mention of the steady state error. So it is an option to include the Integral component. There may be many other considerations such as rise time, coupling between K terms, etc. For simplicity, these considerations were not included in this discussion.

The input power equation may be described as follows:-

$$u = Kp \times e + Kd \times de/dt + Ki \times \int e\, dt$$

where:

u = is the input power to the fuser
Kp = proportional gain (fine tuned and specified by controls engineer)
Kd = derivative gain (fine tuned and specified by controls engineer)
Ki = integral gain (fine tuned and specified by controls engineer)
e = target temperature - current temperature
de/dt = rate of change of error = (e2 – e1) / sampling time
$\int e\, dt$ = summation of error over sampling time period



Armed with the understanding of this set of requirements, the software engineer can then write code to perform PD control. This generic control code takes gain values, measured temperature and target temperature as inputs. A few months down the road, another requirement comes along for controlling motor using the same control strategy. The same PD code that has already been written to control the fuser can also be used for controlling the motor.

In addition to being reusable and extensible, common code needs only be documented once. Suppose the current fuser temperature set point and other control parameters are found to be ineffective in the field and they need to be modified. The only portion that needs to be tweaked is the set of gain values and temperature set point. Such process becomes data driven.

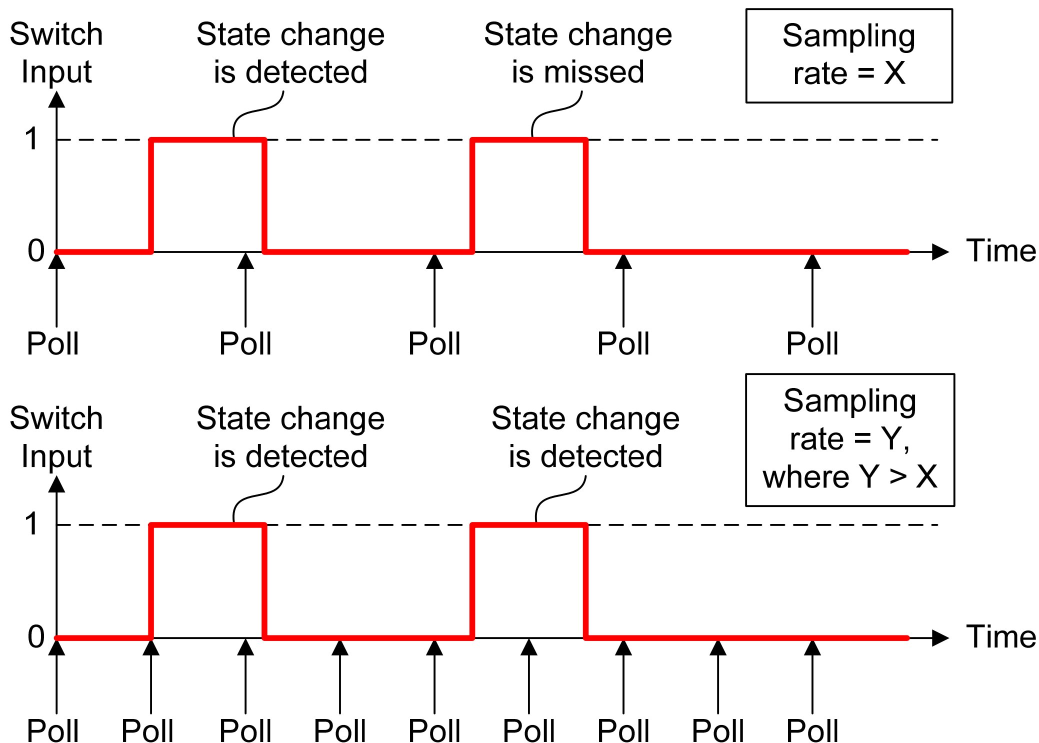
In many systems, data is further separated from control code. Data can be specified in separate data files in one of several formats. Designers often deem certain systems too complicated to model. If a system is generally non-linear, one may break the system down into several control regions. In most cases, there may be linear portions in which controls theory could be applied. In such cases, multiple transfer functions can be designed to selectively run in various regions.

Software engineers are often faced with the lack of requirements. A software implementation often involves two distinct steps. The first step is the design of an engineering solution for a problem and reduction of that engineering solution to a set of requirements. The second is to take the requirements and design a software solution.

During or after the development stage, software must be tested to ensure it meets its product goals. In control type applications, control engineers often rely on a variety of commercial software such as Matlab, MatrixX, etc., to design their control systems. Software engineers may rely on these software programs to generate control data to be verified against results from the real time control system they developed. A sign of trouble arises when software engineers have tremendous freedom to freelance an engineering solution.

Issues of Data Acquisition

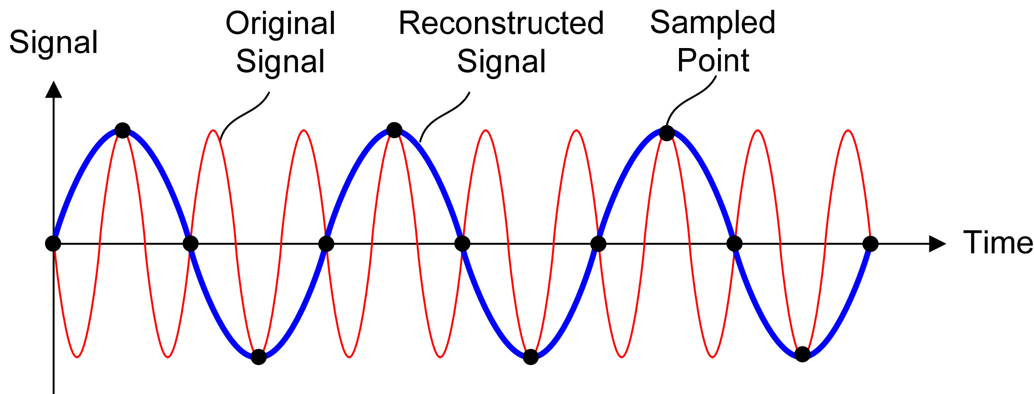
A frequent but often overlooked engineering issue is sampling frequency. In control software engineering, a signal (e.g. a sensor output) that is not sampled at a high enough frequency creates a problem. Consider the following two plots. In the first plot, the sampling rate of X is too low in capturing the second switch state change since the state change occurs between two polling points. If the sampling rate is increased to exceed the switch input frequency, as diagrammed in the second plot, all switch state changes would be properly captured.



Nyquist theorem states that sampling frequency has to be more than twice the rate of the maximum signal frequency. This minimum sampling frequency is called the *Nyquist rate*. In other words, if there is a chance for a sensor to switch state every second, we must sample at a period of less than 500 ms. However, strict application of this theorem ignores real world variables that must be taken into account for a properly functioning control system. Careful selection of data acquisition methods may address some of these factors.

In practice, system frequency variation due to aging, wear and tear, environmental changes, interferences, etc. must be taken into account when setting the sampling rate. For example, an older motor may draw more electrical current for a specified load than a new motor. In a system that uses this current draw as an input to its control system, this could mean a delay in system response if the current draw is polled at a rate originally designed for the system.

In periodic sampling, *aliasing* is another well known problem in which multiple distinct frequencies are represented in a single data set. The following example illustrates that there could be two signals at two distinct frequencies that share the same data points. Aliasing can be avoided by sampling at a rate recommended by Nyquist theorem.



Signal data may be sampled via interrupt or polling. *Polling* refers to a periodic pinging of a machine state, sensor reading, etc., regardless of whether the machine state or sensor reading has changed. Polling is more likely to encounter some of the previously described sampling issues. *Interrupt* refers to asynchronous signal due to a change in machine state or other conditions from hardware indicating the need for attention.

An interrupt occurs when some measured quantity changes. Interrupts are utilized to avoid busy-waiting for an update or change in a measured quantity. As the name suggests, they “interrupt” whatever is happening so that the operating system can respond to this interrupt and return to what it was doing without losing information. Ideally, all systems should use interrupts. Unfortunately, due to system performance demands, economic considerations and hardware related constraints, only a limited number of interrupts may be available in a given system.

Issues of Real Time Control Systems

A *real time control system* is a system in which the proper functioning of the hardware and software it controls depends not only on the logics it executes but also upon the time at which a task is performed by it. As an illustration, a control system for helicopter blade pitch requires cooperation between both software and hardware to respond to an external event. For instance, a helicopter must be capable of adjusting its blade pitch in order to increase or decrease thrust to respond to a wind gust. A helicopter hovering amongst high rises must maintain its position to avoid bumping into buildings or close-by structures. There exists a

time constraint in which a process must occur to correct a deviation to avoid catastrophic failure.

Image Credit: U.S. Department of Energy – Western Area Power Administration, http://www.wapa.gov.

In the example of a vehicle, an antilock brake system must react within a split second to prevent the brakes from locking when wheel slippage is detected.

A multithread system is important in a real time system such that no one time-consuming task is allowed to consume the entire processing power of a microprocessor. In computing language, a *thread* is equivalent to a task. Threads or multithreads are a way for a program to split itself into two or more simultaneously running tasks. The word simultaneous is used loosely to describe a process in which various tasks seem to execute concurrently. In a single processor system, the processor processes a task for a short finite amount of time before moving on to the next task. The task switching occurs so fast that the processor appears to run many tasks simultaneously. Take your PC as an example. While playing back a video on your PC, you are still able to move your mouse to perform other tasks. Without a multithread system, your PC would have to finish playing the video before servicing other requests.

There are potential pitfalls in a multithreaded system. Data access must be properly controlled to ensure data integrity. Generally, task priorities and locks are used to enforce a concurrency control policy between resources. A lock is a mechanism that provides a means to ensure that only one thread can execute a section of code at a time. There are pitfalls associated with lock mechanisms. If not used properly, deadlocks can be created in which task execution is inadvertently locked and does not progress from one task to the next.

Priority may also be used in conjunction with locks. Each task is assigned a priority. A higher priority task pre-empts a currently executing lower priority task. However, care must be taken to avoid priority inversions. *Priority inversion* occurs when a low priority task acquires the same resource shared by a high priority task and the low priority task blocks the execution of the high priority task with a lock. If a medium priority task attempts to run, the medium priority task will pre-empt the low priority task since it does not require the same resource as the low priority task. While the resource shared by the high and low priority tasks is still locked by the low priority task, the high priority task is not able to pre-empt the medium priority task. So the result is delayed execution of the higher priority task. One common solution to this problem is to assign the low priority blocking task the highest priority of the tasks waiting to run to avoid the medium priority task from taking over control.

Let's look at an example. In a helicopter control system, elevation control is achieved by varying the main rotor blade pitch which in turn changes the torque generated by the main rotor. This change in torque tends to change the yaw angle of the helicopter. In response to this change, there must be a change in the tail rotor pitch to counteract the change in torque generated by the main rotor. Consider the following code snippet:

```
Void ElevationControl()
{
  MainRotorBladePitchControl(elevation);
  TailRotorBladePitchControl(elevation);
}

void ElevationSensor()
{
  elevation = ReadSensor();
}
```

Assume the tasks ElevationControl() and ElevationSensor() are set to the same priority. ElevationSensor() is executed periodically. Shortly after executing the function MainRotorBladePitchControl(elevation), ElevationSensor() interrupts and modifies the data resource "elevation." Function MainRotorBladePitchControl(elevation) takes an older copy of "elevation" parameter while TailRotorBladePitchControl(elevation) takes a newly modified copy of "elevation."

It is clear that this could create a problem since the two corresponding functions are now operating on the same resource "elevation" but two different "elevation" values. A *race condition* is said to have occurred. There are two common solutions for this problem. The first solution involves setting ElevationControl() to a higher priority than ElevationSensor(). Therefore, once executed, ElevationControl() must complete before passing the next task on to ElevationSensor(). The second solution involves locking ElevationControl() to prevent ElevationSensor() from interrupting.

In cases where pointers are used, a race condition may even cause program crashes. A *pointer* is a data type whose value directly refers to (or points to) another value stored elsewhere in the memory so an unnecessary copy of a data point can be avoided. If a pointer is deleted prior to being used or otherwise uninitialized, a program crash may occur. In safety critical applications, it is vital to avoid this condition or have an acceptable crash (or exception) handling procedure.

Object-oriented Programming

Many of us have heard of Object-Oriented Programming (OOP). OOP is a programming paradigm that uses abstraction to model real world problems. In simple hardware and software systems, it is possible to add functions to a collection of functions when implementing a new feature without considering the overall design architecture. What if a system requires control functions for multiple types of motors? What if there are multiple units of each type of these motors? Let's look at two types of motor controls in an airplane. The traditional non-object oriented approach would be to write two different motor turn-on functions with each calling some basic functions. With an airplane's many subsystems (e.g., the landing gear, jet engine, control surface, air conditioning, etc.), the number of functions required can quickly balloon and get out of control as illustrated below.

Suppose we want to write code to turn on landing gear and control surface of the airplane. In a non-object oriented approach, we would write a function to turn on a landing gear motor, LandingGearTurnOn(), and a function to turn on a control surface motor, ControlSurfaceTurnOn(). The following example code snippet assumes the landing gear and control surface each has only one motor. TurnOnDCMotor() and TurnOnServoMotor() define how a DC motor and a servo motor should be turned on respectively. Suppose there are other types of motors that we want to add to the landing gear system. There would be a turn on function for each of these motors. Suppose we want to turn off motors, there would be twice as many functions as before.

```
void TurnOnDCMotor()
{
  operatingVoltage = 24;
  TurnPowerOn();
  StartDCMotor();
}

void TurnOnServoMotor()
{
  operatingVoltage = 24;
  TurnPowerOn();
  ServoControlLoop();
}
```

```
void LandingGearTurnOn()
{
  TurnOnDCMotor();
}

void ControlSurfaceTurnOn()
{
  TurnOnServoMotor();
}

Void AircraftController()
{
  // turn on landing gear motor
  LandingGearTurnOnMotor();

  // turn on control surface motor
  ControlSurfaceTurnOnMotor();
}
```

Thanks to OOP, the complexity of this example can be greatly reduced. OPP forces software developer to first come up with some level of design prior to coding. Let's see how OOP can reduce the complexity of this example. Looking at our program, we have some basic motor functions which we could put in a class called Motor(). A *class* defines the attributes or properties and the behavior of the object. These attributes and properties are called member variables and the behavior is implemented as functions called member functions or methods. An object is an instantiation of a class. Put more simply, think of class as a type or mould from which an object is constructed. A basic Motor() class and its TurnOn() implementation may look like the following:-

```
Class Motor
{
  operatingVoltage = 24;
  virtual void TurnOn();
}

Void Motor::TurnOn()
{
  TurnPowerOn();
}
```

Next, let's go on to define other classes. A *base class* has attributes and methods that each motor has, regardless of its type. A *derived class* is a more specialized version of a class that inherits the attributes and methods from the base class. A derived class is a base object *plus* special attributes and methods.

In our example below, DC and servo can be treated as more specialized versions of class Motor, that is, derived classes. In addition to having the basic attributes and methods of class Motor, each derived class (DC and servo) now can have its

own special attributes and/or methods. In our example, the special function is TurnOn().

```
class DCMotor : Motor
{
  Void TurnOn();
}

void DCMotor::TurnOn()
{
  Motor::TurnOn(); // execute TurnOn() method of the base class
  StartDCMotor(); // execute method special to DC motor
}

class ServoMotor : Motor
{
  Void TurnOn();
}

void ServoMotor::TurnOn()
{
  Motor::TurnOn(); // execute basic turn-on method
  ServoControlLoop();// execute method special to servo motor
}
```

But wait! How can a method having the same name as the base class's function be special? In C++, *overriding* a method is possible. A derived class method having the same name as a base class method is executed when this method is called on the derived class. What if the derived class object has been cast to the base class type? In our example, notice the designator "*virtual"* preceding the TurnOn() declaration in class Motor. A derived class method having the same name as a base class virtual method is executed when this method is called on the derived class. Without the virtual designator, the base class method would instead be called. If the implementation of a method does not exist in the derived class, the base class version would be used.

This concept is called *polymorphism*. Given a base class Motor, polymorphism enables the programmer to define different TurnOn() methods for any number of derived classes, such as DCMotor and ServoMotor in our example. No matter what Motor an object is, applying the TurnOn() method to it will return the correct results. In this example, since we override the TurnOn() method, the method TurnPowerOn() would not be called if the base class version is overridden.

One way to solve this problem is to call the base class TurnOn() method explicitly as in Motor::TurnOn() in addition to methods specific to the derived classes. As an illustration, suppose a DCMotor does not require a special turn on method, TurnOn() method would not be implemented in the DCMotor class. In this case, when TurnOn() is called on a DCMotor, the base class TurnOn() would be invoked. In a non-OOP, the software developer would have to create long function names which have no correlations from the design point of view.

Another important feature of OOP is in the ability to *encapsulate* implementation details of a class from an object that uses it. As shown in the last code snippet, each type of motor sweats the details about how it should turn itself on. Therefore, future modifications to the code details can be made only to the TurnOn() methods themselves without affecting other classes. If the modifications were found to introduce software bugs and the program ceased to work properly, the root cause can be more easily traced and contained since the code details are encapsulated. This in turn promotes software maintainability and readability.

Suppose you want to add a turn off method called TurnOff() to the LandingGear class. Simply add TurnOff() method to LandingGear class and define its implementation as follows:

```
class LandingGear
{
  ServoMotor sMotor;
  Void TurnOn();
  Void TurnOff();
}

void LandingGear::TurnOn()
{
  sMotor.TurnOn();
}

void LandingGear::TurnOff()
{
  // define implementation here
}

Class ControlSurface
{
  DCMotor.dcMotor;
  Void TurnOn();
}

void ControlSurface::TurnOn()
{
  sMotor.TurnOn();
}

Void AircraftController()
{
  LandingGear* landingGear = new LandingGear;
  ControlSurface* controlSurface = new ControlSurface;

  // turn on landing gear motor
  landingGear->TurnOn();

  // turn on control surface motor
  controlSurface->TurnOn();
}
```

In AircraftController(), a TurnOn() call on a landingGear object pointer tells the program to invoke the TurnOn() method of a servo motor. A TurnOn() call on controlSurface on the other hand invokes the TurnOn() method of a DCMotor.

Most OOP languages support the use of a feature called *method overloading*. Multiple methods can have the same name but they can take different number and/or type of parameters and they can return data of different types. These methods are said to have different signatures and the signature of a function call determines which method is invoked. It is resolved at compile time which of these methods having the same name will be used. The method overloading feature provides an alternative to case logic as commonly used in a non-OOP implementation. There are several popular OOP software in the marketplace. Most common ones are C++, C#, Java, JavaScript, Python, PHP, Objective-C, etc.

Unified Modeling Language

An Object-Oriented (OO) discussion would not be complete without touching upon its design aspects. *The Unified Modeling Language (UML)* is the lingua franca of software development and its standards are currently defined at the Object Management Group. UML is a collection of best engineering practices that have proven effective in visualizing, communicating, constructing and documenting the artifacts of software systems. The use of UML diagrams encourages exchange of ideas among software engineers and engineers without adequate software development experience since the understanding of such diagrams require only basic knowledge of UML standards. There are several popular UML diagramming software programs in the marketplace. The most common ones are Rational Rose, Smart Draw and Microsoft Visio Professional.

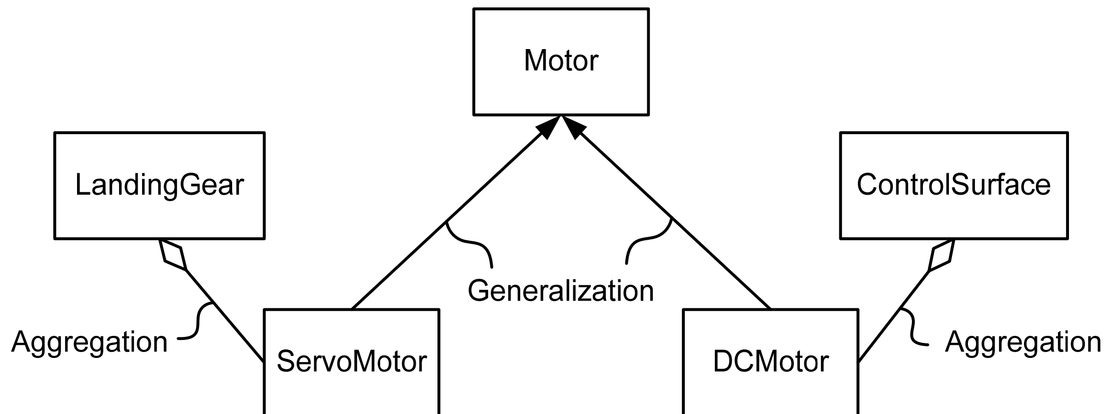
Flowcharts are traditionally used and adequate for designing and documenting small processes that involve few steps. In a non-OOP language, one would model a control system based on its steps. While one may still model the steps at the lowest level in an OOP language, there are other tools which visually communicate to us the relationships between classes (inheritance, containment, class variables, class methods, etc.), how they interact (what methods are executed and what data is passed between objects with respect to time progression, etc.) There are many types of UML diagrams which can be used for designing and documenting the relationships between elements of software models. Several types that are commonly used in control applications are:

- Class diagrams
- Interaction diagrams (or sometimes called sequence or scenario diagrams)
- State diagrams
- Use case diagrams

©Tracy P. Jong and Cheng–Ning Jong

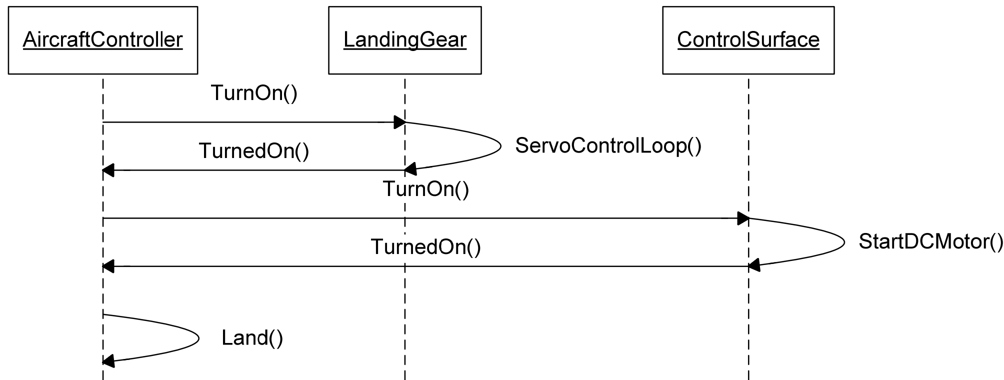
A class diagram is a diagram showing a collection of classes and interfaces and collaborations and relationships among classes and interfaces. An interaction diagram is a diagram showing a collection of objects and interactions among the objects (message exchanges) arranged in time sequence. A state diagram is a diagram showing a collection of finite number of states, transitions between those states, actions that cause these transitions and the actions taken as a result of the transitions. A use case diagram is a diagram showing primary elements (called actors) and processes (called use cases) that form the system. The use case diagram shows interactions between these actors and use cases.

Let's look at our aircraft example below in which a class diagram is utilized to model the relationships between classes. ServoMotor and DCMotor classes both "inherit from" Motor or ServoMotor "is a" Motor and ControlSurface "is a" Motor. Inheritance models the "is a" relationship and captures the generalization of ServoMotor and ControlSurface in code. Inheritance is represented by drawing an arrow pointing from the derived class to the base class. Another important relationship is the "has a" relationship. Generally one object is composed of many other objects. In the same example below, we can say that LandingGear "has a" ServoMotor and ControlSurface "has a" DCMotor. Containment models the "has a" relationship by drawing a line with a diamond from the containing object to the contained object.



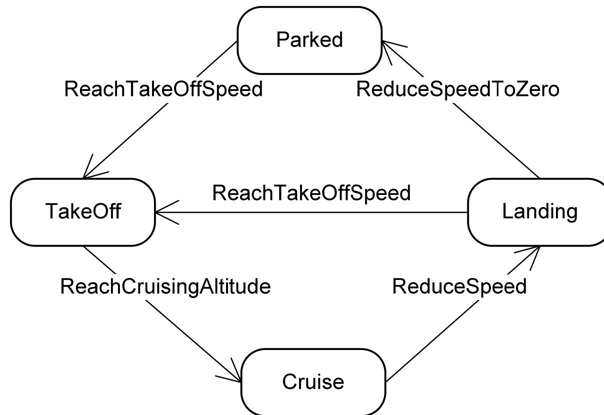
In addition to modeling the system based upon the relationships among classes, it is important to model *how* they interact. *Interaction diagrams* show the interaction among a number of objects in time progression. In this example, an interaction diagram can be drawn to model what we want the LandingGear and ControlSurface to do. This diagram shows that the aircraft controller first turns on the landing gear. The landing gear is deployed by calling the ServoControlLoop() function. Upon execution of this function, the landing gear notifies the aircraft controller that the landing gear has been deployed and that it is time to adjust the control surface to prepare for landing. The aircraft controller then turns on the control surface by calling TurnOn() on the control surface. As a result StartDCMotor() is executed and the control surface is adjusted. Once

adjusted, the control surface notifies the aircraft controller that the aircraft is ready to land. The aircraft controller then executes the landing function.



Obviously, this is an unrealistic example which involves only three objects. In a real project, there would be many more objects, inputs (sensors, customers, etc.) and outputs (display result, etc.).

Let's look at an example of a state diagram below. The rounded rectangles represent states and the arrows represent transitions. The text of each arrow represents the event causing the state transition. At any moment, the aircraft is in one of the four states. While in the parked state, increased ground speed to the take off speed causes the aircraft to transition from the parked state to the take off state. In the take off state, the aircraft maintains its ascent by adjusting the engine propulsion and control surface settings. Eventually the aircraft reaches the desired cruising altitude and aircraft transitions to the cruise state. Upon reaching its destination, the air speed is reduced to anticipate landing. When the landing speed is achieved, the aircraft transitions into the landing state. In this state, the landing gear is deployed and the aircraft adjusts its bearing to land on the runway. Upon landing, the aircraft then taxies to the terminal and eventually comes to a halt. The aircraft then transitions into the parked state to deplane.



Computer Aided Design (CAD)

Computer-Aided Design (CAD) is the use of computer tools in design activities. In a product life-cycle, CAD is also tightly coupled with Computer-Aided Manufacturing (CAM). CAD software can be used to generate a 2D or 3D model of a component. A CAM system takes a CAD model and generates Computer Numerical Control (CNC) code which drives numerically controlled machine tools. Many component manufacturers have also made their scaled CAD drawings and models available to customers and potential customers to determine how their components may be incorporated into end products. There are numerous commercial 2D design software packages. Most commercial 3D CAD software packages have built-in 2D capability. Although not considered a CAD tool, Microsoft Visio and SmartDraw have gained prominence in recent years as a rapid diagramming tool for engineers in many disciplines.

2D models are made up of some combination of geometrical primitives such as points, lines, polylines, curves, circles, polygons, and the like. Many CAD software vendors also include templates which can be used for rapidly setting up a model substructure such as layers, unit of measure, drawing aids and orientation. The layering concept is an important one since it allows one to focus on a particular feature of a drawing. For instance, in a building plan, a basic building layout of rooms may be complemented with data wiring, electrical wiring and plumbing diagrams. Much like software development, CAD drafting requires planning for extensibility. A well thought out drawing would have each of these key diagrams drawn in separate layers. For instance, while dealing with an electrical contractor, all but the basic room layout and electrical wiring diagram can be turned off to focus discussion on electrical concerns. If the contractor is interested in potential spatial interference of electrical wiring with plumbing, then the plumbing layer could also be enabled.

Although not required, 3D modeling has become the starting point for most well-planned hardware design activities. 3D modeling can be considered as the superset of 2D modeling. All modern 3D CAD software has the ability to

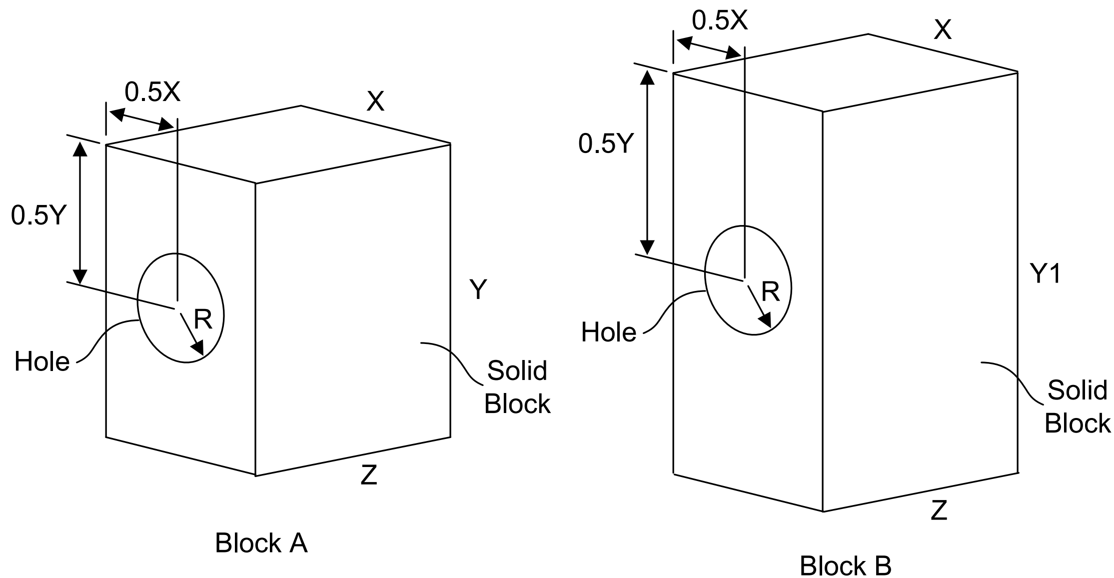
generate 2D drawings from 3D models. Once a view is captured from the 3D model, a 2D drawing can be dimensioned and annotated much like a typical 2D drawing.

During the early days of 3D modeling, a part might be created using a combination of the these operations: extrusion or sweeping along a path of a 2D line drawing, subtraction, interference, cutting and union operations of various primitive 3D components such as cylinders, spheres, cones, etc. If a new part of similar features but different dimensions were desired, a new model had to be created from scratch. Alternatively, steps of the modeling process had to be reversed to revert to the primitive solid to be modified. Solids created using such means are termed “dumb solids.” Some of the most well known software packages using this technology are AutoCAD and CADKEY.

One notable feature of new generations of 3D CAD tool is their parametric feature based modeling capability. SolidWorks, SolidEdge, Autodesk Inventor and Pro-Engineer are among software packages that incorporate parametric feature based modeling. In parametric feature based modeling, each feature has attributes which can be parameterized. For example, a pipe may be defined by two features. The first feature is a base cylinder which has an area (parameter) and a length (parameter). The second feature is a hole which has a radius (parameter). In order to define a pipe, the hole must be constrained to have a diameter smaller than that of the base cylinder and if the wall thickness is to be uniform, the two solids must also be concentric.

A solid created using parametric feature based modeling is defined by the collection and order of creation of features. This order of creation requirement forces the designer to consider manufacturing steps in addition to design steps.

Let’s look at another example. Consider the parts below. Block A has a combination of two features, i.e., a block and a hole. Each feature has attributes associated with them. Block A is composed of a block of volume XYZ and a hole of radius R centered on the XY surface. If Block A were a dumb solid, it would take as much effort to create Block B as in Block A because all of the operations in creating the solid block and a hole would be repeated. However, if parametric feature based modeling is employed, Block A can be reused and modified to create block B. Dimension Y of Block A is changed to Y1 to make Block B. Since the hole feature has been defined in Block A to be centered on the XY face and having radius of R, the relative placement of the hole with respect to XY face remains the same and needs no modification.



Finite Element Analysis

Finite Element Analysis (FEA) is a computer simulation technique used in engineering analysis. There are many commercial software programs designed to perform FEA. Algor, Pro-Engineer, CATIA, ABACUS, Fluent, etc. are popular software packages. FEA is generally used whenever a closed form analytical solution is not possible. In FEA, numerical method is performed on a *network of nodes* to provide an approximation of the objective function a model is designed to compute. Numerical methods are used extensively not only to calculate stresses and deformations in physical structures, but also used extensively in computational fluid dynamics (CFD), heat transfer and vibration analysis.

A wide variety of objective functions are available for optimization. Stress, strain, volume, temperature, displacement, pressure, etc. are commonly analyzed in engineering design. One of the most important aspects of FEA is the ease in which failure analysis can be conducted without ever building a physical model.

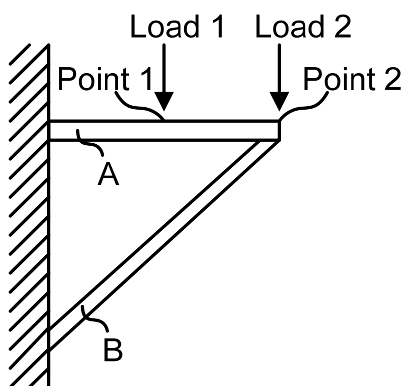
An FEA project includes three distinct steps. The first portion is called *pre-processing* and it involves defining a model using nodes or elements. It often starts with generating a geometric model in a CAD software package and importing it into an FEA package. This geometric model can be represented in 1-D, 2-D or 3-D depending on the goals of an analysis. As an illustration, a simple beam deflection due to weight may be represented as a 2-D model while a 3-D model may be required when wind loading is considered in addition to weight.

The FEA modeling decision of a part is typically based on theory of elasticity and deformation theory of elasticity. An FEA software package typically has a suite

of features, for example, holes, thin wall, beam, spring, viscous damping, material properties, etc. A model would be comprised of a combination of such features approximating the object being modeled. A complex model may be simplified by removing non-essential features so that effort can be focused on analyzing key features. Decisions may also be made to simplify elements by replacing thin wall with shell elements.

Once a model is available, the model is further broken into elements through a process called *meshing*. Each element is bounded by a set of nodes. A *mesh* is a grid of nodes - think of it as a result of discretization of an object that is physically continuous. Generally, discretization error is minimized by increasing the mesh resolution at the cost of increased computation. Regions of interest, e.g., ones thought to experience higher stress for instance, fracture points, corners, fillets and other types of stress concentrators, are assigned higher node density.

The next step involves analysis of the model. Suppose one wish to find the maximum allowable load applied to a bracket. This bracket has a first bar (A) which is fixedly supported on the first end and fixedly supported on the second end by a second bar (B) whose other end is also fixedly supported. Suppose an analytical solution does not exist for this problem. By inspection, there could be two weak points in this bracket, i.e., the center (point 1) and the second end (point 2) of bar A. A point load is applied to the center of bar A and increased until deformation in the bracket occurs or when stress exceeds the material yield strength. This point load simulation is repeated for the second end of bar A. The allowable load is the lower load of the two test points causing deformation at point 1 and point 2.



There is a variety of loading conditions that are commonly tested for: gravity, centrifugal force, pressure, thermal, heat flux, enforced displacements, etc. Without simulation, destructive testing would be required. What if management decides to change the length and diameter of those bars? A whole new set of brackets would have to be built and a new destructive testing would need to be performed.

There are several numerical methods commonly used in underlying computation of an FEA. Finite Element Method (FEM), Finite Difference Method (FDM), Finite Volume Method (FVM), Boundary Element Method (BEM), etc. BEM are more suited for surface type analysis while FDM is one of the simplest ways in approximating differential operators and in solving differential equations.

The next step involves interpreting the simulation result using some form of visualization aid. Though the result is available in its raw form, color coding and graphics enable rapid inspection for problem areas and opportunities for optimizations.