## Overview of Software Applications

It is somewhat difficult to develop meaningful generic categories for software applications as the increasing complexity of software has made it difficult to classify applications into neat compartments. However the following software areas indicate the breadth of potential applications.

### System Software

System software is highly specific to one domain and not easily adaptable to other environments. System software can be classified in one of two ways

- It is a collection of programs written to service other programs. Examples include
  - Operating systems
  - Compilers
  - Editors
  - Device drivers
- Software written specifically to solve one well defined and highly specific problem, e.g. control of an industrial application or process such as the production line of an automobile plant, a nuclear reactor or a fly-by-wire aircraft. In this case, system software is often an embedded application when it may not be apparent to the user that there is indeed a computer inside the system.

In general system software is characterised by heavy interaction with computer hardware and highly specialised applications. These characteristics are what make such software difficult to 'port' of translate to other environments.

## Real-Time Systems and Software

Real-time software is an example of both *system software* and, more often than not, *embedded software*. That is such software concerns itself with software solutions targeted at highly specific problems in which the computer and software may not be visible to the user.

There is no single, all embracing definition of what constitutes a real-time system or its software. Indeed some popular definitions put forward may well apply to situations that may be classed as *non* real-time, but the most popular of these definitions are listed below.

It should be pointed out that a real-time system does not have to meet *all* of these definitions to be so classified. Furthermore, an actual real-time system may act contrary to one or more of these definitions, but agree with others. The definitions are listed mearly to give an indication of the sort of behaviour one could expect from a real-time system.

## Popular Real-Time Systems and Software Definitions

. . . *"A real time system is a* ***controlling*** *system taking in information from its environment, processing it and responding to it."*

. . . *"A real time system reacts, responds and alters its actions so as to effect the environment in which it is placed."*

. . . *"A real time system implies some air of* ***criticality*** *in the response of the system to its external environment."*

. . . *"A real time system is one where the* ***correct*** *answer at the wrong time is the* ***wrong*** *answer"*

. . . *"A real-time system does not have to mean fast, it just means 'timely' which varies from mSec to mins depending upon system"*

. . . *"A real time system has a guaranteed, calculatable (deterministic), worst case response time to an event under its control.".*

## Classifications of Real-Time Systems

Broadly speaking, real-time systems can be classified into two categories, based upon their responsiveness to the external environment, the categories include

➢Hard Real-time systems

➢Soft Real-time Systems

**Hard Real Time Systems.**

A hard real-time system is one in which a failure to meet a specified response results in overall system failure.

A hard real-time system will have a specified maximum delay to a response, which can then be used to judge failure.

Examples of a hard real time system might include

(1) A robot or production line failing to assemble a component within the time allotted to it.

(2) A Railway level crossing system failing to detect a train approaching in time.

(3) Fuel injection management system in a car.

In other words, failure of a hard real-time system usually means some catastrophic failure of the system perhaps resulting in loss of life, or causing damage. These systems are 'time critical' and often safety critical.

**Soft Real Time Systems**

A soft real-time system implies that failure to meet a specified response time merely results in system degradation not necessarily outright failure.

Of course system degradation ultimately becomes system failure if the response is intolerable. This can happen if the response that may have initiated the action has disappeared before it can acted upon.

> *"...A Soft Real time system has a **typical**, or **average** response time against which degradation can be judged. This response time is not fixed and may improve or degrade depending upon loading"*

**Example Soft real time systems**

(1) An Elevator controller. There is no maximum delay specified for the system by which failure can be judged, but the manufacturers may specify a suggested or average response time to a request.

(2) Cash dispenser for a bank.

## Business Software

Business software is probably the largest application area for software development today. Examples of business software include

- Information systems
- Databases
- Payroll

Software in these areas often access large information data bases and re-structure the information to present it in many different ways to facilitate management decision making. This is why such software is often referred to as MIS software of Management information systems software.

A simple example of this might be an excel spreadsheet that can access information from a file and display it in literally dozens of different ways from tables to pie-charts to histograms etc, in other words the emphasis is on the way that data is summarised and presented. Other examples might include

- Revenue Canada ability to access all your tax contributions based upon the entry of a S.I.N number
- The ability of the police to access your criminal record based on an ID or address.
- The ability of ICBC to recall the terms and conditions of your Vehicle insurance based upon a licence plate.
- A personnel departments ability to access information about your employment (Position, home address, terms and conditions and contract, salary, length of service etc) based on your name and department

## Engineering and Scientific Software

Traditionally this field of software development has encapsulated mostly number crunching applications and/or the production of libraries of algorithms to solve mathematical problems. Traditional applications include

- Astronomy, e.g. imaging enhancement algorithms, predicting orbits, mapping star/planet orbits
- Volcanology and earth quake prediction
- Finite element analysis for predicting stress in materials and how shapes, (such as car) buckle and deform in impacts

More recently the emphasis has been on computer simulation and computer aided design, e.g. designing virtual components such as Aircraft, cars, production line robots etc.

## Embedded software

Embedded software includes a broad range of applications where the use of a computer in the production of a system may not be obvious to the end user. Typically embedded software is based around small embedded micro-controllers such as Intel 8051, Motorola 6811 and, at an even simpler level, PIC devices

Examples of embedded software include microwave ovens, CD players, engine management systems in Cars. Think about how many small embedded devices exist in your Home PC, you should easily be able to come up with 10. Now think about how many embedded system exist within a typical luxury Car.

Rather than put this into my own words, here is an excellent article outlining the nature of and problems associated with designing real-time embedded systems.

(Extract from: Real-Time UML,BP Douglass, Addison-Wesley ISBN:0-201-65784-8)

If you read the popular computer press, you would come away with the impression that most computers sit on a desktop (or lap) and run Windows. In terms of the number of deployed systems, embedded real-time systems are orders of magnitude more common than their more-visible desktop cousins. A tour of the average affluent American home might find one or even two standard desktop computers, but literally dozens of smart consumer devices, each containing one or more processors. From the washing machine and microwave oven to the telephone, stereo, television, and automobile, embedded computers are everywhere They help us to toast our muffins and to identify mothers-in-law calling on the phone. Embedded computers are even more prevalent in industry. Trains, switching systems, aircraft, chemical process control, and nuclear power plants all use computers to safely and conveniently improve our productivity and quality of life (not to mention, they also keep a significant number of us gainfully employed) .

The software for these embedded computers is more difficult to construct than it is for the desktop. Real-time systems have all the problems of desktop applications plus many more. Non-real-time systems do not concern themselves with timelines, robustness, or safety - at least not to the same extent as real-time systems. Real-time systems often do not have a conventional computer display or keyboard, but lie at the heart of some apparently non-computerized device. The user of these devices may never be aware of the CPU embedded within, making decisions about how and when the system should act. The user is not intimately involved with such a device as a computer per se, but rather as an electrical or mechanical appliance that provides services. Such systems must often operate for days or even years, in the most hostile environments, without stopping. The services and controls provided must be autonomous and timely. Frequently, these devices have the potential to do great harm if they fail *unsafely*.

An *embedded system* contains a computer as part of a larger system; it does not exist primarily to provide standard computing services to a user. A desktop PC is not an embedded system unless it is within a tomographical imaging scanner or some other device. A computerized microwave oven or VCR is an embedded system because it does no "standard computing." In both cases, the embedded computer is part of a larger system that provides some noncomputing feature to the user, such as popping corn or showing Schwarzenegger ripping telephone booths from the floor.'

Most embedded systems interact directly with electrical devices and indirectly with mechanical ones. Frequently, custom software, written specifically for the application, must control the device. This is why embedded programmers have the reputation of being "bare-metal code pounders." You cannot buy a standard device driver or Windows VxD to talk to custom hardware components. Programming these device drivers requires very low-level manipulation and intimate knowledge of the electrical properties and timing characteristics of the actual devices.

Virtually all embedded systems either monitor or control hardware, or both. Sensors provide information to the system about the state of its external environment. Medical monitoring devices, such as electrocardiography (EGG) machines, use sensors to monitor patient and machine status. Air speed, engine thrust, attitude, and altitude sensors provide aircraft information for proper execution of flight-control plans. Linear and angular position sensors sense a robot's arm position and adjust it via DC or stepper motors.

Many embedded systems use actuators to control their external environment or guide some external processes. Flight-control computers command engine thrust and wing and tail control surface orientation so that the aircraft follows the intended flight path. Chemical process control systems control when, what kind, and the amounts of reagents added to mixing vats. Pacemakers make the heart beat at appropriate intervals, with electrical leads attached to the walls inside the (right-side) heart chambers.

Naturally, most systems containing actuators also contain sensors. While there are some open-loop control systems,[2] the majority of control systems use environmental feedback to ensure that the control loop is acting properly.

Standard computing systems react almost entirely to the user and nothing else.[3] embedded systems, on the other hand, may interact with the user but have more concern for interactions with their sensors and actuators.

One problem that arises with environmental interaction is that the universe has an annoying habit of disregarding our opinions of how and when it ought to behave. External events are frequently not predictable. The system must react to events when they occur rather than when it might be convenient. To be of value, an ECG monitor must alarm quickly following the cessation of cardiac activity. The system cannot delay alarm processing until later that evening, when the processor load is less. Many embedded systems are reactive in nature, and their responses to external events must be tightly bounded in time. Control loops, as we shall see later, are very sensitive to time delays. Delayed actuations destabilize control loops.

Most embedded systems do one or a small set of high-level tasks. The actual execution of those high-level tasks requires many simultaneous lower-level activities. This is called *concurrency*. Since single-processor systems can do only one thing at a time, they implement a *scheduling* policy that controls when tasks execute. In multiple-processor systems, true concurrency is achievable because the processors execute asynchronously. Individual processors within such systems schedule many threads pseudo-concurrently (only a single thread may execute at any given time, but the active thread changes according to some scheduling policy), as well.

Embedded systems are usually constructed with the least expensive (and, therefore, less powerful) computers that can meet the functional and performance requirements. Embedded systems ship the hardware along with the software, as part of a complete system package. As many products are extremely cost sensitive, marketing and sales concerns push for using smaller processors and less memory. Providing smaller CPUs with less memory lowers the manufacturing cost. This per-shipped-item cost is called *recurring cost*; it recurs as each device is manufactured. Software has no significant recurring cost, all the costs are bound up in development, maintenance, and support activities, making it appear to be free.[4] This means that choices are most often made to decrease hardware costs while increasing software development costs.

Under UNIX, a developer needing a big array might just allocate space for 1,000,000 floats with little thought of the consequences. If the program doesn't use all that space, who cares? The workstation has hundreds of megabytes of RAM and gigabytes of virtual memory in the form of hard disk storage. The embedded-systems developer cannot make these simplifying assumptions. He or she must do more with less, which often results in convoluted algorithms and extensive performance optimization. Naturally, this makes the real-time software more complex and expensive to develop and maintain.

Embedded developers often use tools hosted on PCs and workstations but targeted to smaller, less-capable computer platforms. This means they must use cross-compiler tools, which are often more temperamental than the more widely used desktop tools. In addition, the hardware facilities available on the target platform, such as timers, A/D converters, and sensors, cannot be easily simulated on a workstation. The discrepancy between the development and the target environments adds time and effort for the developer wanting to execute and test his or her code. The lack of sophisticated debugging tools on most small targets complicates testing, as well. Small embedded targets often do not even have a display on which to view error and diagnostic messages.

Frequently, the embedded developer must design and write software for hardware that does not yet exist. This creates very real challenges because the developer cannot validate his or her understanding of how the hardware functions. Integration and validation testing become more difficult and lengthy.

Embedded systems must often run continuously for long periods of time. It would be awkward to have to reset your flight-control computer because of a General Protection Fault while you're in the air above Newark airport. The same applies to cardiac pacemakers, which last up to 10 years after implantation. Unmanned space probes must function properly for years on nuclear or solar power supplies. This is different from desktop computers that may be frequently reset. It may be acceptable to reboot your desktop

PC when you discover one of those hidden Excel "features," but it is much less acceptable for a life support ventilator or the control avionics of a commercial passenger jet.

Embedded system environments are often computer-hostile. In surgical operating rooms, electrosurgical units create electrical arcs to cauterize incisions. These produce extremely high EMI (electromagnetic interference) and can physically damage unprotected computer electronics. Even if the damage is not permanent, it is possible to corrupt memory storage, degrading performance or inducing a systems failure.

Apart from increased reliability concerns, software is finding its way ever more frequently into safety systems. Medical devices are perhaps the most obvious safety related computing devices, but computers control many kinds of vehicles, such as aircraft, spacecraft, trains, and even automobiles. Software controls weapons systems and ensures the safety of nuclear power and chemical plants. There is compelling evidence that the scope of industrial and transportation accidents is increasing[5]

For all the reasons mentioned above, developing for embedded software is generally much more difficult than for other types of software. The development environments have fewer tools, and the ones that exist are often less capable than those for desktop environments or for Big Iron mainframes. Embedded targets are slower and have less memory, yet must still perform within tight deadlines. These additional concerns translate into more complexity for the developer, which means more time, more effort, and (unless we're careful, indeed) more defects than standard desktop software of the same size.

[2] An open loop system is one in which feedback about the performed action is not used to control the action. A closed loop system is one in which the action is monitored and that sensory data is used to modify the action.

[3] It is true that behind the scenes even desktop computers must interface with printers, mice, keyboards, and networks. The point is that they do this only to facilitate the user's whim

[4] Unfortunately, many companies opt for decreasing (primarily hardware) recurring costs without considering all the development cost ramifications.

[5] It is not a question of whether safety-critical software developers are paranoid. The real question is, "*are they paranoid enough?*"

## Web Based (Client-Server) Software

Web based software is a fairly new (year 2000+) area of software development but is exploding rapidly.

Web based software is based around the idea of a Client and at least one Server computer connected via a network such as the World Wide Web. The client is the machine the customer sits in front. He/She interrogates a server machine with the aid of a *'browser'*, a package able to display Hypertext mark up language (HTML) content which is both graphical, textual and occasionally multi-media (sound and pictures) and can be produced easily from within a package such as Microsoft word. Typical browsers include Internet Explorer or Netscape Navigator to name but two.

The idea is simple. A business wishing to advertise some product or service publishes a web-page on their server outlining, in HTML, anything they wish to say or advertise. A potential customer wishing to read this content directs their client computer browser to the location of the web-page on the server using a *URL* (universal resource locator) which is a unique address. The server downloads the web-page (HTML content) file to the clients browser which then displays it to the user.

An important aspect of web-based development is the ability of the user to *'surf'* from one web-page to another (possibly on another machine in another country) with content of similar interest by following a trail of *Hyperlinks* within the web-page itself. These hyperlinks are shortcuts to other URLs and are documented in the web-page itself, the user just clicks them and is taken there instantly. At a simple level they can be used to add structure and depth to a web-page in much the same way that directories add structure and depth for organising files of related documents.

The first generation web-pages were fairly static affairs that offered nothing more than static content to the client machine for display. In other words there was no interaction. More recently, web-based *forms* have emerged that allow the user to enter and submit information to the server and get further forms or information in return.

This in turn has given rise to the concept of *e-commerce* and the ability to order/reserve/purchase products on-line using a simple browser connecting to a server hosted 'form' via the Web. In other words, web-based development these days is about developing the applications that sit on the back-end of web-pages rather than the web-pages themselves.

Until recently much of the software used to develop simple web-based applications has been based around languages such as Pearl, Javascript and the Common Gateway Interface (CGI) which have been developed specifically to be browser and web-aware and make the job of interacting over the web easier, however they are very primitive and frustrating to write (a bit like going back to writing scripts in BASIC).

More recently Java has been employed to create 'servlets' which are small bits of java code embedded within the web-page which are downloaded to the client machine and execute on that machine to process the information contained in the form.

Applets by contrast are complete applications that are downloaded from the server to the client and run within a secure environment. In other words, the processing is 'hived off' to the client machine. Those interactive games you sometimes come across on the web, or the more sophisticated applications such as those run by Amazon.com are generally developed as an Applet.

Applets are generally written using full blown Java and are designed with the look and feel of a standard windows (for example) interface. However, instead of running directly under in their own 'window' they run under a window provided by the browser which provides a protective wrapper around the application to ensure that it cannot access or corrupt sensitive facilities of the client machine. Without such security, applets would be the perfect vehicle for distributing viruses.

More recently technologies like CORBA (an open/international standard) and COM (a Microsoft proprietary standard) have been promoting the use of a distributed object model. Here the objects that form the application can be put on many different machines distributed throughout the world and applications can be put together by locating these objects at run time to provide a service.

For example an on-line reservation system for booking a flight could have a 'user-front-end' object that provided the GUI interface for the customer to fill in their reservation situated in the UK, while the database to keep track of bookings could be in the USA. The billing/debiting object for charging customers could be in Canada. In other words, your application doesn't have to sit on one machine anymore, it can be distributed around the world.

Many of the issues surrounding e-commerce and web-based applications today have to do with making on-line decisions safe and secure for the user, using encryption and digital certificates so that confidential details such as personal information and credit card details do not fall into the wrong hands. Other types of Client Server software include

- File servers (i.e. mapped directory drives)
- Print servers
- Email
- Databases

## Artificial Intelligence (AI) Software

This type of software concerns itself with solving complex problems for which there is no readily available or understood algorithm that can be applied. In other words the solution may not be amenable to computation or straightforward analysis. Such systems are designed to learn from their exposure to a problem and gradually, through a process of feedback, evolve a 'best fit' solution, such systems are sometimes know as *expert systems* and often employ *genetic algorithms* designed to mutate the software leading hopefully to software that gets better each time. Examples include speech recognition, simulated intelligence (for use in robots) and Game playing strategies (chess computers for example).

## Safety Critical Systems/Software

In this type of system, a failure can result in injury, loss of life or major environmental damage and thus the overriding concern is to make the system safe in the event of failure. Example include fly by wire aircraft, nuclear power stations process control systems, chemical plants etc.

As one might expect, the costs of failure for a critical system are often very high, and may well include the cost of the equipment being controlled (which may well be destroyed), and the subsequent compensation and clean-up costs that may arise.

Generally speaking, failures in systems can occur for a number of reasons including

- System hardware may fail because of mistakes in its design or because components fail due to manufacturing or fatigue (i.e. they get old)
- Software may fail because of mistakes in its specification, design, coding or test
- Human operators that fail to interact with the system in the way it was anticipated

## Dependability of Safety Critical Systems

The dependability of a system essentially means the degree of user confidence that the system will operate as they expect and that the system will not ‘fail’ under normal circumstances. In essence dependability relies upon.

- **Availability** – The probability at any instant in time that the system is up and running and able to deliver the service it has been designed. For example a system that has 99% availability may mean that a user can probably access that system 99 times out of 100. In other words it may be ‘down’ or unavailable for 1 min in every 100.
- **Probability of Failure on Demand (PFD)** – A measure of the likelihood that a system will fail when a request is made of it. A PFD of 0.001 means that the system is likely to fail once every 1000 requests
- **Reliability** – The probability that over a given period of time the system will deliver correct service, i.e. if the system is in continuous use, how long a period of time could the user expect the system to work for without a failure.
- **Safety** – This is a judgement of how likely it is that the system will cause damage to people and/or environment. Safety is a difficult term to quantify numerically but systems can be thought of in terms of **Safety Integrity levels** (SILs) with 1 being the lowest and 4 being the highest. For example, a railway signalling system might be at SIL 3. (Take a look at http://www.iceweb.com.au/sis/target_sis.htm for more details on SILs)

A qualitative view of SIL has slowly developed over the last few years as the concept of SIL has been adopted at many chemical and petrochemical plants and standardised by IEC 61508 and ANSI/ISA S84.01-1996. As shown below this qualitative view can be expressed in terms of the consequence of the SIS failure, in terms of facility damage, personnel injury, and the public or community exposure.

**Qualitative view of SIL**

| SIL | Generalized View |
|---|---|
| **4** | Catastrophic Community Impact |
| **3** | Employee and Community Impact |
| **2** | Major Property and Production Protection. Possible Injury to employee |
| **1** | Minor Property and Production Protection |

The table below explains what the various SIL levels mean in terms of availability and the probability of failure on demand

| Safety Integrity Level | | | Availability Required | Probability to Fail on Demand | 1/PFD |
|---|---|---|---|---|---|
| IEC 61508 | | **4** | >99.99% | E-005 to E-004 | 100,000 to 10,000 |
| | ISA S84 | **3** | 99.90-99.99% | E-004 to E-003 | 10,000 to 1,000 |
| | | **2** | 99.00 - 99.90% | E-003 to E-002 | 1,000 to 100 |
| | | **1** | 90.00 - 99.00% | E-002 to E-001 | 100 to 10 |

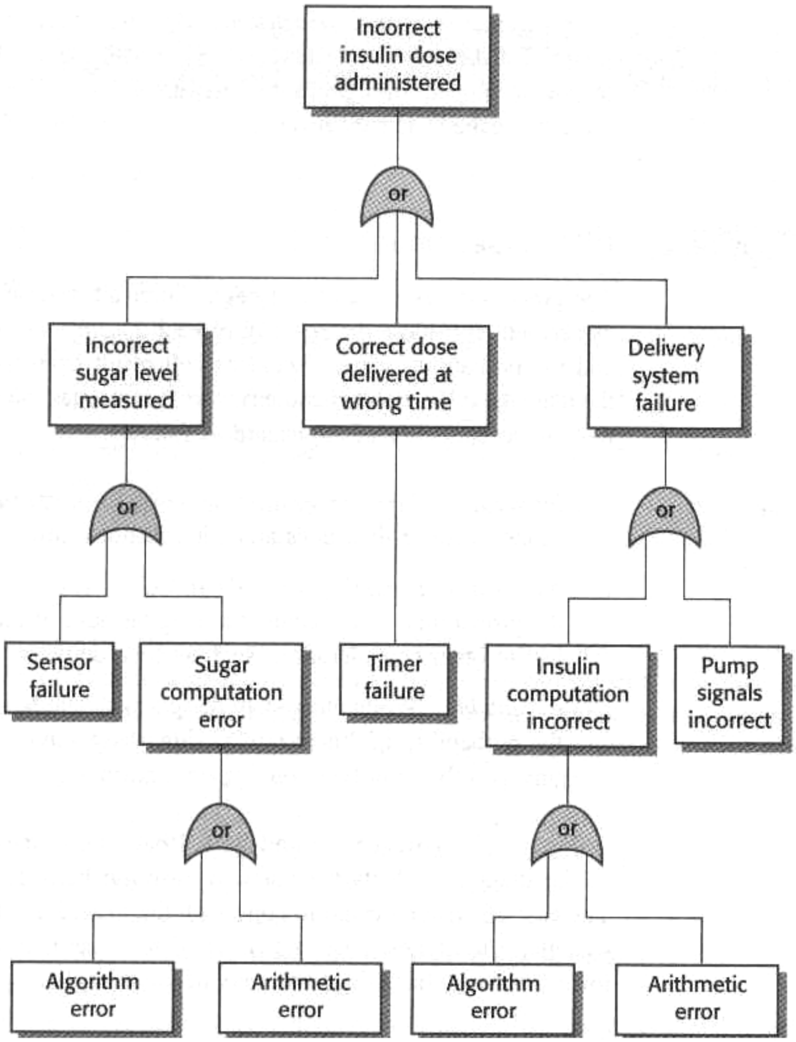
To limit the scope of such a failure, safety critical systems often employ the following techniques to improve the reliability of a system

- Fault Avoidance - here the system takes active steps to limit the possibility of mistakes or to trap mistakes if they happen. A good example of this is rigorous human input validation to make sure users cannot enter data or perform operation that might lead to an unsafe system, from example the Airbus A310 can override the pilots input on the ‘stick’ if it would take the aircraft outside it safe operating envelope.
- Fault Detection – here the system is able to detect a fault and recover from it, or fail-safe (the exact meaning of which depends upon the nature of the system, for instance shutting down is not an option for an aircraft, but might be appropriate for a nuclear reactor.). Hardware devices such a watch dog timers can restart a system if it does not carry out tasks periodically, or extra code can be incorporated into software to ensure that checks are continuously made while the system is running. A simple example might be checking that the system has not accessed an illegal array element or entered an unsafe state.
- Fault Tolerance – here the system introduces both hardware and software redundancy into the solution, i.e. replicated systems often developed by independent teams coupled with a majority voting system, so that in the event of a failure or disagreement in one part of the system, the other replicated services can continue. The space shuttle is a good example of a system employing redundant systems.

## Hazard Analysis Techniques – Fault Tree Analysis

Before the dependability and safety of a system can be determined, the development team have to carry out a detailed hazard analysis of the system they are designing and the effects of failure of that system on the surrounding environment.

One popular technique is to consider all possible hazards and then work backward to consider what faults could give rise to that hazard. Fault Tree analysis is a graphical method of representing this process. An example is shown below for an insulin administering machine



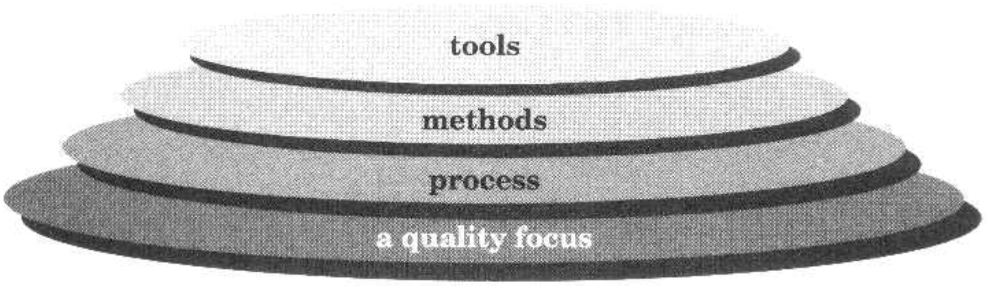
From this, the risk of each individual error/failure can be considered thus giving an estimation of the probability of the hazard occurring.

## What is Software Engineering ?

Software Engineering has been described previously as

> *"...The establishment and use of sound engineering principles in order to obtain economically, software that is reliable, maintainable and works efficiently on real machines"*

but it is important to realise that Software Engineering is also a *layered* technology as shown below, i.e. it is more than just one simple activity. Software Engineering forms the 'glue' that holds the technology layers of 'quality' 'process', 'method' and 'tools' together and enables timely development of computer software.



**What is meant by a Software Engineering Process?**

During the formation/creation of any engineering product, such as the design of a car, building or bridge, there often exists a plan or road map comprising a set of predictable, tried and tested steps that will guide you to making the finished article.

A software engineering process is just like any other engineering process. It is a framework for the tasks that are required to build deliver and maintain large scale high quality software. A software process defines the *approach* that is taken as software is engineered, it is not about deciding whether to use C++ or Java, it is about managing team development, project planning (costs, delivery dates etc), quality assurance, tracking changing requirements, software releases, version control etc. In fact the IEEE have established a document IEEE-1074 which describes the phases and processes required to engineer software, take a look at Elec 310 home page for the simple overview

**Who is involved in the Process?**

Software engineers who have to design it, managers who have to manage it and, often overlooked, the customers who have requested the software and thus need to see it delivered on time, on budget and with the minimum of defects.

**What are the steps involved in the process?**

The process itself very much depends upon the type of software that you are building, in much the same way that designing and manufacturing cars is different from that of aircraft or bridges so the process of engineering information systems (such as databases) is different to that of engineering real-time, safety critical systems for the aircraft industry.

In other words, there is not one simple process that everybody can follow, rather a number of processes exist that are tried and tested for a particular type of software development. Your job is to adopt one for the field of software engineering that you find yourself in.

**What are the results of this process?**

From the point of view of the software engineer, the results of the process are the programs, documents, data, test cases that form the finished product. From the point of view of the customer, it is the finished program, user manual, documentation, certification (if it is a safety critical system), training and maintenance/support contract.

## What is meant by Software Engineering Methods?

Methods provide the technical *'how to's* of Software Engineering including tasks such as

- Requirements analysis – i.e. understanding what the system should do.
- Design – how it should do it, e.g. architecture, algorithms, data structures, human-computer interfaces
- Coding – translation of design into program code.
- Testing – making sure it works correctly and in accordance with requirements.

## What is meant by Software Engineering Tools?

Software Engineering Tools provide automated or semi-automated support for the process and methods layers in software engineering. Some examples of tools that are frequently used in the methods phase might include:

- Modelling tools to capture the requirements of the system. For example a CASE tool that allows a developer to graphically capture and model

  - ➢ Information flow and actions within the system.
  - ➢ Typical interactions between a human and a computer system.
  - ➢ Relationships between entries in a database.

  Examples might include

  - ➢ The Rational Rose UML modelling system, or, if not working in an object based world,
  - ➢ The 'Select Yourdon' structured analysis and design tool.
  - ➢ Microsoft Access can be thought of as modelling tools targeted at database design.

- Code generation tools to facilitate the simple, fast translation of models into high level code such as C++ or Java Classes.
- A compiler to translate High Level Code into machine dependent code to run under a particular operating system
- An automated test code generator to exercise the finished product and verify it for correct operation.
- A software analyser that can analyse your code and generate 'metrics' for it, i.e. numerical values that can be used to make value judgements about such things as the quality or complexity of your design or code.

**How is Software Engineered?**

All Software Engineering development regardless of its nature or complexity follows 3 distinct phases from which we can apply a chosen (hopefully appropriate) process, and a set of methods and tools. These phases are defined as:

- The Definition Phase.
- The Development Phase.
- The Support Phase.

**The Definition Phase**

This phase focuses on '*what'* the system is supposed to do not '*how'* it will do it. For example, it is concerned with issues such as uncovering

- What information might be processed, such as names and addresses in a business or database system, or perhaps altitude, speed, pitch, rudder and direction information in a fly-by-wire aircraft.
- What functionality is required, i.e. what does the system do with the information it has been given (an aircraft control system may adjust the flight surfaces of the aircraft to keep it on target)
- What performance is required, a database system is probably not a real-time system and thus the number of transactions per day may not be the most important aspect of the system, whereas an aircraft may need to react within mSec to inputs from the pilot.
- How is a human or other computer expected to interact with the system.

**The Development Phase**

This phase focuses on 'how' the system is to be realised. And concerns itself with issues such as

- Data and its organisation (e.g. lists, trees, databases etc)
- Algorithms, i.e. how is some procedure implemented or carried out.
- The design of any Human Computer Interfaces. (E.g. windows dialog boxes, forms, screens, mouse, keyboard etc.)
- How the design is mapped to a program structure (objects, classes, relationships and interactions etc.)
- How testing is performed.

Essentially we are concerned here with the

- Architecture of the system (i.e. what building blocks will be required such as objects, functions and procedure and how will they be organised).
- Code Generation (translation of architecture into sub-systems, classes, functions and data types)
- Software Testing: Verification and Validation of the design

## The Support or Maintenance Phase

This phase focuses on change, i.e. modifications to the software that arise after its release to the customer. Four types of change are likely for large systems that are in use for long periods of time

- **Corrective Change:** Initiated as a result of bugs that are uncovered by the customer using the system. A large proportion of these are usually uncovered fairly early after release and quickly settle down to a steady trickle but you never quite get rid of them all (a sobering thought!!). As a developer you may have to absorb the cost of these changes yourself.

- **Adaptive Change:** Initiated as a result of the systems external environment changing, for example

  i. the host operating system, or CPU architecture that the software relies upon might no longer be support forcing a migration to another platform (e.g. windows '98 to windows 2000/XP)
  ii. The business rules of the company that the software was designed to implement might change leading to changes in the business logic of the code. A code example of this is the tax calculation system used by Revenue Canada, changes in laws made by politicians might need to be reflected in the code.

  Costs for this type of maintenance can be negotiated between customer and developer. It might depend upon the wording of the maintenance contract.

- **Enhancement Changes**: As software is used, customers and users often realise that small modifications or enhancements to the system could provide significant additional benefits and thus a number of requests may be initiated requiring modification to the systems. The costs for these are always absorbed by the customer.

  **Perfective change** is an enhancement to the system that evolves it way beyond its original functional requirements.

- **Preventative Change**. We know that software deteriorates with any changes made to it because such changes often introduce new bugs. Preventative change attempts to modify, re-organise or re-structure the system so that any changes made to it in the future will have less of an impact. (This is somewhat analogous to changing the oil in a car, i.e. a small bit of maintenance now and again can lead to less maintenance and costly repairs in the future)

## Measuring Process Quality - The SEI Software Capability Maturity Model (CMM)

It has long been know that one of the key factors that influence the quality of a product is the quality of the *process* used in developing it. Now one of the biggest procurers of software based systems is in fact the Department of Defense (DoD) in the U.S. whose budget runs into billions of dollars and there is naturally a long list of contractors lining up to get a piece or that budget.

The problem faced by the DoD was how to assess these contractors and thus determine the quality of their processes and by implication the quality of their software product. To this end, the DoD funds *The Software Engineering Institute (SEI)* whose stated mission is software technology transfer.

The SEI was thus established to measure and by association improve the software capabilities (i.e. how good are they) of companies interested in bidding for DoD contracts.

The outcome of the work done by SEI is the *Software Capability Maturity Model*. In essence it is a grading scheme used by the DoD to rate the capabilities of their contractors and how mature their software engineering processes are. The model ranks software organisations on a scale of 1-5, defined as shown below, but there is no requirement for a contractor to have actually reached a certain level before it can bid for or be given a contract from the DoD, however there is an implicit assumption that those contractors and organisations with higher ratings do have an advantage when bidding. Note that in order to acquire a higher CMM level organisations have to have in place all requirements for all of the lower levels

**Level 1:** The Lowest or Initial Level. Here the software process is characterised as 'ad-hoc' and occasionally chaotic. The organisation does not have effective management procedures or project plans (delivery schedules, costs etc). If formal procedures for project control exist, they are not documented or used consistently. The organisation may well be capable of engineering software but the quality, delivery schedule and cost will be unpredictable. In other words success very much depends upon individual effort. (this is where you are at now, prior to taking this course)

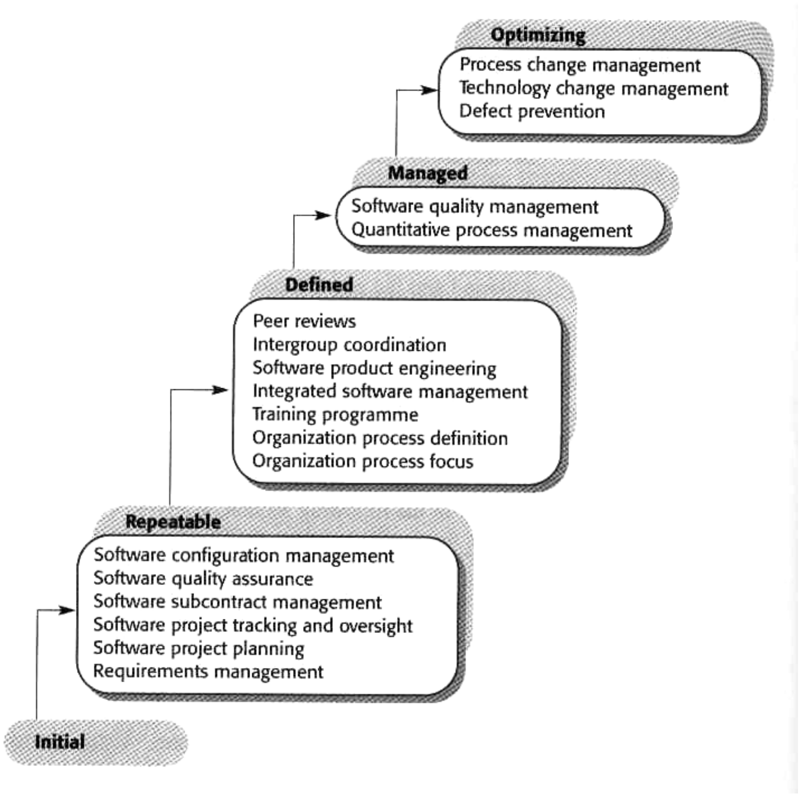
**Level 2:** Repeatable - There is evidence of some basic level of organisational and project management skill within the company to enable it to track costs, software functionality and delivery schedules. More importantly perhaps, there is evidence that a discipline exists within the culture of the company and its developers that would allow for the similar projects to be repeated with similar levels of success, however that success may well depend upon individual managers motivating a team.

**Level 3:** Defined – The software process for both Management and Engineering activities is defined and documented. It is thus in a position to improve itself based on feedback from previous projects.

**Level 4:** Managed – A level 4 organisation has a defined software development process and a formal programme of quantitative data collection. In other words it gathers metrics about its own processes and thus measures the success of that process when applied to software development projects. These measures are then used to improve and refine its own processes.

**Level 5:** Optimising – At this level an organisation is committed to continuous process improvement. Such improvement is budgeted for and is planned. In other words it is an integral part of the organisations success.
After the initial version of the CMM was released, it came in for some criticism for being too imprecise in terms of what organisations were required to do to meet to standards of the various levels.

To remedy this, a second model was released where the original 5 levels were retained but each level was refined in terms of Key Process Areas (KPA's). Improvements could then be made by improving these KPA's rather than just reaching some arbitrary level. This revised model is shown below.


Optimizing
Process change management
Technology change management
Defect prevention
Managed
Software quality management
Quantitative process management
Defined
Peer reviews
Intergroup coordination
Software product engineering
Integrated software management
Training programme
Organization process definition
Organization process focus
Repeatable
Software configuration management
Software quality assurance
Software subcontract management
Software project tracking and oversight
Software project planning
Requirements management
Initial

Even now, the model is still criticised in three important areas:

- The model focuses purely on project management issues rather than product development, i.e. it only measure the ability of the company to *apply a process* and not how effectively it makes use of tools and methods (all part of the development process) to arrive at its goal.
- It excludes risk analysis from the ratings. That is, whether a company does or does not perform risk analysis during the definition and design phases is not deemed important by the model. This should not be the case, as risk analysis has been shown to be very effective in uncovering serious problems before they can impinge on the outcome and success of the process.
- The authors of the model do not state what organisations and scale of project are applicable for the model. Consequently the model has been oversold and used where perhaps it is not applicable, e.g small scale software developments.