NUTAN MAHARASHTRA VIDYA PRASARAK MANDAL'S
NUTAN COLLEGE OF ENGINEERING & RESEARCH (NCER)
Department of Computer Science & Engineering

BTCOC701

Software Engineering

Teaching Notes

Lecture Number	Topic to be covered (Unit 1)
	Introduction (6 Hrs)
1	**Introduction** Professional software development
2	Software engineering ethics
3	Case studies
4	**Software processes:** Software process models,
5	Process activities, Coping with change,
6	The rational unified process.

: Submitted by:

Prof. P. B. Dhore

Nutan College Of Engineering & Research,
Talegaon Dabhade, Pune- 410507

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Software Engineering

Unit 1:- Introduction

What is Software Engineering?

- The term software engineering is the product of two words, software, and engineering.
- The software is a collection of integrated programs.
- Software subsists of carefully-organized instructions and code written by developers on any of various particular computer languages.
- Computer programs and related documentation such as requirements, design models and user manuals.
- Engineering is the application of scientific and practical knowledge to invent, design, build, maintain, and improve frameworks, processes, etc

Software Engineering is an engineering branch related to the evolution of software product using well-defined scientific principles, techniques, and procedures. The result of software engineering is an effective and reliable software product.

Why is Software Engineering required?

Software Engineering is required due to the following reasons:

- To manage Large software
- For more Scalability
- Cost Management
- To manage the dynamic nature of software
- For better quality Management

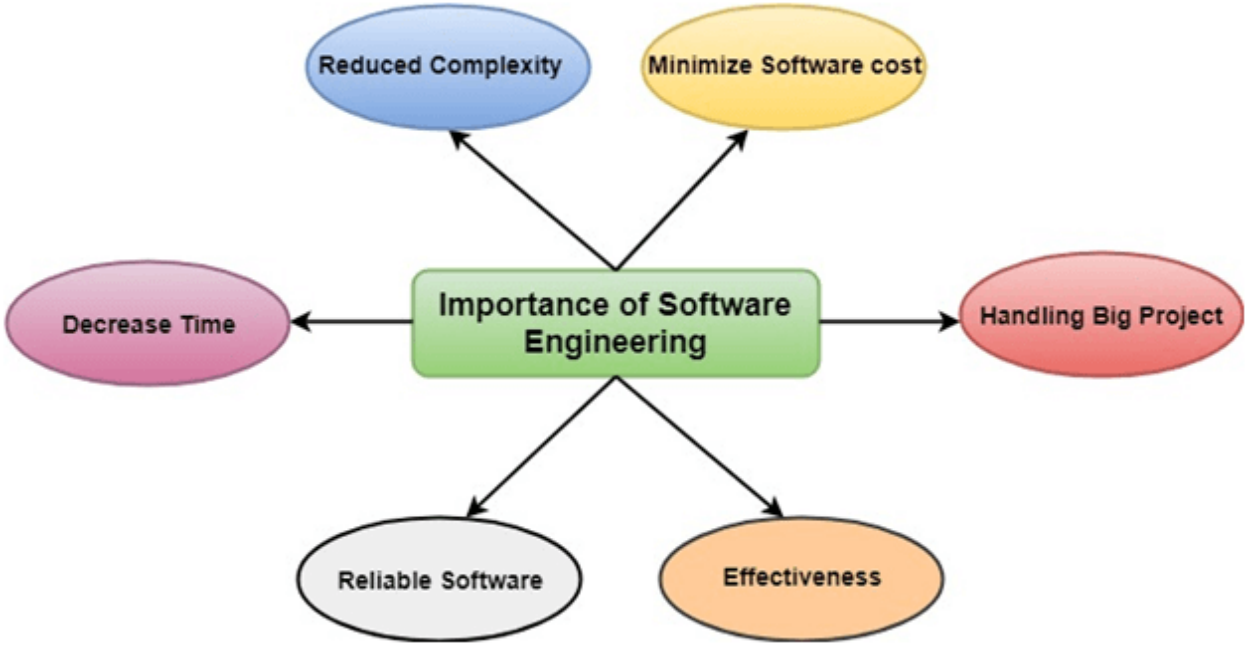
Need of Software Engineering

The necessity of software engineering appears because of a higher rate of progress in user requirements and the environment on which the program is working.

- **Huge Programming:** It is simpler to manufacture a wall than to a house or building, similarly, as the measure of programming become extensive engineering has to step to give it a scientific process.
- **Adaptability:** If the software procedure were not based on scientific and engineering ideas, it would be simpler to re-create new software than to scale an existing one.
- **Cost:** As the hardware industry has demonstrated its skills and huge manufacturing has let down the cost of computer and electronic hardware. But the cost of programming remains high if the proper process is not adapted.
- **Dynamic Nature:** The continually growing and adapting nature of programming hugely depends upon the environment in which the client works. If the quality of the software is continually changing, new upgrades need to be done in the existing one.
- **Quality Management:** Better procedure of software development provides a better and quality software product.

Importance of Software Engineering

The importance of Software engineering is as follows:



Reduces complexity: Big software is always complicated and challenging to progress. Software engineering has a great solution to reduce the complication of any project. Software engineering divides big problems into various small issues. And then start solving each small issue one by one. All these small problems are solved independently to each other.

To minimize software cost: Software needs a lot of hardwork and software engineers are highly paid experts. A lot of manpower is required to develop software with a large number of codes. But in software engineering, programmers project everything and decrease all those things that are not needed. In turn, the cost for software productions becomes less as compared to any software that does not use software engineering method.

To decrease time: Anything that is not made according to the project always wastes time. And if you are making great software, then you may need to run many codes to get the definitive running code. This is a very time-consuming procedure, and if it is not

well handled, then this can take a lot of time. So if you are making your software according to the software engineering method, then it will decrease a lot of time.

Handling big projects: Big projects are not done in a couple of days, and they need lots of patience, planning, and management. And to invest six and seven months of any company, it requires heaps of planning, direction, testing, and maintenance. No one can say that he has given four months of a company to the task, and the project is still in its first stage. Because the company has provided many resources to the plan and it should be completed. So to handle a big project without any problem, the company has to go for a software engineering method.

Reliable software: Software should be secure, means if you have delivered the software, then it should work for at least its given time or subscription. And if any bugs come in the software, the company is responsible for solving all these bugs. Because in software engineering, testing and maintenance are given, so there is no worry of its reliability.

Effectiveness: Effectiveness comes if anything has made according to the standards. Software standards are the big target of companies to make it more effective. So Software becomes more effective in the act with the help of software engineering.

Software Engineering Tutorial Index

Professional software development

Software development is the process of conceiving, specifying, designing, programming, documenting, testing, and bug fixing involved in creating and maintaining applications, frameworks, or other software components. Software development is a process of writing and maintaining the source code, but in a broader sense, it includes all that is involved between the conception of the desired software through to the final manifestation of the software, sometimes in a planned and structured process.

Therefore, software development may include research, new development, prototyping, modification, reuse, re-engineering, maintenance, or any other activities that result in software products.

The software can be developed for a variety of purposes, the three most common being to meet specific needs of a specific client/business (the case with custom software), to meet a perceived need of some set of potential users (the case with commercial and open source software), or for personal use (e.g. a scientist may write software to

automate a mundane task). Embedded software development, that is, the development of embedded software, such as used for controlling consumer products, requires the development process to be integrated with the development of the controlled physical product. System software underlies applications and the programming process itself, and is often developed separately.

The need for better quality control of the software development process has given rise to the discipline of software engineering, which aims to apply the systematic approach exemplified in the engineering paradigm to the process of software development.

There are many approaches to software project management, known as software development life cycle models, methodologies, processes, or models. The waterfall model is a traditional version, contrasted with the more recent innovation of agile software development.

Software engineers are concerned with developing software products (i.e., soft- ware which can be sold to a customer). There are two kinds of software products:

1. Generic products These are stand-alone systems that are produced by a develop- ment organization and sold on the open market to any customer who is able to buy them. Examples of this type of product include software for PCs such as databases, word processors, drawing packages, and project-management tools. It also includes so-called vertical applications designed for some specific pur- pose such as library information systems, accounting systems, or systems for maintaining dental records.

2. Customized (or bespoke) products These are systems that are commissioned by a particular customer. A software contractor develops the software especially for that customer. Examples of this type of software include control systems for electronic devices, systems written to support a particular business process, and air traffic control systems.

An important difference between these types of software is that, in generic products, the organization that develops the software controls the software specification. For cus- tom products, the specification is usually developed and controlled by the organization that is buying the software. The software developers must work to that specification.
However, the distinction between these system product types is becoming increasingly blurred. More and more systems are now being built with a generic product as a base, which is then adapted to suit the requirements of a customer. Enterprise Resource Planning (ERP) systems, such as the SAP system, are the best examples of this approach. Here, a large and complex system is adapted for a com- pany by incorporating information about business rules and processes, reports required, and so on.

Software engineering

Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use. In this definition, there are two key phrases:

Product characteristics Description

Maintainability:- Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.

Dependability and security :- Software dependability includes a range of characteristics including reliability, security, and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.

Efficiency:- Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilization, etc.

Acceptability:- Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable, and compatible with other systems that they use and always try to discover solutions to problems even when there are no appli- cable theories and methods. Engineers also recognize that they must work to organizational and financial constraints so they look for solutions within these constraints.
1.Engineering discipline Engineers make things work. They apply theories, meth- ods, and tools where these are appropriate. However, they use them selectively

2. All aspects of software production Software engineering is not just concerned with the technical processes of software development. It also includes activities such as software project management and the development of tools, methods, and theories to support software production.

Software engineering is important for two reasons:

1. More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.
2. It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems rather than just write the programs as if it was a personal programming project. For most types of systems, the majority of costs are the costs of changing the software after it has gone into use.

The systematic approach that is used in software engineering is sometimes called a software process. A software process is a sequence of activities that leads to the production of a software product. There are four fundamental activities that are common to all software processes. These activities are:

1. Software specification, where customers and engineers define the software that is to be produced and the constraints on its operation.
2. Software development, where the software is designed and programmed.
3. Software validation, where the software is checked to ensure that it is what the customer requires.
4. Software evolution, where the software is modified to reflect changing customer and market requirements.

Software engineering is related to both computer science and systems engineering:

1. Computer science is concerned with the theories and methods that underlie com- puters and software systems, whereas software engineering is concerned with the practical problems of producing software. Some knowledge of computer science is essential for software engineers in the same way that some knowledge of physics is essential for electrical engineers. Computer science theory, however, is often most applicable to relatively small programs. Elegant theories of computer science cannot always be applied to large, complex problems that require a soft- ware solution.

2. System engineering is concerned with all aspects of the development and evo- lution of complex systems where software plays a major role. System engineer- ing is therefore concerned with hardware development, policy and process design and system deployment, as well as software engineering. System engi- neers are involved in specifying the system, defining its overall architecture, and then integrating the different parts to create the finished system. They are less concerned with the engineering of the system components (hardware, software, etc.).

three general issues that affect many different types of software:

1. Heterogeneity:- Increasingly, systems are required to operate as distributed systems across networks that include different types of computer and mobile devices. As well as running on general-purpose computers, software may also have to execute on mobile phones. You often have to integrate new software with older legacy sys- tems written in different programming languages. The challenge here is to develop techniques for building dependable software that is flexible enough to cope with this heterogeneity.

2. Business and social change:- Business and society are changing incredibly quickly as emerging economies develop and new technologies become available. They need to be able to change their existing software and to rapidly develop new soft- ware. Many traditional software engineering techniques are time consuming and delivery of new systems often takes longer than planned. They need to evolve so that the time required for software to deliver value to its customers is reduced

3. Security and trust:- As software is intertwined with all aspects of our lives, it is essential that we can trust that software. This is especially true for remote soft- ware systems accessed through a web page or web service interface. We have to make sure that malicious users cannot attack our software and that information security is maintained.

Software engineering diversity

Software engineering is a systematic approach to the production of software that takes into account practical cost, schedule, and dependability issues, as well as the needs of software customers and producers. How this systematic approach is actu- ally implemented varies dramatically depending on the organization developing the software, the type of software, and the people involved in the development process. There are no universal software engineering methods and techniques that are suit- able for all systems and all companies.

There are many different types of application including:

1. Stand-alone applications:- These are application systems that run on a local com- puter, such as a PC. They include all necessary functionality and do not need to be connected to a network. Examples of such applications are office applica- tions on a PC, CAD programs, photo manipulation software, etc.
2. Interactive transaction-based applications:- These are applications that execute on a remote computer and that are accessed by users from their own PCs or terminals. Obviously, these include web applications such as e-commerce appli- cations where you can interact with a remote system to buy goods and services. This class of application also includes business systems, where a business provides access to its systems through a web browser or special-purpose client program and cloud-based services, such as mail and photo sharing. Interactive applications often incorporate a large data store that is accessed and updated in each transaction.
3. Embedded control systems:- These are software control systems that control and manage hardware devices. Numerically, there are probably more embedded sys- tems than any other type of system. Examples of embedded systems include the software in a mobile (cell) phone, software that controls anti-lock braking in a car, and software in a microwave oven to control the cooking process.
4. Batch processing systems:- These are business systems that are designed to process data in large batches. They process large numbers of individual inputs to create

corresponding outputs. Examples of batch systems include periodic billing systems, such as phone billing systems, and salary payment systems.

5. Entertainment systems:- These are systems that are primarily for personal use and which are intended to entertain the user. Most of these systems are games of one kind or another. The quality of the user interaction offered is the most important distinguishing characteristic of entertainment systems.

6. Systems for modeling and simulation:- These are systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting objects. These are often computationally intensive and require high-performance parallel systems for execution.

7. Data collection systems:- These are systems that collect data from their environ- ment using a set of sensors and send that data to other systems for processing. The software has to interact with sensors and often is installed in a hostile envi- ronment such as inside an engine or in a remote location.

8. Systems of systems:- These are systems that are composed of a number of other software systems. Some of these may be generic software products, such as a spreadsheet program. Other systems in the assembly may be specially written for that environment.

Software engineering and the Web

The development of the World Wide Web has had a profound effect on all of our lives. Initially, the Web was primarily a universally accessible information store and it had little effect on software systems. These systems ran on local computers and were only accessible from within an organization. Around 2000, the Web started to evolve and more and more functionality was added to browsers. This meant that web-based systems could be developed where, instead of a special-purpose user interface, these systems could be accessed using a web browser. This led to the development of a vast range of new system products that delivered innovative serv- ices, accessed over the Web. These are often funded by adverts that are displayed on the user's screen and do not involve direct payment from users.

As well as these system products, the development of web browsers that could run small programs and do some local processing led to an evolution in business and organizational software. Instead of writing software and deploying it on users' PCs, the software was deployed on a web server. This made it much cheaper to change and upgrade the software, as there was no need to install the software on every PC. It also reduced costs, as user interface development is particularly expensive. Consequently, wherever it has been possible to do so, many businesses have moved to web-based interaction with company software systems.

The next stage in the development of web-based systems was the notion of web services. Web services are software components that deliver specific, useful function- ality and which are accessed over the Web. Applications are constructed by integrating these web services, which may be provided by different companies. In principle, this

linking can be dynamic so that an application may use different web services each time that it is executed.

In the last few years, the notion of ‘software as a service’ has been developed. It has been proposed that software will not normally run on local computers but will run on ‘computing clouds’ that are accessed over the Internet. If you use a service such as web-based mail, you are using a cloud-based system. A computing cloud is a huge number of linked computer systems that is shared by many users. Users do not buy software but pay according to how much the software is used or are given free access in return for watching adverts that are displayed on their screen.

The advent of the web, therefore, has led to a significant change in the way that business software is organized. Before the web, business applications were mostly monolithic, single programs running on single computers or computer clusters. Communications were local, within an organization. Now, software is highly distributed, sometimes across the world. Business applications are not programmed from scratch but involve extensive reuse of components and programs.

This radical change in software organization has, obviously, led to changes in the ways that web-based systems are engineered. For example:

1. Software reuse has become the dominant approach for constructing web-based systems. When building these systems, you think about how you can assemble them from pre-existing software components and systems.
2. It is now generally recognized that it is impractical to specify all the requirements for such systems in advance. Web-based systems should be developed and delivered incrementally.
3. User interfaces are constrained by the capabilities of web browsers. Although technologies such as AJAX (Holdener, 2008) mean that rich interfaces can be created within a web browser, these technologies are still difficult to use. Web forms with local scripting are more commonly used. Application interfaces on web-based systems are often poorer than the specially designed user interfaces on PC system products.

Software engineering ethics

software engineering is carried out within a social and legal framework that limits the freedom of people working in that area. As a software engineer, you must accept that your job involves wider responsibilities than simply the application of technical skills. You must also behave in an ethical and morally responsible way if you are to be respected as a professional engineer.

It goes without saying that you should uphold normal standards of honesty and integrity. You should not use your skills and abilities to behave in a dishonest way or in a way that will bring disrepute to the software engineering profession. However, there are

areas where standards of acceptable behavior are not bound by laws but by the more tenuous notion of professional responsibility. Some of these are:

1. Confidentiality:- You should normally respect the confidentiality of your employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.
2. Competence:- You should not misrepresent your level of competence. You should not knowingly accept work that is outside your competence.
3. Intellectual property rights:- You should be aware of local laws governing the use of intellectual property such as patents and copyright. You should be careful to ensure that the intellectual property of employers and clients is protected.
4. Computer misuse:- You should not use your technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses or other malware).

Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

1. PUBLIC — Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER — Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. PRODUCT — Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT — Software engineers shall maintain integrity and independence in their professional judgment.
5. MANAGEMENT — Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION — Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES — Software engineers shall be fair to and supportive of their colleagues.
8. SELF — Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Computers have a central and growing role in commerce, industry, government, medicine, education, entertainment and society at large. Software engineers are those who contribute by direct participation or by teaching, to the analysis, spec- ification, design, development, certification, maintenance and testing of software

systems. Because of their roles in developing software systems, software engi- neers have significant opportunities to do good or cause harm, to enable others to do good or cause harm, or to influence others to do good or cause harm. To ensure, as much as possible, that their efforts will be used for good, software engi- neers must commit themselves to making software engineering a beneficial and respected profession. In accordance with that commitment, software engineers shall adhere to the following Code of Ethics and Professional Practice.

The Code contains eight Principles related to the behaviour of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession. The Principles identify the ethically responsible relationships in which individuals, groups, and organizations participate and the primary obligations within these relationships. The Clauses of each Principle are illus- trations of some of the obligations included in these relationships. These obli- gations are founded in the software engineer's humanity, in special care owed to people affected by the work of software engineers, and the unique elements of the practice of software engineering. The Code prescribes these as obliga- tions of anyone claiming to be or aspiring to be a software engineer.

Case studies

To illustrate software engineering concepts, I use examples from three different types of systems throughout the book. The reason why I have not used a single case study is that one of the key messages in this book is that software engineering prac- tice depends on the type of systems being produced. I therefore choose an appropri- ate example when discussing concepts such as safety and dependability, system modeling, reuse, etc.
The three types of systems that I use as case studies are:

1. An embedded system This is a system where the software controls a hardware device and is embedded in that device. Issues in embedded systems typically

include physical size, responsiveness, power management, etc. The example of an embedded system that I use is a software system to control a medical device.
2. An information system This is a system whose primary purpose is to manage and provide access to a database of information. Issues in information systems include security, usability, privacy, and maintaining data integrity. The example of an information system that I use is a medical records system.
3. A sensor-based data collection system This is a system whose primary purpose is to collect data from a set of sensors and process that data in some way. The key requirements of such systems are reliability, even in hostile environmental conditions, and maintainability. The example of a data collection system that I use is a wilderness weather station.

I introduce each of these systems in this chapter, with more information about each of them available on the Web.

An insulin pump control system

An insulin pump is a medical system that simulates the operation of the pancreas (an internal organ). The software controlling this system is an embedded system, which collects information from a sensor and controls a pump that delivers a controlled dose of insulin to a user.

People who suffer from diabetes use the system. Diabetes is a relatively common condition where the human pancreas is unable to produce sufficient quantities of a hormone called insulin. Insulin metabolises glucose (sugar) in the blood. The conventional treatment of diabetes involves regular injections of genetically engineered insulin. Diabetics measure their blood sugar levels using an external meter and then calculate the dose of insulin that they should inject.

The problem with this treatment is that the level of insulin required does not just depend on the blood glucose level but also on the time of the last insulin injection. This can lead to very low levels of blood glucose (if there is too much insulin) or very high levels of blood sugar (if there is too little insulin). Low blood glucose is, in the short term, a more serious condition as it can result in temporary brain malfunctioning and, ultimately, unconsciousness and death. In the long term, however, continual high levels of blood glucose can lead to eye damage, kidney damage, and heart problems. Current advances in developing miniaturized sensors have meant that it is now pos- sible to develop automated insulin delivery systems. These systems monitor blood sugar levels and deliver an appropriate dose of insulin when required. Insulin delivery systems like this already exist for the treatment of hospital patients. In the future, it may be possible for many diabetics to have such systems permanently attached to their bodies.

A software-controlled insulin delivery system might work by using a micro- sensor embedded in the patient to measure some blood parameter that is proportional to the sugar level. This is then sent to the pump controller. This controller computes the sugar level and the amount of insulin that is needed. It then sends signals to a miniaturized pump to deliver the insulin via a permanently attached needle.

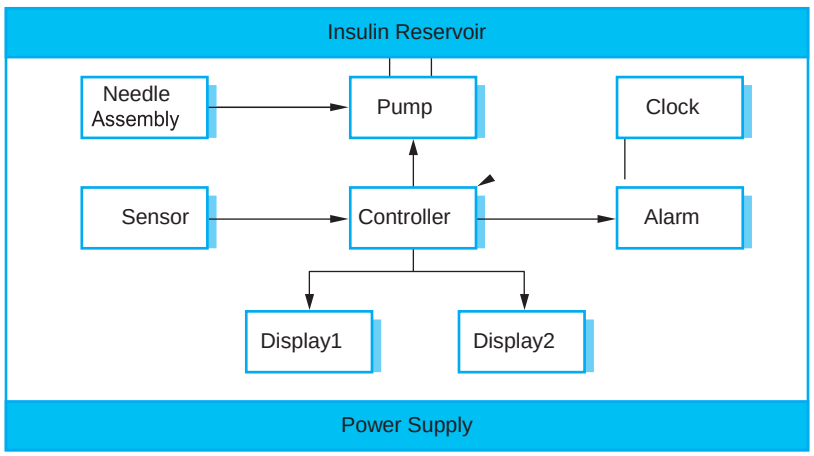


Figure1 :- Insulin pump hardware

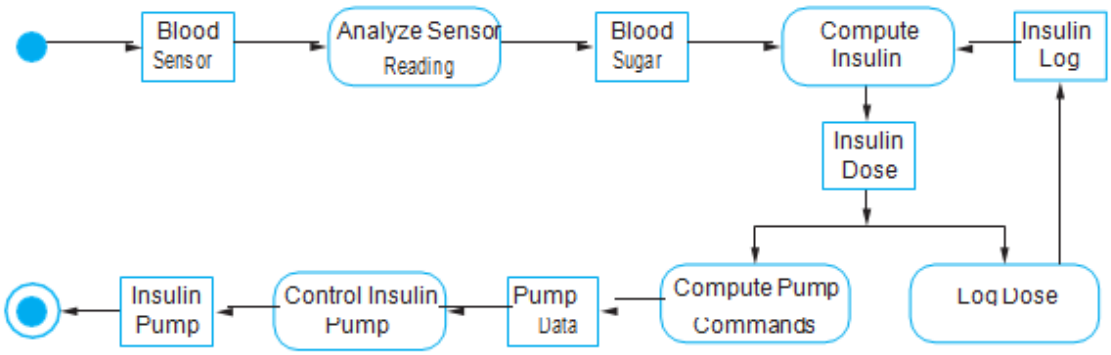


Figure 2:- Activity model of the insulin pump

Figure-1 shows the hardware components and organization of the insulin pump. To understand the examples in this book, all you need to know is that the blood sensor measures the electrical conductivity of the blood under different conditions and that these values can be related to the blood sugar level. The insulin pump delivers one unit of insulin in response to a single pulse from a con- troller. Therefore, to deliver 10 units of insulin, the controller sends 10 pulses to the pump. Figure-2 is a UML activity model that illustrates how the software transforms an input blood sugar level to a sequence of commands that drive the insulin pump.

Clearly, this is a safety-critical system. If the pump fails to operate or does not operate correctly, then the user's health may be damaged or they may fall into a coma because their blood sugar levels are too high or too low. There are, therefore, two essential high-level requirements that this system must meet:

1. The system shall be available to deliver insulin when required.
2. The system shall perform reliably and deliver the correct amount of insulin to counteract the current level of blood sugar.

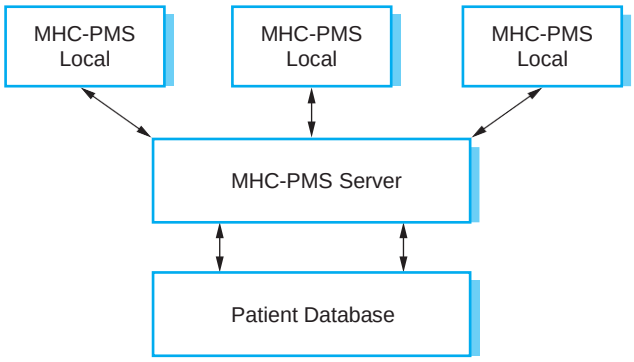


Figure -3:- The organization of the MHC-PMS

The system must therefore be designed and implemented to ensure that the sys- tem always meets these requirements. More detailed requirements and discussions of how to ensure that the system is safe are discussed in later chapters.

A patient information system for mental health care

A patient information system to support mental health care is a medical informa- tion system that maintains information about patients suffering from mental health problems and the treatments that they have received. Most mental health patients do not require dedicated hospital treatment but need to attend specialist clinics regularly where they can meet a doctor who has detailed knowledge of their problems. To make it easier for patients to attend, these clinics are not just run in hospitals. They may also be held in local medical practices or community centers.

The MHC-PMS (Mental Health Care-Patient Management System) is an informa- tion system that is intended for use in clinics. It makes use of a centralized database of patient information but has also been designed to run on a PC, so that it may be accessed and used from sites that do not have secure network connectivity. When the local sys- tems have secure network access, they use patient information in the database but they can download and use local copies of patient records when they are disconnected. The system is not a complete medical records system so does not maintain information about other medical conditions. However, it may interact and exchange data with other clinical information systems. Figure-3 illustrates the organization of the MHC-PMS.

The MHC-PMS has two overall goals:

1. To generate management information that allows health service managers to assess performance against local and government targets.
2. To provide medical staff with timely information to support the treatment of patients.

Users of the system include clinical staff such as doctors, nurses, and health visi- tors (nurses who visit people at home to check on their treatment). Nonmedical users include receptionists who make appointments, medical records staff who maintain the records system, and administrative staff who generate reports.

The system is used to record information about patients (name, address, age, next of kin, etc.), consultations (date, doctor seen, subjective impressions of the patient, etc.),

conditions, and treatments. Reports are generated at regular intervals for med- ical staff and health authority managers. Typically, reports for medical staff focus on information about individual patients whereas management reports are anonymized and are concerned with conditions, costs of treatment, etc.
The key features of the system are:

1. Individual care management Clinicians can create records for patients, edit the information in the system, view patient history, etc. The system supports data summaries so that doctors who have not previously met a patient can quickly learn about the key problems and treatments that have been prescribed.
2. Patient monitoring The system regularly monitors the records of patients that are involved in treatment and issues warnings if possible problems are detected. Therefore, if a patient has not seen a doctor for some time, a warning may be issued. One of the most important elements of the monitoring system is to keep track of patients who have been sectioned and to ensure that the legally required checks are carried out at the right time.
3. Administrative reporting The system generates monthly management reports showing the number of patients treated at each clinic, the number of patients who have entered and left the care system, number of patients sectioned, the drugs prescribed and their costs, etc.

Two different laws affect the system. These are laws on data protection that govern the confidentiality of personal information and mental health laws that govern the com- pulsory detention of patients deemed to be a danger to themselves or others. Mental health is unique in this respect as it is the only medical speciality that can recommend the detention of patients against their will. This is subject to very strict legislative safe- guards. One of the aims of the MHC-PMS is to ensure that staff always act in accor- dance with the law and that their decisions are recorded for judicial review if necessary. As in all medical systems, privacy is a critical system requirement. It is essential that patient information is confidential and is never disclosed to anyone apart from author- ized medical staff and the patient themselves.

Software processes

Objectives
The objective of this chapter is to introduce you to the idea of a software process—a coherent set of activities for software production. When you have read this chapter you will:

- understand the concepts of software processes and software process models;
- have been introduced to three generic software process models and when they might be used;
- know about the fundamental process activities of software requirements engineering, software development, testing, and evolution;

- understand why processes should be organized to cope with changes in the software requirements and design;
- understand how the Rational Unified Process integrates good software engineering practice to create adaptable software processes.

A software process is a set of related activities that leads to the production of a software product. These activities may involve the development of software from scratch in a standard programming language like Java or C. However, business applications are not necessarily developed in this way. New business software is now often devel- oped by extending and modifying existing systems or by configuring and integrating off-the-shelf software or system components.
There are many different software processes but all must include four activities that are fundamental to software engineering:

1. Software specification The functionality of the software and constraints on its operation must be defined.
2. Software design and implementation The software to meet the specification must be produced.
3. Software validation The software must be validated to ensure that it does what the customer wants.
4. Software evolution The software must evolve to meet changing customer needs.

In some form, these activities are part of all software processes. In practice, of course, they are complex activities in themselves and include sub-activities such as requirements validation, architectural design, unit testing, etc. There are also supporting process activities such as documentation and software configuration management. When we describe and discuss processes, we usually talk about the activities in these processes such as specifying a data model, designing a user interface, etc., and the ordering of these activities. However, as well as activities, process descriptions
may also include:

1. Products, which are the outcomes of a process activity. For example, the outcome of the activity of architectural design may be a model of the software architecture.
2. Roles, which reflect the responsibilities of the people involved in the process. Examples of roles are project manager, configuration manager, programmer, etc.
3. Pre- and post-conditions, which are statements that are true before and after a process activity has been enacted or a product produced. For example, before architectural design begins, a pre-condition may be that all requirements have been approved by the customer; after this activity is finished, a post-condition might be that the UML models describing the architecture have been reviewed.

Software processes are complex and, like all intellectual and creative processes, rely on people making decisions and judgments. There is no ideal process and most organizations have developed their own software development processes. Processes

have evolved to take advantage of the capabilities of the people in an organization and the specific characteristics of the systems that are being developed. For some

systems, such as critical systems, a very structured development process is required. For business systems, with rapidly changing requirements, a less formal, flexible process is likely to be more effective.
Sometimes, software processes are categorized as either plan-driven or agile processes. Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan. In agile processes, which I discuss in Chapter 3, planning is incremental and it is easier to change the process to reflect changing customer requirements. As Boehm and Turner (2003) discuss, each approach is suitable for different types of software. Generally, you need to find a balance between plan-driven and agile processes.
Although there is no 'ideal' software process, there is scope for improving the software process in many organizations. Processes may include outdated techniques or may not take advantage of the best practice in industrial software engineering. Indeed, many organizations still do not take advantage of software engineering methods in their software development.
Software processes can be improved by process standardization where the diver- sity in software processes across an organization is reduced. This leads to improved communication and a reduction in training time, and makes automated process sup- port more economical. Standardization is also an important first step in introducing new software engineering methods and techniques and good software engineering practice.

Software process models

software process model is a simplified representation of a software process. Each process model represents a process from a particular per- spective, and thus provides only partial information about that process. For example, a process activity model shows the activities and their sequence but may not show the roles of the people involved in these activities. In this section, I introduce a num- ber of very general process models (sometimes called 'process paradigms') and present these from an architectural perspective. That is, we see the framework of the process but not the details of specific activities.

These generic models are not definitive descriptions of software processes. Rather, they are abstractions of the process that can be used to explain different approaches to software development. You can think of them as process frameworks that may be extended and adapted to create more specific software engineering processes.
The process models that I cover here are:

1. The waterfall model This takes the fundamental process activities of specifica- tion, development, validation, and evolution and represents them as separate process

phases such as requirements specification, software design, implemen- tation, testing, and so on.

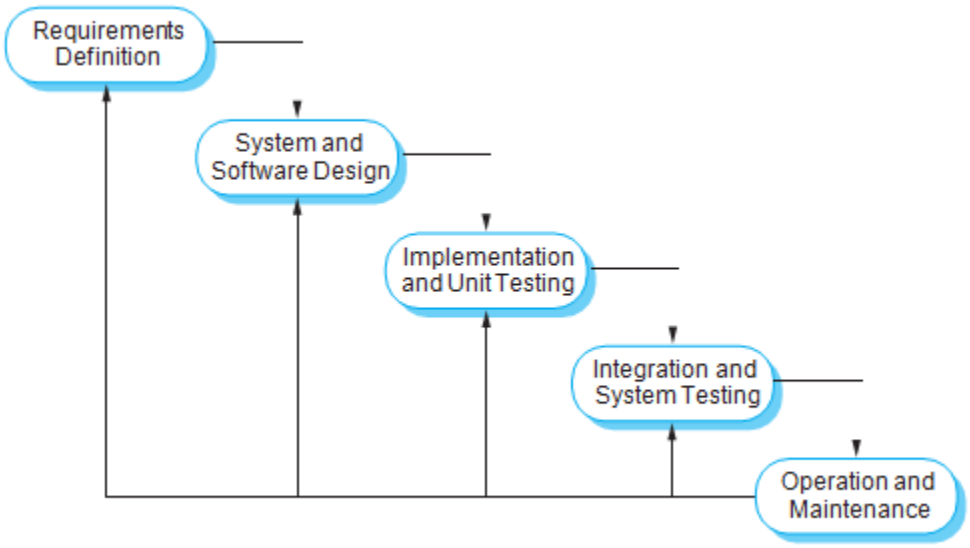


Figure:- The waterfall model

Incremental development This approach interleaves the activities of specifica- tion, development, and validation. The system is developed as a series of versions (increments), with each version adding functionality to the previous version.

Reuse-oriented software engineering This approach is based on the existence of a significant number of reusable components. The system development process focuses on integrating these components into a system rather than developing them from scratch.

These models are not mutually exclusive and are often used together, especially for large systems development. For large systems, it makes sense to combine some of the best features of the waterfall and the incremental development models. You need to have information about the essential system requirements to design a soft- ware architecture to support these requirements. You cannot develop this incremen- tally. Sub-systems within a larger system may be developed using different approaches. Parts of the system that are well understood can be specified and devel- oped using a waterfall-based process. Parts of the system which are difficult to specify in advance, such as the user interface, should always be developed using an incremental approach.

The waterfall model

The first published model of the software development process was derived from more general system engineering processes (Royce, 1970). This model is illustrated in above Figure. Because of the cascade from one phase to another, this model is known as the 'waterfall model' or software life cycle. The waterfall model is an example of a plan-driven process—in principle, you must plan and schedule all of the process activities before starting work on them.

The principal stages of the waterfall model directly reflect the fundamental development activities:

1. Requirements analysis and definition The system's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.
2. System and software design The systems design process allocates the requirements to either hardware or software systems by establishing an overall system architecture. Software design involves identifying and describing the fundamen- tal software system abstractions and their relationships.
3. Implementation and unit testing During this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.
4. Integration and system testing The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.
5. Operation and maintenance Normally (although not necessarily), this is the longest life cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system's services as new requirements are discovered.

In principle, the result of each phase is one or more documents that are approved ('signed off'). The following phase should not start until the previous phase has finished. In practice, these stages overlap and feed information to each other. During design, problems with requirements are identified. During coding, design problems are found and so on. The software process is not a simple linear model but involves feedback from one phase to another. Documents produced in each phase may then have to be modified to reflect the changes made.

Because of the costs of producing and approving documents, iterations can be costly and involve significant rework. Therefore, after a small number of iterations, it is normal to freeze parts of the development, such as the specification, and to con- tinue with the later development stages. Problems are left for later resolution, ignored, or programmed around. This premature freezing of requirements may mean that the system won't do what the user wants. It may also lead to badly structured systems as design problems are circumvented by implementation tricks.

During the final life cycle phase (operation and maintenance) the software is put into use. Errors and omissions in the original software requirements are discovered. Program and design errors emerge and the need for new functionality is identified. The system must therefore evolve to remain useful. Making these changes (software maintenance) may involve repeating previous process stages.

The waterfall model is consistent with other engineering process models and documentation is produced at each phase. This makes the process visible so managers can monitor progress against the development plan. Its major problem is the inflexible partitioning of the project into distinct stages. Commitments must be made at an early stage in the process, which makes it difficult to respond to changing customer requirements.
In principle, the waterfall model should only be used when the requirements are well understood and unlikely to change radically during system development. However, the waterfall model reflects the type of process used in other engineering projects. As is easier to use a common management model for the whole project, software processes based on the waterfall model are still commonly used.

An important variant of the waterfall model is formal system development, where a mathematical model of a system specification is created. This model is then refined, using mathematical transformations that preserve its consistency, into exe- cutable code. Based on the assumption that your mathematical transformations are correct, you can therefore make a strong argument that a program generated in this way is consistent with its specification.

Incremental development
Incremental development is based on the idea of developing an initial implementa- tion, exposing this to user comment and evolving it through several versions until an adequate system has been developed (Figure). Specification, development, and
Concurrent Activities

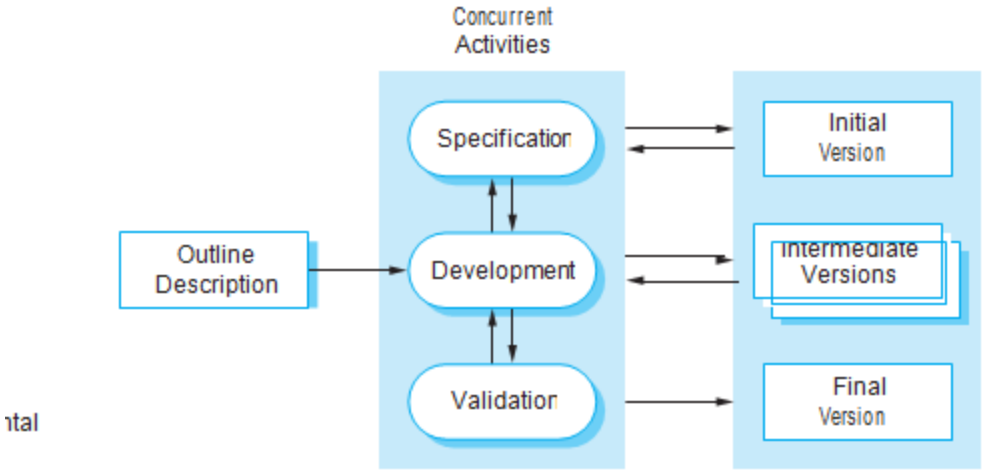


Figure:- Incremental development

validation activities are interleaved rather than separate, with rapid feedback across activities.
Incremental software development, which is a fundamental part of agile approaches, is better than a waterfall approach for most business, e-commerce, and personal systems. Incremental development reflects the way that we solve prob- lems. We rarely work out a complete problem solution in advance but move toward a solution in a series of steps, backtracking when we realize that we have made a mistake. By developing the software incrementally, it is cheaper and easier to make changes in the software as it is being developed.

Each increment or version of the system incorporates some of the functionality that is needed by the customer. Generally, the early increments of the system include the most important or most urgently required functionality. This means that the customer can evaluate the system at a relatively early stage in the development to see if it delivers what is required. If not, then only the current increment has to be changed and, possibly, new functionality defined for later increments.
Incremental development has three important benefits, compared to the waterfall model:

1. The cost of accommodating changing customer requirements is reduced. The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.
2. It is easier to get customer feedback on the development work that has been done. Customers can comment on demonstrations of the software and see how much has been implemented. Customers find it difficult to judge progress from software design documents.
3. More rapid delivery and deployment of useful software to the customer is possi- ble, even if all of the functionality has not been included. Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

Incremental development in some form is now the most common approach for the development of application systems. This approach can be either plan-driven, agile, or, more usually, a mixture of these approaches. In a plan-driven approach, the system increments are identified in advance; if an agile approach is adopted, the early increments are identified but the development of later increments depends on progress and customer priorities.
From a management perspective, the incremental approach has two problems:

1. The process is not visible. Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.
2. System structure tends to degrade as new increments are added. Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increas- ingly difficult and costly.

The problems of incremental development become particularly acute for large, complex, long-lifetime systems, where different teams develop different parts of the system. Large systems need a stable framework or architecture and the responsibili- ties of the different teams working on parts of the system need to be clearly defined with respect to that architecture. This has to be planned in advance rather than devel- oped incrementally.

Reuse-oriented software engineering

In the majority of software projects, there is some software reuse. This often happens informally when people working on the project know of designs or code that are similar to what is required. They look for these, modify them as needed, and incor- porate them into their system.

This informal reuse takes place irrespective of the development process that is used. However, in the 21st century, software development processes that focus on the reuse of existing software have become widely used. Reuse-oriented approaches rely on a large base of reusable software components and an integrating framework for the composition of these components. Sometimes, these components are systems in their own right (COTS or commercial off-the-shelf systems) that may provide spe- cific functionality such as word processing or a spreadsheet.

A general process model for reuse-based development is shown in Figure. Although the initial requirements specification stage and the validation stage are comparable with other software processes, the intermediate stages in a reuse- oriented process are different. These stages are:

1. Component analysis Given the requirements specification, a search is made for components to implement that specification. Usually, there is no exact match and the components that may be used only provide some of the functionality required.
2. Requirements modification During this stage, the requirements are analyzed using information about the components that have been discovered. They are then mod- ified to reflect the available components. Where modifications are impossible, the component analysis activity may be re-entered to search for alternative solutions.
3. System design with reuse During this phase, the framework of the system is designed or an existing framework is reused. The designers take into account the components that are reused and organize the framework to cater for this. Some new software may have to be designed if reusable components are not available.
4. Development and integration Software that cannot be externally procured is developed, and the components and COTS systems are integrated to create the new system. System integration, in this model, may be part of the development process rather than a separate activity.

There are three types of software component that may be used in a reuse-oriented process:

1. Web services that are developed according to service standards and which are available for remote invocation.
2. Collections of objects that are developed as a package to be integrated with a component framework such as .NET or J2EE.
3. Stand-alone software systems that are configured for use in a particular environment.

Reuse-oriented software engineering has the obvious advantage of reducing the amount of software to be developed and so reducing cost and risks. It usually also leads to faster delivery of the software. However, requirements compromises are inevitable and this may lead to a system that does not meet the real needs of users. Furthermore, some control over the system evolution is lost as new versions of the reusable components are not under the control of the organization using them.

Process activities

Real software processes are interleaved sequences of technical, collaborative, and managerial activities with the overall goal of specifying, designing, implementing, and testing a software system. Software developers use a variety of different software tools in their work. Tools are particularly useful for supporting the editing of different types of document and for managing the immense volume of detailed information that is generated in a large software project.

The four basic process activities of specification, development, validation, and evo- lution are organized differently in different development processes. In the waterfall model, they are organized in sequence, whereas in incremental development they are

interleaved. How these activities are carried out depends on the type of software, people, and organizational structures involved. In extreme programming, for example, specifications are written on cards. Tests are executable and developed before the program itself. Evolution may involve substantial system restructuring or refactoring.

Software specification

Software specification or requirements engineering is the process of understanding and defining what services are required from the system and identifying the con- straints on the system's operation and development. Requirements engineering is a particularly critical stage of the software process as errors at this stage inevitably lead to later problems in the system design and implementation.

The requirements engineering process (Figur) aims to produce an agreed requirements document that specifies a system satisfying stakeholder requirements. Requirements are usually presented at two levels of detail. End-users and customers need a high-level statement of the requirements; system developers need a more detailed system specification.

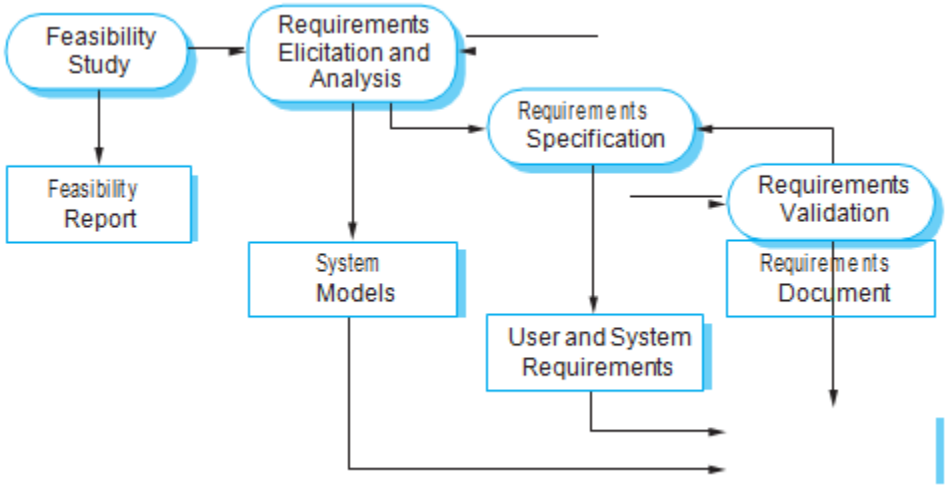


Figure:- The requirements engineering process

There are four main activities in the requirements engineering process:

1. Feasibility study An estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies. The study considers whether the proposed system will be cost-effective from a business point of view and if it can be developed within existing budgetary constraints. A feasibility study should be relatively cheap and quick. The result should inform the decision of whether or not to go ahead with a more detailed analysis.

2. Requirements elicitation and analysis This is the process of deriving the system requirements through observation of existing systems, discussions with poten- tial users and procurers, task analysis, and so on. This may involve the develop- ment of one or more system models and prototypes. These help you understand the system to be specified.
3. Requirements specification Requirements specification is the activity of trans- lating the information gathered during the analysis activity into a document that defines a set of requirements. Two types of requirements may be included in this document. User requirements are abstract statements of the system require- ments for the customer and end-user of the system; system requirements are a more detailed description of the functionality to be provided.
4. Requirements validation This activity checks the requirements for realism, consis- tency, and completeness. During this process, errors in the requirements document are inevitably discovered. It must then be modified to correct these problems.

Software design and implementation

The implementation stage of software development is the process of converting a system specification into an executable system. It always involves processes of soft- ware design and programming but, if an incremental approach to development is used, may also involve refinement of the software specification.

A software design is a description of the structure of the software to be implemented, the data models and structures used by the system, the interfaces between system com- ponents and, sometimes, the algorithms used. Designers do not arrive at a finished design immediately but develop the design iteratively. They add formality and detail as they develop their design with constant backtracking to correct earlier designs. Figure 2.5 is an abstract model of this process showing the inputs to the design process, process activities, and the documents produced as outputs from this process.

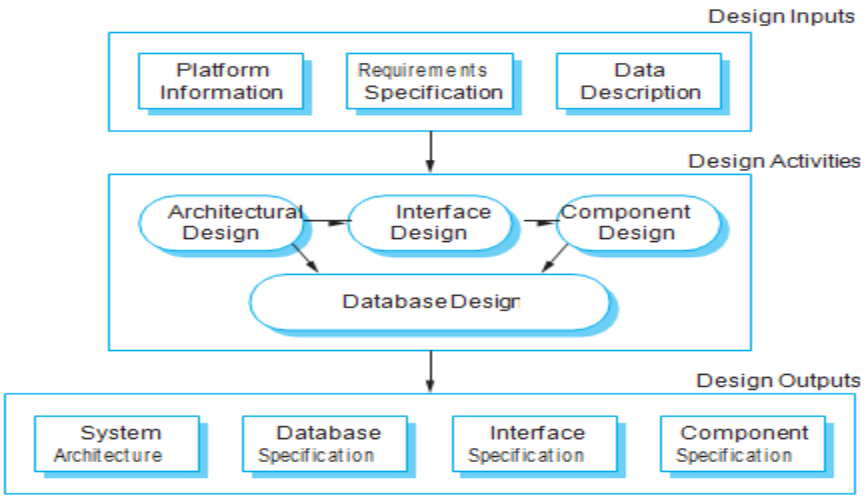


Figure:- A general model of the design process

The diagram suggests that the stages of the design process are sequential. In fact, design process activities are interleaved. Feedback from one stage to another and consequent design rework is inevitable in all design processes.

Most software interfaces with other software systems. These include the operating system, database, middleware, and other application systems. These make up the 'software platform', the environment in which the software will execute. Information about this platform is an essential input to the design process, as designers must decide how best to integrate it with the software's environment. The requirements specification is a description of the functionality the software must provide and its performance and dependability requirements. If the system is to process existing data, then the description of that data may be included in the platform specification; otherwise, the data description must be an input to the design process so that the system data organization to be defined. The activities in the design process vary, depending on the type of system being developed. For example, real-time systems require timing design but may not include a database so there is no database design involved.

Figure shows four activities that may be part of the design process for information systems:

1. Architectural design, where you identify the overall structure of the system, the principal components (sometimes called sub-systems or modules), their rela- tionships, and how they are distributed.
2. Interface design, where you define the interfaces between system components. This interface specification must be unambiguous. With a precise interface, a component can be used without other components having to know how it is implemented. Once interface specifications are agreed, the components can be designed and developed concurrently.
3. Component design, where you take each system component and design how it will operate. This may be a simple statement of the expected functionality to be implemented, with the specific design left to the programmer. Alternatively, it may be a list of changes to be made to a reusable component or a detailed design model. The design model may be used to automatically generate an implementation.
4. Database design, where you design the system data structures and how these are to be represented in a database. Again, the work here depends on whether an existing database is to be reused or a new database is to be created.

Software validation

Software validation or, more generally, verification and validation (V&V) is intended to show that a system both conforms to its specification and that it meets the expectations of the system customer. Program testing, where the system is exe- cuted using simulated test data, is the principal validation technique. Validation may also involve checking processes, such as inspections and reviews, at each stage of the software process from user requirements definition to program development. Because of the predominance of testing, the majority of validation costs are incurred during and after implementation.

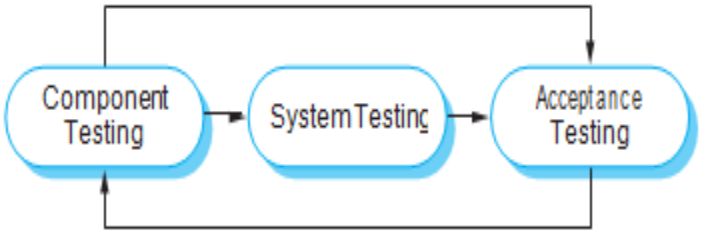


Figure:- Stages of Testing

Except for small programs, systems should not be tested as a single, monolithic unit. Figure shows a three-stage testing process in which system components are tested then the integrated system is tested and, finally, the system is tested with the customer's data. Ideally, component defects are discovered early in the process, and interface problems are found when the system is integrated. However, as defects are discovered, the program must be debugged and this may require other stages in the testing process to be repeated. Errors in program components, say, may come to light during system testing. The process is therefore an iterative one with information being fed back from later stages to earlier parts of the process.
The stages in the testing process are:

1. Development testing The components making up the system are tested by the people developing the system. Each component is tested independently, without other system components. Components may be simple entities such as functions or object classes, or may be coherent groupings of these entities. Test automa- tion tools, such as JUnit (Massol and Husted, 2003), that can re-run component tests when new versions of the component are created, are commonly used.

2. System testing System components are integrated to create a complete system. This process is concerned with finding errors that result from unanticipated interactions between components and component interface problems. It is also concerned with showing that the system meets its functional and non-functional requirements, and testing the emergent system properties. For large systems, this may be a multi-stage process where components are integrated to form sub- systems that are individually tested before these sub-systems are themselves integrated to form the final system.
3. Acceptance testing This is the final stage in the testing process before the system is accepted for operational use. The system is tested with data supplied by the system customer rather than with simulated test data. Acceptance testing may reveal errors and omissions in the system requirements definition, because the real data exercise the system in different ways from the test data. Acceptance testing may also reveal requirements problems where the system's facilities do not really meet the user's needs or the system performance is unacceptable.

Normally, component development and testing processes are interleaved. Programmers make up their own test data and incrementally test the code as it is developed. This is an economically sensible approach, as the programmer knows the component and is therefore the best person to generate test cases.

If an incremental approach to development is used, each increment should be tested as it is developed, with these tests based on the requirements for that incre- ment. In extreme programming, tests are developed along with the requirements before development starts. This helps the testers and developers to understand the requirements and ensures that there are no delays as test cases are created.

When a plan-driven software process is used (e.g., for critical systems develop- ment), testing is driven by a set of test plans. An independent team of testers works from these pre-formulated test plans, which have been developed from the system specification and design. Figure illustrates how test plans are the link between testing and development activities. This is sometimes called the V-model of develop- ment (turn it on its side to see the V).

Acceptance testing is sometimes called ‘alpha testing’. Custom systems are developed for a single client. The alpha testing process continues until the system developer and the client agree that the delivered system is an acceptable implemen- tation of the requirements.

When a system is to be marketed as a software product, a testing process called ‘beta testing’ is often used. Beta testing involves delivering a system to a number of potential customers who agree to use that system. They report problems to the sys- tem developers. This exposes the product to real use and detects errors that may not have been anticipated by the system builders. After this feedback, the system is mod- ified and released either for further beta testing or for general sale.

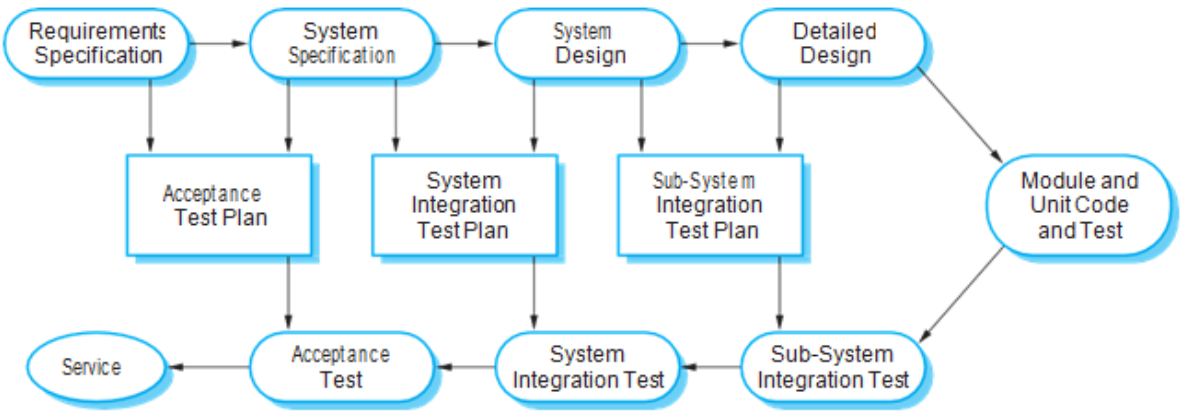


Figure:- Testing Phase in a plan-driven software process

Software evolution

The flexibility of software systems is one of the main reasons why more and more software is being incorporated in large, complex systems. Once a decision has been made to manufacture hardware, it is very expensive to make changes to the hardware design. However, changes can be made to software at any time during or after the system development. Even extensive changes are still much cheaper than corresponding changes to system hardware.

Historically, there has always been a split between the process of software development and the process of software evolution (software maintenance). People think of software development as a creative activity in which a software system is devel- oped from an initial concept through to a working system. However, they sometimes think of software maintenance as dull and uninteresting. Although the costs of main- tenance are often several times the initial development costs, maintenance processes are sometimes considered to be less challenging than original software development. This distinction between development and maintenance is increasingly irrelevant.

Hardly any software systems are completely new systems and it makes much more sense to see development and maintenance as a continuum. Rather than two separate processes, it is more realistic to think of software engineering as an evolutionary process (Figure) where software is continually changed over its lifetime in response to changing requirements and customer needs.

Coping with change

Change is inevitable in all large software projects. The system requirements change as the business procuring the system responds to external pressures and management priorities change. As new technologies become available, new design and implementation possibilities emerge. Therefore whatever software process model is used, it is essential that it can accommodate changes to the software being developed.

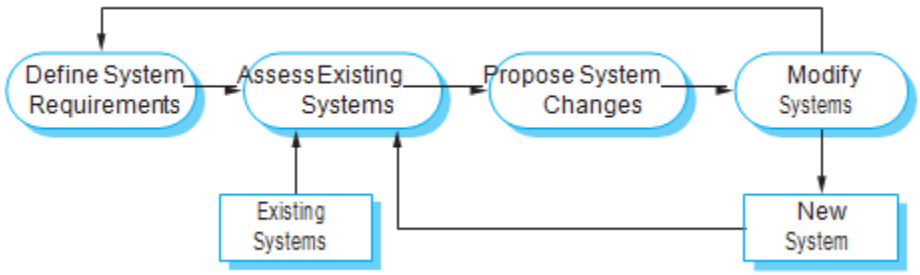


Figure:- System evolution

Change adds to the costs of software development because it usually means that work that has been completed has to be redone. This is called rework. For example, if the relationships between the requirements in a system have been analyzed and new requirements are then identified, some or all of the requirements analysis has to be repeated. It may then be necessary to redesign the system to deliver the new require- ments, change any programs that have been developed, and re-test the system.
There are two related approaches that may be used to reduce the costs of rework:

1. Change avoidance, where the software process includes activities that can antic- ipate possible changes before significant rework is required. For example, a pro- totype system may be developed to show some key features of the system to customers. They can experiment with the prototype and refine their require- ments before committing to high software production costs.
2. Change tolerance, where the process is designed so that changes can be accom- modated at relatively low cost. This normally involves some form of incremen- tal development. Proposed changes may be implemented in increments that have not yet been developed. If this is impossible, then only a single increment (a small part of the system) may have to be altered to incorporate the change.

In this section, I discuss two ways of coping with change and changing system requirements. These are:

1. System prototyping, where a version of the system or part of the system is developed quickly to check the customer's requirements and the feasibility of some design decisions. This supports change avoidance as it allows users to experiment with the system before delivery and so refine their requirements. The number of require- ments change proposals made after delivery is therefore likely to be reduced.
2. Incremental delivery, where system increments are delivered to the customer for comment and experimentation. This supports both change avoidance and change tolerance. It avoids the premature commitment to requirements for the whole system and allows changes to be incorporated into later increments at rel- atively low cost.

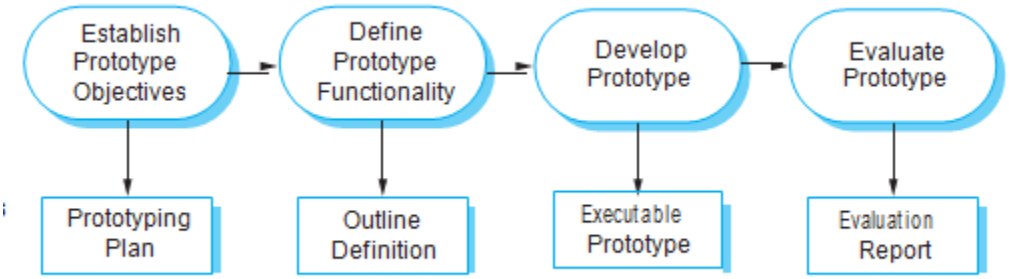


Figure:- The process of prototype development

Prototyping

A prototype is an initial version of a software system that is used to demonstrate concepts, try out design options, and find out more about the problem and its possi- ble solutions. Rapid, iterative development of the prototype is essential so that costs are controlled and system stakeholders can experiment with the prototype early in the software process.

A software prototype can be used in a software development process to help anticipate changes that may be required:

1. In the requirements engineering process, a prototype can help with the elicitation and validation of system requirements.
2. In the system design process, a prototype can be used to explore particular software solutions and to support user interface design.

System prototypes allow users to see how well the system supports their work. They may get new ideas for requirements, and find areas of strength and weakness in the software. They may then propose new system requirements. Furthermore, as the prototype is developed, it may reveal errors and omissions in the requirements that have been proposed. A function described in a specification may seem useful and well defined. However, when that function is combined with other functions, users often find that their initial view was incorrect or incomplete. The system specification may then be modified to reflect their changed understanding of the requirements.

A system prototype may be used while the system is being designed to carry out design experiments to check the feasibility of a proposed design. For example, a database design may be prototyped and tested to check that it supports efficient data access for the most common user queries. Prototyping is also an essential part of the user interface design process. Because of the dynamic nature of user interfaces, tex- tual descriptions and diagrams are not good enough for expressing the user interface requirements. Therefore, rapid prototyping with end-user involvement is the only sensible way to develop graphical user interfaces for software systems.

A process model for prototype development is shown in Figure 2.9. The objec- tives of prototyping should be made explicit from the start of the process. These may be to develop a system to prototype the user interface, to develop a system to validate functional system requirements, or to develop a system to demonstrate the feasibility

of the application to managers. The same prototype cannot meet all objectives. If the objectives are left unstated, management or end-users may misunderstand the function of the prototype. Consequently, they may not get the benefits that they expected from the prototype development.

The next stage in the process is to decide what to put into and, perhaps more importantly, what to leave out of the prototype system. To reduce prototyping costs and accelerate the delivery schedule, you may leave some functionality out of the prototype. You may decide to relax non-functional requirements such as response time and memory utilization. Error handling and management may be ignored unless the objective of the prototype is to establish a user interface. Standards of reliability and program quality may be reduced.

The final stage of the process is prototype evaluation. Provision must be made during this stage for user training and the prototype objectives should be used to derive a plan for evaluation. Users need time to become comfortable with a new sys- tem and to settle into a normal pattern of usage. Once they are using the system nor- mally, they then discover requirements errors and omissions.

A general problem with prototyping is that the prototype may not necessarily be used in the same way as the final system. The tester of the prototype may not be typ- ical of system users. The training time during prototype evaluation may be insuffi- cient. If the prototype is slow, the evaluators may adjust their way of working and avoid those system features that have slow response times. When provided with bet- ter response in the final system, they may use it in a different way.

Developers are sometimes pressured by managers to deliver throwaway proto- types, particularly when there are delays in delivering the final version of the soft- ware. However, this is usually unwise:

1. It may be impossible to tune the prototype to meet non-functional requirements, such as performance, security, robustness, and reliability requirements, which were ignored during prototype development.
2. Rapid change during development inevitably means that the prototype is undoc- umented. The only design specification is the prototype code. This is not good enough for long-term maintenance.
3. The changes made during prototype development will probably have degraded the system structure. The system will be difficult and expensive to maintain.
4. Organizational quality standards are normally relaxed for prototype development.

Prototypes do not have to be executable to be useful. Paper-based mock-ups of the system user interface (Rettig, 1994) can be effective in helping users refine an interface design and work through usage scenarios. These are very cheap to develop and can be constructed in a few days. An extension of this technique is a Wizard of Oz prototype where only the user interface is developed. Users interact with this interface but their requests are passed to a person who interprets them and outputs the appropriate response.

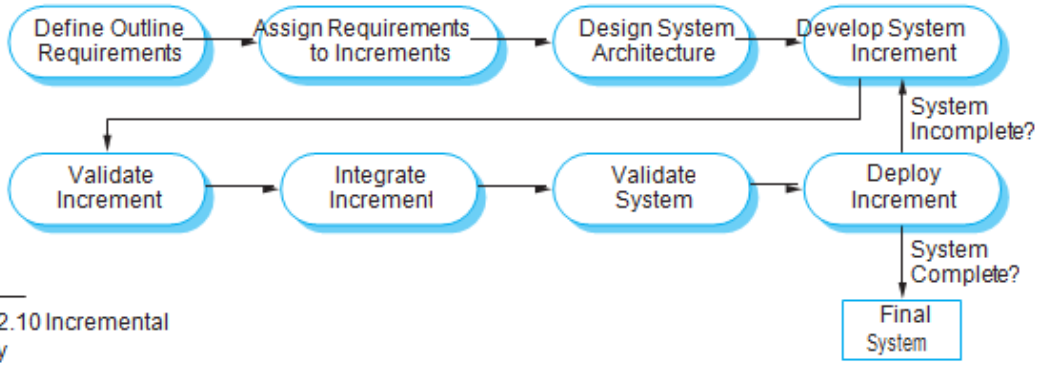


Figure 2.10 Incremental delivery

Figure:- Incremental Delivery

Incremental delivery

Incremental delivery (Figure 2.10) is an approach to software development where some of the developed increments are delivered to the customer and deployed for use in an operational environment. In an incremental delivery process, customers iden- tify, in outline, the services to be provided by the system. They identify which of the services are most important and which are least important to them. A number of delivery increments are then defined, with each increment providing a sub-set of the system functionality. The allocation of services to increments depends on the service priority, with the highest-priority services implemented and delivered first.

Once the system increments have been identified, the requirements for the serv- ices to be delivered in the first increment are defined in detail and that increment is developed. During development, further requirements analysis for later increments can take place but requirements changes for the current increment are not accepted. Once an increment is completed and delivered, customers can put it into service.

This means that they take early delivery of part of the system functionality. They can experiment with the system and this helps them clarify their requirements for later sys- tem increments. As new increments are completed, they are integrated with existing increments so that the system functionality improves with each delivered increment.
Incremental delivery has a number of advantages:

1. Customers can use the early increments as prototypes and gain experience that informs their requirements for later system increments. Unlike prototypes, these are part of the real system so there is no re-learning when the complete system is available.
2. Customers do not have to wait until the entire system is delivered before they can gain value from it. The first increment satisfies their most critical require- ments so they can use the software immediately.
3. The process maintains the benefits of incremental development in that it should be relatively easy to incorporate changes into the system.
4. As the highest-priority services are delivered first and increments then inte- grated, the most important system services receive the most testing. This means

that customers are less likely to encounter software failures in the most impor- tant parts of the system.

However, there are problems with incremental delivery:

1. Most systems require a set of basic facilities that are used by different parts of the system. As requirements are not defined in detail until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments.
2. Iterative development can also be difficult when a replacement system is being developed. Users want all of the functionality of the old system and are often unwilling to experiment with an incomplete new system. Therefore, getting use- ful customer feedback is difficult.
3. The essence of iterative processes is that the specification is developed in conjunc- tion with the software. However, this conflicts with the procurement model of many organizations, where the complete system specification is part of the system development contract. In the incremental approach, there is no complete system specification until the final increment is specified. This requires a new form of contract, which large customers such as government agencies may find difficult to accommodate.

There are some types of system where incremental development and delivery is not the best approach. These are very large systems where development may involve teams working in different locations, some embedded systems where the software depends on hardware development and some critical systems where all the require- ments must be analyzed to check for interactions that may compromise the safety or security of the system.
These systems, of course, suffer from the same problems of uncertain and chang- ing requirements. Therefore, to address these problems and get some of the benefits of incremental development, a process may be used in which a system prototype is developed iteratively and used as a platform for experiments with the system requirements and design. With the experience gained from the prototype, definitive requirements can then be agreed.

Boehm's spiral model

A risk-driven software process framework (the spiral model) was proposed by Boehm (1988). This is shown in Figure 2.11. Here, the software process is repre- sented as a spiral, rather than a sequence of activities with some backtracking from one activity to another. Each loop in the spiral represents a phase of the software process. Thus, the innermost loop might be concerned with system feasibility, the next loop with requirements definition, the next loop with system design, and so on. The spiral model combines change avoidance with change tolerance. It assumes that changes are a result of project risks and includes explicit risk management activities to reduce these risks.

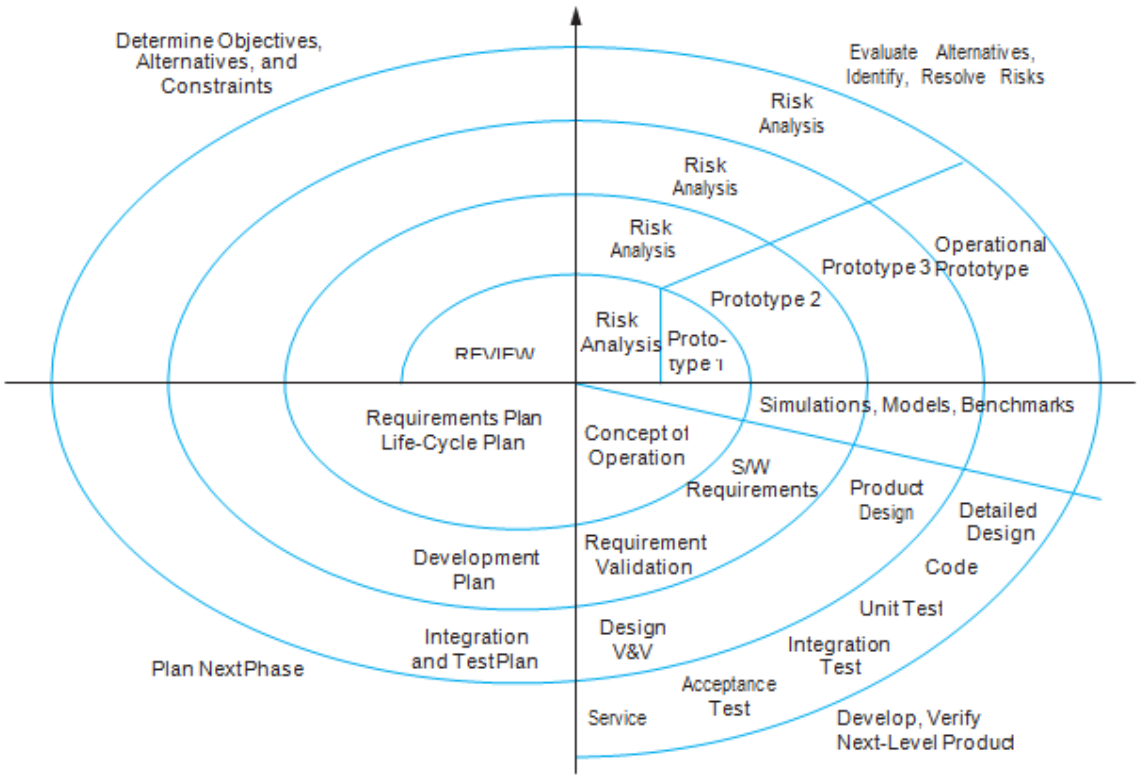


Figure:- Boehm's spiral model of the software process

Each loop in the spiral is split into four sectors:

1. Objective setting Specific objectives for that phase of the project are defined. Constraints on the process and the product are identified and a detailed manage- ment plan is drawn up. Project risks are identified. Alternative strategies, depending on these risks, may be planned.
2. Risk assessment and reduction For each of the identified project risks, a detailed analysis is carried out. Steps are taken to reduce the risk. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.
3. Development and validation After risk evaluation, a development model for the system is chosen. For example, throwaway prototyping may be the best devel- opment approach if user interface risks are dominant. If safety risks are the main consideration, development based on formal transformations may be the most appropriate process, and so on. If the main identified risk is sub-system integra- tion, the waterfall model may be the best development model to use.
4. Planning The project is reviewed and a decision made whether to continue with a further loop of the spiral. If it is decided to continue, plans are drawn up for the next phase of the project.

The main difference between the spiral model and other software process models is its explicit recognition of risk. A cycle of the spiral begins by elaborating objectives such as performance and functionality. Alternative ways of achieving these objec- tives, and dealing with the constraints on each of them, are then enumerated. Each alternative is assessed against each objective and sources of project risk are identi- fied. The next step is to resolve these risks by information-gathering activities such as more detailed analysis, prototyping, and simulation.

Once risks have been assessed, some development is carried out, followed by a plan- ning activity for the next phase of the process. Informally, risk simply means something that can go wrong. For example, if the intention is to use a new programming language, a risk is that the available compilers are unreliable or do not produce sufficiently effi- cient object code. Risks lead to proposed software changes and project problems such as schedule and cost overrun, so risk minimization is a very important project management activity.

The Rational Unified Process

The Rational Unified Process (RUP) (Krutchen, 2003) is an example of a modern process model that has been derived from work on the UML and the associated Unified Software Development Process (Rumbaugh, et al., 1999; Arlow and Neustadt, 2005). I have included a description here, as it is a good example of a hybrid process model.

The RUP recognizes that conventional process models present a single view of the process. In contrast, the RUP is normally described from three perspectives:

1. A dynamic perspective, which shows the phases of the model over time.
2. A static perspective, which shows the process activities that are enacted.
3. A practice perspective, which suggests good practices to be used during the process.

Most descriptions of the RUP attempt to combine the static and dynamic perspec- tives in a single diagram (Krutchen, 2003). I think that makes the process harder to understand, so I use separate descriptions of each of these perspectives.
The RUP is a phased model that identifies four discrete phases in the software process. However, unlike the waterfall model where phases are equated with process activities, the phases in the RUP are more closely related to business rather than technical concerns.

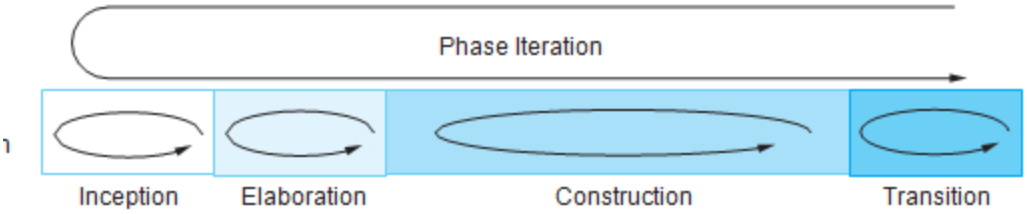


Figure:- Phases in the Rational Unified Process

1. Inception The goal of the inception phase is to establish a business case for the system. You should identify all external entities (people and systems) that will interact with the system and define these interactions. You then use this infor- mation to assess the contribution that the system makes to the business. If this contribution is minor, then the project may be cancelled after this phase.
2. Elaboration The goals of the elaboration phase are to develop an understanding of the problem domain, establish an architectural framework for the system, develop the project plan, and identify key project risks. On completion of this phase you should have a requirements model for the system, which may be a set of UML use-cases, an architectural description, and a development plan for the software.
3. Construction The construction phase involves system design, programming, and testing. Parts of the system are developed in parallel and integrated during this phase. On completion of this phase, you should have a working software system and associated documentation that is ready for delivery to users.
4. Transition The final phase of the RUP is concerned with moving the system from the development community to the user community and making it work in a real environment. This is something that is ignored in most software process models but is, in fact, an expensive and sometimes problematic activity. On completion of this phase, you should have a documented software system that is working correctly in its operational environment.

Iteration within the RUP is supported in two ways. Each phase may be enacted in an iterative way with the results developed incrementally. In addition, the whole set of phases may also be enacted incrementally, as shown by the looping arrow from Transition to Inception in Figure

The static view of the RUP focuses on the activities that take place during the development process. These are called workflows in the RUP description. There are six core process workflows identified in the process and three core supporting work- flows. The RUP has been designed in conjunction with the UML, so the workflow description is oriented around associated UML models such as sequence models, object models, etc. The core engineering and support workflows are described in Figure

The advantage in presenting dynamic and static views is that phases of the development process are not associated with specific workflows. In principle at least, all of the RUP workflows may be active at all stages of the process. In the early phases of the process, most effort will probably be spent on workflows such as business modelling and requirements and, in the later phases, in testing and deployment.

The practice perspective on the RUP describes good software engineering prac- tices that are recommended for use in systems development. Six fundamental best practices are recommended:

1. Develop software iteratively Plan increments of the system based on customer priorities and develop the highest-priority system features early in the develop- ment process.
2. Manage requirements Explicitly document the customer's requirements and keep track of changes to these requirements. Analyze the impact of changes on the system before accepting them.
3. Use component-based architectures Structure the system architecture into components, as discussed earlier in this chapter.
4. Visually model software Use graphical UML models to present static and dynamic views of the software.
5. Verify software quality Ensure that the software meets the organizational quality standards.
6. Control changes to software Manage changes to the software using a change management system and configuration management procedures and tools.

Prof. P. B. Dhore
(Subject In-charge)