SOFTWARE ENGINEERING NOTES

CMP-3310

TALHA SHAHAB

Table of Contents

Software Engineering

1. Software and Software Engineering

"Software is a place where dreams are planted and nightmares harvested, an abstract, mystical swamp where terrible demons compete with magical panaceas, a world of werewolves and silver bullets."

-Brad J. Cox

Software

"Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market."

Software is:

- ✓ Instructions (computer programs) that when executed provide desired features, function, and performance;
- ✓ Data structures that enable the programs to adequately manipulate information
- ✓ Descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

There are two fundamental types of software product:

1. **Generic products:** These are stand-alone systems that are produced by a development 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.
2. **Customized (or bespoke) products**: These are systems which 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.

Legacy software systems . . . were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms. The proliferation of such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve.

What are the attributes of good software?

1. **Maintainability:** Software should be written in such a way that it may evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable consequence of a changing business environment.
2. **Dependability:** Software dependability has a range of characteristics, including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure.
3. **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.
4. **Usability:** Software must be usable, without undue effort, by the type of user for whom it is designed. This means that it should have an appropriate user interface and adequate documentation.

Software Engineering

Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification to maintaining the system after it has gone into use. The notion of

software engineering was first proposed in 1968 at a conference held to discuss what was then called the 'software crises'. This software crisis resulted directly from the introduction of new computer hardware based on integrated circuits.

Engineers make things work. They apply theories, methods and tools where these are appropriate, but they use them selectively and always try to discover solutions to problems even when there are no applicable theories and methods. Engineers also recognize that they must work to organizational and financial constraints, so they look for solutions within these constraints.

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

"In general, software engineers adopt a systematic and organized approach to their work, as this is often the most effective way to produce high-quality software."

1.1 Nature of Software

Today, software takes on a dual role. It is a product, and at the same time, the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or more broadly, by a network of computers that are accessible by local hardware. As the vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments).

Software has characteristics that are considerably different than those of hardware:

1. **Software is developed or engineered; it is not manufactured in the classical sense.**

Although some similarities exist between software development and hardware manufacturing, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software. Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different.

2. **Software doesn't "wear out."**

Software doesn't wear out, but it does deteriorate. Hardware exhibits relatively high failure rates early in its life, defects are corrected and the failure rate drops to a steady-state level (hopefully, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

Software is not susceptible to the environmental maladies that cause hardware to wear out.

3. **Although the industry is moving toward component-based construction, most software continues to be custom built.**

A software component should be designed and implemented so that it can be reused in many different programs. Modern reusable components encapsulate both data and the processing that is applied to the data, enabling the software engineer to create new applications from reusable parts.

1.2 Software Application Domains

Today, seven broad categories of computer software present continuing challenges for software engineers:

1. **System software:** A collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) processes complex, but determinate, information structures. Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data. In either case, the systems software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.
2. **Application software:** Stand-alone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making.
3. **Engineering/scientific software:** has been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing.
4. **Embedded software**: resides within a product or system and is used to implement and control features and functions for the end user and for the system itself.
5. **Product-line software:** designed to provide a specific capability for use by many different customers. Product-line software can focus on a limited and esoteric marketplace (e.g., inventory control products) or address mass consumer markets (e.g., word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications).
6. **Web applications:** called "WebApps," this network-centric software category spans a wide array of applications. In their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics.
7. **Artificial intelligence software:** makes use of non-numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing

1.3 The Unique Nature of WebApps

In the early days of the World Wide Web (circa 1990 to 1995), websites consisted of little more than a set of linked hypertext files that presented information using text and limited graphics. As time passed, the augmentation of HTML by development tools (e.g., XML, Java) enabled Web engineers to provide computing capability along with informational content. Web-based systems and applications (I refer to these collectively as WebApps) were born.

Today, WebApps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications.

The following attributes are encountered in the vast majority of WebApps.

1. **Network intensiveness.** A WebApp resides on a network and must serve the needs of a diverse community of clients. The network may enable worldwide access and communication (i.e., the Internet) or more limited access and communication (e.g., a corporate Intranet)
2. **Concurrency.** A large number of users may access the WebApp at one time. In many cases, the patterns of usage among end users will vary greatly.
3. **Unpredictable load.** The number of users of the WebApp may vary by orders of magnitude from day to day.
4. **Performance.** If a WebApp user must wait too long (for access, for server-side processing, for client-side formatting and display), he or she may decide to go elsewhere.

5. **Availability.** Although expectation of 100 percent availability is unreasonable, users of popular WebApps often demand access on a 24/7/365 basis.
6. **Data driven.** The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end user. In addition, WebApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment (e.g., e-commerce or financial applications)
7. **Content sensitive.** The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.
8. **Continuous evolution.** Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously. It is not unusual for some WebApps (specifically, their content) to be updated on a minute-by-minute schedule or for content to be independently computed for each request.
9. **Immediacy.** Although immediacy—the compelling need to get software to market quickly—is a characteristic of many application domains, WebApps often exhibit a time-to-market that can be a matter of a few days or weeks.
10. **Security.** Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end users who may access the application. In order to protect sensitive content and provide secure modes of data transmission, strong security measures must be implemented throughout the infrastructure that supports a WebApp and within the application itself.
11. **Aesthetics.** An undeniable part of the appeal of a WebApp is its look and feel. When an application has been designed to market or sell products or ideas, aesthetics may have as much to do with success as technical design.

1.4 Software Engineering

"Software Engineering: The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software."

In order to build software that is ready to meet the challenges of the twenty-first century, you must recognize a few simple realities:

- ✓ Understand the problem before you build a solution. It follows that a concerted effort should be made to understand the problem before a software solution is developed.
- ✓ Design is a pivotal software engineering activity. The information technology requirements demanded by individuals, businesses, and governments grow increasing complex with each passing year. Large teams of people now create computer programs that were once built by a single individual. The complexity of these new computer-based systems and products demands careful attention to the interactions of all system elements. It follows that design becomes a pivotal activity.
- ✓ Both quality and maintainability are an outgrowth of good design. Individuals, businesses, and governments increasingly rely on software for strategic and tactical decision making as well as day-to-day operations and control. If the software fails, people and major enterprises can experience anything from minor inconvenience to catastrophic failures. It follows that software should exhibit high quality.

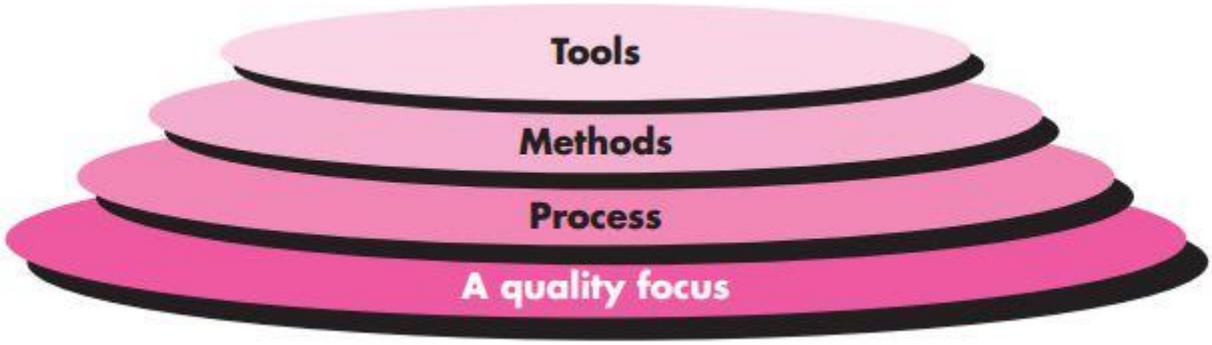
[Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

Software engineering is a layered technology. Total quality management, Six Sigma, and similar philosophies foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering. The bedrock that supports software engineering is a quality focus.

Process defines a framework that must be established for effective delivery of software engineering technology. The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products are produced, milestones are established, quality is ensured, and change is properly managed.

Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

Software engineering tools provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering, is established.



What is CASE?

The acronym CASE stands for Computer-Aided Software Engineering. It covers a wide range of different types of programs that are used to support software process activities such as requirements analysis, system modelling, debugging and testing. All methods now come with associated CASE technology such as editors for the notations used in the method, analysis modules which check the system model according to the method rules and report generators to help create system documentation. The CASE tools may also include a code generator that automatically generates source code from the system model and some process guidance for software engineers.

1.5 The software process

A software process is the set of activities and associated results that produce a software product.

"A process is a collection of activities, actions, and tasks that are performed when some work product is to be created."

- ✓ An **activity** strives to achieve a broad objective (e.g., communication with stakeholders) and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied.
- ✓ An **action** (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural design model).
- ✓ A **task** focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

A process framework establishes the foundation for a complete software engineering process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size or

complexity. In addition, the process framework encompasses a set of umbrella activities that are applicable across the entire software process. A generic process framework for software engineering encompasses five activities:

1. **Communication:** Before any technical work can commence, it is critically important to communicate and collaborate with the customer to understand customer's objectives for the project and to gather requirements that help define software features and functions.
2. **Planning:** It defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule. A software process model is a simplified description of a software process that presents one view of that process. Process models may include activities that are part of the software process, software products and the roles of people involved in soft
3. **Modeling:** A software process model is a simplified description of a software process that presents one view of that process. Process models may include activities that are part of the software process, software products and the roles of people involved in software engineering.
4. **Construction**: This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code. It is where the software is designed and programmed.
5. **Deployment**: The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

For many software projects, framework activities are applied iteratively as a project progresses. That is, communication, planning, modeling, construction, and deployment are applied repeatedly through a number of project iterations. Each project iteration produces a software increment that provides stakeholders with a subset of overall software features and functionality.

1.5.1 Umbrella Activity

Software engineering process framework activities are complemented by a number of umbrella activities. In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk. Typical umbrella activities include:

- ✓ **Software project tracking and control:** allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.
- ✓ **Risk management:** assesses risks that may affect the outcome of the project or the quality of the product.
- ✓ **Software quality assurance**: defines and conducts the activities required to ensure software quality.
- ✓ **Technical reviews:** assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.
- ✓ **Measurement:** defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs; can be used in conjunction with all other framework and umbrella activities.
- ✓ **Software configuration management:** manages the effects of change throughout the software process.
- ✓ **Reusability management**: defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.
- ✓ **Work product preparation and production:** encompasses the activities required to create work products such as models, documents, logs, forms, and lists.

1.5.2 Software Process Models

A software process model is a simplified description of a software process that presents one view of that process. Process models may include activities that are part of the software process, software products and the roles of people involved in software engineering. Some examples of the types of software process model that may be produced are:

- ✓ **A workflow model:** This shows the sequence of activities in the process along with their inputs, outputs and dependencies. The activities in this model represent human actions.
- ✓ **A dataflow or activity model:** This represents the process as a set of activities, each of which carries out some data transformation. It shows how the input to the process, such as a specification, is transformed to an output, such as a design. The activities here may represent transformations carried out by people or by computers.
- ✓ **A role/action model:** This represents the roles of the people involved in the software process and the activities for which they are responsible.

Most software process models are based on one of three general models or paradigms of software development:

1. **The waterfall approach:** This takes the above activities and represents them as separate process phases such as requirements specification, software design, implementation, testing and so on. After each stage is defined it is 'signed-off', and development goes on to the following stage.
2. **Iterative development:** This approach interleaves the activities of specification, development and validation. An initial system is rapidly developed from very abstract specifications. This is then refined with customer input to produce a system that satisfies the customer's needs. The system may then be delivered. Alternatively, it may be re-implemented using a more structured approach to produce a more robust and maintainable system.
3. **Component-based software engineering (CBSE):** This technique assumes that parts of the system already exist. The system development process focuses on integrating these parts rather than developing them from scratch.

1.6 Software Engineering Practice

In the sections that follow, you'll gain a basic understanding of the generic concepts and principles that apply to framework activities.

1.6.1 The Essence of Practice

George Polya outlined the essence of problem solving, and consequently, the essence of software engineering practice:

1 Understand the problem (communication and analysis).
2 Plan a solution (modeling and software design).
3 Carry out the plan (code generation).
4 Examine the result for accuracy (testing and quality assurance).

Their details are as follows:

1. Understand the problem

"There is a grain of discovery in the solution of any problem."

-George Polya

It's sometimes difficult to admit, but most of us suffer from hubris when we're presented with a problem. We listen for a few seconds and then think, Oh yeah, I understand, let's get on with solving this thing. Unfortunately, understanding isn't always that easy. It's worth spending a little time answering a few simple questions:

- ✓ Who has a stake in the solution to the problem? That is, who are the stakeholders?
- ✓ What are the unknowns? What data, functions, and features are required to properly solve the problem?
- ✓ Can the problem be compartmentalized? Is it possible to represent smaller problems that may be easier to understand?
- ✓ Can the problem be represented graphically? Can an analysis model be created?

2. Plan the solution

Now you understand the problem (or so you think) and you can't wait to begin coding. Before you do, slow down just a bit and do a little design:

- ✓ Have you seen similar problems before? Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- ✓ Has a similar problem been solved? If so, are elements of the solution reusable?
- ✓ Can sub problems be defined? If so, are solutions readily apparent for the sub problems?
- ✓ Can you represent a solution in a manner that leads to effective implementation?
- ✓ Can a design model be created?

3. Carry out the plan

The design you've created serves as a road map for the system you want to build. There may be unexpected detours, and it's possible that you'll discover an even better route as you go, but the "plan" will allow you to proceed without getting lost.

- ✓ Does the solution conform to the plan? Is source code traceable to the design model?
- ✓ Is each component part of the solution provably correct? Have the design and code been reviewed, or better, have correctness proofs been applied to the algorithm?

4. Examine the result

You can't be sure that your solution is perfect, but you can be sure that you've designed a sufficient number of tests to uncover as many errors as possible.

- ✓ Is it possible to test each component part of the solution? Has a reasonable testing strategy been implemented?
- ✓ Does the solution produce results that conform to the data, functions, and features that are required? Has the software been validated against all stakeholder requirements?

It shouldn't surprise you that much of this approach is common sense. In fact, it's reasonable to state that a commonsense approach to software engineering will never lead you astray.

1.6.2 General Principles

David Hooker [Hoo96] has proposed seven principles that focus on software engineering practice as a whole. They are reproduced in the following paragraphs:

1. The Reason It All Exists

A software system exists for one reason: to provide value to its users. All decisions should be made with this in mind. Before specifying a system requirement, before noting a piece of system functionality, before determining

the hardware platforms or development processes, ask yourself questions such as: “Does this add real value to the system?” If the answer is “no,” don’t do it. All other principles support this one.

2. KISS (Keep It Simple, Stupid!)

Software design is not a haphazard process. There are many factors to consider in any design effort. All design should be as simple as possible, but no simpler. This facilitates having a more easily understood and easily maintained system. This is not to say that features, even internal features, should be discarded in the name of simplicity. Indeed, the more elegant designs are usually the more simple ones. Simple also does not mean “quick and dirty.” In fact, it often takes a lot of thought and work over multiple iterations to simplify. The payoff is software that is more maintainable and less error-prone.

“There is a certain majesty in simplicity which is far above all the quaintness of wit.”

-Alexander Pope

3. Maintain the Vision

A clear vision is essential to the success of a software project. Without one, a project almost unfailingly ends up being “of two [or more] minds” about itself. Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws. . . . Compromising the architectural vision of a software system weakens and will eventually break even the well-designed systems. Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.

4. What You Produce, Others Will Consume

Seldom is an industrial-strength software system constructed and used in a vacuum. In some way or other, someone else will use, maintain, document, or otherwise depend on being able to understand your system. So, always specify, design, and implement knowing someone else will have to understand what you are doing. The audience for any product of software development is potentially large. Specify with an eye to the users. Design, keeping the implementers in mind. Code with concern for those that must maintain and extend the system. Someone may have to debug the code you write, and that makes them a user of your code. Making their job easier adds value to the system.

5. Be Open to the Future

A system with a long lifetime has more value. In today’s computing environments, where specifications change on a moment’s notice and hardware platforms are obsolete just a few months old, software lifetimes are typically measured in months instead of years. However, true “industrial-strength” software systems must endure far longer. To do this successfully, these systems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start. Never design yourself into a corner. Always ask “what if,” and prepare for all possible answers by creating systems that solve the general problem, not just the specific one. This could very possibly lead to the reuse of an entire system.

6. Plan Ahead for Reuse

Reuse saves time and effort.15Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system. The reuse of code and designs has been proclaimed as a major benefit of using object-oriented technologies. However, the return on this investment is not automatic. To leverage the reuse possibilities that object-oriented [or conventional] programming provides requires forethought and planning. There are many techniques to realize reuse at every level of the system development process. . . . Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.

7. Think!

This last principle is probably the most overlooked. Placing clear, complete thought before action almost always produces better results. When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes a valuable experience. A side effect of thinking is learning to recognize when you don't know something, at which point you can research the answer. When clear thought has gone into a system, value comes out. Applying the first six principles requires intense thought, for which the potential rewards are enormous.

1.7 Software Myths

Software myths—erroneous beliefs about software and the process that is used to build it—can be traced to the earliest days of computing. Myths have a number of attributes that make them insidious.

Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike. However, old attitudes and habits are difficult to modify, and remnants of software myths remain.

A. Management myths
B. Customer myths
C. Practitioner's myths

Management myths

Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

i. Myth:

We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?

Reality:

The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it adaptable? Is it streamlined to improve time-to-delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."

ii. Myth:

If we get behind schedule, we can add more programmers and catch up (sometimes called the "Mongolian horde" concept).

Reality:

Software development is not a mechanistic process like manufacturing. In the words of Brooks "adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well-coordinated manner.

iii. Myth:

If I decide to outsource the software project to a third party, I can just relax and let that firm build it.

Reality:
If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

Customer myths

A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and, ultimately, dissatisfaction with the developer.

i. **Myth:**
A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.

Reality:
Although a comprehensive and stable statement of requirements is not always possible, an ambiguous "statement of objectives" is a recipe for disaster. Unambiguous requirements (usually derived iteratively) are developed only through effective and continuous communication between customer and developer.

ii. **Myth:**
Software requirements continually change, but change can be easily accommodated because software is flexible.

Reality:
It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small. However, as time passes, the cost impact grows rapidly—resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification.

Practitioner's myths

Myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. During the early days, programming was viewed as an art form. Old ways and attitudes die hard.

i. **Myth:**
Once we write the program and get it to work, our job is done.

Reality:
Someone once said that "the sooner you begin 'writing code,' the longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

ii. **Myth:**
Until I get the program "running" I have no way of assessing its quality.

Reality:
One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the technical review. Software reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

iii. Myth:

The only deliverable work product for a successful project is the working program.

Reality:

A working program is only one part of a software configuration that includes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering and, more important, guidance for software support.

iv. Myth:

Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.

Reality:

Software engineering is not about creating documents. It is about creating a quality product. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

2. Process Models

- Generic Process Models
- Perspective Process Models
- Specialized Process Models

When you work to build a product or system, it's important to go through a series of predictable steps—a road map that helps you create a timely, high-quality result. The road map that you follow is called a "software process." Because it provides stability, control, and organization to an activity that can, if left uncontrolled, become quite chaotic. However, a modern software engineering approach must be "agile." It must demand only those activities, controls, and work products that are appropriate for the project team and the product that is to be produced.

Although there are many software processes, some fundamental activities are common to all software processes:

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

Explanation of some software processes are as follows:

2.1 Generic Process Model

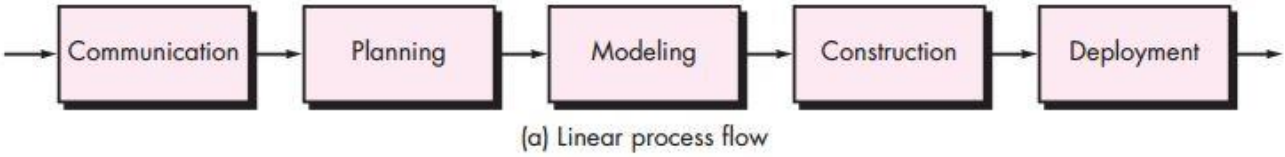
A process is defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks reside within a framework or model that defines their relationship with the process and with one another.

Each framework activity is populated by a set of software engineering actions. Each software engineering action is defined by a **task set** that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.

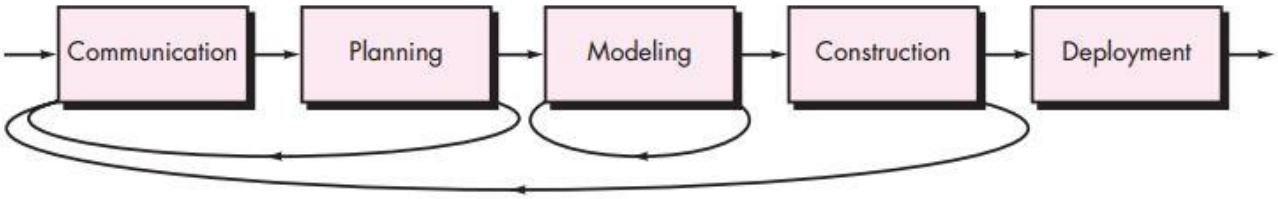
A **generic process framework** for software engineering defines five framework activities: communication, planning, modeling, construction, and deployment. In addition, a set of umbrella activities: project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others are applied throughout the process.

Process flow describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time.

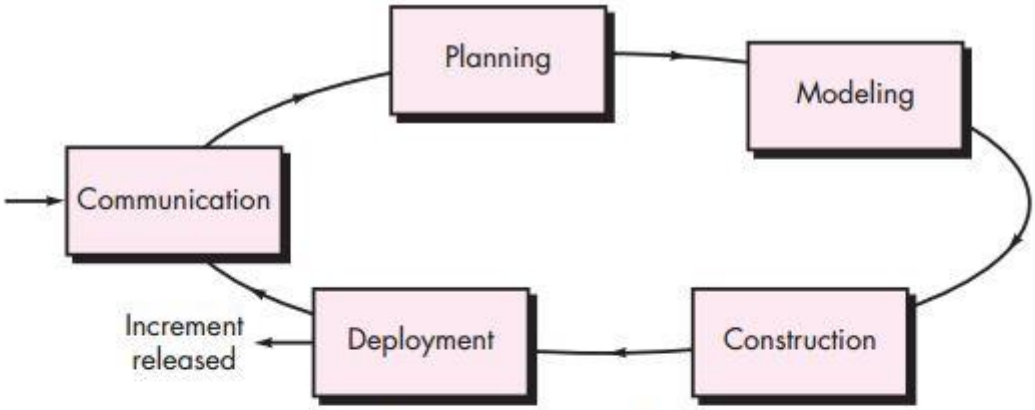
- ✓ **Linear process flow:** A linear process flow executes each of the five framework activities in sequence, beginning with communication and culminating with deployment.
- ✓ **Iterative process flow:** An iterative process flow repeats one or more of the activities before proceeding to the next.
- ✓ **Evolutionary process flow:** An evolutionary process flow executes the activities in a "circular" manner. Each circuit through the five activities leads to a more complete version of the software.
- ✓ **Parallel process flow:** A parallel process flow executes one or more activities in parallel with other activities (e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software).



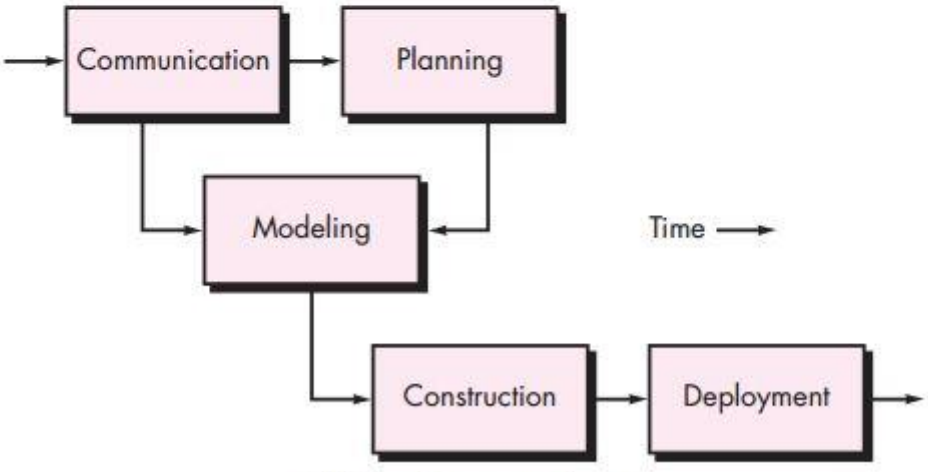
(a) Linear process flow



(b) Iterative process flow



(c) Evolutionary process flow



(d) Parallel process flow

2.1.1 Framework Activity

A generic process framework for software engineering defines five framework activities: communication, planning, modeling, construction, and deployment. (Explained in topic 1.5) If the project is considerably more complex with many stakeholders, each with a different set of (sometime conflicting) requirements, the communication activity might have six distinct actions: inception, elicitation, elaboration, negotiation,

specification, and validation. Each of these software engineering actions would have many work tasks and a number of distinct work products.

2.1.2 Task Set

"A task set defines the actual work to be done to accomplish the objectives of a software engineering action."

Different projects demand different task sets. The software team chooses the task set based on problem and project characteristics. Software engineering action can be represented by a number of different task sets each a collection of software engineering work tasks, related work products, quality assurance points, and project milestones. You should choose a task set that best accommodates the needs of the project and the characteristics of your team.

2.1.3 Process Patterns

A process pattern describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem.

Stated in more general terms, a process pattern provides you with a template a consistent method for describing problem solutions within the context of the software process. By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project.

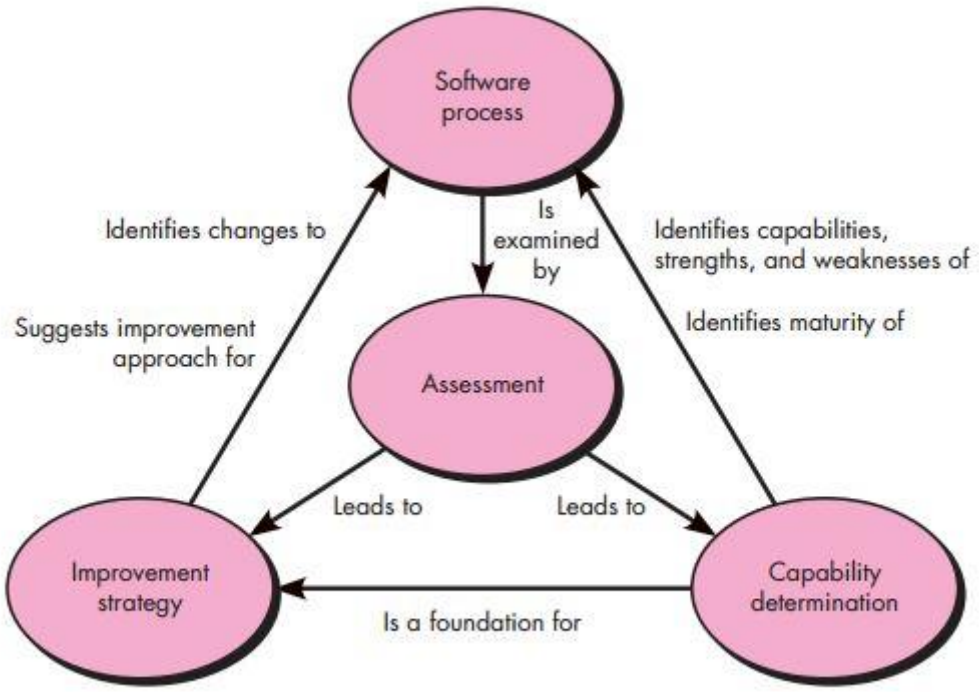
- ✓ **Stage pattern:** defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions and work tasks, a stage pattern incorporates multiple task patterns that are relevant to the stage (framework activity). An example of a stage pattern might be Establishing Communication. This pattern would incorporate the task pattern Requirements Gathering and others.
- ✓ **Task pattern:** defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., Requirements Gathering is a task pattern).
- ✓ **Phase pattern:** define the sequence of framework activities that occurs within the process, even when the overall flow of activities is iterative in nature. An example of a phase pattern might be Spiral Model or Prototyping.

Patterns can be defined at any level of abstraction. In some cases, a pattern might be used to describe a problem (and solution) associated with a complete process model (e.g., prototyping). In other situations, patterns can be used to describe a problem (and solution) associated with a framework activity (e.g., planning) or an action within a framework activity (e.g., project estimating).

2.1.4 Process Assessment and Improvement

The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics. Process patterns must be coupled with solid software engineering practice.

"Software process improvement encompasses a set of activities that will lead to a better software process and, as a consequence, higher-quality software delivered in a more-timely manner."



A number of different approaches to software process assessment and improvement have been proposed over the past few decades:

- **Standard CMMI Assessment Method for Process Improvement (SCAMPI):** provides a five-step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting, and learning. The SCAMPI method uses the **SEI** (Software Engineering Institute) **CMMI** (Capability Maturity Model Integration) as the basis for assessment.
- **CMM-Based Appraisal for Internal Process Improvement (CBA IPI)**: provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment.
- **SPICE (ISO/IEC15504):** a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process.
- **ISO 9001:2000 for Software:** a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies.

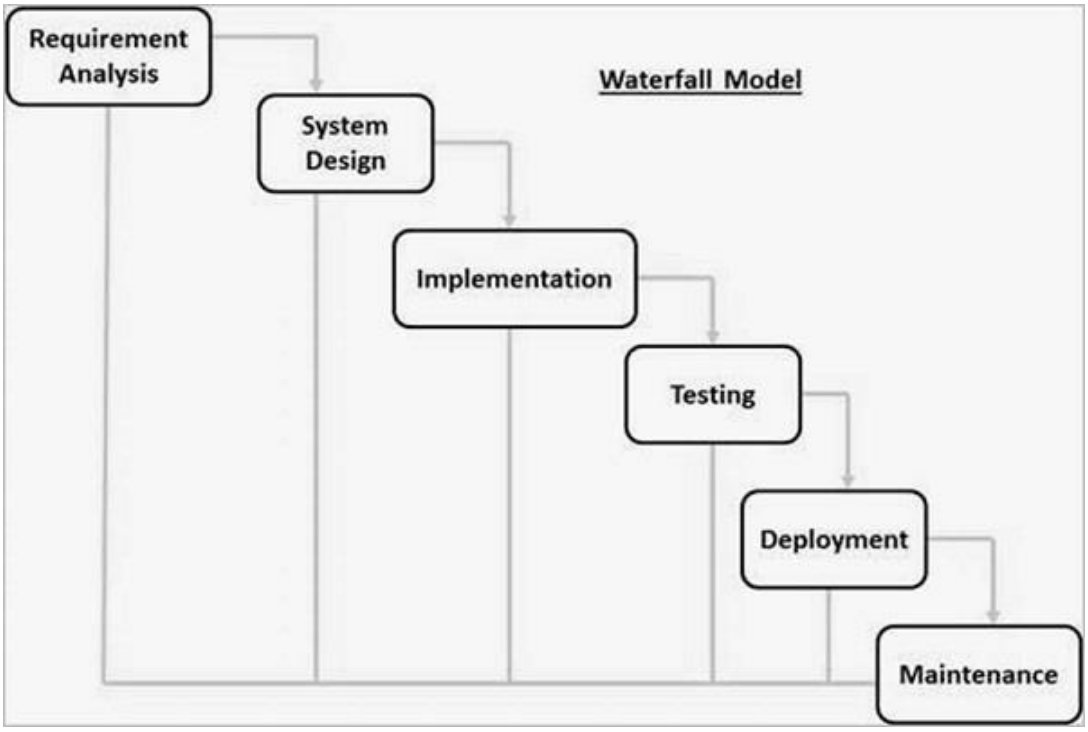
2.2 Perspective Process Models

Prescriptive process models were originally proposed to bring order to the chaos of software development. History has indicated that these traditional models have brought a certain amount of useful structure to software engineering work and have provided a reasonably effective road map for software teams. However, software engineering work and the product that it produces remain on “the edge of chaos.”

All software process models can accommodate the generic framework activities, but each applies a different emphasis to these activities and defines a process flow that invokes each framework activity (as well as software engineering actions and tasks) in a different manner.

2.2.1 The Waterfall Model

The waterfall model, sometimes called the classic life cycle, suggests a systematic, sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software.



The sequential phases in Waterfall model are:

1. **Requirement Gathering and analysis**: All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document.
2. **System Design**: The requirement specifications from first phase are studied in this phase and the system design is prepared. This system design helps in specifying hardware and system requirements and helps in defining the overall system architecture.
3. **Implementation**: With inputs from the system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality, which is referred to as Unit Testing.
4. **Integration and Testing**: All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.
5. **Deployment of system**: Once the functional and non-functional testing is done; the product is deployed in the customer environment or released into the market.
6. **Maintenance**: There are some issues which come up in the client environment. To fix those issues, patches are released. Also to enhance the product some better versions are released. Maintenance is done to deliver these changes in the customer environment.

All these phases are cascaded to each other in which progress is seen as flowing steadily downwards (like a waterfall) through the phases. The next phase is started only after the defined set of goals are achieved for previous phase and it is signed off, so the name "Waterfall Model". In this model, phases do not overlap.

- **Waterfall Model Applications**

Every software developed is different and requires a suitable SDLC approach to be followed based on the internal and external factors. Some situations where the use of Waterfall model is most appropriate are:

- ✓ Requirements are very well documented, clear and fixed.
- ✓ Product definition is stable.
- ✓ Technology is understood and is not dynamic.
- ✓ There are no ambiguous requirements.
- ✓ Ample resources with required expertise are available to support the product.
- ✓ The project is short.

➢ Advantages of Waterfall Model

The advantages of waterfall development are that it allows for departmentalization and control. A schedule can be set with deadlines for each stage of development and a product can proceed through the development process model phases one by one.

Development moves from concept, through design, implementation, testing, installation, troubleshooting, and ends up at operation and maintenance. Each phase of development proceeds in strict order.

Some of the major advantages of the Waterfall Model are as follows:

- ✓ Simple and easy to understand and use
- ✓ Easy to manage due to the rigidity of the model. Each phase has specific deliverables and a review process.
- ✓ Phases are processed and completed one at a time.
- ✓ Works well for smaller projects where requirements are very well understood.
- ✓ Clearly defined stages.
- ✓ Well understood milestones.
- ✓ Easy to arrange tasks.
- ✓ Process and results are well documented.

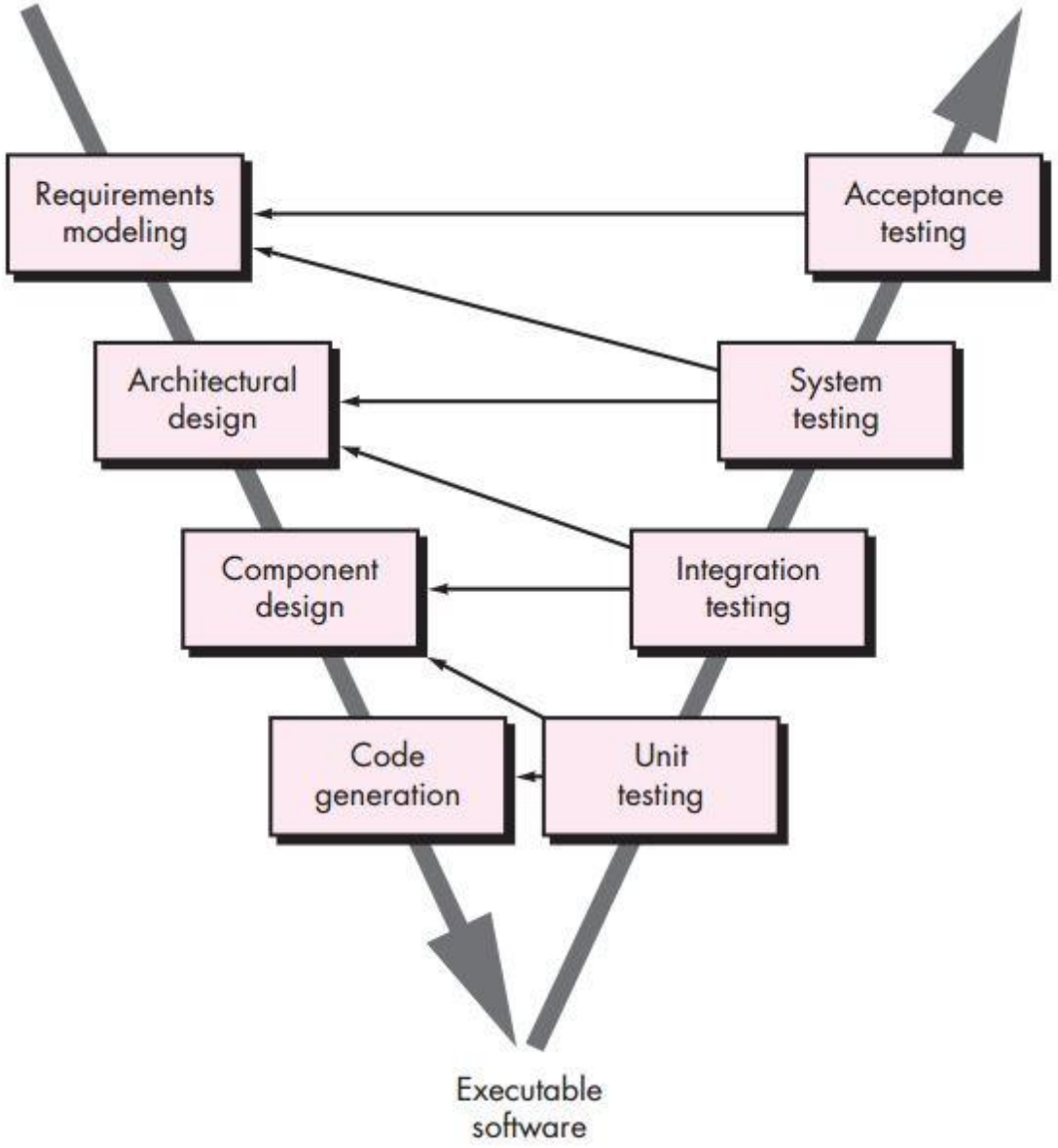
➢ Disadvantages of Waterfall Model

The disadvantage of waterfall development is that it does not allow much reflection or revision. Once an application is in the testing stage, it is very difficult to go back and change something that was not well-documented or thought upon in the concept stage.

The major disadvantages of the Waterfall Model are as follows

- ✓ No working software is produced until late during the life cycle.
- ✓ High amounts of risk and uncertainty.
- ✓ Not a good model for complex and object-oriented projects.
- ✓ Poor model for long and ongoing projects.
- ✓ Not suitable for the projects where requirements are at a moderate to high risk of changing. So, risk and uncertainty is high with this process model.
- ✓ It is difficult to measure progress within stages.
- ✓ Cannot accommodate changing requirements.
- ✓ Adjusting scope during the life cycle can end a project.
- ✓ Integration is done as a "big-bang. at the very end, which doesn't allow identifying any technological or business bottleneck or challenges early.

A variation in the representation of the waterfall model is called the **V-model**. The V-model depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities. As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution.



Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moved down the left side.

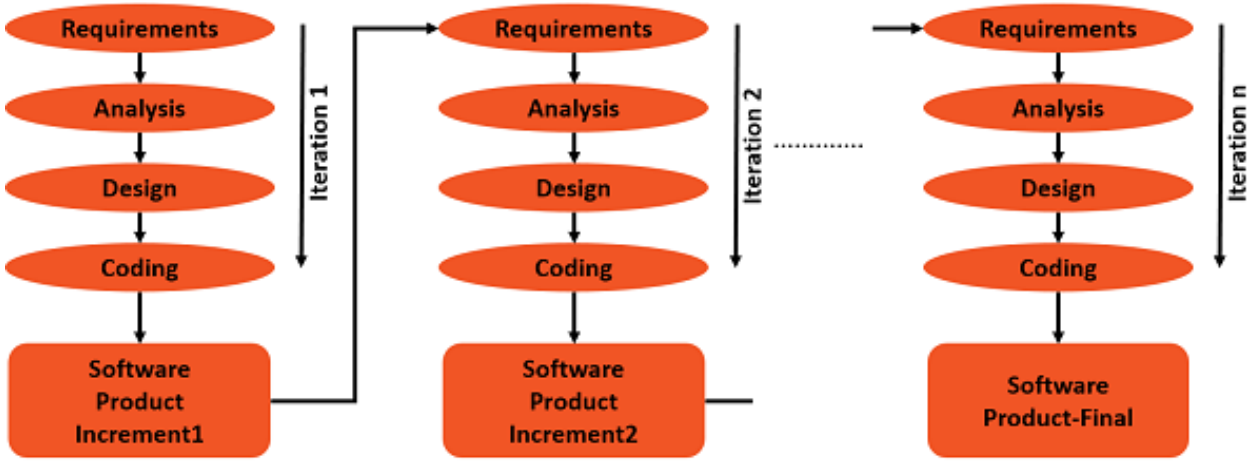
Today, software work is fast-paced and subject to a never-ending stream of changes (to features, functions, and information content). The waterfall model is often inappropriate for such work. However, it can serve as a useful process model in situations where requirements are fixed and work is to proceed to completion in a linear manner.

2.2.2 Incremental Process Model

The incremental model delivers a series of releases, called increments that provide progressively more functionality for the customer as each increment is delivered. The incremental model combines elements of linear and parallel process flows *(at page 13)*.

"Incremental Model is a process of software development where requirements are broken down into multiple standalone modules of software development cycle. Incremental development is done in steps from analysis design, implementation, testing/verification, maintenance."

When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed evaluation). As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.



➢ Advantages of Incremental Process Model

The advantages or strengths of Iterative Incremental model are:

- ✓ You can develop prioritized requirements first.
- ✓ Initial product delivery is faster.
- ✓ Customers gets important functionality early.
- ✓ Lowers initial delivery cost.
- ✓ Each release is a product increment, so that the customer will have a working product at hand all the time.
- ✓ Customer can provide feedback to each product increment, thus avoiding surprises at the end of development.
- ✓ Requirements changes can be easily accommodated.

➢ Disadvantages of Incremental Process Model

The disadvantages of the Iterative Incremental model are:

- ✓ Requires effective planning of iterations.
- ✓ Requires efficient design to ensure inclusion of the required functionality and provision for changes later.
- ✓ Requires early definition of a complete and fully functional system to allow the definition of increments.
- ✓ Well-defined module interfaces are required, as some are developed long before others are developed.
- ✓ Total cost of the complete system is not lower.

When to Use Incremental Process Model?

Iterative Incremental model can be used when:

- ✓ Most of the requirements are known up-front but are expected to evolve over time.
- ✓ The requirements are prioritized.
- ✓ There is a need to get the basic functionality delivered fast.
- ✓ A project has lengthy development schedules.
- ✓ A project has new technology.
- ✓ The domain is new to the team.

2.2.3 Evolutionary Process Models

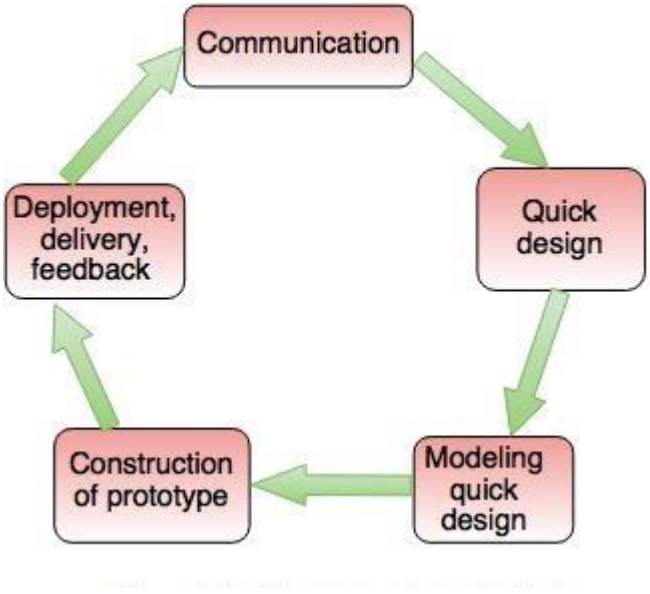
Evolutionary process models produce an increasingly more complete version of the software with each iteration. Evolutionary models are iterative type models. They allow to develop more complete versions of the software.

Following are the evolutionary process models.

A. The prototyping model
B. The spiral model
C. Concurrent development model

Their details are as follows:

A. The Prototyping model



Prototype is defined as first or preliminary form using which other forms are copied or derived. Prototype model is a set of general objectives for software. It does not identify the requirements like detailed input, output. It is software working model of limited functionality. In this model, working programs are quickly produced.

The different phases of Prototyping model are:

1. **Communication**: In this phase, developer and customer meet and discuss the overall objectives of the software.
2. **Quick design:** Quick design is implemented when requirements are known. It includes only the important aspects like input and output format of the software. It focuses on those aspects which are visible to the user rather than the detailed plan. It helps to construct a prototype.
3. **Modeling quick design:** This phase gives the clear idea about the development of software because the software is now built. It allows the developer to better understand the exact requirements.
4. **Construction of prototype**: The prototype is evaluated by the customer itself.
5. **Deployment, delivery, feedback:** If the user is not satisfied with current prototype then it refines according to the requirements of the user. The process of refining the prototype is repeated until all the requirements of users are met. When the users are satisfied with the developed prototype then the system is developed on the basis of final prototype.

Advantages of Prototyping Model

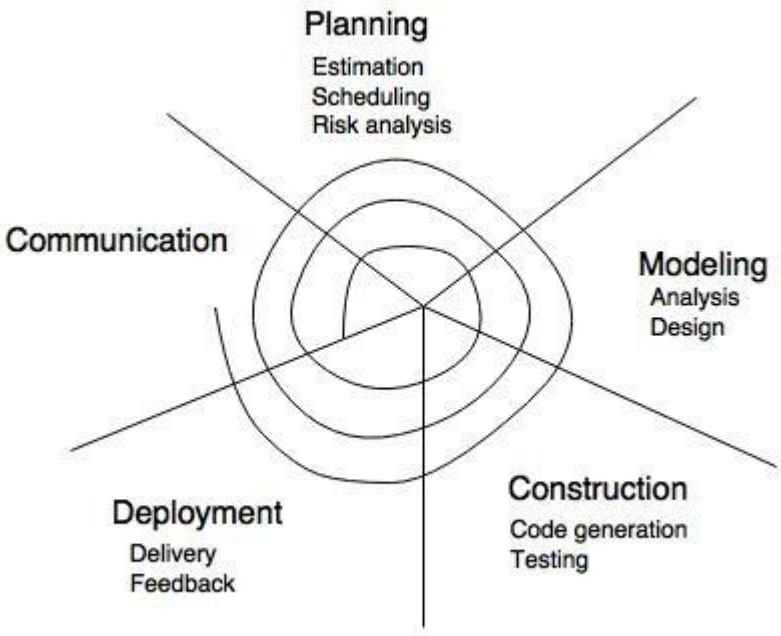
- ✓ Prototype model need not know the detailed input, output, processes, adaptability of operating system and full machine interaction.
- ✓ In the development process of this model users are actively involved.
- ✓ The development process is the best platform to understand the system by the user.
- ✓ Errors are detected much earlier.
- ✓ Gives quick user feedback for better solutions.
- ✓ It identifies the missing functionality easily. It also identifies the confusing or difficult functions.

Disadvantages of Prototyping Model:

- ✓ The client involvement is more and it is not always considered by the developer.
- ✓ It is a slow process because it takes more time for development.
- ✓ Many changes can disturb the rhythm of the development team.
- ✓ It is a thrown away prototype when the users are confused with it.

B. The Spiral model

Spiral model is a risk driven process model. It is used for generating the software projects. In spiral model, an alternate solution is provided if the risk is found in the risk analysis, then alternate solutions are suggested and implemented. It is a combination of prototype and sequential model or waterfall model. In one iteration all activities are done, for large projects the output is small. The framework activities of the spiral model are as shown in the following figure.



<u>The description of the phases of the spiral model is same as that of the process model.</u>

Advantages of Spiral Model

- ✓ It reduces high amount of risk.
- ✓ It is good for large and critical projects.
- ✓ It gives strong approval and documentation control.
- ✓ In spiral model, the software is produced early in the life cycle process.

Disadvantages of Spiral Model

- ✓ It can be costly to develop a software model.
- ✓ It is not used for small projects.

C. The concurrent development model

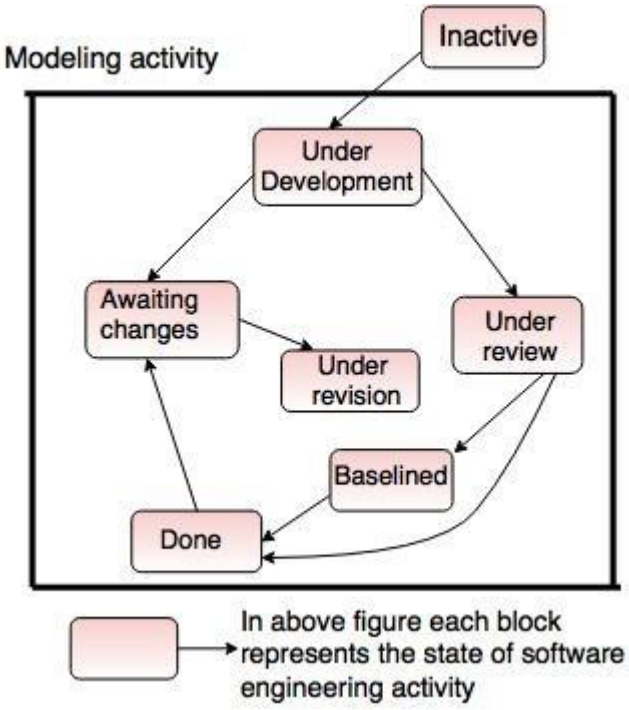
The concurrent development model is called as concurrent model. The communication activity has completed in the first iteration and exits in the awaiting changes state. The modeling activity completed its initial communication and then go to the underdevelopment state. If the customer specifies the change in the requirement, then the modeling activity moves from the under development state into the awaiting change state. The concurrent process model activities moving from one state to another state.

Advantages of the concurrent development model

- ✓ This model is applicable to all types of software development processes.
- ✓ It is easy for understanding and use.
- ✓ It gives immediate feedback from testing.
- ✓ It provides an accurate picture of the current state of a project.

Disadvantages of the concurrent development model

- ✓ It needs better communication between the team members. This may not be achieved all the time.
- ✓ It requires to remember the status of the different activities.



2.3 Specialized Process Model

There are four types of specialized process model:

- **Component Based Development**
- **The Formal Methods**
- **Aspect-Oriented Software Development** *(Not include in Syllabus)*
- **The Unified Process** *(Not include in Syllabus)*

Their details are as follows:

2.3.1 Component Based Development

Component-based development focuses on the decomposition of the design into individual functional or logical components that represent well-defined communication interfaces containing methods, events, and properties. It provides a higher level of abstraction and divides the problem into sub-problems, each associated with component partitions.

The primary objective of component-based development is to ensure **component reusability**. A component encapsulates functionality and behaviors of a software element into a reusable and self-deployable binary unit.

What is a Component?

A component is a modular, portable, replaceable, and reusable set of well-defined functionality that encapsulates its implementation and exporting it as a higher-level interface.

A component is a software object, intended to interact with other components, encapsulating certain functionality or a set of functionalities. It has an obviously defined interface and conforms to a recommended behavior common to all components within an architecture.

A software component can be defined as a unit of composition with a contractually specified interface and explicit context dependencies only. That is, a software component can be deployed independently and is subject to composition by third parties.

Advantages of Component Based Development

- ✓ **Ease of deployment:** As new compatible versions become available, it is easier to replace existing versions with no impact on the other components or the system as a whole.
- ✓ **Reduced cost:** The use of third-party components allows you to spread the cost of development and maintenance.
- ✓ **Ease of development:** Components implement well-known interfaces to provide defined functionality, allowing development without impacting other parts of the system.
- ✓ **Reusable:** The use of reusable components means that they can be used to spread the development and maintenance cost across several applications or systems.
- ✓ **Modification of technical complexity:** A component modifies the complexity through the use of a component container and its services.
- ✓ **Reliability:** The overall system reliability increases since the reliability of each individual component enhances the reliability of the whole system via reuse.
- ✓ **System maintenance and evolution:** Easy to change and update the implementation without affecting the rest of the system.
- ✓ **Independent:** Independency and flexible connectivity of components. Independent development of components by different group in parallel. Productivity for the software development and future software development.

2.3.2 The Formal Methods Model

The formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable you to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called cleanroom software engineering is currently applied by some software development organizations.

- ✓ The development of formal models is currently quite time consuming and expensive.
- ✓ Because few software developers have the necessary background to apply formal methods, extensive training is required.
- ✓ It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

Formal methods can be used at a number of levels:

- **Level 0:** Formal specification may be undertaken and then a program developed from this informally. This has been dubbed formal methods lite. This may be the most cost-effective option in many cases.
- **Level 1:** Formal development and formal verification may be used to produce a program in a more formal manner. For example, proofs of properties or refinement from the specification to a program may be undertaken. This may be most appropriate in high-integrity systems involving safety or security.
- **Level 2:** Theorem provers may be used to undertake fully formal machine-checked proofs. This can be very expensive and is only practically worthwhile if the cost of mistakes is extremely high (e.g., in critical parts of microprocessor design).

3. Agile Development

"Agile SDLC model is a combination of iterative and incremental process models with focus on process adaptability and customer satisfaction by rapid delivery of working software product. Agile Methods break the product into small incremental builds. These builds are provided in iterations."

Agile software development refers to a group of software development methodologies based on iterative development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams. Agile methods or Agile processes generally promote a disciplined project management process that encourages frequent inspection and adaptation, a leadership philosophy that encourages teamwork, self-organization and accountability, a set of engineering best practices intended to allow for rapid delivery of high-quality software, and a business approach that aligns development with customer needs and company goals. Agile development refers to any development process that is aligned with the concepts of the Agile Manifesto. The Manifesto was developed by a group fourteen leading figures in the software industry, and reflects their experience of what approaches do and do not work for software development.

"Agile Software Development is an umbrella term for a set of methods and practices based on the values and principles expressed in the Agile Manifesto."

3.1 Agility Principles

The Agile Alliance defines 12 agility principles for those who want to achieve agility:

1) Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2) Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3) Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4) Business people and developers must work together daily throughout the project.
5) Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6) The most efficient and effective method of conveying information to and within a development team is face-to-face conversation
7) Working software is the primary measure of progress.
8) Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9) Continuous attention to technical excellence and good design enhances agility.
10) Simplicity—the art of maximizing the amount of work not done—is essential.
11) The best architectures, requirements, and designs emerge from self–organizing teams.
12) At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Not every agile process model applies these 12 principles with equal weight, and some models choose to ignore (or at least downplay) the importance of one or more of the principles.

3.2 Human Factors

"Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams." The key point in this statement is that the process molds to the needs of the people and team, not the other way around.

A. **Competence:** In an agile development (as well as software engineering) context, "competence" encompasses innate talent, specific software-related skills, and overall knowledge of the process that the team has chosen to apply. Skill and knowledge of process can and should be taught to all people who serve as agile team members.

B. **Common focus:** Although members of the agile team may perform different tasks and bring different skills to the project, all should be focused on one goal—to deliver a working software increment to the customer within the time promised. To achieve this goal, the team will also focus on continual adaptations (small and large) that will make the process fit the needs of the team.

C. **Collaboration:** Software engineering (regardless of process) is about assessing, analyzing, and using information that is communicated to the software team; creating information that will help all stakeholders understand the work of the team; and building information (computer software and relevant databases) that provides business value for the customer. To accomplish these tasks, team members must collaborate—with one another and all other stakeholders.

D. **Decision-making ability:** Any good software team (including agile teams) must be allowed the freedom to control its own destiny. This implies that the team is given autonomy—decision-making authority for both technical and project issues.

E. **Fuzzy problem-solving ability:** Software managers must recognize that the agile team will continually have to deal with ambiguity and will continually be buffeted by change. In some cases, the team must accept the fact that the problem they are solving today may not be the problem that needs to be solved tomorrow. However, lessons learned from any problem-solving activity (including those that solve the wrong problem) may be of benefit to the team later in the project.

F. **Mutual trust and respect**: The agile team must become what DeMarco and Lister call a "jelled" team. A jelled team exhibits the trust and respect that are necessary to make them "so strongly knit that the whole is greater than the sum of the parts."

G. **Self-organization:** In the context of agile development, self-organization implies three things:
   - ✓ The agile team organizes itself for the work to be done,
   - ✓ The team organizes the process to best accommodate its local environment,
   - ✓ The team organizes the work schedule to best achieve delivery of the software increment.

   Self-organization has a number of technical benefits, but more importantly, it serves to improve collaboration and boost team morale. In essence, the team serves as its own management.

3.3 Agile Model - Pros and Cons

Agile methods are being widely accepted in the software world recently. However, this method may not always be suitable for all products. Here are some pros and cons of the Agile model.

The advantages of the Agile Model are as follows:

- ✓ It is a very realistic approach to software development.
- ✓ Promotes teamwork and cross training.
- ✓ Functionality can be developed rapidly and demonstrated.
- ✓ Resource requirements are minimum.
- ✓ Suitable for fixed or changing requirements
- ✓ Delivers early partial working solutions.
- ✓ Good model for environments that change steadily.
- ✓ Minimal rules, documentation easily employed.
- ✓ Enables concurrent development and delivery within an overall planned context.
- ✓ Little or no planning required.

- ✓ Easy to manage.
- ✓ Gives flexibility to developers.

The disadvantages of the Agile Model are as follows:

- × Not suitable for handling complex dependencies.
- × More risk of sustainability, maintainability and extensibility.
- × An overall plan, an agile leader and agile PM practice is a must without which it will not work.
- × Strict delivery management dictates the scope, functionality to be delivered, and adjustments to meet the deadlines.
- × Depends heavily on customer interaction, so if customer is not clear, team can be driven in the wrong direction.
- × There is a very high individual dependency, since there is minimum documentation generated.
- × Transfer of technology to new team members may be quite challenging due to lack of documentation.

3.4 Extreme Programming

Extreme programming (XP) is one of the most important software development framework of Agile models. It is used to improve software quality and responsive to customer requirements. The extreme programming model recommends taking the best practices that have worked well in the past in program development projects to extreme levels.

A. Good practices needs to practiced extreme programming

Some of the good practices that have been recognized in the extreme programming model and suggested to maximize their use are given below:

- ✓ **Code Review:** Code review detects and corrects errors efficiently. It suggests pair programming as coding and reviewing of written code carried out by a pair of programmers who switch their works between them every hour.
- ✓ **Testing:** Testing code helps to remove errors and improves its reliability. XP suggests test-driven development (TDD) to continually write and execute test cases. In the TDD approach test cases are written even before any code is written.
- ✓ **Incremental development:** Incremental development is very good because customer feedback is gained and based on this development team come up with new increments every few days after each iteration.
- ✓ **Simplicity:** Simplicity makes it easier to develop good quality code as well as to test and debug it.
- ✓ **Design:** Good quality design is important to develop a good quality software. So, everybody should design daily.
- ✓ **Integration testing**: It helps to identify bugs at the interfaces of different functionalities. Extreme programming suggests that the developers should achieve continuous integration by building and performing integration testing several times a day.

B. Basic principles of Extreme programming

XP is based on the frequent iteration through which the developers implement User Stories. User stories are simple and informal statements of the customer about the functionalities needed. A User story is a conventional description by the user about a feature of the required system. It does not mention finer details such as the different scenarios that can occur. On the basis of User stories, the project team proposes Metaphors. Metaphors are a common vision of how the system would work. The development team may decide to build a Spike for some feature. A Spike is a very simple program that is constructed to explore the

suitability of a solution being proposed. It can be considered similar to a prototype. Some of the basic activities that are followed during software development by using XP model are given below:

- ✓ **Coding:** The concept of coding which is used in XP model is slightly different from traditional coding. Here, coding activity includes drawing diagrams (modeling) that will be transformed into code, scripting a web-based system and choosing among several alternative solutions.
- ✓ **Testing:** XP model gives high importance on testing and considers it be the primary factor to develop a fault-free software.
- ✓ **Listening:** The developers needs to carefully listen to the customers if they have to develop a good quality software. Sometimes programmers may not have the depth knowledge of the system to be developed. So, it is desirable for the programmers to understand properly the functionality of the system and they have to listen to the customers.
- ✓ **Designing:** Without a proper design, a system implementation becomes too complex and very difficult to understand the solution, thus it makes maintenance expensive. A good design results elimination of complex dependencies within a system. So, effective use of suitable design is emphasized.
- ✓ **Feedback:** One of the most important aspects of the XP model is to gain feedback to understand the exact customer needs. Frequent contact with the customer makes the development effective.
- ✓ **Simplicity:** The main principle of the XP model is to develop a simple system that will work efficiently in present time, rather than trying to build something that would take time and it may never be used. It focuses on some specific features that are immediately needed, rather than engaging time and effort on speculations of future requirements.

C. Applications of Extreme Programming (XP)

Some of the projects that are suitable to develop using XP model are given below:

- ✓ **Small projects:** XP model is very useful in small projects consisting of small teams as face to face meeting is easier to achieve.
- ✓ **Projects involving new technology or Research projects: This** type of projects face changing of requirements rapidly and technical problems. So XP model is used to complete this type of projects.

4. Introduction to Systems Analysis and Design

What is a System?

The word System is derived from Greek word Systema, which means an organized relationship between any set of components to achieve some common cause or objective. A system is a collection of elements or components that are organized for a common purpose.

A system is "an orderly grouping of interdependent components linked together according to a plan to achieve a specific goal."

Systems development is systematic process which includes phases such as planning, analysis, design, deployment, and maintenance. Here, in this tutorial, we will primarily focus on:

a. **Systems analysis**
b. **Systems design**

Their short details are as follows:

a. Systems Analysis

It is a process of collecting and interpreting facts, identifying the problems, and decomposition of a system into its components. System analysis is conducted for the purpose of studying a system or its parts in order to identify its objectives. It is a problem solving technique that improves the system and ensures that all the components of the system work efficiently to accomplish their purpose. **Analysis specifies what the system should do.**

b. Systems Design

It is a process of planning a new business system or replacing an existing system by defining its components or modules to satisfy the specific requirements. Before planning, you need to understand the old system thoroughly and determine how computers can best be used in order to operate efficiently. **System Design focuses on how to accomplish the objective of the system.**

System Analysis and Design (SAD) mainly focuses on:

- Systems
- Processes
- Technology

➢ **Constraints of a System**

A system must have three basic constraints:

- ✓ A system must have some **structure and behavior** which is designed to achieve a predefined objective.
- ✓ **Interconnectivity** and **interdependence** must exist among the system components.
- ✓ The **objectives of the organization** have a **higher priority** than the objectives of its subsystems. For example, traffic management system, payroll system, automatic library system, human resources information system.

4.1 Business Information Systems

An information system (IS) is an organized system for the collection, organization, storage and communication of information.

Information systems are the combination of people, information technology, and business processes to accomplish a business objective. Every information system (IS) has people, processes, and information technology. In fact, many IS professionals add most of their value working with people and processes. They manage the programmers but typically avoid programming themselves. We can represent an information system as a triangle with people, processes, and information technology (computers) on the three vertices. The three parts of an information system are often referred to as the information systems triangle. Every information system is designed to make someone's life easier. Unfortunately, that someone is not always the consumer.

4.1.1 Purpose of Business Information Systems

Information systems are implemented within an organization for the purpose of improving the effectiveness and efficiency of that organization. Capabilities of the information system and characteristics of the organization, its work systems, its people, and its development and implementation methodologies together determine the extent to which that purpose is achieved.

4.2 Information System Components

A system is a set of related components that produces specific results. An information system has five key components hardware, software, data, processes, and people.

- **Hardware:** The term hardware refers to machinery. This category includes the computer itself, which is often referred to as the central processing unit (CPU), and all of its support equipment. Hardware consists of everything in the physical layer of the information system. For example, hardware can include servers, workstations, networks, telecommunications equipment, fiber optic cables, handheld computers, scanners, digital capture devices, and other technology based infrastructure. As new technologies emerge, manufacturers race to market the innovations and reap the rewards.
- **Software:** Software refers to the programs that control the hardware and produce the desired information or results. Software consists of system software and application software. **System software** manages the hardware components, which can include a single workstation or a global network with many thousands of clients. **Application software** consists of programs that support day-to-day business functions and provide users with the information they require. Application software can serve one user or thousands of users throughout the organization. Application software includes horizontal and vertical systems. A **horizontal system** is a system, such as an inventory or payroll application, that can be adapted for use in many different types of companies. A **vertical system** is designed to meet the unique requirements of a specific business or industry, such as a Web-based retailer, a medical practice, or a video chain.
- **Data:** Data is the raw material that an information system transforms into useful information. An information system can store data in various locations, called tables. By linking the tables, the system can extract specific information.
- **Processes:** Processes describe the tasks and business functions that users, managers, and IT staff members perform to achieve specific results. Processes are the building blocks of an information system because they represent actual day-to-day business operations. To build a successful information system, analysts must understand business processes and document them carefully.
- **People:** People who have an interest in an information system are called **stakeholders**. Stakeholders include the management group responsible for the system, the **users** (sometimes called **end users**) inside and outside the company who will interact with the system, and IT staff members, such as systems analysts, programmers, and network administrators who develop and support the system. Each stakeholder group has a vital interest in the information system, but most experienced IT professionals agree that the success or failure of a system usually depends on whether it meets the needs of its users. For that reason, it is essential to understand user requirements and expectations throughout the development process.

4.3 Types of Information Systems

1. Transaction Processing Systems

A transaction processing system provides a way to collect, process, store, display modify or cancel transactions. Most of these systems allow multiple transactions to take place simultaneously. The data that this system collects is usually stored in databases which can be used to produce reports such as billing, wages, inventory summaries, manufacturing schedules, or check registers.

2. Management Information Systems

A management information system is an information system that uses the data collected by the transaction processing system and uses this data to create reports in a way that managers can use it to make routine business decisions in response to problems. Some of the reports that this information system creates are summary, exception and ad hoc reports. All this is done to increase the efficiency of managerial activity.

3. Decision Support Systems

A decision support system helps make decisions by working and analyzing data that can generate statistical projections and data models. This system gives support rather than replacing a manager's judgment while improving the quality of a manager's decision. A DSS helps solve problems while using external data.

4. Expert Systems and Neutral Networks

An expert system, also known as a knowledge-based system, is a computer system that is designed to analyze data and produce recommendations, diagnosis and decisions that are controlled. A neutral system uses computers to foster the way a human brain may process information, learn and remember that information.

5. Information Systems in Organizations

This information system collects, stores and processes data to give an organization real time useful and accurate information. This information system encompasses data gathering information from the people and machines that collect, process, output and store data. Also in the networks that transmit and receive data and the procedures that govern the way data is handled.

4.4 Evaluating Software

Evaluation as a general endeavor can be characterized by the following features:

- Evaluation is a task, which results in one or more reported outcomes.
- Evaluation is an aid for planning, and therefore the outcome is an evaluation of different possible actions.
- Evaluation is goal oriented. The primary goal ist to check results of actions or interventions, in order to improve the quality of the actions or to choose the best action alternative.
- Evaluation is dependent on the current knowledge of science and the methodological standards.

Any evaluation has pragmatically chosen goals. In the domain of software evaluation, the goal can be characterized by one or more of three simple questions:

- **"Which one is better?"** The evaluation aims to compare alternative software systems, e.g. to choose the best fitting software tool for given application, for a decision among several prototypes, or for comparing several versions of a software system.
- **"How good is it?"** This goal aims at the determination of the degree of desired qualities of a finished system. The evaluation of the system with respect to "Usability-Goals" is one of the application of this goal.
- **"Why is it bad?"** The evaluation aims to determine the weaknesses of a software such that the result generates suggestions for further development. A typical instance of this procedure is a system developing approach using prototypes or a re-engineering of an existing system.

Evaluation Techniques

Evaluation techniques are activities of evaluators which can be precisely defined in behavioral and organizational terms. It is important not to confuse "Evaluation techniques" with "Evaluation models", which usually constitute a combination of evaluation techniques. We classify evaluation techniques into two categories, the descriptive evaluation techniques and the predictive evaluation techniques, both of which should be present in every evaluation:

Descriptive evaluation techniques are used to describe the status and the actual problems of the software in an objective, reliable and valid way. These techniques are user based and can be subdivided into several approaches:

- ✓ **Behavior based evaluation techniques** record user behavior while working with a system which "produces" some kind of data. These procedures include observational techniques and "thinking-aloud" protocols.
- ✓ **Opinion based evaluation methods** aim to elicit the user's (subjective) opinions. Examples are interviews, surveys and questionnaires.
- ✓ **Usability Testing stems** from classical experimental design studies. Nowadays, Usability Testing (as a technical term) is understood to be a combination of behavior and opinion based measures with some amount of experimental control, usually chosen by an expert.

The predictive evaluation techniques have as their main aim to make recommendations for future software development and the prevention of usability errors. These techniques are expert – or at least expertise based, such as Walkthrough or inspection techniques. Even though the expert is the driving power in these methods, users may also participate in some instances.

4.5 Make or Buy Decision

The make-or-buy decision is the action of deciding between manufacturing an item internally (or in-house) or buying it from an external supplier (also known as **outsourcing**). Such decisions are typically taken when a firm that has manufactured a part or product, or else considerably modified it, is having issues with current suppliers, or has reducing capacity or varying demand.

Another way to define make-or-buy decision that is closely related to the first definition is this: a decision to perform one of the activities in the value chain in-house, instead of purchasing externally from a supplier. A value chain is the complete range of tasks – such as design, manufacture, marketing and distribution of a product / service that businesses must get done to take a service or product from conception to their customers.

To come to a make-or-buy decision, it is essential to thoroughly analyze, all of the expenses associated with product development in addition to expenses associated with buying the product. The assessment should include qualitative and quantitative factors. It should also separate relevant expenses from irrelevant ones and consider only the former. The study should also look at the availability of the product and its quality under each of the two situations.

A. Quantitative and Qualitative Analysis

Quantitative aspects are essentially the incremental costs stemming from making or purchasing the component. Factors of this type to look at may incorporate things such as availability of manufacturing facilities, needed resources and manufacturing capacity. This may also incorporate variable and fixed expenses that can be found out either by way of estimation or with certainty. Similarly, quantitative expenses would incorporate the cost of the good under consideration as the price is determined by suppliers offering the product for sale in the marketplace.

Qualitative factors to look at call for more subjective assessment. Examples of such factors include control over component quality, the reliability and reputation of the suppliers, the possibility of modifying the decision in the future, the long-term viewpoint concerning manufacture or purchase of the product, and the impact of the decision on customers and suppliers.

B. Relevant and Irrelevant Expenses

As mentioned earlier, distinguishing between these two kinds of expenses is necessary to come to a make-or-buy decision. Relevant costs for manufacturing the good are all the expenses that could be avoided by not manufacturing the product in addition to the opportunity cost resulting from utilizing production facilities to

manufacture the good as against the next best alternative utilization of the manufacturing facilities. Relevant costs for buying the product are all the expenses relating to purchasing a product from suppliers. Irrelevant costs are the expenses involved irrespective of whether the good is produced internally or bought externally.

C. Factors favoring in-house manufacture

- ✓ Wish to integrate plant operations
- ✓ Need for direct control over manufacturing and/or quality
- ✓ Cost considerations (costs less to make the part)
- ✓ Improved quality control
- ✓ No competent suppliers and/or unreliable suppliers
- ✓ Quantity too little to interest a supplier
- ✓ Design secrecy is necessary to protect proprietary technology
- ✓ Control of transportation, lead time, and warehousing expenses
- ✓ Political, environmental, or social reasons
- ✓ Productive utilization of excess plant capacity to assist with absorbing fixed overhead (utilizing existing idle capacity)
- ✓ Wish to keep up a stable workforce (in times when there are declining sales)
- ✓ Greater guarantee of continual supply

D. Factors favoring purchase from outside

- ✓ Suppliers' specialized know-how and research are more than that of the buyer
- ✓ Lack of expertise
- ✓ Small-volume needs
- ✓ Cost aspects (costs less to purchase the item)
- ✓ Wish to sustain a multiple source policy
- ✓ Item not necessary to the firm's strategy
- ✓ Limited facilities for a manufacture or inadequate capacity
- ✓ Brand preference
- ✓ Inventory and procurement considerations

E. Costs for the make analysis

- ✓ Direct labor expenses
- ✓ Incremental inventory-carrying expenses
- ✓ Incremental capital expenses
- ✓ Incremental purchasing expenses
- ✓ Incremental factory operating expenses
- ✓ Incremental managerial expenses
- ✓ Delivered purchased material expenses
- ✓ Any follow-on expenses resulting from quality and associated problems

F. Cost factors for the buy analysis

- ✓ Transportation expenses
- ✓ Purchase price of the part
- ✓ Incremental purchasing expenses
- ✓ Receiving and inspection expenses
- ✓ Any follow-on expenses associated with service or quality

G. How To Arrive At A Make Or Buy Decision?

Here's one example of a process of how businesses can make a sensible make-or-buy decision. Businesses should first carry out an assessment of quantitative aspects before considering qualitative aspects to finalize their make or buy decisions.

Step 1

Carry out the quantitative analysis by comparing the expenses incurred in each option. The expense of purchasing products is the price paid to suppliers to purchase them. On the contrary, the cost of manufacture includes both variable and fixed expenses. For example, a business requires 10 units of its item in 10 consecutive periods. The company can either buy the units at $100 per unit or expend $1,000 to set up manufacture facilities and $8 to manufacture each unit. As the business expends $10,000 to buy the products and $9,000 to manufacture the same quantity of products, with respect to make-or-buy, the business would do better to manufacture the goods, on the basis of only quantitative factors.

Step 2

Think about all the qualitative factors that may have a bearing on the decision to manufacture the products. This incorporates all pertinent factors that cannot be decreased to numbers such as the quality of the business' production department and its experience. An example for this is that it may be possible that the business has zero experience in manufacturing a specific good and its previous experience in manufacturing other goods cannot be applied.

Step 3

Think about qualitative factors that may have a bearing on the decision to buy the products from external suppliers. Such factors include: the quality of the suppliers' management, its dependability and the quality of its goods. An example for this is that it is probable that the supplier has considerable experience in manufacturing the item being considered and the business may want to develop a long-term relationship with a supplier.

Step 4

Factor the qualitative aspects into the quantitative assessment so as to complete it. An example for this in this case is that: even though it is cheaper for the business to manufacture its products, there are grounds to believe that its goods would be of a lower grade than those it can buy. In addition, as the business desires to forge a long-term relationship with its supplier, it may desire to purchase its goods from that supplier so as to commence the relationship.

Step 5

Arrive at a final make-or-buy decision after considering both quantitative and qualitative factors. This would depend on the particular business and what it is doing so as to create profits. Continuing with the above example, even if it is likely that the business may buy better grade products than those it can manufacture in-house, the quality of its goods/products may not have a bearing on its sales on the basis of its business model and what it is putting on the market. If such is the case, the wish to develop a long-term relationship may or may not be adequate to prevail over the $1,000 savings in expenses; instead it depends on how strong is the business' yearning for the relationship and what it hopes to accomplish by starting it.

5. Design Concepts

The goal of design is to produce a model or representation that exhibits firmness, commodity, and delight. The Roman architecture critic Vitruvius advanced the notion that well-designed buildings were those which exhibited firmness, commodity, and delight. The same might be said of good software. **Firmness:** A program should not have any bugs that inhibit its function. **Commodity:** A program should be suitable for the purposes for which it was intended. **Delight:** The experience of using the program should be a pleasurable one. Here we have the beginnings of a theory of design for software.

5.1 Design within the context of software engineering

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing).

The **architectural design** defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented. The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

The **interface design** describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

The **component-level design** transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design. During design you make decisions that will ultimately affect the success of software construction and, as important, the ease with which software can be maintained.

Importance of Software Design

The importance of software design can be stated with a single word—quality. Design is the place where quality is fostered in software engineering. Design provides you with representations of software that can be assessed for quality. Design is the only way that you can accurately translate stakeholder's requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support activities that follow. Without design, you risk building an unstable system—one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process, when time is short and many dollars have already been spent.

5.2 The Design Process

- ☑ Quality Guidelines
- ☑ Quality Attributes
- ☑ The Evolution of Software Design

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction—a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent

refinement leads to design representations at much lower levels of abstraction. These can still be traced to requirements, but the connection is more subtle.

Software Quality Guidelines and Attributes: McGlaughlin suggests three characteristics that serve as a guide for the evaluation of a good design:

- ✓ The design must implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.
- ✓ The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- ✓ The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

5.2.1 Quality Guidelines

In order to evaluate the quality of a design representation, you and other members of the software team must establish technical criteria for good design. For the time being, consider the following guidelines:

1) A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics and (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
2) A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
3) A design should contain distinct representations of data, architecture, interfaces, and components.
4) A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5) A design should lead to components that exhibit independent functional characteristics.
6) A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7) A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
8) A design should be represented using a notation that effectively communicates its meaning.

These design guidelines are not achieved by chance. They are achieved through the application of fundamental design principles, systematic methodology, and thorough review.

5.2.2 Quality Attributes

Hewlett-Packard developed a set of software quality attributes that has been given the acronym FURPS: functionality, usability, reliability, performance, and supportability. The FURPS quality attributes represent a target for all software design:

- ✓ Functionality is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- ✓ Usability is assessed by considering human factors, overall aesthetics, consistency, and documentation.
- ✓ Reliability is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
- ✓ Performance is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.
- ✓ Supportability combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, maintainability—and in addition, testability,

compatibility, configurability (the ability to organize and control elements of the software configuration, the ease with which a system can be installed, and the ease with which problems can be localized.

Not every software quality attribute is weighted equally as the software design is developed. One application may stress functionality with a special emphasis on security. Another may demand performance with particular emphasis on processing speed. A third might focus on reliability. Regardless of the weighting, it is important to note that these quality attributes must be considered as design commences, not after the design is complete and construction has begun.

5.2.3 The Evolution of Software Design

The evolution of software design is a continuing process that has now spanned almost six decades. Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a top-down manner. Procedural aspects of design definition evolved into a philosophy called structured programming. Later work proposed methods for the translation of data flow or data structure into a design definition. Newer design approaches proposed an object-oriented approach to design derivation. More recent emphasis in software design has been on software architecture and the design patterns that can be used to implement software architectures and lower levels of design abstractions. Growing emphasis on aspect-oriented methods, model-driven development, and test-driven development emphasize techniques for achieving more effective modularity and architectural structure in the designs that are created.

Regardless of the design method that is used, you should apply a set of basic concepts to data, architectural, interface, and component-level design. These concepts are considered in the sections that follow.

5.3 Design Concepts

- ☑ Abstraction
- ☑ Architecture
- ☑ Patterns
- ☑ Separation of Concerns
- ☑ Modularity
- ☑ Functional Independence
- ☑ Refinement
- ☑ Aspects
- ☑ Refactoring
- ☑ Design Classes

A set of fundamental software design concepts has evolved over the history of software engineering. Although the degree of interest in each concept has varied over the years, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied.

5.3.1 Abstraction

Abstraction is the act of representing essential features without including the background details or explanations. In the computer science and software engineering domain, the abstraction principle is used to reduce complexity and allow efficient design and implementation of complex software systems. **The process of picking out (abstracting) common features of objects and procedures.**
Some areas of software design and implementation where the abstraction principle is applied include programming languages (mainly in object-oriented programming languages), specification languages, control abstraction, data abstraction and the architecture of software systems.

5.3.2 Architecture

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system". In its simplest form, architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components. In a broader sense, however, components can be generalized to represent major system elements and their interactions.

Shaw and Garlan describe a set of properties that should be specified as part of an architectural design:

- ✓ **Structural properties.** This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.
- ✓ **Extra-functional properties.** The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
- ✓ **Families of related systems.** The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

5.3.3 Patterns

Brad Appleton defines a design pattern in the following manner: "A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns". Stated in another way, a design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine:

- ✓ Whether the pattern is applicable to the current work,
- ✓ Whether the pattern can be reused (hence, saving design time)
- ✓ Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

5.3.4 Separation of Concerns

Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A concern is a feature or behavior that is specified as part of the requirements model for the software. By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve. Separation of concerns is manifested in other related design concepts: modularity, aspects, functional independence, and refinement.

5.3.5 Modularity

Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called modules that are integrated to satisfy problem requirements. It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.

5.3.6 Functional Independence

Functional independence is achieved by developing modules with “single-minded” function and an “aversion” to excessive interaction with other modules. Stated another way, you should design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure. It is fair to ask why independence is important.

Independence is assessed using two qualitative criteria: cohesion and coupling. Cohesion is an indication of the relative functional strength of a module. Coupling is an indication of the relative interdependence among modules.

5.3.7 Refinement

Refinement is actually a process of elaboration. You begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information. You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts. Abstraction enables you to specify procedure and data internally but suppress the need for “outsiders” to have knowledge of low-level details. Refinement helps you to reveal low-level details as design progresses. Both concepts allow you to create a complete design model as the design evolves.

5.3.8 Aspects

An aspect is a representation of a crosscutting concern. As requirements analysis occurs, a set of “concerns” is uncovered. These concerns “include requirements, use cases, features, data structures, quality-of-service issues, variants, intellectual property boundaries, collaborations, patterns and contracts”. Ideally, a requirements model can be organized in a way that allows you to isolate each concern (requirement) so that it can be considered independently. In practice, however, some of these concerns span the entire system and cannot be easily compartmentalized.

As design begins, requirements are refined into a modular design representation. Consider two requirements, A and B. Requirement A crosscuts requirement B “if a software decomposition [refinement] has been chosen in which B cannot be satisfied without taking A into account”.

5.3.9 Refactoring

An important design activity suggested for many agile methods, refactoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. Fowler defines refactoring in the following manner: “Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.”

When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design.

5.3.10 Design Classes

The requirements model defines a set of analysis classes. Each describes some element of the problem domain, focusing on aspects of the problem that are user visible. The level of abstraction of an analysis class is relatively

high. Five different types of design classes, each representing a different layer of the design architecture, can be developed:

- ✓ User interface classes define all abstractions that are necessary for human computer interaction (HCI). In many cases, HCI occurs within the context of a metaphor (e.g., a checkbook, an order form, a fax machine), and the design classes for the interface may be visual representations of the elements of the metaphor.
- ✓ Business domain classes are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.
- ✓ Process classes implement lower-level business abstractions required to fully manage the business domain classes.
- ✓ Persistent classes represent data stores (e.g., a database) that will persist beyond the execution of the software.
- ✓ System classes implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

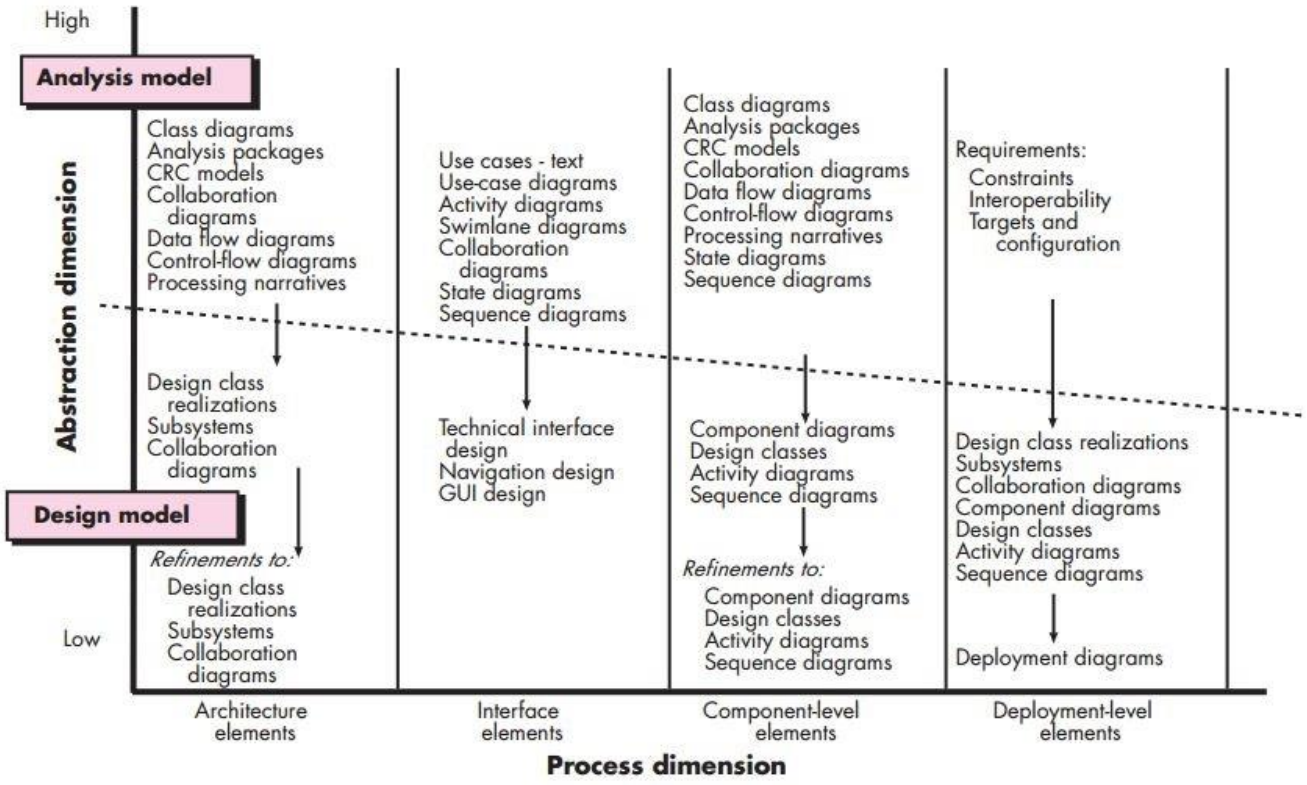
As the architecture forms, the level of abstraction is reduced as each analysis class is transformed into a design representation. That is, analysis classes represent data objects (and associated services that are applied to them) using the jargon of the business domain. Design classes present significantly more technical detail as a guide for implementation.

5.4 The Design Model

- ☑ Data Design Elements
- ☑ Architectural Design Elements
- ☑ Interface Design Elements
- ☑ Component-Level Design Elements
- ☑ Deployment-Level Design Elements

The design model can be viewed in two different dimensions as illustrated in following Figure. The process dimension indicates the evolution of the design model as design tasks are executed as part of the software process. The abstraction dimension represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively. Referring to Figure, the dashed line indicates the boundary between the analysis and design models. In some cases, a clear distinction between the analysis and design models is possible. In other cases, the analysis model slowly blends into the design and a clear distinction is less obvious.

You should note, however, that model elements indicated along the horizontal axis are not always developed in a sequential fashion. In most cases preliminary architectural design sets the stage and is followed by interface design and component-level design, which often occur in parallel. The deployment model is usually delayed until the design has been fully developed.



5.4.1 Data Design Elements

Like other software engineering activities, data design (sometimes referred to as data architecting) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system. In many software applications, the architecture of the data will have a profound influence on the architecture of the software that must process it.

The structure of data has always been an important part of software design. At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications. At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system. At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself. In every case, data design plays an important role.

5.4.2 Architectural Design Elements

The architectural design for software is the equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows that allow movement into and out of the rooms. The floor plan gives us an overall view of the house. Architectural design elements give us an overall view of the software.

The architectural design for software is the equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows that allow movement into and out of the rooms. The floor plan gives us an overall view of the house.

Architectural design elements give us an overall view of the software. The architectural model is derived from three sources:

- ✓ Information about the application domain for the software to be built;
- ✓ Specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand
- ✓ The availability of architectural styles and patterns.

The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model. Each subsystem may have its own architecture (e.g., a graphical user interface might be structured according to a preexisting architectural style for user interfaces).

5.4.3 Interface Design Elements

The interface design elements for software depict information flows into and out of the system and how it is communicated among the components defined as part of the architecture. There are three important elements of interface design:

- ✓ The user interface (UI);
- ✓ External interfaces to other systems, devices, networks, or other producers or consumers of information;
- ✓ Internal interfaces between various designs components. These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

UI design (increasingly called usability design) is a major software engineering action. Usability design incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components). In general, the UI is a unique subsystem within the overall application architecture.

The design of external interfaces requires definitive information about the entity to which information is sent or received. In every case, this information should be collected during requirements engineering and verified once the interface design commences. The design of external interfaces should incorporate error checking and (when necessary) appropriate security features.

The design of internal interfaces is closely aligned with component-level design. Design realizations of analysis classes represent all operations and the messaging schemes required to enable communication and collaboration between operations in various classes. Each message must be designed to accommodate the requisite information transfer and the specific functional requirements of the operation that has been requested. If the classic input-process-output approach to design is chosen, the interface of each software component is designed based on data flow representations and the functionality described in a processing narrative.

5.4.4 Component-Level Design Elements

The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors).

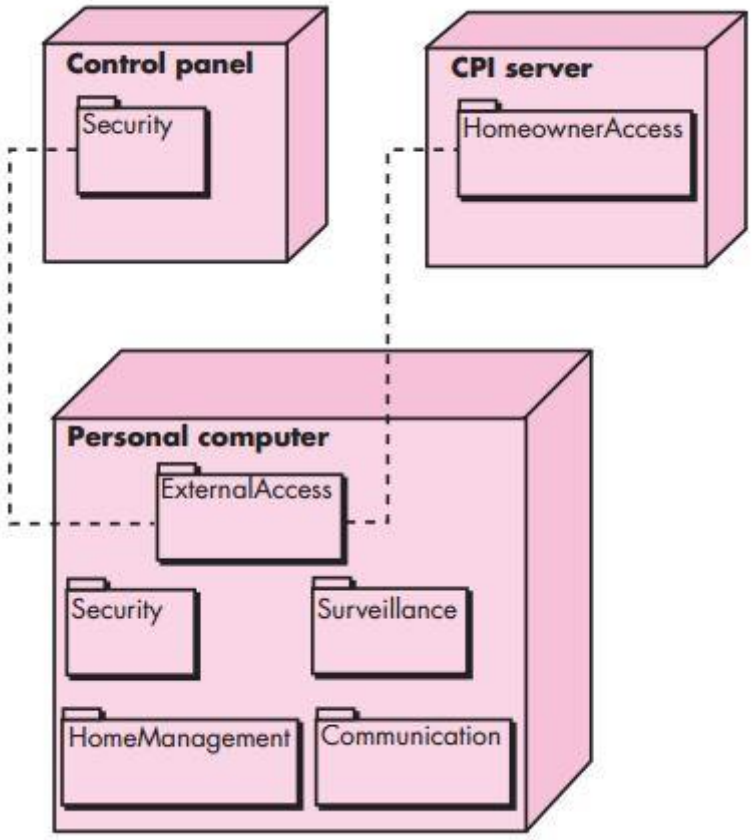
The design details of a component can be modeled at many different levels of abstraction. A UML activity diagram can be used to represent processing logic. Detailed procedural flow for a component can be

represented using either pseudocode or some other diagrammatic form (e.g., flowchart or box diagram). Algorithmic structure follows the rules established for structured programming (i.e., a set of constrained procedural constructs). Data structures, selected based on the nature of the data objects to be processed, are usually modeled using pseudocode or the programming language to be used for implementation.

5.4.5 Deployment-Level Design Elements

Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software. For example, the elements of the *SafeHome* product are configured to operate within three primary computing environments—a home-based PC, the *SafeHome* control panel, and a server housed at CPI Corp. (providing Internet-based access to the system).

During design, a UML deployment diagram is developed and then refined as shown in Figure. In the figure, three computing environments are shown (in actuality, there would be more including sensors, cameras, and others). The subsystems (functionality) housed within each computing element are indicated. For example, the personal computer houses subsystems that implement security, surveillance, home management, and communications features. In addition, an external access subsystem has been designed to manage all attempts to access the *SafeHome* system from an external source. Each subsystem would be elaborated to indicate the components that it implements.



This Page is intentionally left blank.

6. Architectural Design

6.1 Software Architecture

What Is Architecture?

"The architecture of a system is a comprehensive framework that describes its form and structure, its components and how they fit together."

The architecture is not the operational software. Rather, it is a representation that enables you to:

(1) Analyze the effectiveness of the design in meeting its stated requirements,
(2) Consider architectural alternatives at a stage when making design changes is still relatively easy, and
(3) Reduce the risks associated with the construction of the software.

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

This definition emphasizes the role of "software components" in any architectural representation. In the context of architectural design, a software component can be something as simple as a program module or an object-oriented class, but it can also be extended to include databases and "middleware" that enable the configuration of a network of clients and servers. The properties of components are those characteristics that are necessary for an understanding of how the components interact with other components. At the architectural level, internal properties (e.g., details of an algorithm) are not specified. The relationships between components can be as simple as a procedure call from one module to another or as complex as a database access protocol.

Why Is Architecture Important?

There are three key reasons that software architecture is important:

- ✓ Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- ✓ The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- ✓ Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together"

Architectural Descriptions

An architectural description is actually a set of work products that reflect different views of the system. Software architecture description is the set of practices for expressing, communicating and analyzing software architectures (also called architectural rendering), and the result of applying such practices through a work product expressing a software architecture. The IEEE Computer Society has proposed with the following objectives:

- ✓ To establish a conceptual framework and vocabulary for use during the design of software architecture,
- ✓ To provide detailed guidelines for representing an architectural description
- ✓ To encourage sound architectural design practices.

The IEEE standard defines an architectural description (AD) as "a collection of products to document an architecture." The description itself is represented using multiple views, where each view is "a representation of a whole system from the perspective of a related set of [stakeholder] concerns.

Architectural Decisions

Each view developed as part of an architectural description addresses a specific stakeholder concern. To develop each view (and the architectural description as a whole) the system architect considers a variety of alternatives and ultimately decides on the specific architectural features that best meet the concern. Therefore, architectural decisions themselves can be considered to be one view of the architecture. The reasons that decisions were made provide insight into the structure of a system and its conformance to stakeholder concerns.

As a system architect, you can use the template suggested in the sidebar to document each major decision. By doing this, you provide a rationale for your work and establish an historical record that can be useful when design modifications must be made.

6.2 Architectural Styles

An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system. In the case where an existing architecture is to be reengineered, the imposition of an architectural style will result in fundamental changes to the structure of the software including a reassignment of the functionality of components.

An architectural pattern, like an architectural style, imposes a transformation on the design of an architecture. However, a pattern differs from a style in a number of fundamental ways:

- ✓ The scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety;
- ✓ A pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level.
- ✓ Architectural patterns tend to address specific behavioral issues within the context of the architecture (e.g., how real-time applications handle synchronization or interrupts). Patterns can be used in conjunction with an architectural style to shape the overall structure of a system.

Although millions of computer-based systems have been created over the past 60 years, the vast majority can be categorized into one of a relatively small number of architectural styles:

Data-centered architectures. A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a “blackboard” that sends notifications to client software when data of interest to the client changes.

Data-flow architectures. This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern has a set of components, called filters, connected by pipes that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the workings of its neighboring filters.

Call and return architectures. This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of substyles exist within this category:

- ✓ Main program/subprogram architectures. This classic program structure decomposes function into a control hierarchy where a “main” program invokes a number of program components that in turn may invoke still other components.

- ✓ Remote procedure call architectures. The components of a main program/subprogram architecture are distributed across multiple computers on a network.

Object-oriented architectures. The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components are accomplished via message passing.

Layered architectures. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.

What is Architectural Patterns?
Architectural patterns address an application-specific problem within a specific context and under a set of limitations and constraints. The pattern proposes an architectural solution that can serve as the basis for architectural design.

6.3 Golden Rules *(For User Interface Design)*

To build a software with better user interface there are some rules that must be followed. According to Theo Mandel there are three golden rules:

1) Place the user in control.
2) Reduce the user's memory load.
3) Make the interface consistent.

These golden rules actually form the basis for a set of user interface design principles that guide this important aspect of software design. The details of these rules are as follows:

1) Place the User in Control

As a designer, you may be tempted to introduce constraints and limitations to simplify the implementation of the interface. The result may be an interface that is easy to build, but frustrating to use. Mandel defines a number of design principles that allow the user to maintain control:

- ✓ Define interaction modes in a way that does not force a user into unnecessary or undesired actions. (An interaction mode is the current state of the interface.)
- ✓ Provide for flexible interaction. Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multi-touch screen, or voice recognition commands. But every action is not amenable to every interaction mechanism.
- ✓ Allow user interaction to be interruptible and undoable. Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to "undo" any action.
- ✓ Streamline interaction as skill levels advance and allow the interaction to be customized. Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.
- ✓ Hide technical internals from the casual user. The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology. In essence, the interface should never require that the user interact at a level that is "inside" the machine (e.g., a user should never be required to type operating system commands from within application software).

- ✓ Design for direct interaction with objects that appear on the screen. The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing. For example, an application interface that allows a user to "stretch" an object (scale it in size) is an implementation of direct manipulation.

2) Reduce the User's Memory Load

Mandel defines design principles that enable an interface to reduce the user's memory load:

- ✓ Reduce demand on short-term memory. When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions, inputs, and results. This can be accomplished by providing visual cues that enable a user to recognize past actions, rather than having to recall them.
- ✓ Establish meaningful defaults. The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences. However, a "reset" option should be available, enabling the redefinition of original default values.
- ✓ Define shortcuts that are intuitive. When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the print function), the mnemonic should be tied to the action in a way that is easy to remember (e.g., first letter of the task to be invoked).
- ✓ The visual layout of the interface should be based on a real-world metaphor. For example, a bill payment system should use a checkbook and check register metaphor to guide the user through the bill paying process. This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.
- ✓ Disclose information in a progressive fashion. The interface should be organized hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick.

3) Make the Interface Consistent

The interface should present and acquire information in a consistent fashion. This implies that

- ✓ All visual information is organized according to design rules that are maintained throughout all screen displays,
- ✓ Input mechanisms are constrained to a limited set that is used consistently throughout the application,
- ✓ Mechanisms for navigating from task to task are consistently defined and implemented.

Mandel defines a set of design principles that help make the interface consistent:

- ✓ Allow the user to put the current task into a meaningful context. Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand. In addition, the user should be able to determine where he has come from and what alternatives exist for a transition to a new task.
- ✓ Maintain consistency across a family of applications. A set of applications (or products) should all implement the same design rules so that consistency is maintained for all interaction.
- ✓ If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so. Once a particular interactive sequence has become a de facto standard (e.g., the use of alt-S to save a file), the user expects this in every application he encounters. A change (e.g., using alt-S to invoke scaling) will cause confusion.

6.4 User Interface Analysis and Design

The overall process for analyzing and designing a user interface begins with the creation of different models of system function (as perceived from the outside). You begin by delineating the human- and computer-oriented tasks that are required to achieve system function and then considering the design issues that apply to all interface designs. Tools are used to prototype and ultimately implement the design model, and the result is evaluated by end users for quality.

6.4.1 Interface Analysis and Design Models

Four different models come into play when a user interface is to be analyzed and designed. A human engineer (or the software engineer) establishes a **user model**, the software engineer creates a **design model**, the end user develops a mental image that is often called the **user's mental model** or the system perception, and the implementers of the system create an **implementation model**. Unfortunately, each of these models may differ significantly. Your role, as an interface designer, is to reconcile these differences and derive a consistent representation of the interface.

The user model establishes the profile of end users of the system. To build an effective user interface, "all design should begin with an understanding of the intended users, including profiles of their age, gender, physical abilities, education, cultural or ethnic background, motivation, goals and personality".

The user's mental model (system perception) is the image of the system that end users carry in their heads. For example, if the user of a particular word processor were asked to describe its operation, the system perception would guide the response. The accuracy of the description will depend upon the user's profile (e.g., novices would provide a sketchy response at best) and overall familiarity with software in the application domain. A user who understands word processors fully but has worked with the specific word processor only once might actually be able to provide a more complete description of its function than the novice who has spent weeks trying to learn the system.

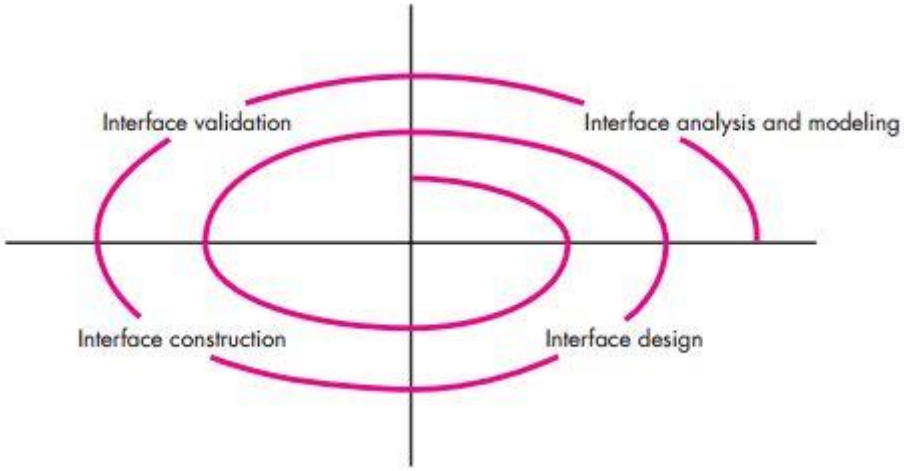
The implementation model combines the outward manifestation of the computer based system (the look and feel of the interface), coupled with all supporting information (books, manuals, videotapes, help files) that describes interface syntax and semantics. When the implementation model and the user's mental model are coincident, users generally feel comfortable with the software and use it effectively. To accomplish this "melding" of the models, the design model must have been developed to accommodate the information contained in the user model, and the implementation model must accurately reflect syntactic and semantic information about the interface.

6.4.2 The Process

The analysis and design process for user interfaces is iterative and can be represented using a spiral model. The user interface analysis and design process begins at the interior of the spiral and encompasses four distinct framework activities:

(1) Interface analysis and modeling,
(2) Interface design,
(3) Interface construction,
(4) Interface validation.

Each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration of requirements and the resultant design. In most cases, the construction activity involves prototyping—the only practical way to validate what has been designed.



Interface analysis focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited.

The goal of interface design is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.

Interface construction normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit may be used to complete the construction of the interface.

Interface validation focuses on:

- ✓ The ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements;
- ✓ The degree to which the interface is easy to use and easy to learn,
- ✓ The users' acceptance of the interface as a useful tool in their work.

6.5 WebApps Interface Design

6.5.1 Interface Design Principles and Guidelines

The user interface of a WebApp is its "first impression." Regardless of the value of its content, the sophistication of its processing capabilities and services, and the overall benefit of the WebApp itself, a poorly designed interface will disappoint the potential user and may cause the user to go elsewhere. In order to design WebApp interfaces that exhibit these characteristics, Tognozzi identifies a set of overriding design principles:

- ✓ **Anticipation.** A WebApp should be designed so that it anticipates the user's next move.
- ✓ **Communication.** The interface should communicate the status of any activity initiated by the user. Communication can be obvious (e.g., a text message) or subtle (e.g., an image of a sheet of paper moving through a printer to indicate that printing is under way).
- ✓ **Consistency.** The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout) should be consistent throughout the WebApp. For example, if underlined blue text implies a navigation link, content should never incorporate blue underlined text that does not imply a link.
- ✓ **Controlled autonomy.** The interface should facilitate user movement throughout the WebApp, but it should do so in a manner that enforces navigation conventions that have been established for the

application. For example, navigation to secure portions of the WebApp should be controlled by userID and password, and there should be no navigation mechanism that enables a user to circumvent these controls.

- ✓ **Efficiency.** The design of the WebApp and its interface should optimize the user's work efficiency, not the efficiency of the developer who designs and builds it or the client server environment that executes it.
- ✓ **Flexibility.** The interface should be flexible enough to enable some users to accomplish tasks directly and others to explore the WebApp in a somewhat random fashion. In every case, it should enable the user to understand where he is and provide the user with functionality that can undo mistakes and retrace poorly chosen navigation paths.
- ✓ **Focus.** The WebApp interface (and the content it presents) should stay focused on the user task(s) at hand. In all hypermedia there is a tendency to route the user to loosely related content. Why? Because it's very easy to do! The problem is that the user can rapidly become lost in many layers of supporting information and lose sight of the original content that she wanted in the first place.
- ✓ **Fitt's law**. "The time to acquire a target is a function of the distance to and size of the target". Based on a study conducted in the 1950s, Fitt's law "is an effective method of modeling rapid, aimed movements, where one appendage (like a hand) starts at rest at a specific start position, and moves to rest within a target area. If a sequence of selections or standardized inputs (with many different options within the sequence) is defined by a user task, the first selection (e.g., mouse pick) should be physically close to the next selection.
- ✓ **Human interface objects.** A vast library of reusable human interface objects has been developed for WebApps. Use them. Any interface object that can be "seen, heard, touched or otherwise perceived" by an end user can be acquired from any one of a number of object libraries.
- ✓ **Latency reduction**. Rather than making the user wait for some internal operation to complete (e.g., downloading a complex graphical image), the WebApp should use multitasking in a way that lets the user proceed with work as if the operation has been completed.
- ✓ **Learnability.** A WebApp interface should be designed to minimize learning time, and once learned, to minimize relearning required when the WebApp is revisited. In general the interface should emphasize a simple, intuitive design that organizes content and functionality into categories that are obvious to the user.
- ✓ **Metaphors.** An interface that uses an interaction metaphor is easier to learn and easier to use, as long as the metaphor is appropriate for the application and the user. A metaphor should call on images and concepts from the user's experience, but it does not need to be an exact reproduction of a real-world experience.
- ✓ **Maintain work product integrity.** A work product (e.g., a form completed by the user, a user-specified list) must be automatically saved so that it will not be lost if an error occurs.
- ✓ **Readability.** All information presented through the interface should be readable by young and old. The interface designer should emphasize readable type styles, font sizes, and color background choices that enhance contrast.
- ✓ **Track state.** When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return later to pick up where she left off. In general, cookies can be designed to store state information. However, cookies are a controversial technology, and other design solutions may be more palatable for some users.
- ✓ **Visible navigation.** A well-designed WebApp interface provides "the illusion that users are in the same place, with the work brought to them". When this approach is used, navigation is not a user concern. Rather, the user retrieves content objects and selects functions that are displayed and executed through the interface.

Nielsen and Wagner suggest a few pragmatic interface design guidelines (based on their redesign of a major WebApp) that provide a nice complement to the principles suggested earlier in this section:

- ✓ Reading speed on a computer monitor is approximately 25 percent slower than reading speed for hardcopy. Therefore, do not force the user to read voluminous amounts of text, particularly when the text explains the operation of the WebApp or assists in navigation.
- ✓ Avoid "under construction" signs—an unnecessary link is sure to disappoint.
- ✓ Users prefer not to scroll. Important information should be placed within the dimensions of a typical browser window.
- ✓ Navigation menus and head bars should be designed consistently and should be available on all pages that are available to the user. The design should not rely on browser functions to assist in navigation.
- ✓ Aesthetics should never supersede functionality. For example, a simple button might be a better navigation option than an aesthetically pleasing, but vague image or icon whose intent is unclear.
- ✓ Navigation options should be obvious, even to the casual user. The user should not have to search the screen to determine how to link to other content or services.

A well-designed interface improves the user's perception of the content or services provided by the site. It need not necessarily be flashy, but it should always be well structured and ergonomically sound.

6.5.2 Interface Design Workflow for WebApps

User interface design begins with the identification of user, task, and environmental requirements. Once user tasks have been identified, user scenarios (use cases) are created and analyzed to define a set of interface objects and actions. Information contained within the requirements model forms the basis for the creation of a screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items. Tools are then used to prototype and ultimately implement the interface design model. The following tasks represent a rudimentary workflow for WebApp interface design:

1) Review information contained in the requirements model and refine as required.
2) Develop a rough sketch of the WebApp interface layout.
3) Map user objectives into specific interface actions. You must answer the following question: "How does the interface enable the user to accomplish each objective?"
4) Define a set of user tasks that are associated with each action. Each interface action (e.g., "buy a product") is associated with a set of user tasks.
5) Storyboard screen images for each interface action. As each action is considered, a sequence of storyboard images (screen images) should be created to depict how the interface responds to user interaction. Content objects should be identified (even if they have not yet been designed and developed), WebApp functionality should be shown, and navigation links should be indicated.
6) Refine interface layout and storyboards using input from aesthetic design. In most cases, you'll be responsible for rough layout and storyboarding, but the aesthetic look and feel for a major commercial site is often developed by artistic, rather than technical, professionals.
7) Identify user interface objects that are required to implement the interface. This task may require a search through an existing object library to find those reusable objects (classes) that are appropriate for the WebApp interface.
8) Develop a procedural representation of the user's interaction with the interface.
9) Develop a behavioral representation of the interface.
10) Describe the interface layout for each state.
11) Refine and review the interface design model. Review of the interface should focus on usability.

7. Software Quality Assurance

- ☑ Background Issues
- ☑ Elements of Software Quality Assurance

Some software developers continue to believe that software quality is something you begin to worry about after code has been generated. Nothing could be further from the truth! Software quality assurance (often called quality management) is an umbrella activity that is applied throughout the software process.

Software quality assurance (SQA) encompasses:

(1) An SQA process,
(2) Specific quality assurance and quality control tasks (including technical reviews and a multitier testing strategy),
(3) Effective software engineering practice (methods and tools),
(4) Control of all software work products and the changes made to them,
(5) A procedure to ensure compliance with software development standards (when applicable),
(6) Measurement and reporting mechanisms.

7.1 Background Issues

Quality control and assurance are essential activities for any business that produces products to be used by others. The first formal quality assurance and control function was introduced at Bell Labs in 1916 and spread rapidly throughout the manufacturing world. During the 1940s, more formal approaches to quality control were suggested. These relied on measurement and continuous process improvement as key elements of quality management.

Today, every company has mechanisms to ensure quality in its products. In fact, explicit statements of a company's concern for quality have become a marketing ploy during the past few decades.

The history of quality assurance in software development parallels the history of quality in hardware manufacturing. During the early days of computing (1950s and 1960s), quality was the sole responsibility of the programmer. Standards for quality assurance for software were introduced in military contract software development during the 1970s and have spread rapidly into software development in the commercial world. Extending the definition presented earlier, software quality assurance is a "planned and systematic pattern of actions" that are required to ensure high quality in software. The scope of quality assurance responsibility might best be characterized by paraphrasing a once-popular automobile commercial: "Quality Is Job #1." The implication for software is that many different constituencies have software quality assurance responsibility software engineers, project managers, customers, salespeople, and the individuals who serve within an SQA group.

7.2 Elements of Software Quality Assurance

Software quality assurance encompasses a broad range of concerns and activities that focus on the management of software quality. These can be summarized in the following manner:

- ✓ **Standards.** The IEEE, ISO, and other standards organizations have produced a broad array of software engineering standards and related documents. Standards may be adopted voluntarily by a software engineering organization or imposed by the customer or other stakeholders. The job of SQA is to ensure that standards that have been adopted are followed and that all work products conform to them.
- ✓ **Reviews and audits.** Technical reviews are a quality control activity performed by software engineers for software engineers. Their intent is to uncover errors. Audits are a type of review performed by SQA

personnel with the intent of ensuring that quality guidelines are being followed for software engineering work. For example, an audit of the review process might be conducted to ensure that reviews are being performed in a manner that will lead to the highest likelihood of uncovering errors.

- ✓ **Testing.** Software testing is a quality control function that has one primary goal—to find errors. The job of SQA is to ensure that testing is properly planned and efficiently conducted so that it has the highest likelihood of achieving its primary goal.
- ✓ **Error/defect collection and analysis**. The only way to improve is to measure how you're doing. SQA collects and analyzes error and defect data to better understand how errors are introduced and what software engineering activities are best suited to eliminating them.
- ✓ **Change management.** Change is one of the most disruptive aspects of any software project. If it is not properly managed, change can lead to confusion, and confusion almost always leads to poor quality. SQA ensures that adequate change management practices have been instituted.
- ✓ **Education.** Every software organization wants to improve its software engineering practices. A key contributor to improvement is education of software engineers, their managers, and other stakeholders. The SQA organization takes the lead in software process improvement and is a key proponent and sponsor of educational programs.
- ✓ **Vendor management.** Three categories of software are acquired from external software vendors—shrink-wrapped packages (e.g., Microsoft Office), a tailored shell that provides a basic skeletal structure that is custom tailored to the needs of a purchaser, and contracted software that is custom designed and constructed from specifications provided by the customer organization. The job of the SQA organization is to ensure that high-quality software results by suggesting specific quality practices that the vendor should follow (when possible), and incorporating quality mandates as part of any contract with an external vendor.
- ✓ **Security management.** With the increase in cybercrime and new government regulations regarding privacy, every software organization should institute policies that protect data at all levels, establish firewall protection for WebApps, and ensure that software has not been tampered with internally. SQA ensures that appropriate process and technology are used to achieve software security.
- ✓ **Safety.** Because software is almost always a pivotal component of human rated systems (e.g., automotive or aircraft applications), the impact of hidden defects can be catastrophic. SQA may be responsible for assessing the impact of software failure and for initiating those steps required to reduce risk.
- ✓ **Risk management.** Although the analysis and mitigation of risk is the concern of software engineers, the SQA organization ensures that risk management activities are properly conducted and that risk-related contingency plans have been established.

8. Software Testing Strategies

- ☑ Strategic Issues
- ☑ Test Strategies for Conventional Software
- ☑ Validation Testing
- ☑ System Testing

Software is tested to uncover errors that were made inadvertently as it was designed and constructed. A strategy for software testing is developed by the project manager, software engineers, and testing specialists. A strategy for software testing provides a road map that describes the steps to be conducted as part of testing, when these steps are planned and then undertaken, and how much effort, time, and resources will be required. Therefore, any testing strategy must incorporate test planning, test case design, test execution, and resultant data collection and evaluation.

A software testing strategy should be flexible enough to promote a customized testing approach. At the same time, it must be rigid enough to encourage reasonable planning and management tracking as the project progresses.

8.1 Strategic Issues

There are some overriding issues, if they are not addressed, then the software testing will fail, even the best one. Those issues are as follows:

- ✓ ***Specify product requirements in a quantifiable manner long before testing commences.*** Although the overriding objective of testing is to find errors, a good testing strategy also assesses other quality characteristics such as portability, maintainability, and usability. These should be specified in a way that is measurable so that testing results are unambiguous.
- ✓ ***State testing objectives explicitly.*** The specific objectives of testing should be stated in measurable terms. For example, test effectiveness, test coverage, meantime-to-failure, the cost to find and fix defects, remaining defect density or frequency of occurrence, and test work-hours should be stated within the test plan.
- ✓ ***Understand the users of the software and develop a profile for each user category.*** Use cases that describe the interaction scenario for each class of user can reduce overall testing effort by focusing testing on actual use of the product.
- ✓ ***Develop a testing plan that emphasizes "rapid cycle testing."*** Gilb recommends that a software team "learn to test in rapid cycles (2 percent of project effort) of customer-useful, at least field 'trialable,' increments of functionality and/or quality improvement." The feedback generated from these rapid cycle tests can be used to control quality levels and the corresponding test strategies.
- ✓ ***Build "robust" software that is designed to test itself.*** Software should be designed in a manner that uses anti-bugging techniques. That is, software should be capable of diagnosing certain classes of errors. In addition, the design should accommodate automated testing and regression testing.
- ✓ ***Use effective technical reviews as a filter prior to testing.*** Technical reviews can be as effective as testing in uncovering errors. For this reason, reviews can reduce the amount of testing effort that is required to produce high quality software.
- ✓ ***Conduct technical reviews to assess the test strategy and test cases themselves.*** Technical reviews can uncover inconsistencies, omissions, and outright errors in the testing approach. This saves time and also improves product quality.
- ✓ ***Develop a continuous improvement approach for the testing process.*** The test strategy should be measured. The metrics collected during testing should be used as part of a statistical process control approach for software testing.

8.2 Test Strategies for Conventional Software

- ☑ Unit Testing
- ☑ Integration Testing
- ☑ Regression Testing
- ☑ Smoke Testing

There are many strategies that can be used to test software. At one extreme, you can wait until the system is fully constructed and then conduct tests on the overall system in hopes of finding errors. This approach, although appealing, simply does not work. It will result in buggy software that disappoints all stakeholders. At the other extreme, you could conduct tests on a daily basis, whenever any part of the system is constructed. This approach, although less appealing to many, can be very effective. Unfortunately, some software developers hesitate to use it.

A testing strategy that is chosen by most software teams falls between the two extremes. It takes an incremental view of testing, beginning with the testing of individual program units, moving to tests designed to facilitate the integration of the units, and culminating with tests that exercise the constructed system. Each of these classes of tests is described in the sections that follow.

8.2.1 Unit Testing

Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The relative complexity of tests and the errors those tests uncover is limited by the constrained scope established for unit testing. The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel for multiple components.

- **Unit-test considerations.**

Data flow across a component interface is tested before any other testing is initiated. If data do not enter and exit properly, all other tests are moot. In addition, local data structures should be exercised and the local impact on global data should be ascertained (if possible) during unit testing.

Selective testing of execution paths is an essential task during the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow.

Boundary testing is one of the most important unit testing tasks. Software often fails at its boundaries. That is, errors often occur when the nth element of an n-dimensional array is processed, when the *ith* repetition of a loop with *i* passes is invoked, when the maximum or minimum allowable value is encountered. Test cases that exercise data structure, control flow, and data values just below, at, and just above maxima and minima are very likely to uncover errors.

Among the potential errors that should be tested when error handling is evaluated are:

(1) Error description is unintelligible,
(2) Error noted does not correspond to error encountered,
(3) Error condition causes system intervention prior to error handling,
(4) Exception-condition processing is incorrect,
(5) Error description does not provide enough information to assist in the location of the cause of the error.

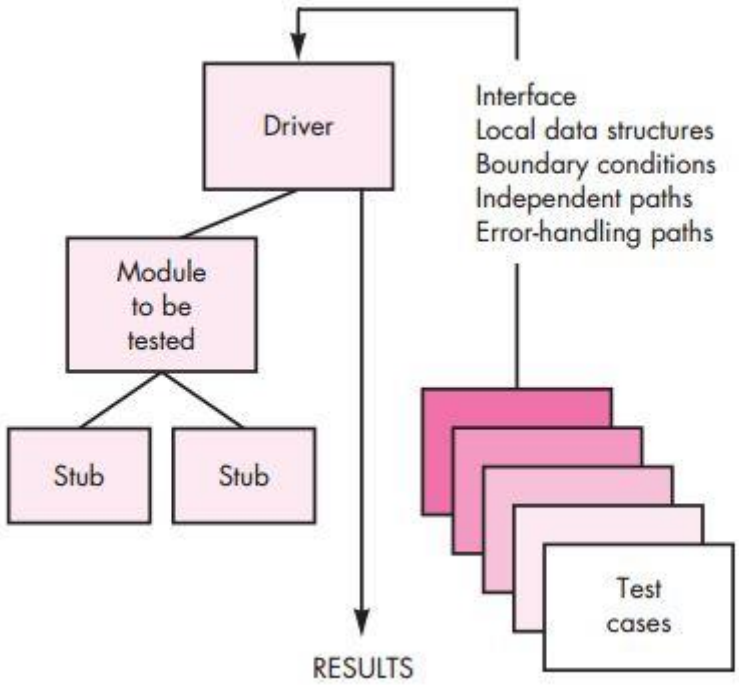
- **Unit-test procedures.**

Unit testing is normally considered as an adjunct to the coding step. The design of unit tests can occur before coding begins or after source code has been generated. A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the categories discussed earlier. Each test case should be coupled with a set of expected results.

Because a component is not a stand-alone program, driver and/or stub software must often be developed for each unit test. In most applications a driver is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results. Stubs serve to replace modules that are subordinate (invoked by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

Drivers and stubs represent testing "overhead." That is, both are software that must be written (formal design is not commonly applied) but that is not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with "simple" overhead software. In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used).

Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.



8.2.2 Integration Testing

Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit-tested components and build a program structure that has been dictated by design.

There is often a tendency to attempt non-incremental integration; that is, to construct the program using a "big bang" approach. All components are combined in advance. The entire program is tested as a whole. And chaos

usually results! A set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.

Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied. In the paragraphs that follow, a number of different incremental integration strategies are discussed.

- **Top-down integration**

Top-down integration testing is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner. The integration process is performed in a series of five steps:

(1) The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
(2) Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
(3) Tests are conducted as each component is integrated.
(4) On completion of each set of tests, another stub is replaced with the real component.
(5) Regression testing (discussed later in this section) may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built. The top-down integration strategy verifies major control or decision points early in the test process. In a "well-factored" program structure, decision making occurs at upper levels in the hierarchy and is therefore encountered first. If major control problems do exist, early recognition is essential. If depth-first integration is selected, a complete function of the software may be implemented and demonstrated. Early demonstration of functional capability is a confidence builder for all stakeholders.

Top-down strategy sounds relatively uncomplicated, but in practice, logistical problems can arise. The most common of these problems occurs when processing at low levels in the hierarchy is required to adequately test upper levels. Stubs replace low-level modules at the beginning of top-down testing; therefore, no significant data can flow upward in the program structure. As a tester, you are left with three choices:

(1) Delay many tests until stubs are replaced with actual modules,
(2) Develop stubs that perform limited functions that simulate the actual module,
(3) Integrate the software from the bottom of the hierarchy upward.

The first approach (delay tests until stubs are replaced by actual modules) can cause you to lose some control over correspondence between specific tests and incorporation of specific modules. The second approach is workable but can lead to significant overhead, as stubs become more and more complex. The third approach, called bottom-up integration is discussed in the paragraphs that follow.

- **Bottom-up integration**

Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure). Because components are integrated from the

bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated. A bottom-up integration strategy may be implemented with the following steps:

(1) Low-level components are combined into clusters (sometimes called builds) that perform a specific software sub-function.
(2) A driver (a control program for testing) is written to coordinate test case input and output.
(3) The cluster is tested.
(4) Drivers are removed and clusters are combined moving upward in the program structure.

8.2.3 Regression testing

Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

In a broader context, successful tests (of any kind) result in the discovery of errors, and errors must be corrected. Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed. Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

- ✓ A representative sample of tests that will exercise all software functions.
- ✓ Additional tests that focus on software functions that are likely to be affected by the change.
- ✓ Tests that focus on the software components that have been changed.

As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions. It is impractical and inefficient to re-execute every test for every program function once a change has occurred.

8.2.4 Smoke testing

Smoke testing is an integration testing approach that is commonly used when product software is developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis. In essence, the smoke-testing approach encompasses the following activities:

(1) Software components that have been translated into code are integrated into a build. A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
(2) A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover “showstopper” errors that have the highest likelihood of throwing the software project behind schedule.
(3) The build is integrated with other builds, and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

The daily frequency of testing the entire product may surprise some readers. However, frequent tests give both managers and practitioners a realistic assessment of integration testing progress. Smoke testing provides a number of benefits when it is applied on complex, time critical software projects:

- ✓ Integration risk is minimized. Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early, thereby reducing the likelihood of serious schedule impact when errors are uncovered.
- ✓ The quality of the end product is improved. Because the approach is construction (integration) oriented, smoke testing is likely to uncover functional errors as well as architectural and component-level design errors. If these errors are corrected early, better product quality will result.
- ✓ Error diagnosis and correction are simplified. Like all integration testing approaches, errors uncovered during smoke testing are likely to be associated with "new software increments"—that is, the software that has just been added to the build(s) is a probable cause of a newly discovered error.
- ✓ Progress is easier to assess. With each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is being made.

8.3 Validation Testing

Like all other testing steps, validation tries to uncover errors, but the focus is at the requirements level, on things that will be immediately apparent to the end user. Testing focuses on user-visible actions and user-recognizable output from the system.

➢ Validation-Test Criteria

Software validation is achieved through a series of tests that demonstrate conformity with requirements. A test plan outlines the classes of tests to be conducted, and a test procedure defines specific test cases that are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all content is accurate and properly presented, all performance requirements are attained, documentation is correct, and usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).

After each validation test case has been conducted, one of two possible conditions exists:

(1) The function or performance characteristic conforms to specification and is accepted

(2) A deviation from specification is uncovered and a deficiency list is created.

Deviations or errors discovered at this stage in a project can rarely be corrected prior to scheduled delivery. It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

➢ Configuration Review

An important element of the validation process is a configuration review. The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support activities. The configuration review, sometimes called an **audit**.

➢ Alpha and Beta Testing

If software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one. Most software product builders use a process called alpha and beta testing to uncover errors that only the end user seems able to find.

The **alpha test** is conducted at the developer's site by a representative group of end users. The software is used in a natural setting with the developer "looking over the shoulder" of the users and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The **beta test** is conducted at one or more end-user sites. Unlike alpha testing, the developer generally is not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, you make modifications and then prepare for release of the software product to the entire customer base.

A variation on beta testing, called **customer acceptance testing**, is sometimes performed when custom software is delivered to a customer under contract. The customer performs a series of specific tests in an attempt to uncover errors before accepting the software from the developer. In some cases (e.g., a major corporate or governmental system) acceptance testing can be very formal and encompass many days or even weeks of testing.

8.4 System Testing

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.

A classic system-testing problem is "finger pointing." This occurs when an error is uncovered, and the developers of different system elements blame each other for the problem. Rather than indulging in such nonsense, you should anticipate potential interfacing problems and:

(1) Design error-handling paths that test all information coming from other elements of the system,
(2) Conduct a series of tests that simulate bad data or other potential errors at the software interface,
(3) Record the results of tests to use as "evidence" if finger pointing does occur, and
(4) Participate in planning and design of system tests to ensure that software is adequately tested.

In the sections that follow, types of system tests that are worthwhile for software-based systems.

8.4.1 Recovery Testing

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), re-initialization, check-pointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

8.4.2 Security Testing

Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. To quote Beizer: "The system's security must, of course, be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack."

During security testing, the tester plays the role(s) of the individual who desires to penetrate the system. Anything goes! The tester may attempt to acquire passwords through external clerical means; may attack the system with custom software designed to break down any defenses that have been constructed; may overwhelm the system, thereby denying service to others; may purposely cause system errors, hoping to penetrate during recovery; may browse through insecure data, hoping to find the key to system entry.

8.4.3 Stress Testing

Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example:

(1) Special tests may be designed that generate ten interrupts per second, when one or two is the average rate,
(2) Input data rates may be increased by an order of magnitude to determine how input functions will respond,
(3) Test cases that require maximum memory or other resources are executed,
(4) Test cases that may cause thrashing in a virtual operating system are designed,
(5) Test cases that may cause excessive hunting for disk-resident data are created.

A variation of stress testing is a technique called **sensitivity testing**. In some situations (the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation. Sensitivity testing

attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

8.4.4 Performance Testing

For real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable. Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as tests are conducted. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.

Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion. External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis. By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.

8.4.5 Deployment Testing

In many cases, software must execute on a variety of platforms and under more than one operating system environment. Deployment testing, sometimes called **configuration testing**, exercises the software in each environment in which it is to operate. In addition, deployment testing examines all installation procedures and specialized installation software (e.g., “installers”) that will be used by customers, and all documentation that will be used to introduce the software to end users.

9. Testing Conventional Applications

- ☑ Internal and External View of Testing
- ☑ White Box Testing
- ☑ Black Box Testing

9.1 Internal and External View of Testing

Any engineered product (and most other things) can be tested in one of two ways:

(1). Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function.

(2). Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised.

The first test approach takes an external view and is called **black-box testing**. The second requires an internal view and is termed **white-box testing**.

9.2 White-Box Testing

White box testing is a testing technique that examines the program structure and derives test data from the program logic/code. The other names of glass box testing are clear box testing, open box testing, logic driven testing or path driven testing or structural testing.

Advantages of White Box Testing:

- ✓ Forces test developer to reason carefully about implementation.
- ✓ Reveals errors in "hidden" code.
- ✓ Spots the Dead Code or other issues with respect to best programming practices.

Disadvantages of White Box Testing:

- ✓ Expensive as one has to spend both time and money to perform white box testing.
- ✓ Every possibility that few lines of code are missed accidentally.
- ✓ In-depth knowledge about the programming language is necessary to perform white box testing.

White Box Testing Techniques:

A. Path Coverage: This technique corresponds to testing all possible paths which means that each statement and branch is covered.

B. Statement Coverage: This technique is aimed at exercising all programming statements with minimal tests.

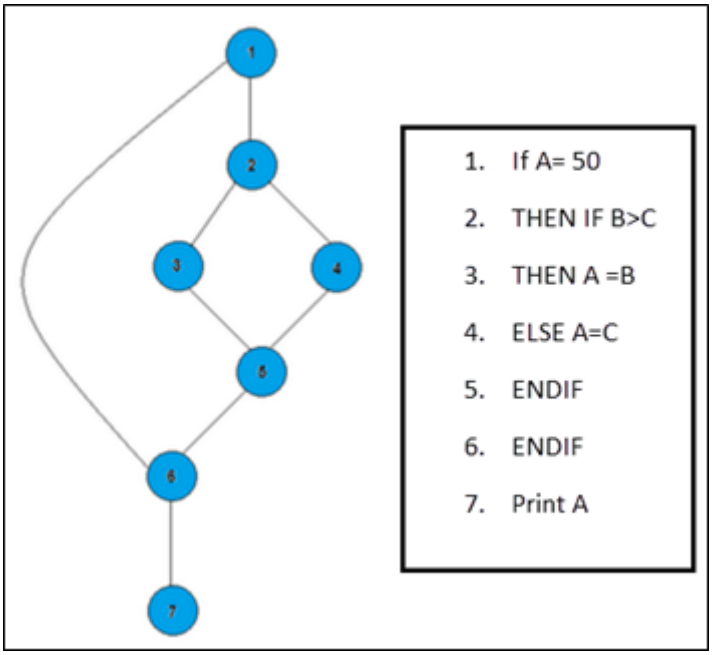
C. Branch Coverage: This technique is running a series of tests to ensure that all branches are tested at least once.

Their short details are as follows:

A. Basis Path Testing (Path Coverage)

The basis path testing is same, but it is based on a White Box Testing method, that defines test cases based on the flows or logical path that can be taken through the program. Basis path testing involves execution of all possible blocks in a program and achieves maximum path coverage with least number of test cases. It is a hybrid of branch testing and path testing methods.

The objective behind basis path testing is that it defines the number of independent paths, thus the number of test cases needed can be defined explicitly (maximizes the coverage of each test case). Here we will take a simple example, to get better idea what is basis path testing include.



In the above example, we can see there are few conditional statements that is executed depending on what condition it suffice. Here there are 3 path or condition that need to be tested to get the output,

- Path 1: 1,2,3,5,6, 7
- Path 2: 1,2,4,5,6, 7
- Path 3: 1, 6, 7

Benefits of basis path testing

- ✓ It helps to reduce the redundant tests
- ✓ It focuses attention on program logic
- ✓ It helps facilitates analytical versus arbitrary case design
- ✓ Test cases which exercise basis set will execute every statement in program at least once

B. Control Structure Testing

Structural testing, also known as glass box testing or white box testing is an approach where the tests are derived from the knowledge of the software's structure or internal implementation. The other names of structural testing includes clear box testing, open box testing, logic driven testing or path driven testing.

Advantages of Structural Testing:

- ✓ Forces test developer to reason carefully about implementation
- ✓ Reveals errors in "hidden" code
- ✓ Spots the Dead Code or other issues with respect to best programming practices.

Disadvantages of Structural Box Testing:

- ✓ Expensive as one has to spend both time and money to perform white box testing.
- ✓ Every possibility that few lines of code is missed accidentally.
- ✓ In depth knowledge about the programming language is necessary to perform white box testing.

9.3 Black-Box Testing

Black box testing is a software testing techniques in which functionality of the software under test (SUT) is tested without looking at the internal code structure, implementation details and knowledge of internal paths of the software. This type of testing is based entirely on the software requirements and specifications.

In Black-Box Testing we just focus on inputs and output of the software system without bothering about internal knowledge of the software program. This method is named so because the software program, in the eyes of the tester, is like a black box; inside which one cannot see. This method attempts to find errors in the following categories:

- ✓ Incorrect or missing functions
- ✓ Interface errors
- ✓ Errors in data structures or external database access
- ✓ Behavior or performance errors
- ✓ Initialization and termination errors

For example: A tester, without knowledge of the internal structures of a website, tests the web pages by using a browser; providing inputs (clicks, keystrokes) and verifying the outputs against the expected outcome.

Black Box Testing - Steps

Here are the generic steps followed to carry out any type of Black Box Testing.

- ✓ Initially requirements and specifications of the system are examined.
- ✓ Tester chooses valid inputs (positive test scenario) to check whether SUT processes them correctly. Also some invalid inputs (negative test scenario) are chosen to verify that the SUT is able to detect them.
- ✓ Tester determines expected outputs for all those inputs.
- ✓ Software tester constructs test cases with the selected inputs.
- ✓ The test cases are executed.
- ✓ Software tester compares the actual outputs with the expected outputs.
- ✓ Defects if any are fixed and re-tested.

Black-Box Techniques

Following are some techniques that can be used for designing black box tests.

- ✓ **Equivalence Partitioning**: It is a software test design technique that involves dividing input values into valid and invalid partitions and selecting representative values from each partition as test data.
- ✓ **Boundary Value Analysis**: It is a software test design technique that involves the determination of boundaries for input values and selecting values that are at the boundaries and just inside/ outside of the boundaries as test data.
- ✓ **Cause-Effect Graphing**: It is a software test design technique that involves identifying the cases (input conditions) and effects (output conditions), producing a Cause-Effect Graph, and generating test cases accordingly.

Advantages of Black-Box Testing

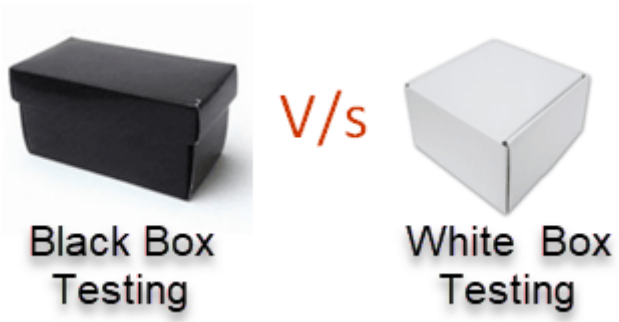
- ✓ Tests are done from a user's point of view and will help in exposing discrepancies in the specifications.
- ✓ Tester need not know programming languages or how the software has been implemented.
- ✓ Tests can be conducted by a body independent from the developers, allowing for an objective perspective and the avoidance of developer-bias.
- ✓ Test cases can be designed as soon as the specifications are complete.

Disadvantages of Black-Box Testing

- ✓ Only a small number of possible inputs can be tested and many program paths will be left untested.
- ✓ Without clear specifications, which is the situation in many projects, test cases will be difficult to design.
- ✓ Tests can be redundant if the software designer/developer has already run a test case.
- ✓ Ever wondered why a soothsayer closes the eyes when foretelling events? So is almost the case in Black Box Testing.

Black-Box Vs White-Box Testing

Black-Box Testing	White-Box Testing
It is a testing approach which is used to test the software without the knowledge of the internal structure of program or application.	It is a testing approach in which internal structure is known to the tester.
It also knowns as data-driven, box testing, data-, and functional testing.	It is also called structural testing, clear box testing, code-based testing, or glass box testing.
Testing is based on external expectations; internal behavior of the application is unknown.	Internal working is known, and the tester can test accordingly.
This type of testing is ideal for higher levels of testing like System Testing, Acceptance testing.	Testing is best suited for a lower level of testing like Unit Testing, Integration testing.
Programming knowledge is not needed to perform Black Box testing.	Programming knowledge is required to perform White Box testing.
Implementation knowledge is not requiring doing Black Box testing.	Complete understanding needs to implement White-Box testing.
Test and programmer are dependent on each other, so it is tough to automate.	White Box testing is easy to automate.
The main objective of this testing is to check what functionality of the system under test.	The main objective of White Box testing is done to check the quality of the code.
Testing can start after preparing requirement specification document.	Testing can start after preparing for Detail design document.
Performed by the end user, developer, and tester.	Usually done by tester and developers.
It is less exhaustive and time-consuming.	Exhaustive and time-consuming method.
Not the best method for algorithm testing.	Best suited for algorithm testing.
Code access is not required for Black Box Testing.	White box testing requires code access. Thereby, the code could be stolen if testing is outsourced.
Low skilled testers can test the application with no knowledge of the implementation of programming language or operating system.	Need an expert tester with vast experience to perform white box testing.



10. Gantt Chart

Gantt chart is a type of a bar chart that is used for illustrating project schedules. Gantt charts can be used in any projects that involve effort, resources, milestones and deliveries.

At present, Gantt charts have become the popular choice of project managers in every field. Gantt charts allow project managers to track the progress of the entire project. Through Gantt charts, the project manager can keep a track of the individual tasks as well as of the overall project progression.

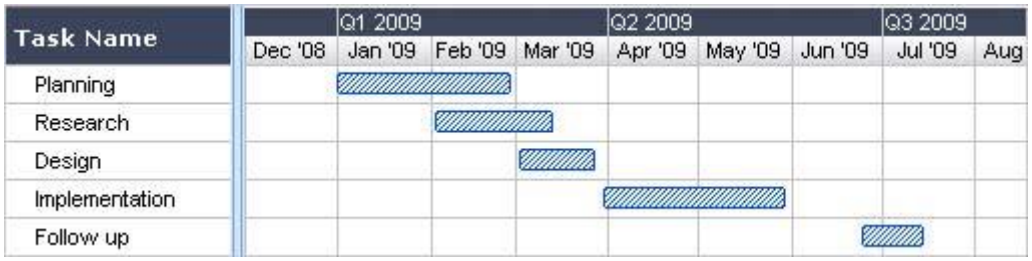
In addition to tracking the progression of the tasks, Gantt charts can also be used for tracking the utilization of the resources in the project. These resources can be human resources as well as materials used.

Gantt chart was invented by a mechanical engineer named Henry Gantt in 1910. Since the invention, Gantt chart has come a long way. By today, it takes different forms from simple paper based charts to sophisticated software packages.

A Gantt chart, commonly used in project management, is one of the most popular and useful ways of showing activities (tasks or events) displayed against time. On the left of the chart is a list of the activities and along the top is a suitable time scale. Each activity is represented by a bar; the position and length of the bar reflects the start date, duration and end date of the activity. This allows you to see at a glance:

- ✓ What the various activities are
- ✓ When each activity begins and ends
- ✓ How long each activity is scheduled to last
- ✓ Where activities overlap with other activities, and by how much
- ✓ The start and end date of the whole project

Example:



Advantages & Disadvantages

The ability to grasp the overall status of a project and its tasks at once is the key advantage in using a Gantt chart tool. Therefore, upper management or the sponsors of the project can make informed decisions just by looking at the Gantt chart tool.

The software-based Gantt charts are able to show the task dependencies in a project schedule. This helps to identify and maintain the critical path of a project schedule.

Gantt chart tools can be used as the single entity for managing small projects. For small projects, no other documentation may be required; but for large projects, the Gantt chart tool should be supported by other means of documentation.

For large projects, the information displayed in Gantt charts may not be sufficient for decision making.

Although Gantt charts accurately represent the cost, time and scope aspects of a project, it does not elaborate on the project size or size of the work elements. Therefore, the magnitude of constraints and issues can be easily misunderstood.

Conclusion

Gantt chart tools make project manager's life easy. Therefore, Gantt chart tools are important for successful project execution. Identifying the level of detail required in the project schedule is the key when selecting a suitable Gantt chart tool for the project. One should not overly complicate the project schedules by using Gantt charts to manage the simplest tasks.

11. Risk Management

- ☑ Proactive vs Reactive Risk Strategies
- ☑ Software Risks

Before reading about the difference between proactive and reactive risk management, let us first look at what risk management is all about. Errors are common in any work environment. Such errors may occur due to human mistakes, unexpected accidents, natural disasters and third party decisions which affect the organization. Such errors can be either avoidable or unavoidable. The plan of minimizing such errors and mitigating its effects during an incident is known as risk management. This involves the identification, assessment and prioritization of risks. The objective of risk management is to deflect the effects of uncertainty in business.

11.1 Proactive vs Reactive Risk Strategies

Reactive risk management is often compared to a firefighting scenario. The reactive risk management kicks into action once an accident happens, or problems are identified after the audit. The accident is investigated, and measures are taken to avoid similar events happening in the future. Further, measures will be taken to reduce the negative impact the incident could cause on business profitability and sustainability.

Contrary to reactive risk management, proactive risk management seeks to identify all relevant risks earlier, before an incident occurs. The present organization has to deal with an era of rapid environmental change that is caused by technological advancements, deregulation, fierce competition, and increasing public concern. So, a risk management which relies on past incidents is not a good choice for any organization. Therefore, new thinking in risk management was necessary, which paved the way for proactive risk management.

Reactive Risk Strategy	Proactive Risk Strategy
Actions in response to hazard/risk occurrence	Actions that address perceived hazard/risk occurrence before it actually occurs
After hazard/risk occurrence, taking measures (i.e., corrective actions) to prevent re-occurrence. Management does this by processing incident/accident reports and	Before an identified hazard occurs, management creates control measures to prevent initial occurrence. Identifying these hazards usually happens through proactive activities, or by reviewing proactive reports.
Once hazard occurs, employees take action to prevent an accident. If risk occurrence is inevitable, employees take actions to mitigate damages. These issues must be reported.	Hazard mechanisms and threats are identified before hazard occurrence (and hazard occurrence is mitigated). These issues are generally "voluntary" reports, but it's a best practice to encourage employees to report these issues.
Examples: MOR, Incident report, Accident report	Examples: Audits/inspections, Voluntary reporting Surveys

11.2 Software Risks

Risk can be defined as the probability of an event, hazard, accident, threat or situation occurring and its undesirable consequences. It is a factor that could result in negative consequences and usually expressed as the product of impact and likelihood.

Project risks threaten the project plan. That is, if project risks become real, it is likely that the project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, stakeholder, and requirements problems and their impact on a software project.

Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems. In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors. Technical risks occur because the problem is harder to solve than you thought it would be.

Known risks are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment).

Predictable risks are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced).

Unpredictable risks are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

Business risks threaten the viability of the software to be built and often jeopardize the project or the product. Candidates for the top five business risks are:

(1). Building an excellent product or system that no one really wants (market risk),
(2). Building a product that no longer fits into the overall business strategy for the company (strategic risk),
(3). Building a product that the sales force doesn't understand how to sell (sales risk),
(4). Losing the support of senior management due to a change in focus or a change in people (management risk),
(5). Losing budgetary or personnel commitment (budget risks).

12. Maintenance and Reengineering

- ☑ Software Maintenance
- ☑ Software Reengineering

12.1 Software Maintenance

Software maintenance is widely accepted part of SDLC now a days. It stands for all the modifications and updations done after the delivery of software product. There are number of reasons, why modifications are required, some of them are briefly mentioned below:

- ✓ **Market Conditions** - Policies, which changes over the time, such as taxation and newly introduced constraints like, how to maintain bookkeeping, may trigger need for modification.
- ✓ **Client Requirements** - Over the time, customer may ask for new features or functions in the software.
- ✓ Host Modifications - If any of the hardware and/or platform (such as operating system) of the target host changes, software changes are needed to keep adaptability.
- ✓ **Organization Changes** - If there is any business level change at client end, such as reduction of organization strength, acquiring another company, organization venturing into new business, need to modify in the original software may arise.

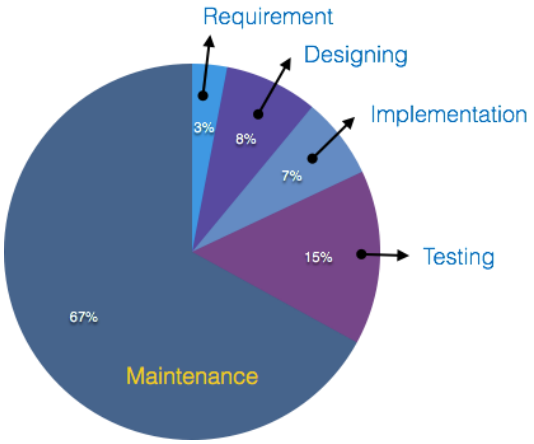
Types of maintenance

In a software lifetime, type of maintenance may vary based on its nature. It may be just a routine maintenance tasks as some bug discovered by some user or it may be a large event in itself based on maintenance size or nature. Following are some types of maintenance based on their characteristics:

- ✓ **Corrective Maintenance** - This includes modifications and updations done in order to correct or fix problems, which are either discovered by user or concluded by user error reports.
- ✓ **Adaptive Maintenance** - This includes modifications and updations applied to keep the software product up-to date and tuned to the ever changing world of technology and business environment.
- ✓ **Perfective Maintenance** - This includes modifications and updates done in order to keep the software usable over long period of time. It includes new features, new user requirements for refining the software and improve its reliability and performance.
- ✓ **Preventive Maintenance** - This includes modifications and updations to prevent future problems of the software. It aims to attend problems, which are not significant at this moment but may cause serious issues in future.

Cost of Maintenance

Reports suggest that the cost of maintenance is high. A study on estimating software maintenance found that the cost of maintenance is as high as 67% of the cost of entire software process cycle.



On an average, the cost of software maintenance is more than 50% of all SDLC phases. There are various factors, which trigger maintenance cost go high, such as:

Real-world factors affecting Maintenance Cost

- ✓ The standard age of any software is considered up to 10 to 15 years.
- ✓ Older softwares, which were meant to work on slow machines with less memory and storage capacity cannot keep themselves challenging against newly coming enhanced softwares on modern hardware.
- ✓ As technology advances, it becomes costly to maintain old software.
- ✓ Most maintenance engineers are newbie and use trial and error method to rectify problem.

- ✓ Often, changes made can easily hurt the original structure of the software, making it hard for any subsequent changes.
- ✓ Changes are often left undocumented which may cause more conflicts in future.

Software-end factors affecting Maintenance Cost

- ✓ Structure of Software Program
- ✓ Programming Language
- ✓ Dependence on external environment
- ✓ Staff reliability and availability

12.2 Software Reengineering

A software product reengineering is a software upgrading procedure or its migration to a more advanced technology platform. At the same time, its current functionality is either saved or undergoes slight modification. Typically, software reengineering process includes one or more procedures from the following list:

- ✓ Source code translation into another programming language;
- ✓ Database reorganization or its transfer;
- ✓ Software architecture optimization;
- ✓ New functionality addition and integration with third-party APIs.

When there is a need for software reengineering process

The modernization process will be an appropriate option in the following cases:

- ☞ **When the programming language or platform is no longer supported.** Improvements, patches for bug fixing and security updates no longer work, which makes the app vulnerable. Also, there are lost the options for integration with third-party systems through modern APIs;
- ☞ **There is a drastic change in technology.** The situation when the initially promising technology is replaced by more successful and advanced alternatives is common in IT. The market is constantly evolving and, if the company wants to keep up with technology, reengineering process becomes a necessity;
- ☞ **Business processes in the company are changing**. If the product was initially confined to the ideal solution of a limited number of tasks in clearly defined processes' conditions, it may be necessary to make changes to the software, provided most of them are to be changed;
- ☞ **If the software is initially poor**. This sometimes happens because of an attempt of excessive saving at development or opting for amateur performers. If the solution used limits the company's performance and does not work well enough, it influences business processes and reengineering might help to achieve a higher-quality product;
- ☞ **A technology appears which is perfect for your goals**. For example, a completely new market can appear, as happened when the first iPhone came out. This does not happen often nowadays, but still, it may well do.

If you fall under any of the above criteria, this is an occasion to consider the need for reengineering process to keep up with competitors and keep developing your product.

Benefits of Software Reengineering Process for Business

At a certain stage, the organization is faced with the choice of creating a new system from scratch or upgrading an existing one. In most cases, it is software reengineering process that will be the right choice, as it provides a number of significant advantages:

- ☞ **Productivity increase**. By optimizing the code and database the speed of work is increased;

- **Improvement opportunity**. You can not only refine the existing product, but also expand its capabilities by adding new features;
- **Risk reduction**. Development from scratch is always a more risky exercise, as opposed to a phased upgrade of the existing system;
- **Time saving**. Instead of starting development from scratch, the existing solution is simply transferred to a new platform, saving all business-logic;
- **Optimization potential**. You can refine the system functionality and increase its flexibility, ensuring better compliance with the enterprise's current objectives;
- **Processes continuity**. The old product can be used while testing the new system until all work is completed.

Thus, we have an optimal solution since we have to transfer the software to a new platform/technology, while ensuring the continuous operation of processes of the enterprise.

12.2.1 Software Reengineering Activities

Inventory analysis. Every software organization should have an inventory of all applications. The inventory can be nothing more than a spreadsheet model containing information that provides a detailed description (e.g., size, age, business criticality) of every active application. By sorting this information according to business criticality, longevity, current maintainability and supportability, and other locally important criteria, candidates for reengineering appear. Resources can then be allocated to candidate applications for reengineering work.

Document restructuring. Weak documentation is the trademark of many legacy systems. But what can you do about it? What are your options?

1. Creating documentation is far too time consuming. If the system works, you may choose to live with what you have. In some cases, this is the correct approach. It is not possible to re-create documentation for hundreds of computer programs. If a program is relatively static, is coming to the end of its useful life, and is unlikely to undergo significant change, let it be!
2. Documentation must be updated, but your organization has limited resources. You'll use a "document when touched" approach. It may not be necessary to fully re-document an application. Rather, those portions of the system that are currently undergoing change are fully documented. Over time, a collection of useful and relevant documentation will evolve.
3. The system is business critical and must be fully re-documented. Even in this case, an intelligent approach is to pare documentation to an essential minimum.

Each of these options is viable. Your software organization must choose the one that is most appropriate for each case.

Reverse engineering. The term reverse engineering has its origins in the hardware world. A company disassembles a competitive hardware product in an effort to understand its competitor's design and manufacturing "secrets." These secrets could be easily understood if the competitor's design and manufacturing specifications were obtained. But these documents are proprietary and unavailable to the company doing the reverse engineering. In essence, successful reverse engineering derives one or more design and manufacturing specifications for a product by examining actual specimens of the product.

Reverse engineering for software is quite similar. In most cases, however, the program to be reverse engineered is not a competitor's. Rather, it is the company's own work (often done many years earlier). The "secrets" to be understood are obscure because no specification was ever developed. Therefore, reverse engineering for software is the process of analyzing a program in an effort to create a representation of the program at a higher

level of abstraction than source code. Reverse engineering is a process of design recovery. Reverse engineering tools extract data, architectural, and procedural design information from an existing program.

Code restructuring. The most common type of reengineering (actually, the use of the term reengineering is questionable in this case) is code restructuring. 4 Some legacy systems have a relatively solid program architecture, but individual modules were coded in a way that makes them difficult to understand, test, and maintain. In such cases, the code within the suspect modules can be restructured.

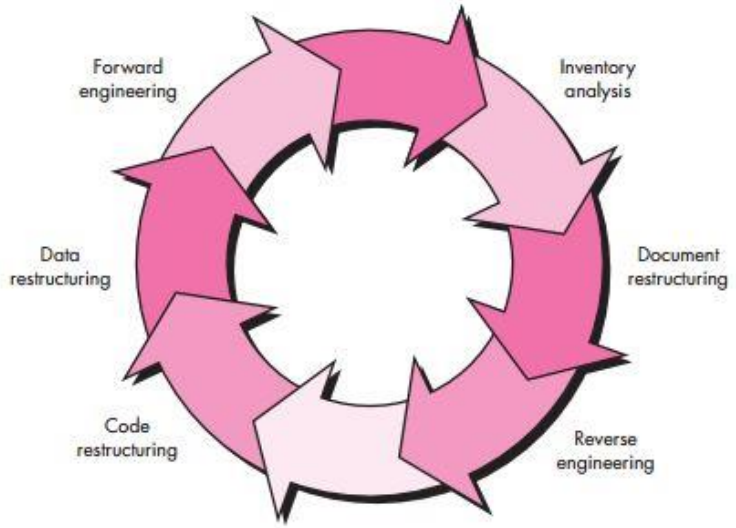
To accomplish this activity, the source code is analyzed using a restructuring tool. Violations of structured programming constructs are noted and code is then restructured (this can be done automatically) or even rewritten in a more modern programming language. The resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced. Internal code documentation is updated.

Data restructuring. A program with weak data architecture will be difficult to adapt and enhance. In fact, for many applications, information architecture has more to do with the long-term viability of a program than the source code itself. Unlike code restructuring, which occurs at a relatively low level of abstraction, data restructuring is a full-scale reengineering activity. In most cases, data restructuring begins with a reverse engineering activity. Current data architecture is dissected, and necessary data models are defined. Data objects and attributes are identified, and existing data structures are reviewed for quality.

When data structure is weak (e.g., flat files are currently implemented, when a relational approach would greatly simplify processing), the data are reengineered. Because data architecture has a strong influence on program architecture and the algorithms that populate it, changes to the data will invariably result in either architectural or code-level changes.

Forward engineering. In an ideal world, applications would be rebuilt using an automated "reengineering engine." The old program would be fed into the engine, analyzed, restructured, and then regenerated in a form that exhibited the best aspects of software quality. In the short term, it is unlikely that such an "engine" will appear, but vendors have introduced tools that provide a limited subset of these capabilities that addresses specific application domains (e.g., applications that are implemented using a specific database system). More important, these reengineering tools are becoming increasingly more sophisticated.

Forward engineering not only recovers design information from existing software but uses this information to alter or reconstitute the existing system in an effort to improve its overall quality. In most cases, reengineered software re-implements the function of the existing system and also adds new functions and/or improves overall performance.



13. Understanding Requirements

- ☑ Requirements Engineering
- ☑ Establishing the Groundwork
- ☑ Eliciting Requirements
- ☑ Developing Use Cases
- ☑ Building the Requirements Model

13.1 Requirements Engineering

The process of collecting the software requirement from the client then understand, evaluate and document it is called as requirement engineering. Requirement engineering constructs a bridge for design and construction. Requirement Engineering is the process of defining, documenting and maintaining the requirements. It is a process of gathering and defining service provided by the system. Requirements Engineering Process consists of the following main activities:

☞ Inception

Inception is a task where the requirement engineering asks a set of questions to establish a software process. In this task, it understands the problem and evaluates with the proper solution. It collaborates with the relationship between the customer and the developer. The developer and customer decide the overall scope and the nature of the question.

☞ Elicitation

Elicitation means to find the requirements from anybody. The requirements are difficult because the following problems occur in elicitation.

- ✓ **Problem of scope:** The customer give the unnecessary technical detail rather than clarity of the overall system objective.
- ✓ **Problem of understanding**: Poor understanding between the customer and the developer regarding various aspect of the project like capability, limitation of the computing environment.
- ✓ **Problem of volatility**: In this problem, the requirements change from time to time and it is difficult while developing the project.

☞ Elaboration

In this task, the information taken from user during inception and elaboration and are expanded and refined in elaboration. Its main task is developing pure model of software using functions, feature and constraints of a software.

☞ Negotiation

In negotiation task, a software engineer decides the how will the project be achieved with limited business resources. To create rough guesses of development and access the impact of the requirement on the project cost and delivery time.

☞ Specification

In this task, the requirement engineer constructs a final work product. The work product is in the form of software requirement specification. In this task, formalize the requirement of the proposed software such as informative, functional and behavioral. The requirement are formalize in both graphical and textual formats.

☞ Validation

The work product is built as an output of the requirement engineering and that is accessed for the quality through a validation step. The formal technical reviews from the software engineer, customer and other stakeholders helps for the primary requirements validation mechanism.

☞ Requirement management

It is a set of activities that help the project team to identify, control and track the requirements and changes can be made to the requirements at any time of the ongoing project. These tasks start with the identification and assign a unique identifier to each of the requirement. After finalizing the requirement traceability table is developed. The examples of traceability table are the features, sources, dependencies, subsystems and interface of the requirement.

13.2 Establishing the groundwork

There are the following steps required to establish the groundwork for an understanding of software requirements, to get the project started in a way that will keep it moving forward toward a successful solution.

(A) Identifying Stakeholders

Anyone who benefits in a direct or indirect way from the system which is being developed is called stakeholders. Each stakeholder has a different view of the system, achieves different benefits when the system is successfully developed, and is open to different risks if the development effort should fail. At inception, you should create a list of people who will contribute input as requirements are elicited.

(B) Recognizing Multiple Viewpoints

Because many different stakeholders exist, the requirements of the system will be explored from many different points of view. For example, the marketing group is interested in functions and features that will excite the potential market, making the new system easy to sell. Business managers are interested in a feature set that can be built within budget and that will be ready to meet defined market windows. End users may want features that are familiar to them and that are easy to learn and use. Software engineers may be concerned with functions that are invisible to nontechnical stakeholders but that enable an infrastructure that supports more marketable functions and features. Support engineers may focus on the maintainability of the software.

(C) Working toward Collaboration

The job of a requirements engineer is to identify areas of commonality (i.e., requirements on which all stakeholders agree) and areas of conflict or inconsistency (i.e., requirements that are desired by one stakeholder but conflict with the needs of another stakeholder). Collaboration does not necessarily mean that requirements are defined by committee. In many cases, stakeholders collaborate by providing their view of requirements, but a strong “project champion” (e.g., a business manager or a senior technologist) may make the final decision about which requirements make the cut.

(D) Asking the First Questions

Questions asked at the inception of the project should be “context free”. The first set of context-free questions focuses on the customer and other stakeholders, the overall project goals and benefits. For example, you might ask:

- ✓ Who is behind the request for this work?
- ✓ Who will use the solution?
- ✓ What will be the economic benefit of a successful solution?
- ✓ Is there another source for the solution that you need?

These questions help to identify all stakeholders who will have interest in the software to be built. In addition, the questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development.

The next set of questions enables you to gain a better understanding of the problem and allows the customer to voice his or her perceptions about a solution:

- ✓ How would you characterize "good" output that would be generated by a successful solution?
- ✓ What problem(s) will this solution address?
- ✓ Can you show me (or describe) the business environment in which the solution will be used?
- ✓ Will special performance issues or constraints affect the way the solution is approached?

The final set of questions focuses on the effectiveness of the communication activity itself.

- ✓ Are you the right person to answer these questions? Are your answers "official"?
- ✓ Are my questions relevant to the problem that you have?
- ✓ Am I asking too many questions?
- ✓ Can anyone else provide additional information?
- ✓ Should I be asking you anything else?

13.3 Eliciting Requirements

Requirements elicitation (also called requirements gathering) combines elements of problem solving, elaboration, negotiation, and specification. In order to encourage a collaborative, team-oriented approach to requirements gathering, stakeholders work together to identify the problem, propose elements of the solution, negotiate different approaches and specify a preliminary set of solution requirements.

Collaborative Requirements Gathering

Many different approaches to collaborative requirements gathering have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:

- ✓ Meetings are conducted and attended by both software engineers and other stakeholders.
- ✓ Rules for preparation and participation are established.
- ✓ An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- ✓ A "facilitator" (can be a customer, a developer, or an outsider) controls the meeting.
- ✓ A "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room, or virtual forum) is used.

The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal.

Quality Function Deployment

Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. QFD "concentrates on maximizing customer satisfaction from the software engineering process". To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process. QFD identifies three types of requirements:

Normal requirements. The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied. Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.

Expected requirements. These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction. Examples of expected requirements are: ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.

Exciting requirements. These features go beyond the customer's expectations and prove to be very satisfying when present. For example, software for a new mobile phone comes with standard features, but is coupled with a set of unexpected capabilities (e.g., multi-touch screen, visual voice mail) that delight every user of the product.

Usage Scenarios

As requirements are gathered, an overall vision of system functions and features begins to materialize. However, it is difficult to move into more technical software engineering activities until you understand how these functions and features will be used by different classes of end users. To accomplish this, developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called use cases.

Elicitation Work Products

The work products produced as a consequence of requirements elicitation will vary depending on the size of the system or product to be built. For most systems, the work products include:

- ✓ A statement of need and feasibility.
- ✓ A bounded statement of scope for the system or product.
- ✓ A list of customers, users, and other stakeholders who participated in requirements elicitation.
- ✓ A description of the system's technical environment.
- ✓ A list of requirements (preferably organized by function) and the domain constraints that apply to each.
- ✓ A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.

Any prototypes developed to better define requirements. Each of these work products is reviewed by all people who have participated in requirements elicitation.

13.4 Developing Use Cases

A **use case** is a description of a particular use of the system by a user of the system. It also defines the sequence of actions and behavior of the system. Anything that users would like to do with the IT system has to be made available as a use case. Each use case will describe the relation the actor has with the system in order to achieve a specific task. The success measurement for an effective written use case is one that is easily understood, and ultimately the developers can build the right product the first time.

Use case diagrams are behavior diagrams which are used to define a set of actions that system should or can perform in collaboration with one or more external users of the system (actors). Each use case should provide some observable and valuable result to the actors or other stakeholders of the system.

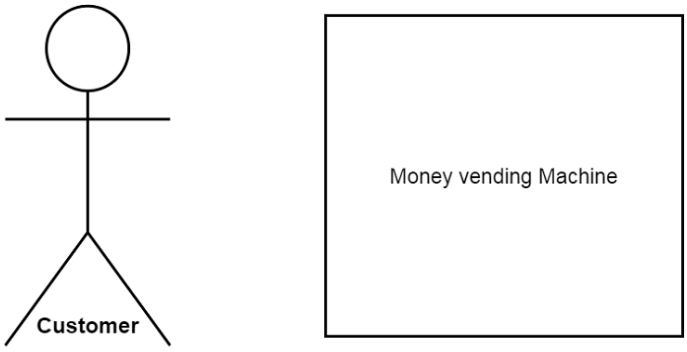
How do you write a Use Case? Use case diagrams are designed with 4 major elements:

- ☞ Actors
- ☞ System
- ☞ Use Cases
- ☞ Relationships between actors and use cases

Actors

Actor is an external entity, any "object" or person that has behavior associated with it. You can picture an actor as a user of the IT system. Generally, the users are actors but often systems can be actors as well. Actor names should not correspond to job titles.

Ticket vending machine allows commuters to buy tickets. So Commuter is our actor in this case. Customer is an actor for the money vending machine.



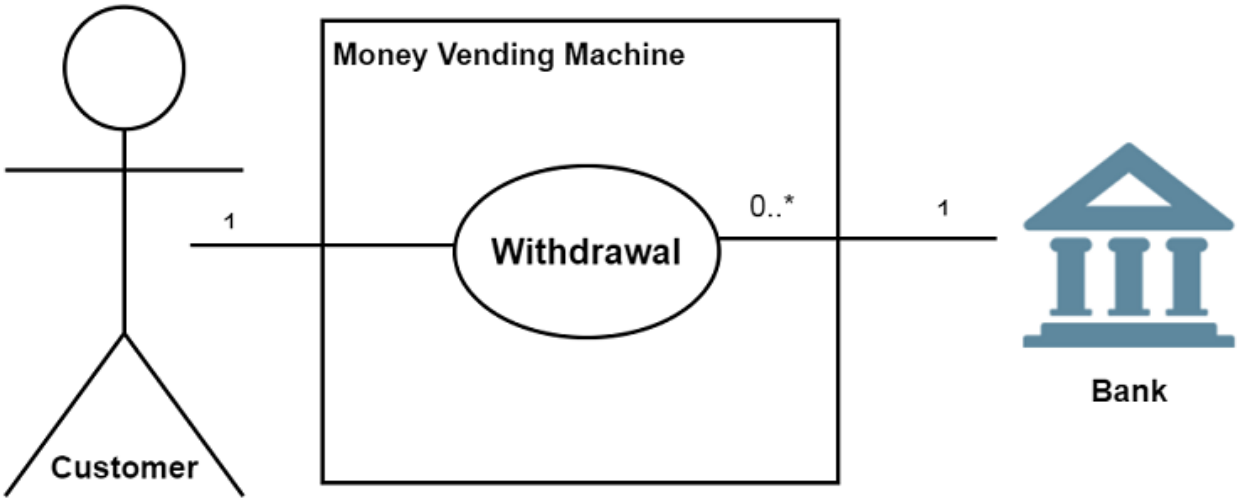
System

System is used to define the scope of the use case and drawn as a rectangle. This an optional element but useful when your visualizing large systems. For example you can create all the use cases and then use the system object to define the scope covered by your project. Or you can even use it to show the different areas covered in different releases.

Use Case

Use cases represent the activities that actors perform with the help of your system in the pursuit of a particular goal. We need to define what those users (actors) need from the system. Use case should reflect user needs and goals, and should be initiated by an actor. Business actor Customer participating in the business use case should be connected to the use case by association.

The ultimate goal of the customer in relation to our money vending machine is to withdraw money. So we are adding **Withdrawal** use case. Withdrawing money from the vending machine might involve a bank for the transactions to be made. So we are also adding another actor – **Bank**. Both actors participating in the use case should be connected to the use case by association.



Money vending machine provides Withdrawal use case for the customer and Bank actors.

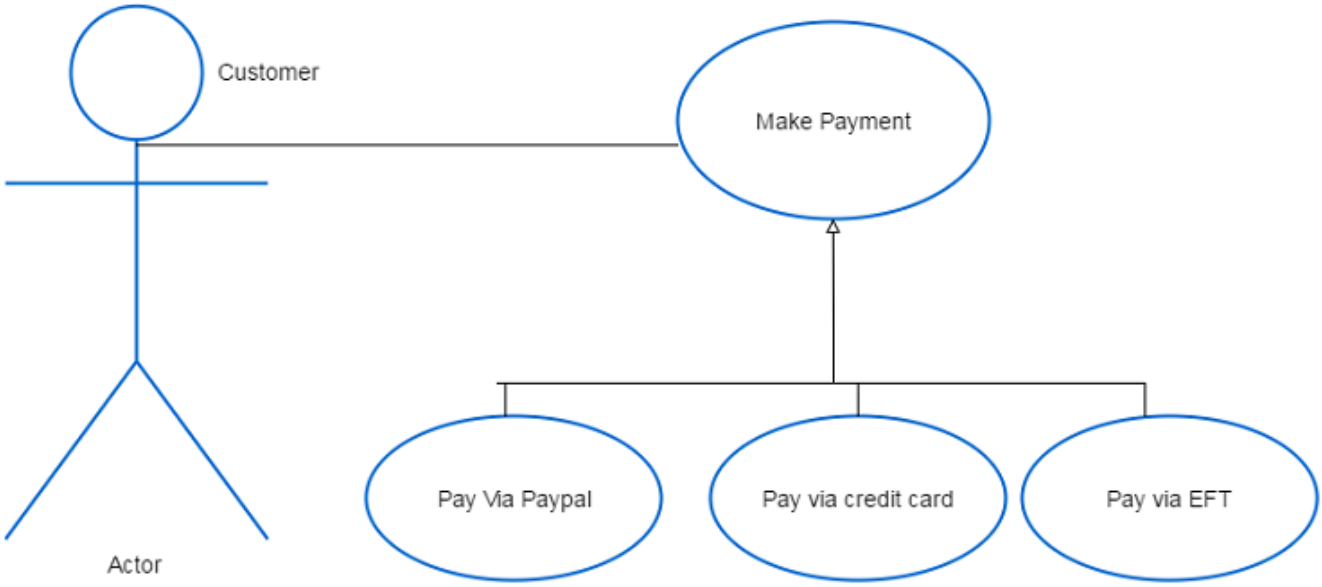
Relationships between Actors and Use cases

Use cases could be organized using following relationships:

- ✓ Generalization
- ✓ Association
- ✓ Extend
- ✓ Include

Generalization between Use Cases

There may be instances where actors are associated with similar use cases. In such case a Child use case inherits the properties and behavior of the parent use. Hence we need to generalize the actor to show the inheritance of functions. They are represented by a solid line with a large hollow triangle arrowhead.



Association between Use Cases

Associations between actors and use cases are indicated in use case diagrams by solid lines. An association exists whenever an actor is involved with an interaction described by a use case.

Extend

There are some functions that are triggered optionally. In such cases the extend relationship is used and the extension rule is attached to it. Thing to remember is that the base use case should be able to perform a function on its own even if the extending use case is not called.

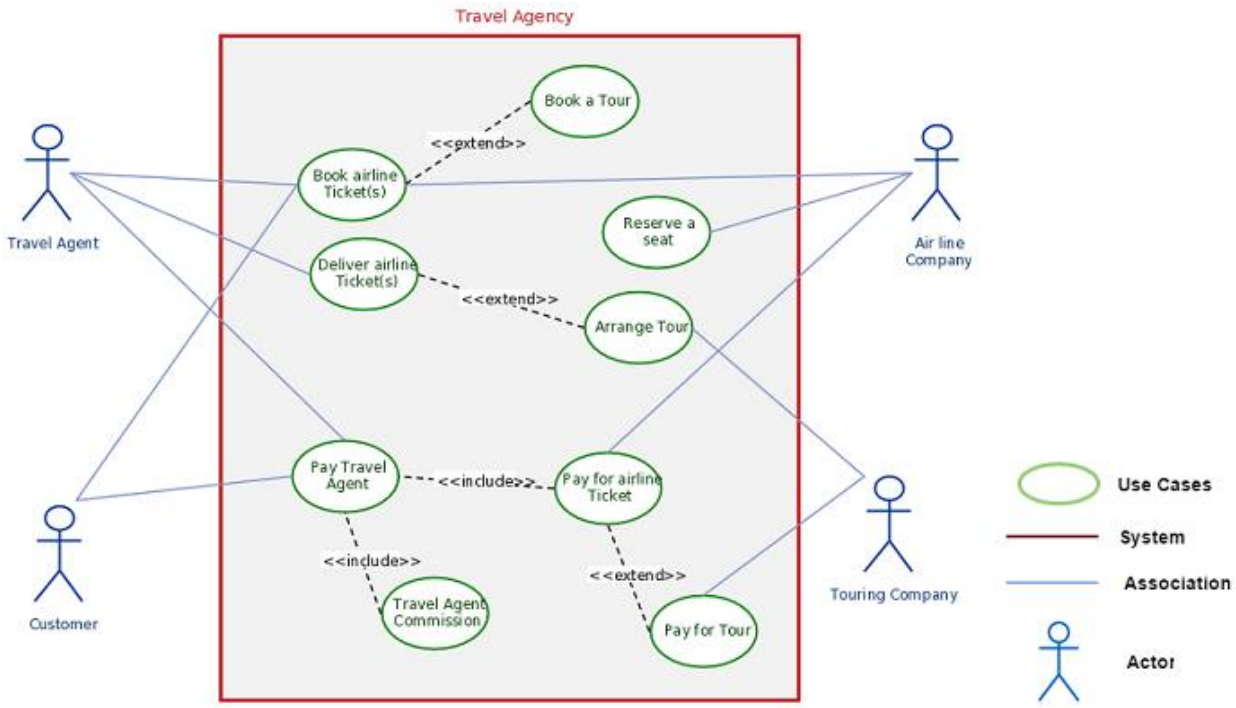
Extend relationship is shown as a dashed line with an open arrowhead directed from the extending use case to the extended (base) use case. The arrow is labeled with the keyword «extend».

Include

It is used to extract use case fragments that are duplicated in multiple use cases. It is also used to simplify large use case by splitting it into several use cases and to extract common parts of the behaviors of two or more use cases.

Include relationship between use cases which is shown by a dashed arrow with an open arrowhead from the base use case to the included use case. The arrow is labeled with the keyword «include».

Use cases deal only in the functional requirements for a system. Other requirements such as business rules, quality of service requirements, and implementation constraints must be represented separately. The diagram shown below is an example of a simple use case diagram with all the elements marked.



Basic Principles for Successful Application of Use cases

- Keep it simple by telling stories
- Be productive without perfection
- Understand the big picture
- Identify reuse opportunity for use cases
- Focus on value
- Build the system in slices
- Deliver the system in increments
- Adapt to meet the team's needs

What is the importance of Use Cases?

Use cases have been used extensively over the past few decades. The advantages of Use cases includes:

- The list of goal names provides the shortest summary of what the system will offer
- It gives an overview of the roles of each and every component in the system. It will help us in defining the role of users, administrators etc.
- It helps us in extensively defining the user's need and exploring it as to how it will work.
- It provides solutions and answers to many questions that might pop up if we start a project unplanned.

13.5 Building the requirement model

The intent of the requirement model is to provide a description of the required informational, functional, and behavioral domains for a computer-based system. The model changes dynamically as you learn more about the system to be built, and other stakeholders understand more about what they really require.

As the requirements model evolves, certain elements will become relatively stable, providing a solid foundation for the design tasks that follow. However, other elements of the model may be more volatile, indicating that stakeholders do not yet fully understand requirements for the system.

13.5.1 Elements of the Requirements Model

The specific elements of the requirements model are dictated by the analysis modeling method that is to be used. However, a set of generic elements is common to most requirements models.

- **Scenario-based elements:** The system is described from the user's point of view using a scenario-based approach. For example, basic use cases and their corresponding use-case diagrams evolve into more elaborate template-based use cases. Scenario-based elements of the requirements model are often the first part of the model that is developed. As such, they serve as input for the creation of other modeling elements.
- **Class-based elements:** Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system. These objects are categorized into classes—a collection of things that have similar attributes and common behaviors.
- **Behavioral elements:** The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied. Therefore, the requirements model must provide modeling elements that depict behavior. The state diagram is one method for representing the behavior of a system by depicting its states and the events that cause the system to change state. A state is any externally observable mode of behavior.
- **Flow-oriented elements:** Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms, applies functions to transform it, and produces output in a variety of forms. Input may be a control signal transmitted by a transducer, a series of numbers typed by a human operator, a packet of information transmitted on a network link, or a voluminous data file retrieved from secondary storage.

13.5.2 Analysis Patterns

Anyone who has done requirements engineering on more than a few software projects begins to notice that certain problems reoccur across all projects within a specific application domain. These analysis patterns suggest solutions (e.g., a class, a function, a behavior) within the application domain that can be reused when modeling many applications.

First, analysis patterns speed up the development of abstract analysis models that capture the main requirements of the concrete problem by providing reusable analysis models with examples as well as a description of advantages and limitations. Second, analysis patterns facilitate the transformation of the analysis model into a design model by suggesting design patterns and reliable solutions for common problems.

Analysis patterns are integrated into the requirement model by reference to the pattern name. They are also stored in a repository so that requirements engineers can use search facilities to find and apply them.

Functional vs. Non-Functional Requirements

A **functional requirement** specifies something that the application or system should do. Often this is defined as a behavior of the system that takes input and provides output. For example, a traveler fills out a form in an airline's mobile application with his/her name and passport details (input), submits the form, and the application generates a boarding pass with the traveler's details (output).

Non-functional requirements, sometimes also called quality requirements, describe how the system should be, as opposed to what it should do. Non-functional requirements of a system include performance (e.g. response time), maintainability, and scalability, among many others. In the airline application example, the requirement that the application must display the boarding pass after a maximum of five seconds from the time the traveler presses the 'submit' button would be a non-functional requirement.

- ✓ Functional requirement is specified by User, while non-functional requirement is specified by technical peoples e.g. Architect, Technical leaders and software developers.
- ✓ Functional requirement is also the activity System must perform, on other hand non-functional are depending upon criticality of application. For example, if your application is not critical and you can live with downtime, you may not need to develop complex failover and disaster recovery code, reducing your application total development time.
- ✓ Functional requirements defines a software's functionality i.e. what can they do, while non-functional requirements defines, other things which is not required by user but requirement by service provider or software itself e.g. logging is a non-functional requirement to support an application, not directly used by user but essential to troubleshoot any issue in production environment.
- ✓ Non-functional requirements are sometimes defined in terms of metrics (something that can be measured about the system) to make them more tangible.
- ✓ Non-functional requirements may also describe aspects of the system that don't relate to its execution, but rather to its evolution over time (e.g. maintainability, extensibility, documentation, etc.).

14. Requirement Modelling Strategies

There are two types of requirement modelling strategies one is Flow Oriented Modelling and second is Class-based Modelling. Their details are as follows:

(1). Flow Oriented Modelling

Flow models focus on the flow of data objects as they are transformed by processing functions. Derived from structured analysis, flow models use the data flow diagram, a modeling notation that depicts how input is transformed into output as data objects move through the system. Each software function that transforms data is described by a process specification or narrative. In addition to data flow, this modeling element also depicts control flow.

Data flow oriented modeling is the most widely used analysis notation. Flow oriented modeling focuses on structured analysis and design, follows a top to down methodology and uses a graphical technique depicting information flows and the transformations that are applied as data moves from input to output.

The Flow oriented elements are:

(A) Data flow model

- ☞ It is a graphical technique. It is used to represent information flow.
- ☞ The data objects are flowing within the software and transformed by processing the elements.

- ☞ The data objects are represented by labeled arrows. Transformation are represented by circles called as bubbles.
- ☞ DFD shown in a hierarchical fashion. The DFD is split into different levels. It also called as 'context level diagram'.

(B) Control flow model

- ☞ Large class applications require a control flow modeling.
- ☞ The application creates control information instated of reports or displays.
- ☞ The applications process the information in specified time.
- ☞ An event is implemented as a boolean value.
  For example, the boolean values are true or false, on or off, 1 or 0.

(C) Control Specification

- ☞ A short term for control specification is CSPEC.
- ☞ It represents the behaviour of the system.
- ☞ The state diagram in CSPEC is a sequential specification of the behaviour.
- ☞ The state diagram includes states, transitions, events and activities.
- ☞ State diagram shows the transition from one state to another state if a particular event has occurred.

(D) Process Specification

- ☞ A short term for process specification is PSPEC.
- ☞ The process specification is used to describe all flow model processes.
- ☞ The content of process specification consists narrative text, Program Design Language(PDL) of the process algorithm, mathematical equations, tables or UML activity diagram.

(2). Class-based Modeling

Class based modeling represents the object. The system manipulates the operations. The elements of the class based model consist of classes and object, attributes, operations, class – responsibility - collaborator (CRS) models.

☞ Classes

Classes are determined using underlining each noun or noun clause and enter it into the simple table. Classes are found in following forms:

- ✓ **External entities**: The system, people or the device generates the information that is used by the computer based system.
- ✓ **Things**: The reports, displays, letter, signal are the part of the information domain or the problem.
- ✓ **Occurrences or events**: A property transfer or the completion of a series or robot movements occurs in the context of the system operation.
- ✓ **Roles**: The people like manager, engineer, sales person are interacting with the system.
- ✓ **Organizational units:** The division, group, team are suitable for an application.
- ✓ **Places**: The manufacturing floor or loading dock from the context of the problem and the overall function of the system.
- ✓ **Structures**: The sensors, computers are defined a class of objects or related classes of objects.

☞ Attributes

Attributes are the set of data objects that are defining a complete class within the context of the problem. For example, 'employee' is a class and it consists of name, Id, department, designation and salary of the employee are the attributes.

☞ Operations

The operations define the behavior of an object. The operations are characterized into following types:

- ✓ The operations manipulate the data like adding, modifying, deleting and displaying etc.
- ✓ The operations perform a computation.
- ✓ The operation monitors the objects for the occurrence of controlling an event.

☞ CRS Modeling

- ✓ The CRS stands for Class-Responsibility-Collaborator.
- ✓ It provides a simple method for identifying and organizing the classes that are applicable to the system or product requirement.
- ✓ Class is an object-oriented class name. It consists of information about sub classes and super class.
- ✓ Responsibilities are the attributes and operations that are related to the class.
- ✓ Collaborations are identified and determined when a class can achieve each responsibility of it. If the class cannot identify itself, then it needs to interact with another class.

15. Difference Between Structured Analysis and Object Oriented Analysis

Structured Analysis

Structured analysis is a software engineering technique that uses graphical diagrams to develop and portray system specifications that are easily understood by users. These diagrams describe the steps that need to occur and the data required to meet the design function of a particular software. This type of analysis mainly focuses on logical systems and functions, and aims to convert business requirements into computer programs and hardware specifications. Structured analysis is a fundamental aspect of system analysis.

The major steps involved in the structured analysis process are:

- ☞ Studying the current business environment
- ☞ Modeling the old logical system
- ☞ Modeling a new logical system
- ☞ Modeling a new physical environment
- ☞ Evaluating alternatives
- ☞ Selecting the best design
- ☞ Creating structured specifications

There are three orthogonal views related to structured analysis:

- ☞ **Functional View**: This involves data flow diagrams, which define the work that has been done and the flow of data between things done, thereby providing the primary structure of a solution.
- ☞ **Data View**: This comprises the entity relationship diagram and is concerned with what exists outside the system that is being monitored.
- ☞ **Dynamic View**: This includes state transition diagrams and defines when things happen and the conditions under which they may happen.

Object Oriented Analysis

Object oriented analysis, focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements.

The use of modeling to define and analyze the requirements necessary for success of a system. Object-oriented analysis is a process that groups items that interact with one another, typically by class, data or behavior, to create a model that accurately represents the intended purpose of the system as a whole. Object-oriented analysis does not factor implementation limitations into the model.

16. Difference between FDD Diagrams and UML Diagrams

What is UML Diagram?

UML (**Unified Modeling Language)** is a modeling language used in object oriented software design. When developing object oriented software, UML is used to specify and visualize the components that make up a software system. UML diagrams mainly represent the structural view and the behavioral view of a system.

UML is a modeling language used in object oriented software design. UML provides capabilities to specify and visualize the components that make up a software system. UML diagrams mainly represent the structural view and the behavioral view of a system. Structural view of the system is represented using diagrams like class diagrams, composite structure diagrams, etc. Dynamic view of the system is represented using diagrams such as sequence diagrams, activity diagrams, etc. UML version 2.2 includes fourteen diagrams, which includes seven diagrams for representing the structural view and other seven representing the behavioral view. Among the seven behavioral diagrams, four diagrams can be used to represent interactions with the system. There are tools that can be used for UML modeling such as IBM Rational Rose.

What is FDD Diagram?

The purpose of the **functional decomposition diagram** is to show on a single page the capabilities of an organization that are relevant to the consideration of an architecture. By examining the capabilities of an organization from a functional perspective, it is possible to quickly develop models of what the organization does without being dragged into an extended debate on how the organization does it. Once a basic functional decomposition diagram has been developed, it becomes possible to layer heat-maps on top of this diagram to show scope and decisions. For example, the capabilities to be implemented during the different phases of a change program.

Functional decomposition corresponds to the various functional relationships as how the original complex business function was developed. It mainly focusses on how the overall functionality is developed and its interaction between various components. Large or complex functionalities are more easily understood when broken down into pieces using functional decomposition.

17. Data and Process Modelling

- ☑ What is a data flow diagram?
- ☑ Physical and logical data flow diagrams
- ☑ Data flow diagram levels
- ☑ How to create a data flow diagram

Businesses are built on systems and processes—a company couldn't operate without them. From lead nurturing methods to the way a team interacts with customers, nearly everything a business does involves a system of some sort. And, when it comes to systems and processes, efficiency is everything. In some cases, shaving even a minute or two off can lead to substantial savings. There are countless ways to analyze and improve efficiency, but one that stands out is through data flow diagrams.

Whether you are improving an existing process or implementing a new one, a data flow diagram (DFD) will make the task easier. However, if you've never created a DFD before, getting started can be intimidating. There is a lot to take in: different levels of diagrams, symbols and notation, not to mention actually creating the diagram navigating it all will take more than looking at a few examples. If you're new to data flow diagrams, this guide will help get you started.

☞ What is a Data Flow Diagram(DFD)?

A data flow diagram shows the way information flows through a process or system. It includes data inputs and outputs, data stores, and the various sub-processes the data moves through. DFDs are built using standardized symbols and notation to describe various entities and their relationships.

Data flow diagrams visually represent systems and processes that would be hard to describe in a chunk of text. You can use these diagrams to map out an existing system and make it better or to plan out a new system for implementation. Visualizing each element makes it easy to identify inefficiencies and produce the best possible system.

☞ Physical and logical data flow diagrams

Before actually creating your data flow diagram, you'll need to determine whether a physical or logical DFD best suits your needs. If you're new to data flow diagrams, don't worry the distinction is pretty straightforward.

Logical data flow diagrams focus on what happens in a particular information flow: what information is being transmitted, what entities are receiving that info, what general processes occur, etc. The processes described in a logical DFD are business activities—a logical DFD doesn't delve into the technical aspects of a process or system. Non-technical employees should be able to understand these diagrams.

Physical data flow diagrams focus on how things happen in an information flow. These diagrams specify the software, hardware, files, and people involved in an information flow. A detailed physical data flow diagram can facilitate the development of the code needed to implement a data system.

Both physical and logical data flow diagrams can describe the same information flow. In coordination they provide more detail than either diagram would independently. As you decide which to use, keep in mind that you may need both.

☞ Data flow diagram levels

Data flow diagrams are also categorized by level. Starting with the most basic, level 0, DFDs get increasingly complex as the level increases. As you build your own data flow diagram, you will need to decide which level your diagram will be.

Level 0 DFDs, also known as context diagrams, are the most basic data flow diagrams. They provide a broad view that is easily digestible but offers little detail. Level 0 data flow diagrams show a single process node and its connections to external entities.

Level 1 DFDs are still a general overview, but they go into more detail than a context diagram. In a level 1 data flow diagram, the single process node from the context diagram is broken down into sub-processes. As these processes are added, the diagram will need additional data flows and data stores to link them together.

Level 2+ DFDs simply break processes down into more detailed sub-processes. In theory, DFDs could go beyond level 3, but they rarely do. Level 3 data flow diagrams are detailed enough that it doesn't usually make sense to break them down further.

☞ DFD Diagram Notations

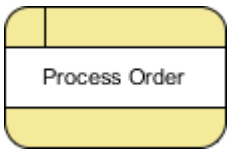
Now we'd like to briefly introduce to you a few diagram notations which you'll see in the tutorial below.

External Entity

An external entity can represent a human, system or subsystem. It is where certain data comes from or goes to. It is external to the system we study, in terms of the business process. For this reason, people used to draw external entities on the edge of a diagram.

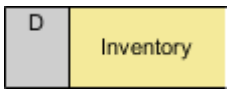
Process

A process is a business activity or function where the manipulation and transformation of data takes place. A process can be decomposed to finer level of details, for representing how data is being processed within the process.



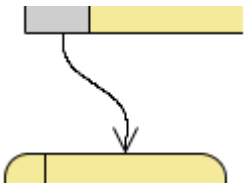
Data Store

A data store represents the storage of persistent data required and/or produced by the process. Here are some examples of data stores: membership forms, database table, etc.



Data Flow

A data flow represents the flow of information, with its direction represented by an arrow head that shows at the end(s) of flow connector.



☞ How to create a data flow diagram

Now that you have some background knowledge on data flow diagrams and how they are categorized, you're ready to build your own DFD. The process can be broken down into 5 steps:

1. Identify major inputs and outputs in your system

Nearly every process or system begins with input from an external entity and ends with the output of data to another entity or database. Identifying such inputs and outputs gives a macro view of your system—it shows the broadest tasks the system should achieve. The rest of your DFD will be built on these elements, so it is crucial to know them early on.

2. Build a context diagram

Once you've identified the major inputs and outputs, building a context diagram is simple. Draw a single process node and connect it to related external entities. This node represents the most general process information undergoes to go from input to output.

The example below shows how information flows between various entities via an online community. Data flows to and from the external entities, representing both input and output. The center node, "online community," is the general process.

3. Expand the context diagram into a level 1 DFD

The single process node of your context diagram doesn't provide much information—you need to break it down into sub-processes. In your level 1 data flow diagram, you should include several process nodes, major databases, and all external entities. Walk through the flow of information: where does the information start and what needs to happen to it before each data store?

4. Expand to a level 2+ DFD

To enhance the detail of your data flow diagram, follow the same process as in step 3. The processes in your level 1 DFD can be broken down into more specific sub-processes. Once again, ensure you add any necessary data stores and flows—at this point you should have a fairly detailed breakdown of your system. To progress beyond a level 2 data flow diagram, simply repeat this process. Stop once you've reached a satisfactory level of detail.

5. Confirm the accuracy of your final diagram

When your diagram is completely drawn, walk through it. Pay close attention to the flow of information: does it make sense? Are all necessary data stores included? By looking at your final diagram, other parties should be able to understand the way your system functions. Before presenting your final diagram, check with co-workers to ensure your diagram is comprehensible.

18. Introduction to SDLC

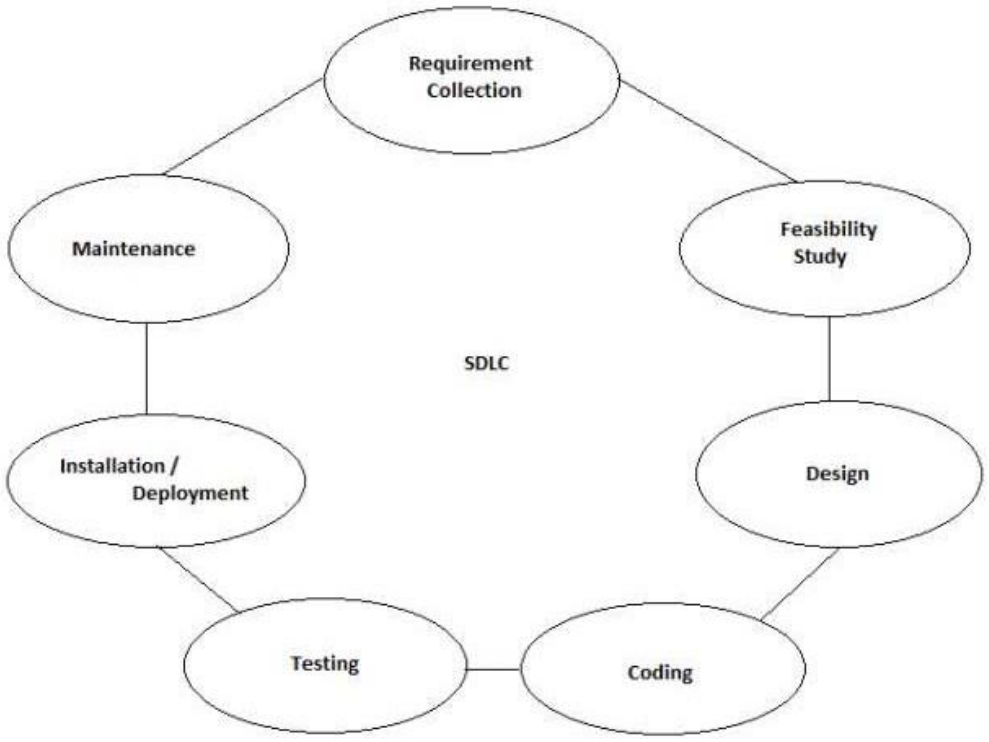
SDLC stands for Software Development Life Cycle. The software development life cycle (SDLC) is a process which is used to develop software. SDLC is a step by step procedure need to be followed by the organization to design and develop a high quality product. The phases of software development life cycle are which describes that how to develop, maintain particular software. The life cycle aims to develop a good quality product/software. SDLC produces intermediate products that can be reviewed to check whether they work according to customer requirement.

- ✓ SDLC is also known as Software development process.
- ✓ SDLC is an approach creates considerable documentation where this documentation helpful to make sure that requirement can be traced back to stated business requirements.
- ✓ It is a framework which has a set of tasks performed at each phase in the software development process.

SDLC is a step by step procedure or systematic approach to develop software and it is followed within a software organization. It consists of various phases which describe how to design, develop, enhance and maintain particular software.

18.1 SDLC Phases

System Development Life Cycle phases are as follows:



Phase 1: Requirement collection and analysis:

The requirement is the first stage in the SDLC process. It is conducted by the senior team members with inputs from all the stakeholders and domain experts in the industry. Planning for the quality assurance requirements and recognition of the risks involved is also done at this stage.

This stage gives a clearer picture of the scope of the entire project and the anticipated issues, opportunities, and directives which triggered the project. Requirements Gathering stage need teams to get detailed and precise requirements. This helps companies to finalize the necessary timeline to finish the work of that system.

Phase 2: Feasibility study:

Once the requirement analysis phase is completed the next step is to define and document software needs. This process conducted with the help of 'Software Requirement Specification' document also known as 'SRS' document. It includes everything which should be designed and developed during the project life cycle.

There are mainly five types of feasibilities checks:

- ✓ Economic: Can we complete the project within the budget or not?
- ✓ Legal: Can we handle this project as cyber law and other regulatory framework/compliances?
- ✓ Operation feasibility: Can we create operations which is expected by the client?
- ✓ Technical: Need to check whether the current computer system can support the software
- ✓ Schedule: Decide that the project can be completed within the given schedule or not.

Phase 3: Design:

In this third phase, the system and software design documents are prepared as per the requirement specification document. This helps define overall system architecture. This design phase serves as input for the next phase of the model. There are two kinds of design documents developed in this phase:

High-Level Design (HLD)

- ✓ Brief description and name of each module
- ✓ An outline about the functionality of every module
- ✓ Interface relationship and dependencies between modules
- ✓ Database tables identified along with their key elements
- ✓ Complete architecture diagrams along with technology details

Low-Level Design (LLD)

- ✓ Functional logic of the modules
- ✓ Database tables, which include type and size
- ✓ Complete detail of the interface
- ✓ Addresses all types of dependency issues
- ✓ Listing of error messages
- ✓ Complete input and outputs for every module

Phase 4: Coding:

Once the system design phase is over, the next phase is coding. In this phase, developers start build the entire system by writing code using the chosen programming language. In the coding phase, tasks are divided into units or modules and assigned to the various developers. It is the longest phase of the Software Development Life Cycle process. In this phase, Developer needs to follow certain predefined coding guidelines. They also need to use programming tools like compiler, interpreters, debugger to generate and implement the code.

Phase 5: Testing:

Once the software is complete, and it is deployed in the testing environment. The testing team starts testing the functionality of the entire system. This is done to verify that the entire application works according to the customer requirement.

During this phase, QA and testing team may find some bugs/defects which they communicate to developers. The development team fixes the bug and send back to QA for a re-test. This process continues until the software is bug-free, stable, and working according to the business needs of that system.

Phase 6: Installation/Deployment:

Once the software testing phase is over and no bugs or errors left in the system then the final deployment process starts. Based on the feedback given by the project manager, the final software is released and checked for deployment issues if any.

Phase 7: Maintenance:

Once the system is deployed, and customers start using the developed system, following 3 activities occur

- ✓ Bug fixing - bugs are reported because of some scenarios which are not tested at all
- ✓ Upgrade - Upgrading the application to the newer versions of the Software
- ✓ Enhancement - Adding some new features into the existing software

The main focus of this SDLC phase is to ensure that needs continue to be met and that the system continues to perform as per the specification mentioned in the first phase.

SDLC Models:

The models of SDLC are the methodologies that are selected for the software development is depending on the project's aims and goals. These models are mainly used to develop software, based on the requirement, cost, customer and time, decide which model to be followed to develop software. Each model follows sequential steps of its own type, to develop high quality software. The types of SDLC models are:

- ✓ Waterfall Model
- ✓ Spiral Model
- ✓ Prototype Model
- ✓ V-Model
- ✓ Iterative model
- ✓ Agile Model

19. SWOT Analysis

SWOT is an acronym for **Strengths**, **Weaknesses**, **Opportunities** and **Threats**. By definition, Strengths (S) and Weaknesses (W) are considered to be internal factors over which you have some measure of control. Also, by definition, Opportunities (O) and Threats (T) are considered to be external factors over which you have essentially no control.

SWOT Analysis is the most renowned tool for audit and analysis of the overall strategic position of the business and its environment. Its key purpose is to identify the strategies that will create a firm specific business model that will best align an organization's resources and capabilities to the requirements of the environment in which the firm operates. An overview of the four factors (Strengths, Weaknesses, Opportunities and Threats) is given below-

(1). Strengths

Strengths are the qualities that enable us to accomplish the organization's mission. These are the basis on which continued success can be made and continued/sustained. Strengths can be either tangible or intangible. These

are what you are well-versed in or what you have expertise in, the traits and qualities your employees possess (individually and as a team) and the distinct features that give your organization its consistency.

Strengths are the beneficial aspects of the organization or the capabilities of an organization, which includes human competencies, process capabilities, financial resources, products and services, customer goodwill and brand loyalty. Examples of organizational strengths are huge financial resources, broad product line, no debt, committed employees, etc.

(2). Weaknesses

Weaknesses are the qualities that prevent us from accomplishing our mission and achieving our full potential. These weaknesses deteriorate influences on the organizational success and growth. Weaknesses are the factors which do not meet the standards we feel they should meet.

Weaknesses in an organization may be depreciating machinery, insufficient research and development facilities, narrow product range, poor decision-making, etc. Weaknesses are controllable. They must be minimized and eliminated. For instance - to overcome obsolete machinery, new machinery can be purchased. Other examples of organizational weaknesses are huge debts, high employee turnover, complex decision making process, narrow product range, large wastage of raw materials, etc.

(3). Opportunities

Opportunities are presented by the environment within which our organization operates. These arise when an organization can take benefit of conditions in its environment to plan and execute strategies that enable it to become more profitable. Organizations can gain competitive advantage by making use of opportunities.

Organization should be careful and recognize the opportunities and grasp them whenever they arise. Selecting the targets that will best serve the clients while getting desired results is a difficult task. Opportunities may arise from market, competition, industry/government and technology. Increasing demand for telecommunications accompanied by deregulation is a great opportunity for new firms to enter telecom sector and compete with existing firms for revenue.

(4). Threats

Threats arise when conditions in external environment jeopardize the reliability and profitability of the organization's business. They compound the vulnerability when they relate to the weaknesses. Threats are uncontrollable. When a threat comes, the stability and survival can be at stake. Examples of threats are - unrest among employees; ever changing technology; increasing competition leading to excess capacity, price wars and reducing industry profits; etc.

☞ Advantages of SWOT Analysis

SWOT Analysis is instrumental in strategy formulation and selection. It is a strong tool, but it involves a great subjective element. It is best when used as a guide, and not as a prescription. Successful businesses build on their strengths, correct their weakness and protect against internal weaknesses and external threats. They also keep a watch on their overall business environment and recognize and exploit new opportunities faster than its competitors. SWOT Analysis helps in strategic planning in following manner:

- ✓ It is a source of information for strategic planning.
- ✓ Builds organization's strengths.
- ✓ Reverse its weaknesses.
- ✓ Maximize its response to opportunities.
- ✓ Overcome organization's threats.
- ✓ It helps in identifying core competencies of the firm.
- ✓ It helps in setting of objectives for strategic planning.

- ✓ It helps in knowing past, present and future so that by using past and current data, future plans can be chalked out.

SWOT Analysis provide information that helps in synchronizing the firm's resources and capabilities with the competitive environment in which the firm operates.

☞ Limitations of SWOT Analysis

SWOT Analysis is not free from its limitations. It may cause organizations to view circumstances as very simple because of which the organizations might overlook certain key strategic contact which may occur. Moreover, categorizing aspects as strengths, weaknesses, opportunities and threats might be very subjective as there is great degree of uncertainty in market. SWOT Analysis does stress upon the significance of these four aspects, but it does not tell how an organization can identify these aspects for itself. There are certain limitations of SWOT Analysis which are not in control of management. These include:

- ✓ Price increase;
- ✓ Inputs/raw materials;
- ✓ Government legislation;
- ✓ Economic environment;
- ✓ Searching a new market for the product which is not having overseas market due to import restrictions; etc.

Internal limitations may include:

- ✓ Insufficient research and development facilities;
- ✓ Faulty products due to poor quality control;
- ✓ Poor industrial relations;
- ✓ Lack of skilled and efficient labor, etc.

20. Importance of Strategic Planning

Abstract Strategic planning is a management technique that helps organizations set future goals and objectives to achieve more stable and predictable growth. Strategic planning also identifies the actions required to reach these goals. With another parlance, it is a methodology by which a specific roadmap is recognized for growing a doable, coherent and strong business or organization. The concept of strategic planning, on the other hand, is defined as an instrument that allows making long-term plans in consideration of the risks and opportunities faced by the organization, and improving efficiency by acting in line with these plans. The selected sample business is a European Café/fast food restaurant. This paper outlines and discusses the importance of executing and crafting a strategic plan for the success of the selected businesses. As a strategic plan should comprise mission and vision statements, first, it highlights the importance of a mission statement for the company; explains how the mission will be essential to the company's success. Then, it emphasizes how the vision statement supports the company's mission. This report determines the five key objectives for the company encompassing operational, financial, and human resource aspects of the business, and additionally justifies why each of these objectives is essential to the success of the business and how they support the mission and vision statements.

Simply put, a strategic plan is the formalized road map that describes how your company executes the chosen strategy. A plan spells out where an organization is going over the next year or more and how it's going to get there. Typically, the plan is organization-wide or focused on a major function, such as a division or a department.

A strategic plan is a management tool that serves the purpose of helping an organization do a better job, because a plan focuses the energy, resources, and time of everyone in the organization in the same direction.

A strategic plan is a management tool that C-level managers need to master and is for established businesses and business owners who are serious about growth. It also does the following:

- ✓ Helps build your competitive advantage
- ✓ Communicates your strategy to staff
- ✓ Prioritizes your financial needs
- ✓ Provides focus and direction to move from plan to action

21. Information Systems Projects

This section discusses reasons for systems projects and the internal and external factors that affect systems projects. The section also includes a preview of project management.

☞ Main Reasons for Systems Projects

The starting point for most projects is called a systems request, which is a formal way of asking for IT support. A systems request might propose enhancements for an existing system, the correction of problems, the replacement of an older system, or the development of an entirely new information system that is needed to support a company's current and future business needs.

The main reasons for systems requests are improved service to customers, better performance, support for new products and services, more information, stronger controls, and reduced cost.

☞ Improved Service

Systems requests often are aimed at improving service to customers or users within the company. Allowing mutual fund investors to check their account balances on a Web site, storing data on rental car customer

preferences, or creating an online college registration system are examples that provide valuable services and increased customer satisfaction.

☞ Support For New Products and Services

New products and services often require new types or levels of IT support. For example, a software vendor might offer an automatic upgrade service for subscribers; or a package delivery company might add a special service for RFID-tagged shipments. In situations like these, it is most likely that additional IT support will be required. At the other end of the spectrum, product obsolescence can also be an important factor in IT planning. As new products enter the marketplace, vendors often announce that they will no longer provide support for older versions. A lack of vendor support would be an important consideration in deciding whether or not to upgrade.

☞ Better Performance

The current system might not meet performance requirements. For example, it might respond slowly to data inquiries at certain times, or it might be unable to support company growth. Performance limitations also result when a system that was designed for a specific hardware configuration becomes obsolete when new hardware is introduced.

☞ More Information

The system might produce information that is insufficient, incomplete, or unable to support the company's changing information needs. For example, a system that tracks customer orders might not be capable of analyzing and predicting marketing trends. In the face of intense competition and rapid product development cycles, managers need the best possible information to make major decisions on planning, designing, and marketing new products and services.

☞ Stronger Controls

A system must have effective controls to ensure that data is secure and accurate. Some common security controls include passwords, various levels of user access, and **encryption**, or coding of data to keep it safe from unauthorized users. Hardware-based security controls include **biometric devices** that can identify a person by a retina scan or by mapping a facial pattern. A new biometric tool scans hands, rather than faces. The technology uses infrared scanners that create images with thousands of measurements of hand and finger characteristics.

In addition to being secure, data also must be accurate. Controls should minimize data entry errors whenever possible. For example, if a user enters an invalid customer number, the order processing system should reject the entry immediately and prompt the user to enter a valid number. Data entry controls must be effective without being excessive. If a system requires users to confirm every item with an "Are you sure? Y/N" message, internal users and customers might complain that the system is not user-friendly.

22. Evaluation of Systems Requests

In most organizations, the IT department receives more systems requests than it can handle. Many organizations assign responsibility for evaluating systems requests to a group of key managers and users. Many companies call this group a systems review committee or a computer resources committee. Regardless of the name, the objective is to use the combined judgment and experience of several managers to evaluate systems projects.

☞ Systems Request Forms

Many organizations use a special form for systems requests. A properly designed form streamlines the request process and ensures consistency. The form must be easy to understand and include clear instructions. It should

include enough space for all required information and should indicate what supporting documents are needed. Many companies use online systems request forms that can be filled in and submitted electronically.

When a systems request form is received, a systems analyst or IT manager examines it to determine what IT resources (staff and time) are required for the preliminary investigation. A designated person or a committee then decides whether to proceed with a preliminary investigation. Occasionally a situation will arise that requires an immediate response. For example, if the problem involves a mission-critical system, an IT maintenance team would attempt to restore normal operations. When the system is functioning properly, the team conducts a review and prepares a systems request to cover the work that was performed.

☞ Systems Review Committee

Most large companies use a systems review committee to evaluate systems requests. Instead of relying on a single individual, a committee approach provides a variety of experience and knowledge. With a broader viewpoint, a committee can establish priorities more effectively than an individual, and one person's bias is less likely to affect the decisions. A typical committee consists of the IT director and several managers from other departments. The IT director usually serves as a technical consultant to ensure that committee members are aware of crucial issues, problems, and opportunities.

Although a committee offers many advantages, some disadvantages exist. For example, action on requests must wait until the committee meets. To avoid delay, committee members typically use e-mail and teleconferencing to communicate. Another potential disadvantage of a committee is that members might favor projects requested by their own departments, and internal political differences could delay important decisions.

Many smaller companies rely on one person to evaluate system requests instead of a committee. If only one person has the necessary IT skills and experience, that person must consult closely with users and managers throughout the company to ensure that business and operational needs are considered carefully.

Whether one person or a committee is responsible, the goal is to evaluate the requests and set priorities. Suppose four requests must be reviewed: a request from the marketing group to analyze current customer spending habits and forecast future trends; a request from the technical support group for a cellular link so service representatives can download technical data instantly; a request from the accounting department to redesign customer statements and allow Internet access; and a request from the production staff for an inventory control system that can exchange data with major suppliers. Which of those projects should the firm pursue? What criteria should be applied? How should priorities be determined? To answer those questions, the individual or the committee must assess the feasibility of each systems request.

23. Preliminary Investigation

A systems analyst conducts a preliminary investigation to study the systems request and recommend specific action. After obtaining an authorization to proceed, the analyst interacts with managers and users. The analyst gathers facts about the problem or opportunity, project scope and constraints, project benefits, and estimated development time and costs. The end product of the preliminary investigation is a report to management.

☞ Interaction with Managers and Users

Before beginning a preliminary investigation, a memo or an e-mail message should let people know about the investigation and explain your role. You should meet with key managers, users, and IT staff to describe the project, explain your responsibilities, answer questions, and invite comments. This starts an important dialogue with users that will continue throughout the entire development process.

When interacting with users, you should be careful in your use of the word problem, because generally it has a negative meaning. When you ask users about problems, some will stress current system limitations rather than desirable new features or enhancements. Instead of focusing on difficulties, you should question users about additional capability they would like to have. Using this approach, you highlight ways to improve the user's job, you get a better understanding of operations, and you build better, more positive relationships with users.

☞ Planning the Preliminary Investigation

Step 1: Understand the Problem or Opportunity

If the systems request involves a new information system or a substantial change in an existing system, systems analysts might need to develop a business profile that describes business processes and functions. Even where the request involves relatively minor changes or enhancements, you need to understand how those modifications will affect business operations and other information systems.

In many cases, the systems request does not reveal the underlying problem, but only a symptom. A popular technique for investigating causes and effects is called a **fishbone** diagram, or Ishikawa diagram. A fishbone diagram is an analysis tool that represents the possible causes of a problem as a graphical outline. When using a fishbone diagram, an analyst first states the problem and draws a main bone with sub-bones that represent possible causes of the problem. The **Pareto chart** is another widely used tool for visualizing and prioritizing issues that need attention.

Step 2: Define the Project Scope and Constraints

Determining the project scope means defining the specific boundaries, or extent, of the project. For example, a statement that, *payroll is not being produced accurately is very general*, compared with the statement *overtime pay is not being calculated correctly for production workers on the second shift at the Yorktown plant.* Similarly, the statement, *the project scope is to modify the accounts receivable system,* is not as specific as the statement, *the project scope is to allow customers to inquire online about account balances and recent transactions.*

Projects with very general scope definitions are at risk of expanding gradually, without specific authorization, in a process called project creep. To avoid this problem, you should define project scope as clearly as possible. You might want to use a graphical model that shows the systems, people, and business processes that will be affected. The scope of the project also establishes the boundaries of the preliminary investigation itself. A systems analyst should limit the focus to the problem at hand and avoid unnecessary expenditure of time and money.

Along with defining the scope of the project, you need to identify any constraints on the system. A constraint is a requirement or condition that the system must satisfy or an outcome that the system must achieve. A constraint can involve hardware, software, time, policy, law, or cost. System constraints also define project scope.

Step 3: Perform Fact-Finding

The objective of fact-finding is to gather data about project usability, costs, benefits, and schedules. Fact-finding involves various techniques, which are described below. Depending on what information is needed to investigate the systems request, fact-finding might consume several hours, days, or weeks. For example, a change in a report format or data entry screen might require a single telephone call or e-mail message to a user, whereas a new inventory system would involve a series of interviews. During fact-finding, you might analyze organization charts, conduct interviews, review current documentation, observe operations, and carry out a user survey.

Step 4: Analyze Project Usability, Cost, Benefit, and Schedule Data

During fact-finding, you gathered data about the project's predicted costs, anticipated benefits, and schedule issues that could affect implementation. Before you can evaluate feasibility, you must analyze this data carefully. If you conducted interviews or used surveys, you should tabulate the data to make it easier to understand. If you observed current operations, you should review the results and highlight key facts that will be useful in the feasibility analysis. If you gathered cost and benefit data, you should be able to prepare financial analysis and impact statements using spreadsheets and other decision support tools.

Step 5: Evaluate Feasibility

You have analyzed the problem or opportunity, defined the project scope and constraints, and performed fact-finding to evaluate project usability, costs, benefits, and time constraints. Now you are ready to evaluate the project's feasibility.

Step 6: Present Results and Recommendations to Management

At this stage, you have several alternatives. You might find that no action is necessary or that some other strategy, such as additional training, is needed. To solve a minor problem, you might implement a simple solution without performing further analysis. In other situations, you will recommend that the project proceed to the next development phase, which is systems analysis.

The final task in the preliminary investigation is to prepare a report to management, and possibly deliver a presentation. The report includes an evaluation of the systems request, an estimate of costs and benefits, and a case for action, which is a summary of the project request and a specific recommendation.

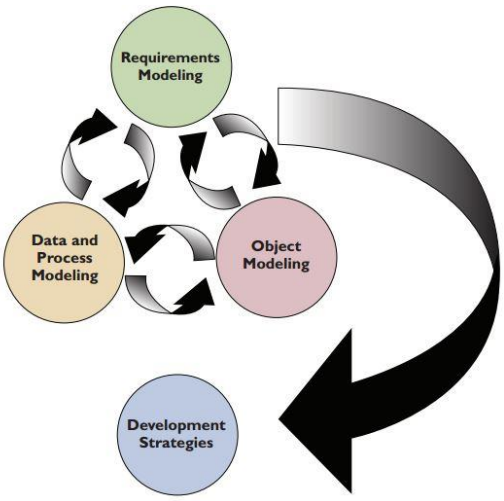
24. Systems Analysis

The overall objective of the systems analysis phase is to understand the proposed project, ensure that it will support business requirements, and build a solid foundation for system development. In this phase, you use models and other documentation tools to visualize and describe the proposed system.

Systems Analysis Activities

- **Requirement Modelling**

Requirements modeling involves fact-finding to describe the current system and identification of the requirements for the new system, such as outputs, inputs, processes, performance, and security. Outputs refer to electronic or printed information produced by the system. Inputs refer to necessary data that enters the system, either manually or in an automated manner. Processes refer to the logical rules that are applied to transform the data into meaningful information. Performance refers to system characteristics such as speed, volume, capacity, availability, and reliability. Security refers to hardware, software, and procedural controls that safeguard and protect the system and its data from internal or external threats.



- **Data and Process Modelling**

In Data and Process Modeling, you will continue the modeling process by learning how to represent graphically system data and processes using traditional structured analysis techniques.

☞ Object Modelling

Object modeling is another popular modeling technique. While structured analysis treats processes and data as separate components, object-oriented analysis (O-O) combines data and the processes that act on the data into things called objects. These objects represent actual people, things, transactions, and events that affect the system. During the system development process, analysts often use both modeling methods to gain as much information as possible.

☞ Development Strategies

Development Strategies, you will consider various development options and prepare for the transition to the systems design phase of the SDLC. You will learn about software trends, acquisition and development alternatives, outsourcing, and formally documenting requirements for the new system. The deliverable, or end product, of the systems analysis phase is a system requirements document, which is an overall design for the new system. In addition, each activity within the systems analysis phase has an end product and one or more milestones.

☞ Systems Analysis Skills

You will need strong analytical and interpersonal skills to build an accurate model of the new system. Analytical skills enable you to identify a problem, evaluate the key elements, and develop a useful solution. Interpersonal skills are especially valuable to a systems analyst who must work with people at all organizational levels, balance conflicting needs of users, and communicate effectively. Because information systems affect people throughout the company, you should consider team-oriented strategies as you begin the systems analysis phase.

25. Fact Finding Techniques

To study any system the analyst needs to do collect facts and all relevant information. The facts when expressed in quantitative form are termed as data. The success of any project is depended upon the accuracy of available data. Accurate information can be collected with help of certain methods/ techniques. These specific methods for finding information of the system are termed as fact finding techniques. Interview, Questionnaire, Record View and Observations are the different fact finding techniques used by the analyst. The analyst may use more than one technique for investigation.

☞ Interview

This method is used to collect the information from groups or individuals. Analyst selects the people who are related with the system for the interview. In this method the analyst sits face to face with the people and records their responses. The interviewer must plan in advance the type of questions he/ she is going to ask and should be ready to answer any type of question. He should also choose a suitable place and time which will be comfortable for the respondent.

The information collected is quite accurate and reliable as the interviewer can clear and cross check the doubts there itself. This method also helps gap the areas of misunderstandings and help to discuss about the future problems. Structured and unstructured are the two sub categories of Interview. Structured interview is more formal interview where fixed questions are asked and specific information is collected whereas unstructured interview is more or less like a casual conversation where in-depth areas topics are covered and other information apart from the topic may also be obtained.

☞ Questionnaire

It is the technique used to extract information from number of people. This method can be adopted and used only by a skillful analyst. The Questionnaire consists of series of questions framed together in logical manner.

The questions are simple, clear and to the point. This method is very useful for attaining information from people who are concerned with the usage of the system and who are living in different countries. The questionnaire can be mailed or send to people by post. This is the cheapest source of fact finding.

- ☞ **Record View**

The information related to the system is published in the sources like newspapers, magazines, journals, documents etc. This record review helps the analyst to get valuable information about the system and the organization.

- ☞ **Observation**

Unlike the other fact finding techniques, in this method the analyst himself visits the organization and observes and understand the flow of documents, working of the existing system, the users of the system etc. For this method to be adopted it takes an analyst to perform this job as he knows which points should be noticed and highlighted. In analyst may observe the unwanted things as well and simply cause delay in the development of the new system.

26. References

These notes are outline based of BSCS-Term IV of University Of Sargodha by Talha Shahab. The material is fetched from the following mentioned sources and copied only for educational purpose.

1. Software Engineering: A Practitioner's Approach by Roger S. Pressman; McGraw-Hill 7th Edition ISBN: 978–0–07–337597–7.
2. Systems Analysis and Design by Gary B. Shelly, Harry J. Rosenblatt 8th Edition ISBN-10: 0-324-59766-5.
3. https://www.geeksforgeeks.org/software-engineering/
4. https://www.tutorialspoint.com/software_engineering/
5. And some other websites. ☺

The End