UNIT-1

Introduction to Software Engineering

Software engineering is a discipline in which theories, methods and tools are applied to develop professional software. (Or) Software engineering is the systematic approach to develop and maintain a software product in a cost effective and efficient way.

Characteristics of software

- Software is engineered or developed, it is not manufactured in the classical sense.
- Software doesn't wear out.
- Although the industry is moving toward component based assembly, most software continues to be custom built.

Goals of Software Engineering

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. The goals of software engineering are:

- Software production which consists of developed programs and associated documentation.
- The software product should have the essential product attributes maintainability, dependability, efficiency and acceptability.
- It should also include suggestions for the process to be followed, the notations to be used, system models to be developed and rules governing these models and design guidelines.

The two fundamental types of software product are

Generic products: These are standalone systems developed by organizations and sold on open market to any customer who is able to buy them.

Customized products: These are systems which are commissioned by a particular customer. A software contractor develops the software especially for that customer.

Generic process framework

A process framework establishes the foundation for a complete software 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. Each framework activity is populated by a set of software engineering actions-a collection of related tasks that produces a major software engineering work product. Each action is populated with individual work tasks that accomplish some part of the work implied by the action.

The following generic process framework is applicable to vast majority of software projects:

Communication: This framework activity involves heavy communication and collaboration with the customer and encompasses requirements gathering and other related activities.

Planning: This activity establishes a plan for the software engineering work that follows. It describes 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.

Modeling: The activity encompasses the creation of models that allow the developer and the customer to better understand software requirements and the design that will achieve those requirements.
Construction: This activity combines code generation and the testing that is required to uncover errors in the code.
Deployment: The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.
The modeling activity is composed of two software engineering actions:

- **Analysis** encompasses a set of work tasks requirements gathering, elaboration, negotiation, specification and validation that lead to the creation of the analysis model or requirements specification.
- **Design** encompasses work tasks data design, architectural design, interface design and component-level design and create a design model or design specification.

Each software engineering action is 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. The task set that best accommodates the needs of the project and characteristics of the team is chosen. The framework described in the generic view of software engineering is completed by a number of umbrella activities. Typical activities in this category include:
Software Project Tracking and Control-allows the software team to assess progress against the project plan and take the necessary action to maintain schedule.
Risk Management-assess the 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.
Formal Technical Reviews-assesses software engineering work products in an effort to uncover or remove errors before they are propagated to the next action or activity.
Measurement-defines and collects process, project and product measures that assist the team in delivering software that meets customer needs.
Software Configuration Management-manages the effects of change throughout the software process.
Reusability Management-defines criteria for work product reuse and establish 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. All process models can be categorized within the process framework discussed. But process models do differ fundamentally in:

- The overall flow of activities and tasks and the interdependencies among activities and tasks.
- The degree to which work tasks are defined within each framework activity.
- The degree to which work products are identified and required.
- The manner which quality assurance activities are applied.
- The manner in which project tracking and control activities are applied.
- The overall degree of detail and rigor with which the process is described.
- The degree to which customer and other stakeholders are involved within the project.
- The level of autonomy given to the software project team.
- The degree to which team organization and roles are prescribed.

CHARACTERISTICS OF SOFTWARE

1. Software is developed or engineered; it is not manufactured in the classical sense. Software costs are concentrated in engineering. this means that software projects cannot be managed as if they were manufacturing projects.

2. Software doesn‘t —wear out.

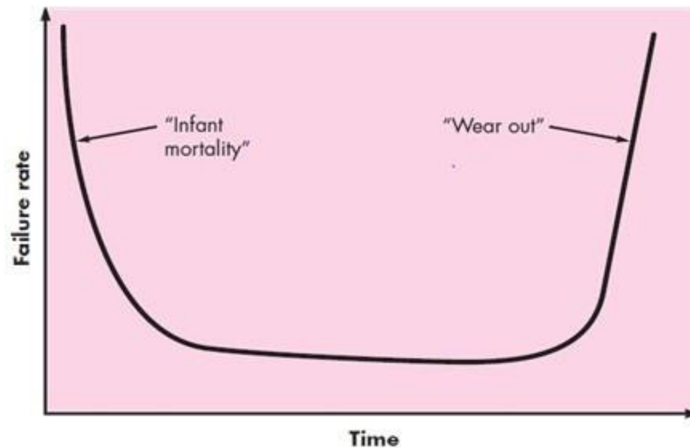
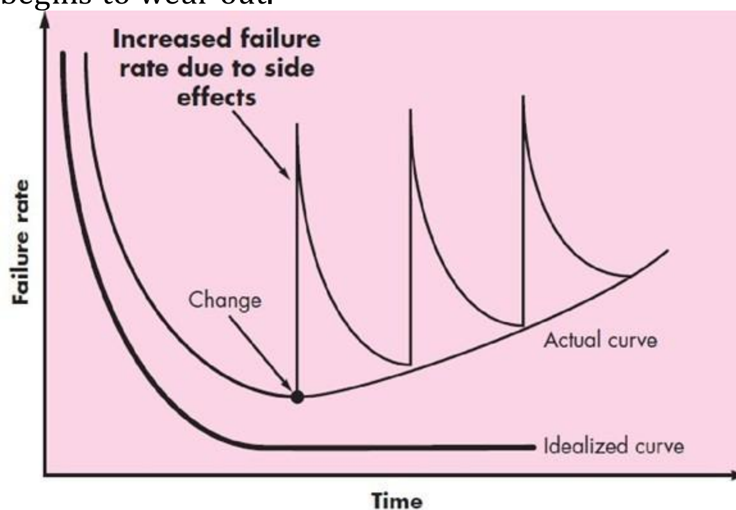


Figure depicts failure rate as a function of time for hardware. The relationship, often called the —bathtub curve, indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the 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.



Considering the time curve ,software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the actual curve. Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

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.

CMMI

The CMMI represents a process meta-model in two different ways: (1) as a continuous model and (2) as a ―staged model. Each process area (e.g., project planning or requirements management) is formally assessed against specific goals and practices and is rated according to the following capability levels:

Level 0: Incomplete—the process area (e.g., requirements management) is either not performed or does not achieve all goals and objectives defined by the CMMI for level 1 capability for the process area.

Level 1: Performed—all of the specific goals of the process area (as defined by the CMMI) have -been satisfied. Work tasks required to produce defined work products are being conducted.

Level 2: Managed—all capability level 1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy; all people doing the work have access to adequate resources to get the job done; stakeholders are actively involved in the process area as required; all work tasks and work products are ―monitored, controlled, and reviewed; and are evaluated for adherence to the process description‖ .

Level 3: Defined—all capability level 2 criteria have been achieved. In addition, the process is ―tailored from the organization's set of standard processes according to the organization's tailoring guidelines, and contributes work products, measures, and other process-improvement information to the organizational process assets‖ .

Level 4: Quantitatively managed—all capability level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment. ―Quantitative objectives for quality and process performance are established and used as criteria in managing the process.

Level 5: Optimized—all capability level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative (statistical) means to meet changing customer needs and to continually improve the efficacy of the process area under consideration.

The CMMI defines each process area in terms of ―specific goals and the ―specific practices required to achieve these goals. *Specific goals* establish the characteristics that must exist if the activities implied by a process area are to be effective. *Specific practices* refine a goal into a set of process-related activities.

A PROCESS FRAMEWORK

A process framework establishes the foundation for a complete software 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.

Each framework activity is populated by a set of software engineering actions-a collection of related tasks that produces a major software engineering work product.

Each action is populated with individual work tasks that accomplish some part of the work implied by the action.

The following generic process framework is applicable to vast majority of software projects:

Communication: this framework activity involves heavy communication and collaboration with the customer and encompasses requirements gathering and other related activities.

Planning: this activity establishes a plan for the software engineering work that follows. It describes 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.

Modeling: the activity encompasses the creation of models that allow the developer and the customer to better understand software requirements and the design that will achieve those requirements.

Construction: this activity combines code generation and the testing that is required to uncover errors in the code.
Deployment: the software is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation. The modeling activity is composed of two software engineering actions:

- **Analysis** encompasses a set of work tasks requirements gathering, elaboration, negotiation, specification and validation that lead to the creation of the analysis model or requirements specification.

- **Design** encompasses work tasks data design, architectural design, interface design and component-level design and create a design model or design specification. Each software engineering action is 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. The task set that best accommodates the needs of the project and characteristics of the team is chosen. The framework described in the generic view of software engineering is completed by a number of umbrella activities. Typical activities in this category include:
  - to create work products such as models, documents, logs, forms and lists. All process models can be categorized within the process framework discussed. But process models do differ fundamentally in:
  - The overall flow of activities and tasks and the interdependencies among activities and tasks.
  - The degree to which work tasks are defined within each framework activity.
  - The degree to which work products are identified and required.
  - The manner which quality assurance activities are applied.
  - The manner in which project tracking and control activities are applied.
  - The overall degree of detail and rigor with which the process is described.
  - The degree to which customer and other stakeholders are involved within the project.
    The level of autonomy is given to the software project team.

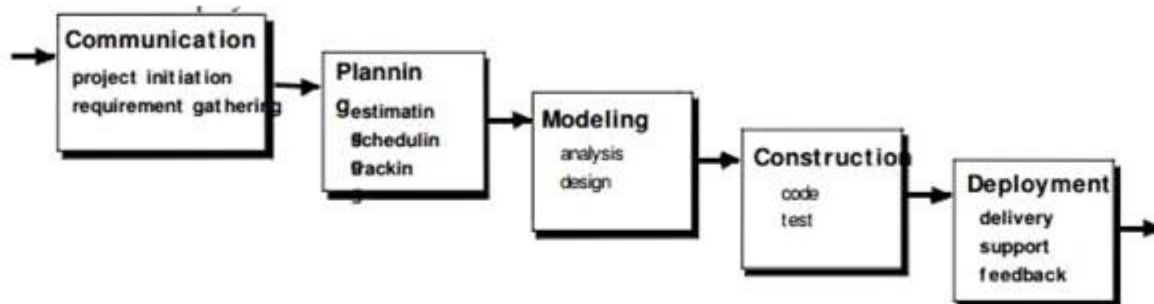
WATERFALL MODEL

When the requirements of a problem are reasonably well understood-when work flows from communication through deployment in a reasonably linear fashion. This situation is encountered when

- Well defined adaptations or enhancements to an existing system must be made.
- In limited number of new development efforts, but only when requirements are well defined and reasonably stable.

The waterfall model, called the classic life cycle suggests a systematic, sequential approach to software development.

- Begins with customer specification of requirements
- Progresses through planning
- Modeling
- Construction
- Deployment



THE WATERFALL MODEL

Problems encountered when Waterfall Model is applied:

- Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.

- It is often difficult for the customer to state all the requirements explicitly. The waterfall model requires this and has difficulty accommodating uncertainty that exists at the beginning of many projects.

- The customer must have patience. A working version of the program will not be available until late in the project life-span. A major blunder, if undetected until the working program is reviewed can be disastrous.

Linear structure of the waterfall model leads to blocking states in which some project team members must wait for other members of the team to complete dependent tasks. Blocking states are more prevalent at the beginning and end of the linear sequential process.

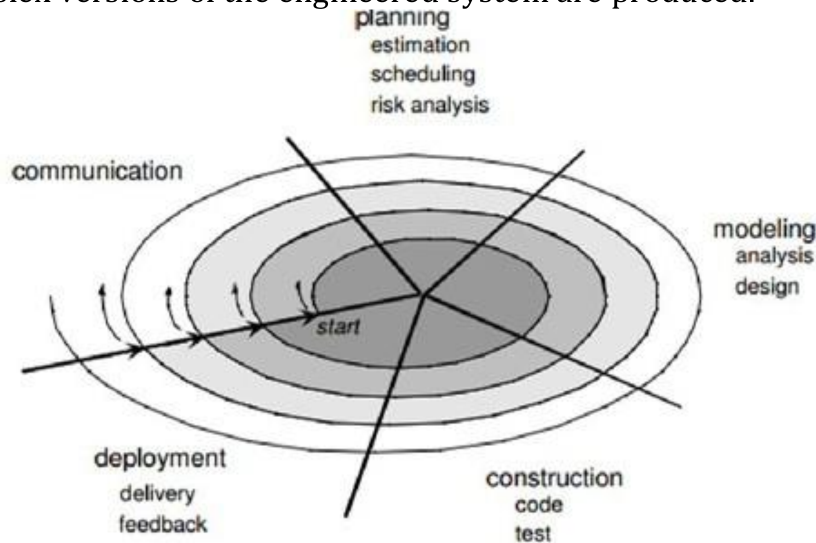
SPIRAL MODEL

The spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complex versions of the software. Boehm definition:

The spiral development model is a risk-driven process model generator. It has two main distinguishing features:

- One is a cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk.
- The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

Using spiral model, software is developed in a series of evolutionary releases. During early iterations, the release is a paper model or prototype. During later iterations, more complex versions of the engineered system are produced.



A TYPICAL SPIRAL MODEL

A spiral model is divided into a set of framework activities defined by the software engineering team. Each of the framework activities represents one segment of the spiral path as shown in the figure.

- As evolution begins the software team performs activities implied by the by the circuit around the spiral, in clockwise direction, beginning at the center.

- Risk is considered at each revolution made.

- Anchor point milestones-a combination of work products and conditions that are attained along the path of the spiral are noted for each evolutionary pass.

- First circuit around the spiral results in the development of product specifications.

- Subsequent passes develop a prototype, progressively lead to sophisticated version of the software.

- and schedule are adjusted based on feedback derived from the customer after delivery.

- Unlike other process models that end when the software is delivered, the spiral model can be adapted throughout the life of the software.

- The first spiral represents a Concept Development Project which starts at core and continues for multiple iterations until concept development is implemented.

- The concept developed to an actual product, proceeds outward on the spiral and a New Product Development Project commences.
- The new product may evolve to represent Product Enhancement Project.

Features of Spiral Model:

- Realistic approach to the development of large scale-systems and software.

- Because the software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level.

- Enables the developer to apply the prototyping approach at any stage in the evolution of the product.

- It maintains the systematic stepwise approach of classic life cycle but in incorporatesit into an iterative framework that realistically reflects the real world.

- It demands direct consideration of technical risks at all stages of the project and if applied properly, reduce risks before they become problematic.

RAD MODEL

Rapid Application Development (RAD) is an incremental software process model that emphasizes a short development cycle. The RAD model is a high-speed adaptation of the waterfall model, in which rapid development is achieved by using component-based construction approach. If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a fully functional system within a very short time period.
Like other process models, RAD approach maps to the generic framework activities as

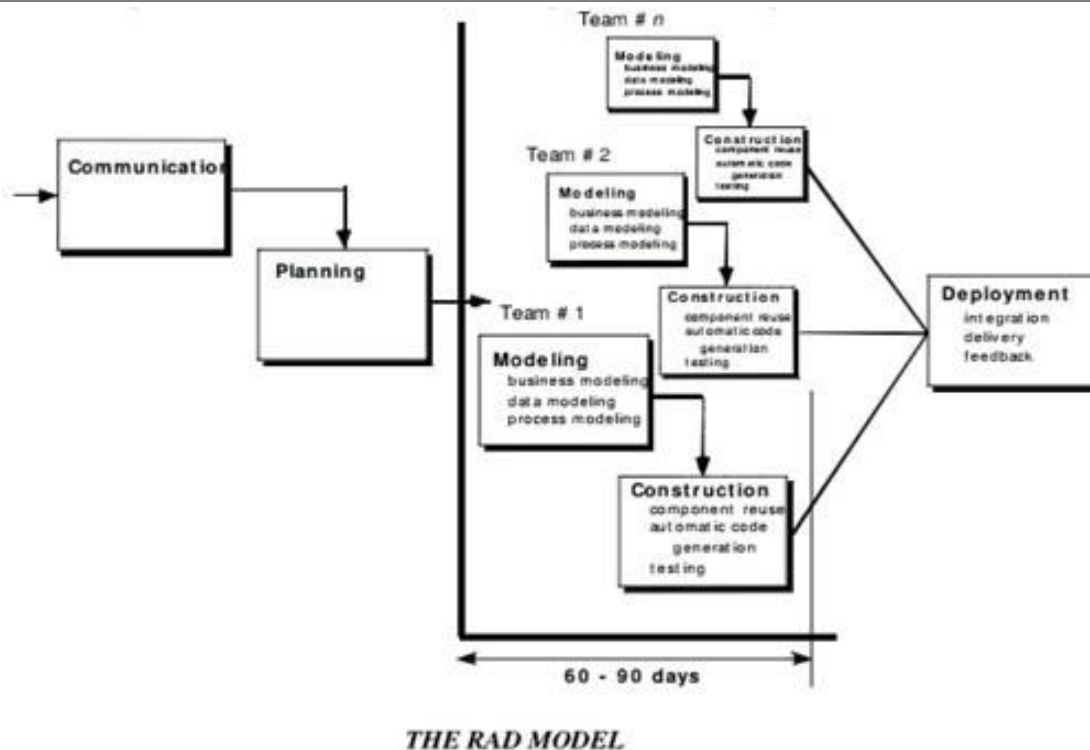
- Communication, works to understand the business problem and the information characteristics that the software must accommodate.

- Planning, is essential because multiple software teams work in parallel on different system functions.

- Modeling encompasses three major phases-business modeling, data modeling and process modeling- and establishes design representations that serve as the basis for RAD‘s construction activity.

- Construction emphasizes the use of pre-existing software components and the application of automatic code generation.

- Deployment, establishes a basis for subsequent iterations, if required.



THE RAD MODEL

If a business application can be modularized in a way that enables each major function to be completed in less than three months, it is a candidate for RAD. Each major function can be addressed by a separate RAD team and then integrated to form a whole.

RAD Drawbacks

- For large, but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.

- If developers and customers are not committed to the rapid fire activities, RAD project fails.

- If a system is not properly modularized, building the components for RAD will be problematic.

  - If high performance is an issue and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work.
  - Not appropriate when technical risks are high or when new technology is used.

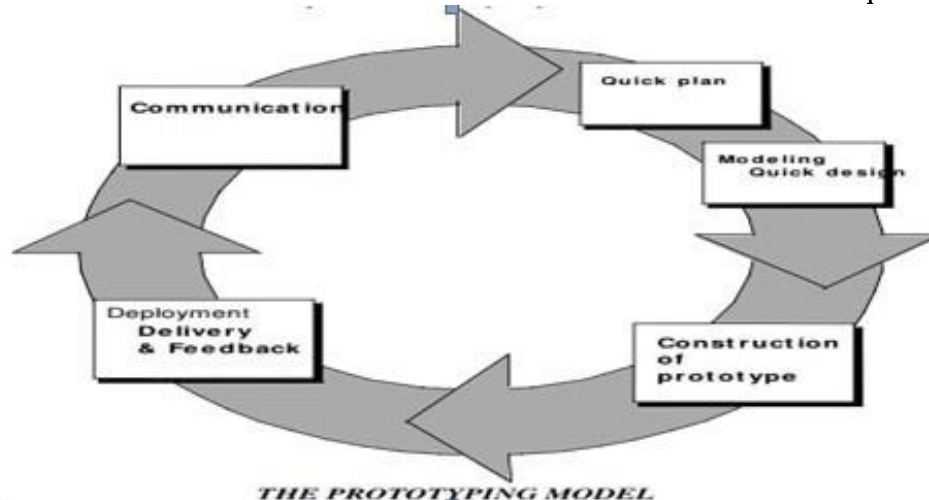
PROTOTYPING MODEL

The customer usually defines a set of general objectives for software, but does not identify detailed input, processing or output requirements. The developer may be unsure of

- The efficiency of an algorithm
- The adaptability of an operating system
- The form that human-machine interaction should take

In these and many other situations, a prototyping paradigm may offer the best approach. Although prototyping can be used as a standalone process model, it is more

often used as a technique that can be implemented within the context of any one of the other process models. The prototyping paradigm assists the software engineer and the customer to better understand what is to be built when the requirements are fuzzy.



THE PROTOTYPING MODEL

- The prototyping paradigm begins with communication. The software engineer and the customer meet and
  - Define the overall objectives for the software
  - Outline areas whether further definition is mandatory
  - Identify whatever requirements are known
- The quick design leads to construction of a prototype.
- The prototype is deployed and then evaluated by customer/user. The feedback is used to refine requirements for the software.
- Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.
- Prototyping serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to make use of existing program fragments or applies tools that enable working programs to be generated quickly.

- The prototype can serve as the first system. Both customers and developers like the prototyping paradigm. Users get a feel for the actual system and developers get to build something immediately.

Problems with Prototyping

The customer sees what appears to be a working version of the software, unaware that in the rush to get it working, the quality or maintainability is not considered. When informed that the software has to be rebuilt, demands for a few fixes to make it a working product.

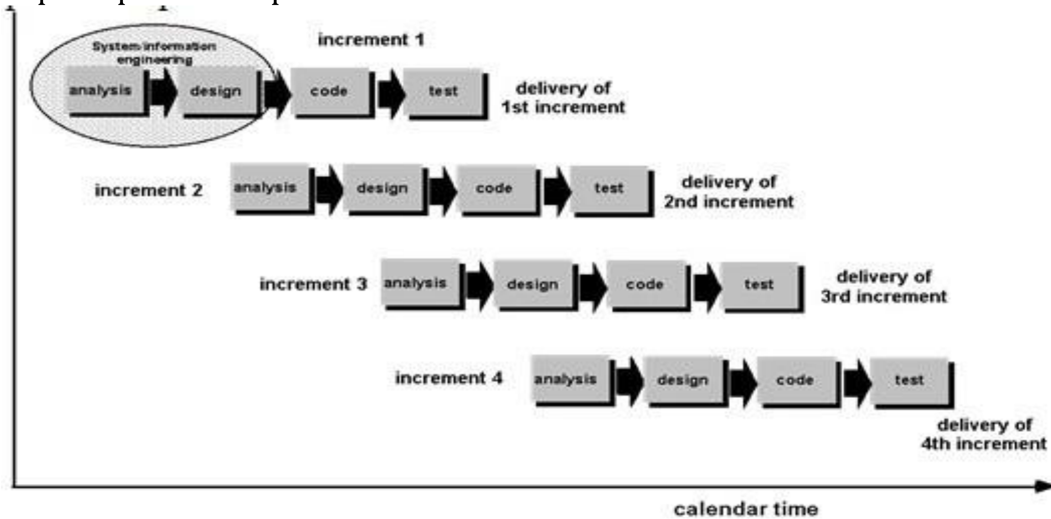
The developer often makes implementation compromises to make the prototype work quickly. An inappropriate OS or programming language is used, but the developer forgets why these choices are inappropriate and gets comfortable with the system. The less-than-ideal choice has now become an integral part of the system.

Incremental models

The incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable —increments of the software in a manner that is similar to the increments produced by an evolutionary process flow.

- When an incremental model is used, the first increment is often a core product.

- 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 model:

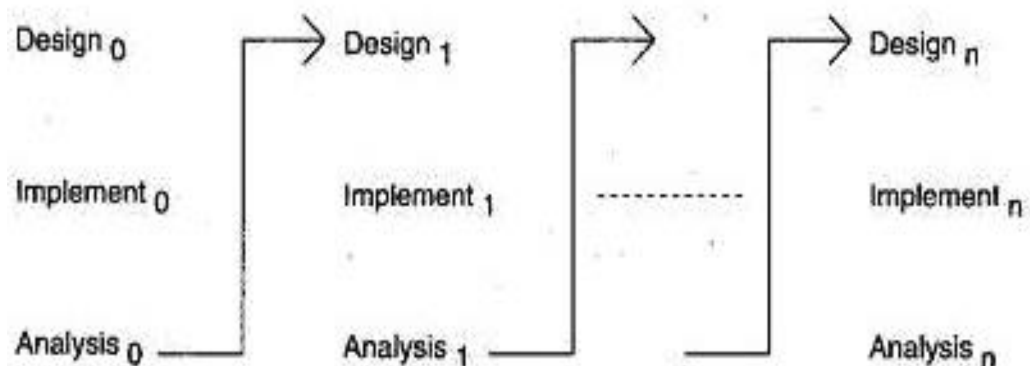
- Generates working software quickly and early during the software life cycle.
- This model is more flexible – less costly to change scope and requirements.
- It is easier to test and debug during a smaller iteration.
- In this model customer can respond to each built.
- Lowers initial delivery cost.
- Easier to manage risk because risky pieces are identified and handled during it‘d iteration.

Disadvantages of Incremental model

- Needs good planning and design.
- Needs a clear and complete definition of the whole system before it can be broken down and built incrementally.
- Total cost is higher than waterfall.

Iterative models

An iterative life cycle model does not attempt to start with a full specification of requirements. Instead, development begins by specifying and implementing just part of the software, which can then be reviewed in order to identify further requirements. This process is then repeated, producing a new version of the software for each cycle of the model.



Advantages of Iterative model

- In iterative model we can only create a high-level design of the application before we actually begin to build the product and define the design solution for the entire product. Later on we can design and built a skeleton version of that, and then evolved the design based on what had been built.

- In iterative model we are building and improving the product step by step. Hence we can track the defects at early stages. This avoids the downward flow of the defects.

- In iterative model we can get the reliable user feedback. When presenting sketches and blueprints of the product to users for their feedback, we are effectively asking them to imagine how the product will work.

- In iterative model less time is spent on documenting and more time is given for designing.

Disadvantages of Iterative model

- Each phase of an iteration is rigid with no overlaps
- Costly system architecture or design issues may arise because not all requirements are gathered up front for the entire lifecycle.

THE CONCURRENT DEVELOPMENT MODEL

The Concurrent Development Model also called Concurrent Engineering, is represented as a series of

- Framework activities
- Software engineering actions and tasks
- And their associated states

The figure provides a representation of one software engineering task within the modeling activity for the concurrent process model. The activity-modeling-may be in any one of the states noted at a given time. Similarly other activities or tasks can be represented in an analogous manner. All activities exist concurrently but reside in different states. For example

- Early in a project the communication activity has completed its first iteration and exists in the awaiting changes state.
- The modeling activity which exists in the none state while initial communication was completed, now makes a transition into the under development state.
- If customer indicates changes in the requirements, the modeling activity moves from the under development state into the awaiting changes state.

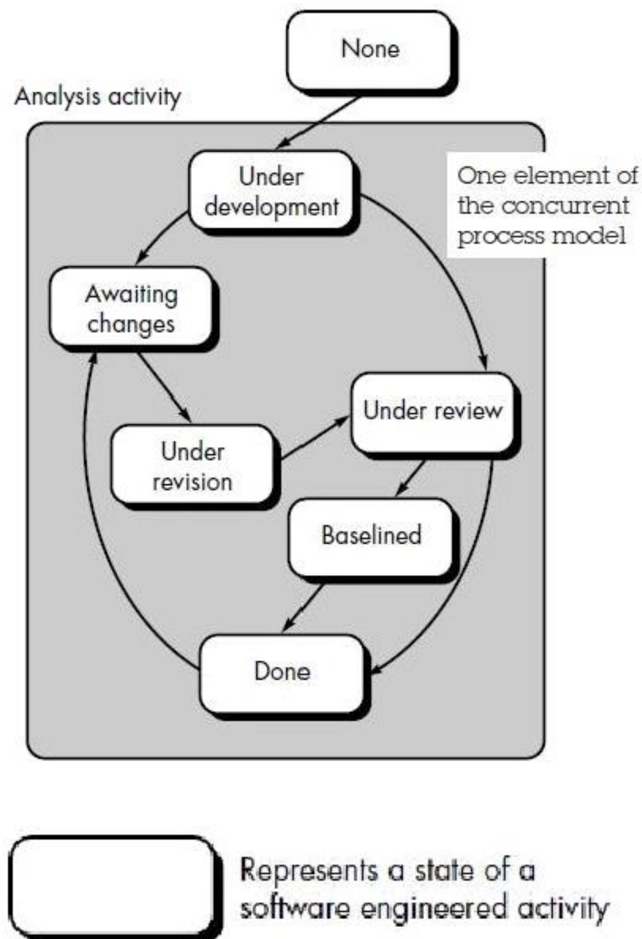
The concurrent process model defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions or tasks. The concurrent process model is applicable to all types of software development and provides an accurate picture of the current state of a project. This model defines a network of activities. Events triggered at one point in the process network trigger transitions among the states.

The Concurrent Development Model also called Concurrent Engineering, is represented as a series of

- Framework activities
- Software engineering actions and tasks
- And their associated states

The figure provides a representation of one software engineering task within the modeling activity for the concurrent process model. The activity-modeling-may be in any one of the states noted at a given time. Similarly other activities or tasks can be represented in an analogous manner. All activities exist concurrently but reside in different states.

For example, Early in a project the communication activity has completed its first iteration and exists in the awaiting changes state. The modeling activity which exists in the none state while initial communication was completed, now makes a transition into the under development state. If customer indicates changes in the requirements, the modeling activity moves from the under development state into the awaiting changes state. The concurrent process model defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions or tasks. The concurrent process model is applicable to all types of software development and provides an accurate picture of the current state of a project. This model defines a network of activities. Events triggered at one point in the process network trigger transitions among the states.



Specialized Process Models

Component-Based Development

The component-based development model incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software. However, the component-based development model constructs applications from pre-packaged software components.

The component-based development model incorporates the following steps:

- Available component-based products are researched and evaluated for the application domain in question.
- Component integration issues are considered.
- A software architecture is designed to accommodate the components.
- Components are integrated into the architecture.
- Comprehensive testing is conducted to ensure proper functionality.

Formal Methods Model

The formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software.

- 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.

Aspect-oriented software development

Aspect-oriented software development (AOSD), often referred to as aspect-oriented programming (AOP), is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing, and constructing aspects—mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern.

Unified Process Model

The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer‘s view of a system. It emphasizes the important role of software architecture and ―helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse. It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

Phases of Unified Process Model

The **inception phase** of the UP encompasses both customer communication and planning activities. By collaborating with stakeholders, business requirements for the software are identified; a rough architecture for the system is proposed; and a plan for the iterative, incremental nature of the ensuing project is developed. Fundamental business requirements are described through a set of preliminary use cases that describe which features and functions each major class of users desires.

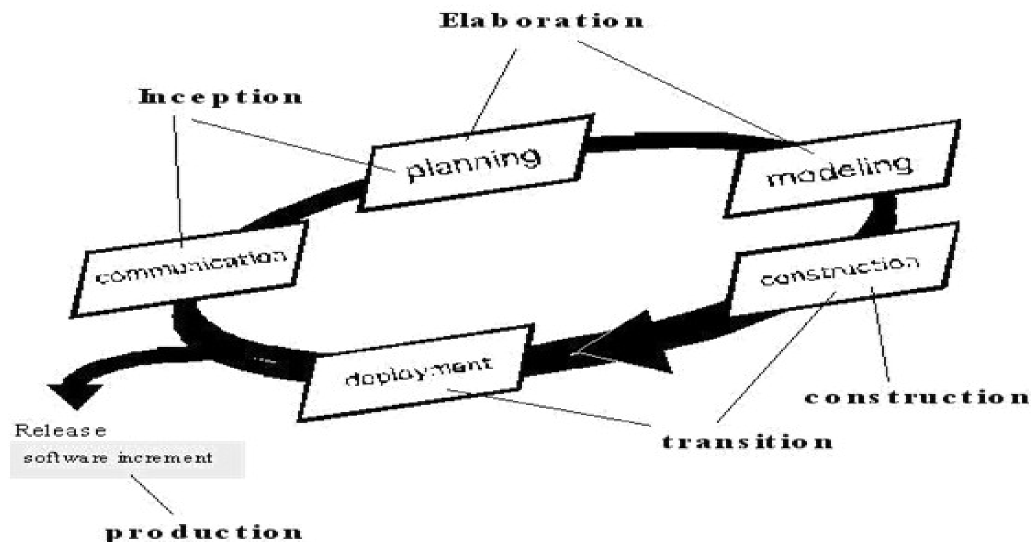
The **elaboration phase** encompasses the communication and modeling activities of the generic process model. Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software—the use case model, the

requirements model, the design model, the implementation model, and the deployment model.

The **construction phase** of the UP is identical to the construction activity defined for the generic software process. Using the architectural model as input, the construction phase develops or acquires the software components that will make each use case operational for end users.

The **transition phase** of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback) activity. Software is given to end users for beta testing and user feedback reports both defects and necessary changes.

The **production phase** of the UP coincides with the deployment activity of the generic process. During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.



Agile planning

Agile methods of software development are iterative approaches where the software is developed and delivered to customers in increments. Unlike plan-driven approaches, the functionality of these increments is not planned in advance but is decided during the development. The decision on what to include in an increment depends on progress and on the customer's priorities.
The most commonly used agile approaches such as Scrum (Schwaber, 2004) and extreme programming (Beck, 2000) have a two-stage approach to planning, corresponding to the startup phase in plan-driven development and development planning:
1. Release planning, which looks ahead for several months and decides on the features that should be included in a release of a system.
2. Iteration planning, which has a shorter-term outlook, and focuses on planning the next increment of a system. This is typically 2 to 4 weeks of work for the team.

Planning in XP

The system specification in XP is based on user stories that reflect the features that should be included in the system. At the start of the project, the team and the customer try to identify a set of stories, which covers all of the functionality that will be included in the final system. Some functionality will inevitably be missing, but this is not important at this stage.
The next stage is an estimation stage. The project team reads and discusses the stories and ranks them in order of the amount of time they think it will take to implement the story. This may involve breaking large stories into smaller stories. Relative estimation is often easier than absolute estimation. People often find it difficult to estimate how much effort or time is needed to do something. However, when they are presented with several things to do, they can make judgments about which stories will take the longest time and most effort. Once the ranking has been completed, the team then allocates notional effort points to the stories.
Each developer knows their individual velocity so should not sign up for more tasks than they can implement in the time.
There are two important benefits from this approach to task allocation:
1. The whole team gets an overview of the tasks to be completed in an iteration. They therefore have an understanding of what other team members are doing and who to talk to if task dependencies are identified.
2. Individual developers choose the tasks to implement; they are not simply allocated tasks by a project manager. They therefore have a sense of ownership in these tasks and this is likely to motivate them to complete the task.

In some agile methods, such as extreme programming, customers are directly involved in deciding whether a change should be implemented. When they propose a change to the system requirements, they work with the team to assess the impact of that change and then decide whether the change should take priority over the features planned for the next increment of the system. However, changes that involve software improvement are left to the discretion of the programmers working on the system. Refactoring, where the software is continually improved, is not seen as an overhead but rather as a necessary part of the development process. As the development team changes software components, they should maintain a record of the changes made to each component.

UNIT - 2

Software Requirements The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed

Requirements may be functional or non-functional.

- Functional requirements describe system services or functions
- Non-functional requirements is a constraint on the system or on the development process

Functional Requirements

- Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
- Describe functionality or system services
- Depend on the type of software, expected users and the type of system where the software is used
- Functional user requirements may be high-level statements of what the system should do but functional system requirements should describe the system services in detail.

Examples of functional requirements

- The user shall be able to search either all of the initial set of databases or select asubset from it.
- The system shall provide appropriate viewers for the user to read documents in the document store.

Requirements completeness and consistency:

Complete

- They should include descriptions of all facilities required

Consistent

- There should be no conflicts or contradictions in the descriptions of the system facilities

In practice, it is impossible to produce a complete and consistent requirements document

Non-functional requirements

- Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
- Define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- Process requirements may also be specified mandating a particular CASE system, programming language or development method

Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless.

Non-functional Classification

Product requirements

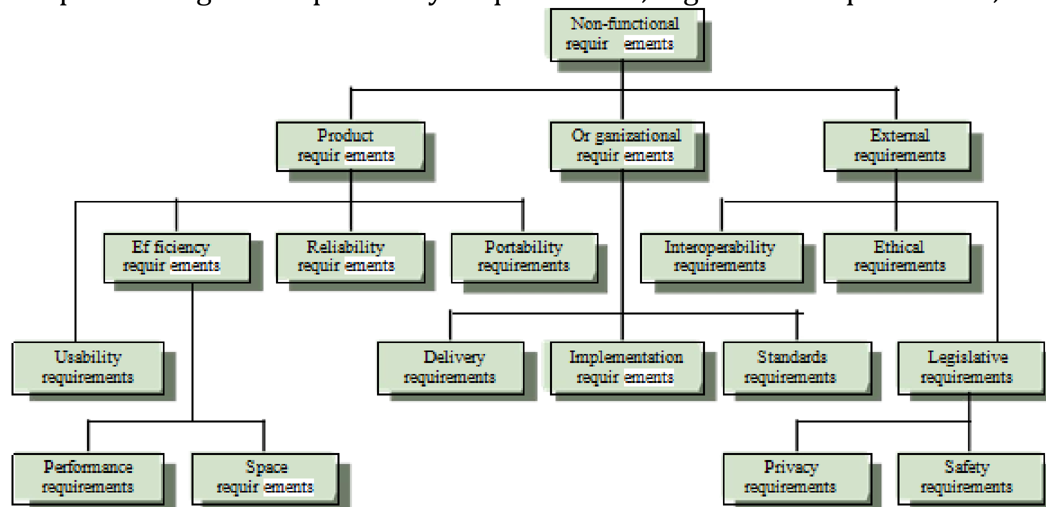
- Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.

Organizational requirements

- Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.

External requirements

Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.



Examples of Non-Functional Requirements

Product Requirement

- It shall be possible for all necessary communication between the APSE and the user to be expressed in the standard Ada character set.

Organizational Requirement

- The system development process and deliverable documents shall conform to the process and deliverables in software organizations.

External Requirement

- The system shall not disclose any personal information about customers apart from their name and reference number to the operators of the system.

Goals and Requirements

Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify.

Goal

- A general intention of the user such as ease of use Verifiable non-functional requirement
- A statement using some measure that can be objectively tested. Goals are helpful to developers as they convey the intentions of the system users.

Speed	Processed transactions /second
	response time Screen refresh time
Size	K Bytes Number of RAM chips
Ease of Use	Training time
	Number of help frames
Reliability	Mean time to failure
	Probability of unavailability
	Rate of failure occurrence
	Availability
Robustness	Time to restart after failure
	Percentage of events causing failure
	Probability of data corruption on failure
Probability	Percentage of target dependent statement
	Number of target systems

User Requirements

User requirements are statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate.

The user requirements for a system should describe the functional and nonfunctional requirements so that they are understandable by system users who don't have detailed technical knowledge. Ideally, they should specify only the external behavior of the system.

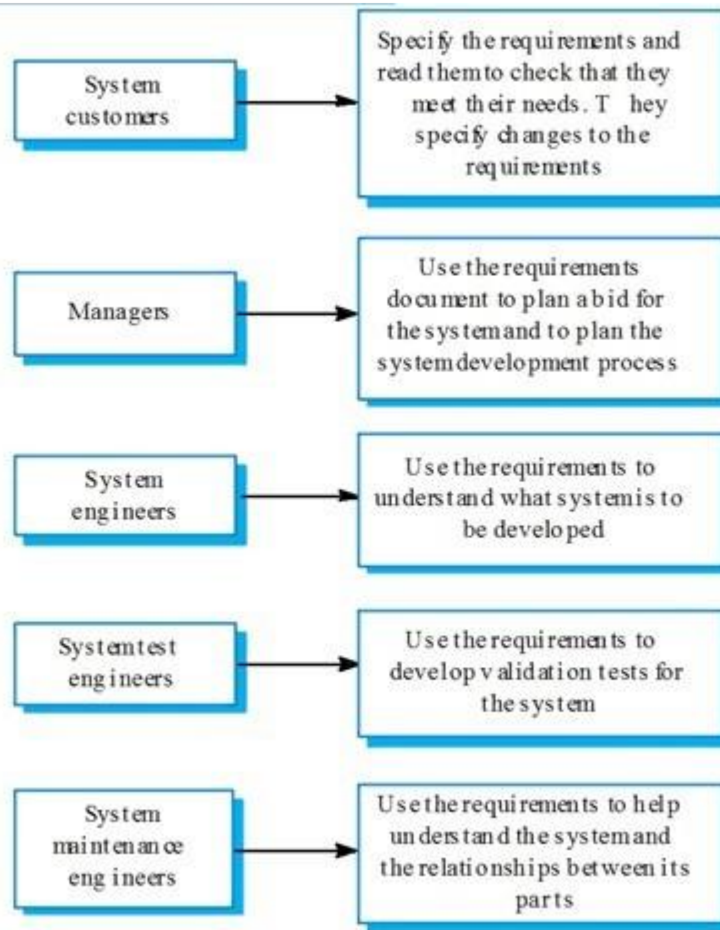
System Requirements

System requirements are more detailed descriptions of the software system's functions, services, and operational constraints. The system requirements document (sometimes called a functional specification) should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.

System requirements are expanded versions of the user requirements that are used by software engineers as the starting point for the system design. They add detail and explain how the user requirements should be provided by the system. They may be used as part of the contract for the implementation of the system and should therefore be a complete and detailed specification of the whole system.

Software Requirements Document

- The requirements document is the official statement of what is required of the system developers.
- Should include both a definition of user requirements and a specification of the system requirements.
- It is NOT a design document. As far as possible, it should set out WHAT the system should do rather than HOW it should do it



IEEE Standard

- Defines a generic structure for a requirements document that must be instantiated for each specific system.
  - Introduction.
  - General description.
  - Specific requirements.
  - Appendices.
  - Index.
- Requirement Document Structure
  - Preface
  - Introduction
  - Glossary
  - User requirements definition
  - System architecture
  - System requirements specification
  - System models
  - System evolution
  - Appendices
  - Index

REQUIREMENT ENGINEERING

Requirement Engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification and managing the requirements as they are transformed into an operational system.

Guidelines Principles for Requirement Engineering

- Understand the problem before beginning the analysis model.
- Develop prototypes that enable a user to understand how human/machine interaction will occur.
- Record the origin of and the reason for each and every requirements.
- Use multiple views of requirements.
- Rank the requirements and eliminate the ambiguity.

REQUIREMENT ENGINEERING PROCESS:

Inception

During inception, the requirements engineer asks a set of questions to establish…

- A basic understanding of the problem
- The people who want a solution
- The nature of the solution that is desired
- The effectiveness of preliminary communication and collaboration between the customer and the developer

Elicitation

Elicitation may be accomplished through two activities

- ✓ Collaborative requirements gathering
- ✓ Quality function deployment

Elaboration

- During elaboration, the software engineer takes the information obtained during inception and elicitation and begins to expand and refine it
- Elaboration focuses on developing a refined technical model of software functions, features, and constraints

Negotiation

- During negotiation, the software engineer reconciles the conflicts between what the customer wants and what can be achieved given limited business resources
- Requirements are ranked (i.e., prioritized) by the customers, users, and other stakeholders
- Risks associated with each requirement are identified and analyzed

Specification

A specification is the final work product produced by the requirements engineer

- It is normally in the form of a software requirements specification
- It serves as the foundation for subsequent software engineering activities

It describes the function and performance of a computer-based system and the constraints that will govern its development

Validation

- During validation, the work products produced as a result of requirements engineering are assessed for quality
- The specification is examined to ensure that
- all software requirements have been stated unambiguously
- inconsistencies, omissions, and errors have been detected and corrected
- the work products conform to the standards established for the process, the project, and the product

The formal technical review serves as the primary requirements validation mechanism

- Members include software engineers, customers, users, and other stakeholders

Requirements Management

- During requirements management, the project team performs a set of activities to identify, control, and track requirements and changes to the requirements at any time as the project proceeds
- Each requirement is assigned a unique identifier
- The requirements are then placed into one or more traceability tables.

FEASIBILITY STUDY

The aims of a feasibility study are to find out whether the system is worth implementing and if it can be implemented, given the existing budget and schedule.

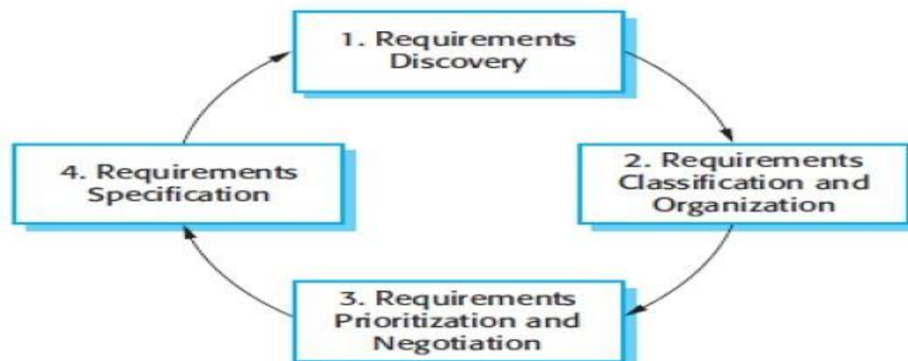
The purpose of feasibility study is not to solve the problem, but to determine whether the problem is worth solving. This helps to decide whether to proceed with the project or not.

The input to the feasibility study is a set of preliminary business requirements, an outline description of the system and how the system is intended to support business processes. The results of the feasibility study should be a report that recommends whether or not it is worth carrying on with the requirements engineering and system development process.

Issues addressed by feasibility study

- Gives focus to the project and outline alternatives.
- Narrows business alternatives
- Identifies new opportunities through the investigative process.
- Identifies reasons not to proceed.
- Enhances the probability of success by addressing and mitigating factors early on that could affect the project.
- Provides quality information for decision making.
- Provides documentation that the business venture was thoroughly investigated.
- Helps in securing funding from lending institutions and other monetary sources.
- Helps to attract equity investment.
- The feasibility study is a critical step in the business assessment process. If properly conducted, it may be the best investment you ever made Carrying out a feasibility study involves information assessment, information collection and report writing.

REQUIREMENTS ELICITATION AND ANALYSIS



The process activities are:

Requirements discovery: This is the process of interacting with stakeholders of the system to discover their requirements. Domain requirements from stakeholders and

documentation are also discovered during this activity.

Requirements classification and organization: This activity takes the unstructured collection of requirements, groups related requirements, and organizes them into coherent clusters. The most common way of grouping requirements is to use a model of the system architecture to identify sub-systems and to associate requirements with each sub-system.

Requirements prioritization and negotiation: Inevitably, when multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation.

Requirements specification: The requirements are documented and input into the next round of the spiral.

Eliciting and understanding requirements from system stakeholders is a difficult process for several reasons:

- Stakeholders often don't know what they want from a computer system except in the most general terms; they may find it difficult to articulate what they want the system to do; they may make unrealistic demands because they don't know what is and isn't feasible.
- Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers, without experience in the customer's domain, may not understand these requirements.
- Different stakeholders have different requirements and they may express these in different ways. Requirements engineers have to discover all potential sources of requirements and discover commonalities and conflict.
- Political factors may influence the requirements of a system. Managers may demand specific system requirements because these will allow them to increase their influence in the organization.
- The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. The importance of particular requirements may change. New requirements may emerge from new stakeholders who were not originally consulted.

Requirements discovery

Requirements discovery (sometime called requirements elicitation) is the process of gathering information about the required system and existing systems, and distilling the user and system requirements from this information.

Sources of information during the requirements discovery phase include documentation, system stakeholders and specifications of similar systems.

Stakeholders range from end-users of a system through managers to external stakeholders such as regulators, who certify the acceptability of the system.

For example, system stakeholders for the mental healthcare patient information system include:

- Patients whose information is recorded in the system.
- Doctors who are responsible for assessing and treating patients.
- Nurses who coordinate the consultations with doctors and administer some treatments.
- Medical receptionists who manage patients' appointments.
- IT staff who are responsible for installing and maintaining the system.
- A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
- Healthcare managers who obtain management information from the system.
- Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been

properly implemented.

Interviewing

The requirements engineering team puts questions to stakeholders about the system that they currently use and the system to be developed. Requirements are derived from the answers to these questions.

Interviews may be of two types:

- Closed interviews, where the stakeholder answers a pre-defined set of questions.
- Open interviews, in which there is no pre-defined agenda. The requirements engineering team explores a range of issues with system stakeholders and hence develop a better understanding of their needs.

It can be difficult to elicit domain knowledge through interviews for two reasons:

All application specialists use terminology and jargon that are specific to a domain. It is impossible for them to discuss domain requirements without using this terminology. They normally use terminology in a precise and subtle way that is easy for requirements engineers to misunderstand.

Some domain knowledge is so familiar to stakeholders that they either find it difficult to explain or they think it is so fundamental that it isn't worth mentioning.

Effective interviewers have two characteristics:

1. They are open-minded, avoid pre-conceived ideas about the requirements, and are willing to listen to stakeholders. If the stakeholder comes up with surprising requirements, then they are willing to change their mind about the system.

2. They prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system. Saying to people 'tell me what you want' is unlikely to result in useful information. They find it much easier to talk in a defined context rather than in general terms.

Scenarios

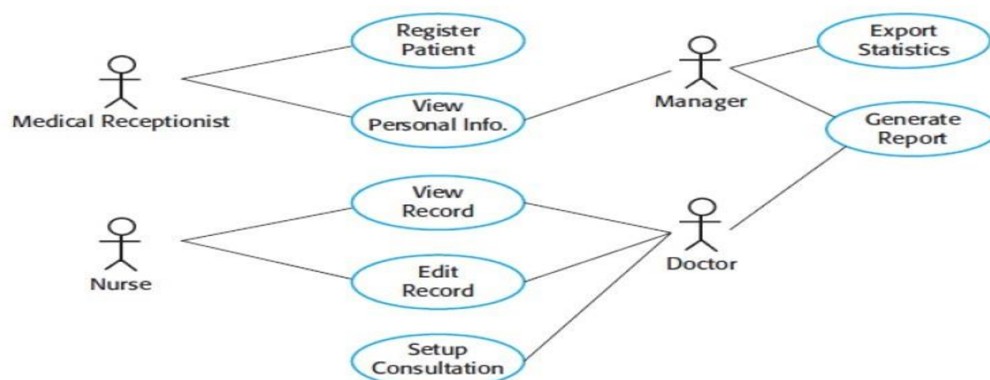
A scenario starts with an outline of the interaction. During the elicitation process, details are added to this to create a complete description of that interaction.

A scenario may include:

- A description of what the system and users expects when the scenario starts.
- A description of the normal flow of events in the scenario.
- A description of what can go wrong and how this is handled.
- Information about other activities that might be going on at the same time.

A description of the system state when the scenario finishes.

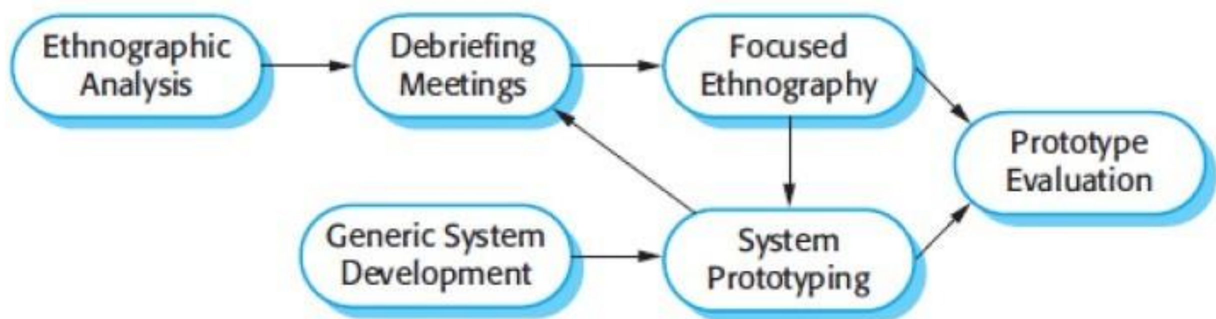
Use cases



Use cases are documented using a high-level use case diagram. The set of use cases represents all of the possible interactions that will be described in the system requirements.

Actors in the process, who may be human or other systems, are represented as stick figures. Each class of interaction is represented as a named ellipse. Lines link the actors with the interaction. Optionally, arrowheads may be added to lines to show how the interaction is initiated.
Use cases identify the individual interactions between the system and its users or other systems. Each use case should be documented with a textual description. These can then be linked to other models in the UML that will develop the scenario in more detail.

Ethnography
Ethnography is an observational technique that can be used to understand operational processes and help derive support requirements for these processes. An analyst immerses himself or herself in the working environment where the system will be used. The day-to-day work is observed and notes made of the actual tasks in which participants are involved. The value of ethnography is that it helps discover implicit system requirements that reflect the actual ways that people work, rather than the formal processes defined by the organization.
Ethnography is particularly effective for discovering two types of requirements:
Requirements that are derived from the way in which people actually work, rather than the way in which process definitions say they ought to work.
Requirements that are derived from cooperation and awareness of other people's activities.
Ethnography can be combined with prototyping
The ethnography informs the development of the prototype so that fewer prototype refinement cycles are required. Furthermore, the prototyping focuses the ethnography by identifying problems and questions that can then be discussed with the ethnographer.



REQUIREMENTS VALIDATION
Requirements validation is the process of checking that requirements actually define the system that the customer really wants. The cost of fixing a requirements problem by making a system change is usually much greater than repairing design or coding errors. The reason for this is that a change to the requirements usually means that the system design and implementation must also be changed.
During the requirements validation process, different types of checks should be carried out on the requirements in the requirements document. These checks include:
Validity checks A user may think that a system is needed to perform certain functions. However, further thought and analysis may identify additional or different functions that are required. Systems have diverse stakeholders with different needs and any set of requirements is inevitably a compromise across the stakeholder community.
Consistency checks Requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system

function.

Completeness checks The requirements document should include requirements that define all functions and the constraints intended by the system user.

Realism checks Using knowledge of existing technology, the requirements should be checked to ensure that they can actually be implemented. These checks should also take account of the budget and schedule for the system development.

Verifiability To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

There are a number of requirements validation techniques that can be used individually or in conjunction with one another:

1. **Requirements reviews:** The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.
2. **Prototyping:** In this approach to validation, an executable model of the system in question is demonstrated to end-users and customers. They can experiment with this model to see if it meets their real needs.
3. **Test-case generation**: Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered. Developing tests from the user requirements before any code is written is an integral part of extreme programming.

REQUIREMENTS MANAGEMENT

Once a system has been installed and is regularly used, new requirements inevitably emerge. It is hard for users and system customers to anticipate what effects the new system will have on their business processes and the way that work is done.

There are several reasons why change is inevitable:

1. The business and technical environment of the system always changes after installation. New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change (with consequent changes in the system support required), and new legislation and regulations may be introduced that the system must necessarily abide by.
2. The people who pay for a system and the users of that system are rarely the same people. System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements and, after delivery; new features may have to be added for user support if the system is to meet its goals.
3. Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory. The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support given to different users has to be changed.

Requirements Management Planning

Planning is an essential first stage in the requirements management process.

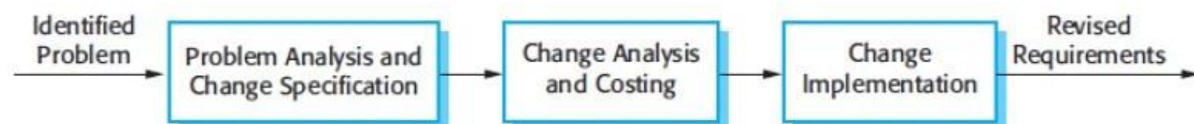
During the requirements management stage, the following is decided

Requirements identification: Each requirement must be uniquely identified so that it can be cross-referenced with other requirements and used in traceability assessments.

Change management process: This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.

Traceability policies: These policies define the relationships between each requirement and between the requirements and the system design that should be recorded. The traceability policy should also define how these records should be maintained.
Tool support Requirements management: involves the processing of large amounts of information about the requirements. Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.
Requirements management needs automated support and the software tools for this should be chosen during the planning phase.
Requirements storage: The requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.
Change management: The process of change management is simplified if active tool support is available.
Traceability management: Tool support for traceability allows related requirements to be discovered. Some tools are available which use natural language processing techniques to help discover possible relationships between requirements.
Requirements change management
Requirements change management should be applied to all proposed changes to a system's requirements after the requirements document has been approved.
There are three principal stages to a change management process:
1. Problem analysis and change specification: The process starts with an identified requirements problem or, sometimes, with a specific change proposal. During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.

2. Change analysis and costing: The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. The cost of making the change is estimated both in terms of modifications to the requirements document and, if appropriate, to the system design and implementation. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.
3. Change implementation: The requirements document and, where necessary, the system design and implementation, are modified. You should organize the requirements document so that you can make changes to it without extensive rewriting or reorganization. As with programs, changeability in documents is achieved by minimizing external references and making the document sections as modular as possible. Thus, individual sections can be changed and replaced without affecting other parts of the document.



Classical Analysis

Structured analysis views a system from the perspective of the data flowing through it. The function of the system is described by processes that transform the data flows. Structured analysis takes advantage of information hiding through successive decomposition (or top down) analysis. This allows attention to be focused on pertinent

details and avoids confusion from looking at irrelevant details. As the level of detail increases, the breadth of information is reduced. The result of structured analysis is a set of related graphical diagrams, process descriptions, and data definitions. They describe the transformations that need to take place and the data required to meet a system's functional requirements. The goal of the classical analysis workflow is to produce a detailed specifications document based on the identified requirements.

Specifications doc is significant because:

- this is contract between developer and client regarding what the system will do
- if developer and client are different organizations, it is a legal contract
- it needs to address both functional and non-functional requirements of the system
- it is blueprint that designer will use for design then programmer to implement

The specifications document thus must be *detailed*, *unambiguous*, and *complete* model of the system.

Function-oriented

- top-down decomposition of business process
- each decomposition results in set of two or more simpler sub processes
- recursively decompose until function becomes trivial or easily understood/expressed
- results in tree structure that resembles organizational chart

Process-oriented

- first apply functional decomposition to identify processes/subprocesses
- use arrows to indicate directed activity
- control flow from process to process is a directed activity
- data flow is a directed activity (e.g. from database to process or vice versa)
- control flow typically expressed using *Flowcharts*
- data flow typically expressed using *Data Flow Diagrams (DFD)*
- functional decomposition and flowcharts dominated for technical/scientific systems development

Data-oriented

- Identify *data entities* in the system
- "something that has separate and distinct existence in the world of the users and is of interest to the users in that they need to record data about it".
- Entities are identified by certain *nouns* in a system description.
- Identify *entity types* by grouping together similar entities. This is an important form of abstraction.
- Each entity is an *occurrence of* its type.
- Then determine what *attributes* those entity types have

- Attributes are the relevant properties that an entity has
- All entities of a type have the same attributes
- Different entities of the same type will have different values for some attributes

- Attribute values will record an entity's *state*
- An attribute whose value uniquely identifies an entity may be selected as a *key*
- If no single attribute has unique value, a combination of attributes may be used.

- Then determine what *associations*, or interactions, those entities have with other entities.

- one entity can take an action with another, or may play a role for another
- Associations are identified by certain *verbs* or verb phrases
- associations are also known as *relationships*
- the *entity-relationship diagram* (ERD) was developed for modeling
- consider some examples: car, bookstore, library, table factory

More on Relationships

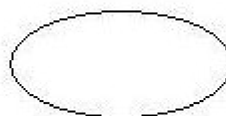
- relationships are also known as associations
- each relationship is between two entity types e.g. in the library example, the phrase "a patron borrows books" describes a relationship "borrows" between two entity types, "patron" and "book"
- a relationship is normally *directed*, e.g. in the library example, when a patron borrows a book, the "borrows" relationship is directed from the patron to the book
- each end of the association is also characterized by its *cardinality*
- Cardinality is number of occurrences of each entity type involved in an association e.g. in the library example, one patron borrows one or more books.

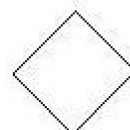
Cardinality on the patron end is 1 and on the book end is many (usually denoted by M or N or * or a triangle)

Entity-Relationship Diagrams

- visual modeling technique, a.k.a. "ERD" or "ER diagram"
- key concepts are described above: entities, attributes, relationships, cardinality
- ERDs describe *static* system views; DFDs show system *dynamics*
- ERDs and DFDs complement each other and Structured Analysis uses both
- database layouts and structures can be designed from ERDs
- as with DFDs, there are several different graphical notations
- most common notation is from Chen (ERD originator) or derivative:

Typically boxes for entities, ovals for attributes, diamonds for named relationships, lines to connect attributes to entities, arrows to connect entities and relationships





Formal Classical Specification Techniques

State Transition Diagrams

Frequently used to model event sequences, GUIs, and much more. Similar to Turing Machines as computational model.

Petri Nets

Graphical notation used to model concurrent processing.

Z specification language

- formal specification notation based on set theory and first order predicate logic.
- named after German mathematician Zermelo
- pronounced "zed", European pronunciation of the letter Z
- used mostly in Europe
- methodology-independent
- produces precise unambiguous specifications
- some mathematical skills required
- fundamental entity is the *schema*
    - data schema consists of: name, subcomponents, invariants
    - operation schema consists of: name, parameters, pre- and post-conditions
- non-standard object-oriented versions exist

Z Example

Sign on an escalator:

SHOES MUST BE WORN.
DOGS MUST BE CARRIED.

Any ambiguities there?!

Here it is, written in Z

$$\forall p : PERSON \bullet \\ (\exists s1, s2 : SHOE \bullet wears(p, s1, s2)) \\ \land (\forall d : DOG \bullet isWith(p, d) \Rightarrow carries(p, d))$$

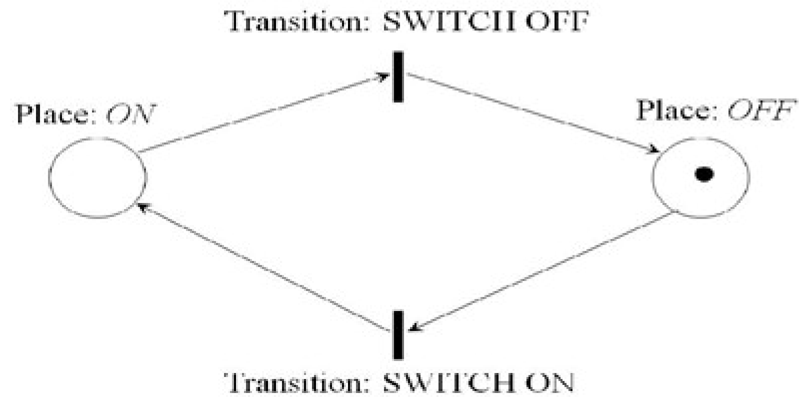
A Petri Nets (PN) comprises places, transitions, and arcs

– Places are system states

– Transitions describe events that may modify the system state

– Arcs specify the relationship between places

Tokens reside in places, and are used to specify the state of a PN



- Two places: Off and On
- Two transitions: Switch Off and Switch On
- Four arcs
- The off condition is true
- A transition can fire if an input token exists
  - – One token is moved from the input place to the output

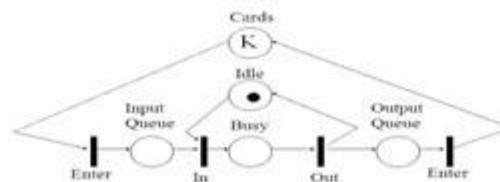
place. PN properties

- 8-tuple mathematical model

– M={P,T,I,O,H,PAR,PRED,MP}
– P - the set of places
– T - the set of transitions
– I,O,H - Input, output, inhibition function
– PAR - the set of parameters
– PRED - Predicates restricting parameter range
– PM - Parameter value

- From this linear algebra can be used to analyze a network

Manufacturing Example



- Very rich modeling
- Easily capable of modeling software project, requirements, architectures, and processes
- Drawbacks
  - – Complex rules
  - – Analysis quite complex

Data Dictionary

Provides definitions for all elements in the system which include:

- Meaning of data flows and stores in DFDs
- Composition of the data flows e.g. customer address breaks down to street number, street name, city and postcode
- Composition of the data in stores e.g. in Customer store include name, date of birth, address, credit rating etc.

Details of the relationships between entities

Data dictionary Notation

=is composed of

+	and
( )	optional (may be present or absent)
{ }	iteration
[ ]	select one of several alternatives
*	comment
@	identifier (key field) for store
❙	separates alternative choices in the [ ] construct

Data dictionary Examples

name = courtesy-title + first-name + (middle-name) + last-name courtesy-title = [Mr. ❙ Miss ❙ Mrs. ❙ Ms. ❙ Dr. ❙ Professor] first-name = {legal-character}
middle-name ={legal-character} last-name = {legal-character} legal-character = [A-Z❙a-z❙0-9❙'❙-❙ ❙]
Current-height =** *units: metres; range: 1.00-2.50*
sex =***values: [M❙F]*

As both are elementary data, no composition need be shown, though an explanation of the relevant units/symbols is needed order = customer-name + shipping-address + 1{item}10 means that an order always has a customer name and a shipping address and has between 1 and 10 items.

UNIT- 3

Software Design

Introduction

A software design creates meaningful engineering representation (or model) of some software product that is to be built. Designers must strive to acquire a repertoire of alternative design information and learn to choose the elements that best match the analysis model. A design model can be traced to the customer's requirements and can be assessed for quality against predefined criteria. During the design process the software requirements model (data, function, behavior) is transformed into design models that describe the details of the data structures, system architecture, interfaces, and components necessary to implement the system. Each design product is reviewed for quality (i.e. identify and correct errors, inconsistencies, or omissions, whether better alternatives exist, and whether the design model can be implemented within the project constraints) before moving to the next phase of software development.

Principles of Software Design

- Encompasses the set of principles, concepts, and practices that lead to the development of a high quality system or product
- Design principles establish and overriding philosophy that guides the designer as the work is performed
- Design concepts must be understood before the mechanics of design practice are applied
- Goal of design engineering is to produce a model or representation that is bug free (firmness), suitable for its intended uses (commodity), and pleasurable to use (delight)
- Software design practices change continuously as new methods, better analysis, and broader understanding evolve

Software Engineering Design

- Data/Class design - created by transforming the analysis model class-based elements (class diagrams, analysis packages, CRC models, collaboration diagrams) into classes and data structures required to implement the software
- Architectural design - defines the relationships among the major structural (fundamental, essential,)elements of the software, it is derived from the class-based elements and flow-oriented elements (data flow diagrams, control flow diagrams, processing narratives) of the analysis model
- Interface design - describes how the software elements, hardware elements, and end-users communicate with one another, it is derived from the analysis model scenario-based elements (use-case text, use-case diagrams, activity diagrams, swim lane diagrams), flow-oriented elements, and behavioral elements (state diagrams, sequence diagrams)
- Component-level design - created by transforming the structural elements defined by the software architecture into a procedural(technical, practical) description of the software components using information obtained from the analysis model class-based elements, flow-oriented elements, and behavioral elements.

Software Quality Attributes

A good design must

- implement all explicit requirements from the analysis model and accommodate all implicit requirements desired by the user
- be readable and understandable guide for those who generate code, test components, or support the system
- provide a complete picture (data, function, behavior) if the software from an implementation perspective

Design Quality Guidelines

A design should

- exhibit an architecture that has been created using recognizable architectural styles or patterns o is composed of components that exhibit good design characteristics o can be implemented in an evolutionary manner
- be modular
- contain distinct representations of data, architecture, interfaces, and components (modules)
- lead to data structures that are appropriate for the objects to be implemented and be drawn from recognizable design patterns
- lead to components that exhibit independent functional characteristics
- lead to interfaces that reduce the complexity of connections between modules and with the external environment
- be derived using a repeatable method that is driven by information obtained during software requirements analysis
- be represented using a notation that effectively communicates its meaning

FURPS Quality Factors

- Functionality – feature set and program capabilities
- Usability – human factors (aesthetics, consistency, documentation)
- Reliability – frequency and severity opf failure
- Performance – processing speed, response time, throughput, efficiency
- Supportability – maintainability (extensibility, adaptability, serviceability), testability, compatibility, configurability

Generic Design Task Set

1. Examine information domain model and design appropriate data structures for data objects and their attributes
2. Select an architectural pattern appropriate to the software based on the analysis model
3. Partition the analysis model into design subsystems and allocate these subsystems within the architecture
    - Be certain each subsystem is functionally cohesive
    - Design subsystem interfaces
    - Allocate analysis class or functions to subsystems
4. Create a set of design classes
    - Translate analysis class into design class
    - Check each class against design criteria and consider inheritance issues o Define methods and messages for each design class

- Evaluate and select design patterns for each design class or subsystem after considering alternatives
- Revise design classes and revise as needed

5. Design any interface required with external systems or devices
6. Design user interface
   - Review task analyses
   - Specify action sequences based on user scenarios o Define interface objects and control mechanisms o Review interface design and revise as needed
7. Conduct component level design
   - Specify algorithms at low level of detail o Refine interface of each component
   - Define component level data structures
   - Review components and correct all errors uncovered
8. Develop deployment model

Design Concepts

- Abstraction – allows designers to focus on solving a problem without being concerned about irrelevant lower level details (*procedural abstraction* - named sequence of events and *data abstraction* – named collection of data objects)
- Software Architecture – overall structure of the software components and the ways in which that structure provides conceptual integrity for a system
  - Structural models – architecture as organized collection of components
  - Framework models – attempt to identify repeatable architectural patterns
  - Dynamic models – indicate how program structure changes as a function of external events
  - Process models – focus on the design of the business or technical process that system must accommodate
  - Functional models – used to represent system functional hierarchy
- Design Patterns – description of a design structure that solves a particular design problem within a specific context and its impact when applied
- Separation of concerns – any complex problem is solvable by subdividing it into pieces that can be solved independently
- Modularity - the degree to which software can be understood by examining its components independently of one another
- Information Hiding – information (data and procedure) contained within a module is inaccessible to modules that have no need for such information
- Functional Independence – achieved by developing modules with single-minded purpose and an aversion to excessive interaction with other models o Cohesion - qualitative indication of the degree to which a module focuses on just one thing. Coupling - qualitative indication of the degree to which a module is connected to other modules and to the outside world
- Refinement – process of elaboration where the designer provides successively more detail for each design component
- Aspects – a representation of a cross-cutting concern that must be accommodated as refinement and modularization occur
- Refactoring – process of changing a software system in such a way internal structure is improved without altering the external behavior or code design

Design Classes

- Refine analysis classes by providing detail needed to implement the classes and implement a software infrastructure the support the business solution
- Five types of design classes can be developed to support the design architecture
  - user interface classes – abstractions needed for human-computer interaction (HCI)
  - business domain classes – refinements of earlier analysis classes o process classes – implement lower level business abstractions
  - persistent classes – data stores that persist beyond software execution
  - System classes – implement software management and control functions

Design Class Characteristics

- Complete (includes all necessary attributes and methods) and sufficient (contains only those methods needed to achieve class intent)
- Primitiveness – each class method focuses on providing one service
- High cohesion – small, focused, single-minded classes
- Low coupling – class collaboration kept to minimum

Design Model

- Process dimension – indicates design model evolution as design tasks are executed during software process
  - Architecture elements
  - Interface elements
  - Component-level elements
  - Deployment-level elements
- Abstraction dimension – represents level of detail as each analysis model element is transformed into a design equivalent and refined
  - High level (analysis model elements)
  - Low level (design model elements)
- Many UML diagrams used in the design model are refinements of diagrams created in the analysis model (more implementation specific detail is provided)
- Design patterns may be applied at any point in the design process

Data Design

- High level model depicting user's view of the data or information
- Design of data structures and operators is essential to creation of high-quality applications
- Translation of data model into database is critical to achieving system business objectives
- Reorganizing databases into a data warehouse enables data mining or knowledge discovery that can impact success of business itself.

DESIGN HEURISTICS

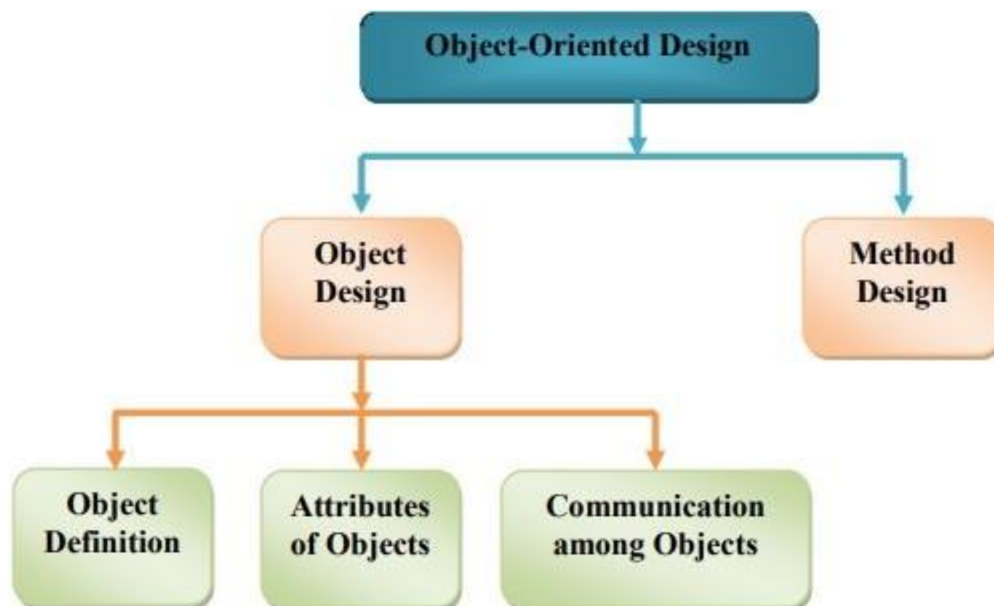
Heuristics are powerful tools in designing software and they provide a more subjective view of software quality. Application of heuristics is a difficult task and this potentially places a greater burden on the developers who must interpret this view since it consists of potentially conflicting indicators with varying degrees of precision and relevance. Heuristics may occur as individual pieces of developers' or as a suite covering multiple

aspects of software development.

Object-oriented design is a method of design encompassing the process of object-oriented decomposing and a notation for depicting logical and physical as well as static and dynamic models of the system under design. Object-oriented paradigm is more closely related to the real world situations. Procedural systems are based on functions and data does not depend upon operations in these systems. But in case of real world problems, data is closely related to operations as it defines the state of a real world object whereas operations define the behavior of that object. This concept is used by Object-Oriented systems to solve a problem. The main emphasis in object oriented design is focused towards objects and classes. Object-oriented design is related to develop an object-oriented module of a software system to serve the identified and analyzed requirements for the system. The main features of an object-oriented design include improved design quality, faster development process, high reusability features and modularity.

Booch introduced a methodology, given in Figure, which shows the necessary features of an object-oriented design. It comprises of a four-step process to design an object-oriented system. These are reproduced as under:

- Identify Classes and Objects: Identify key abstractions in the problem space and label them as potential classes and objects.
- Identify semantics of Classes and Objects: The meanings of the previously identified classes and objects are established, including defining the life cycle of each object from creation to destruction.
- Identification of Class-Object Relationship: Class and object interaction is identified, for example patterns of inheritance and patterns of visibility among objects and classes
- Specify Class and Object Interfaces and Implement Classes and Objects: Detailed internal view is constructed which includes defining methods and their behaviors



Object Oriented Methodology (Booch, 1994)

There are a number of internal attributes of Object-Oriented Design which are described as follows:

- **Coupling:** Coupling is defined as the measure of the relative interdependence among modules. For example, object A is coupled to object B only if A sends a message to B. Thus coupling refers to the number of messages passed between objects.

• **Cohesion:** "Cohesion measures the degree of connectivity among the elements of a single class or object". Cohesion relates to a measure of the relative functional strength of a module. It measures encapsulation within an object and deals with method's data interaction inside an object that makes it internally bonded. A class is said to be cohesive when the methods and variables contained are highly correlated. Cohesion can be used to identify the badly designed classes.

Inheritance: Inheritance is a mechanism in which one object inherits characteristics from one or more than one objects. It occurs at all levels of a class hierarchy. It is used to construct relationship between super classes and subclasses in various ways as inheritance enables the attributes and operations of a class to be inherited by all subclasses and the objects that are instantiated.

Encapsulation: Encapsulation encapsulates data and the operations into a single named object. It is an indirect measure of data abstraction and information hiding. Encapsulation hides internal specification of an object and shows only external interface. The process of compartmentalizing the elements of an abstraction that constitutes its structure and behavior is encapsulation. Encapsulation serves to separate the contractual interface of an abstraction and its implementation. Encapsulation influences software metrics by changing the focus of measurement from a single module to a package of data.

Information Hiding: Information hiding is the process of hiding all the secrets of an object that do not contribute to its essential characteristics. Public interface and a private implementation of an object are kept distinct. All information about a module should be private to the module unless it is specifically declared public. Information hiding plays a strong role in such metrics as object coupling and the degree of information hiding.

Localization: Localization is a characteristic of software that indicates the manner in which information is concentrated within a program. In the object oriented context, information is concentrated by encapsulating both data and functions within the bounds of a class or object. According to Booch, localization is the process of gathering and placing things in close physical proximity to each other. It is based on objects in case of object-oriented design. A design plan is totally dependent upon the localization approach, because one function may involve several objects and one object may provide many functions. Metrics should apply to the class as a complete entity. Even the relationship between functions and classes is not necessarily one-to-one. For that reason, metrics that reflect the manner in which classes collaborate must be capable of accommodating one-to-many and many-to-one relationship.

Heuristics for Object Oriented Design

Design evaluation is effective and beneficial to both expert and novice designers during software development process. Design heuristics are proposed as a more accessible and informal means by which developers can evaluate OO design.

Software heuristics are small, simple, legible, self-contained nuggets of design expertise. They target specific design problems and provide guidance to affect a solution. Unlike metrics, heuristics are outwardly defined in terms of the observable problems that occur during OOD. Moreover, the lessons learned from applying the low level design concepts such as coupling, cohesion and size throughout metric research provide a solid theoretical foundation that heuristics build upon to document recurring and observable problems within OO systems. Design heuristics are available to all developers performing OOD and are applicable within a number of software domains. They permit small, incremental enhancements to maintainability and provide a common vocabulary for expressing design problems. Design heuristics are shown to be concentrated pieces of design expertise that deliver knowledge and experience from the expert to the

novice. These need to be simple and understandable in order for them to be useful to the majority of developers performing design evaluation.

Conflicts among Heuristics

The use of OOD heuristics within the software development process. The heuristics were presented in an example-driven manner and were comprehensively described. However, there were no common mechanisms for documenting the design heuristics as self-contained, transferable pieces of design expertise. Also, the research neither presented the inter relationship among heuristics nor any plan was proposed to deploy these heuristics in support of an informal approach to design evaluation.

Architectural Design

- Provides an overall view of the software product
- Derived from
  - Information about the application domain relevant to software
  - Relationships and collaborations among analysis model elements
  - Availability of architectural patterns and styles
- Usually depicted as a set of interconnected systems that are often derived fromthe analysis packages with in the requirements model

Interface Design

- Interface is a set of operations that describes the externally observable behavior of a class and provides access to its public operations
- Important elements
  - User interface (UI)
  - External interfaces to other systems
  - Internal interfaces between various design components
- Modeled using UML communication diagrams (called collaborationdiagrams in UML 1.x)

Component-Level Design

- Describes the internal detail of each software component
- Defines
  - Data structures for all local data objects
  - Algorithmic detail for all component processing functions
  - Interface that allows access to all component operations

Modeled using UML component diagrams, UML activity diagrams, pseudo code (PDL), and sometimes flowcharts

Deployment-Level Design

- Indicates how software functionality and subsystems will be allocated within the physical computing environment
- Modeled using UML deployment diagrams
- *Descriptor form* deployment diagrams show the computing environment but does not indicate configuration details
- *Instance form* deployment diagrams identifying specific named hardware configurations are developed during the latter stages of design

ARCHITECTURAL DESIGN

Establishing the overall structure of a software system

Objectives

- To introduce architectural design and to discuss its importance
- To explain why multiple models are required to document a software architecture
- To describe types of architectural model that may be used

Architectural (high-level) design = the process of establishing the subsystems of a larger software system and defining a framework for subsystem control and communication
Software architecture = the output of the high-level design process
Defining and documenting the software architecture provides support for:

- Stakeholder communication
- System analysis
- Large-scale software reuse
    - The software architecture of a program or computing system is the structure or structures of the system, which comprise software components
    - architecture is not the operational software
    - Enables a software engineer

Analyze the effectiveness of the design in meeting its stated requirements
Consider architectural alternatives at a stage when making design changes is still relatively easy
Reducing the risks associated with the construction of the software.

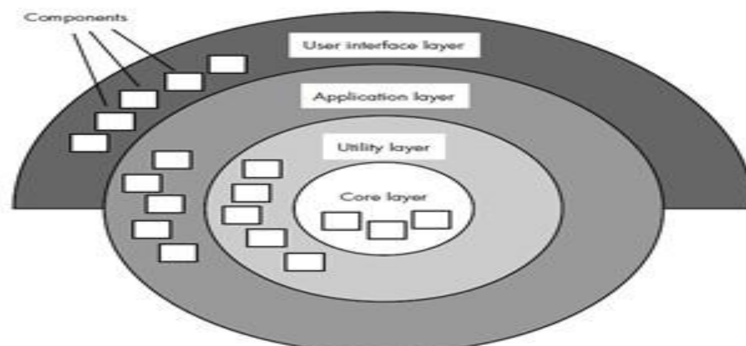
- Representations of software architecture are an enabler for communication between all parties
- The architecture highlights early design decisions that will have a profound impact on all software engineering work
- Architecture —constitutes a relatively small, intellectually graspable model of how the system is structured

Architectural Styles

An Architectural style typically specifies the design vocabulary, constraints on how that vocabulary is used and semantic assumptions about that vocabulary. Each style has several views and structures. An architectural view represents a set of elements and the relationships among them. Thus an architectural style defines a family of such systems in terms of a pattern of structural organization.

Layered Architectural style

This type of architectural style is the hierarchical organization of a system in layers. Layered systems are designed in a modular fashion at each layer in the architecture. There are well-defined interfaces between the layers. Layered system design is based on the increasing level of abstraction. Layered organization of an operating system is a good example of layered architectural style. Other examples can be a database system , an object request broker ,network layers etc. This architecture promotes reuse. However it has some drawbacks. It is not necessary to design all systems in a layered fashion. Adding more number of layers may decrease system performance.



Data-Flow Style

- The data flow is characterized by viewing the system as a series of transformations in a successive manner.
- In this style the input data enters the system and then moves through the components one at a time. and finally the transformed data are produced as output.
- These styles focus on achieving the quality of reuse and modifiability.
- The pipe and filter style follows the component connector structure in which components are filters and pipes are connectors.
- The filter reads stream of data as input ,performs data transformation and forwards the output data stream to another filter.
- The pipe transforms data streams from one filter to another.
- The pipe and filter style processes data streams in a pipeline instead of processing them as a single entity in the batch sequential style.
- For example a compiler executes in a pipe and filter style. It is comprised of several pipelined activities or filters.
- These are lexical analysis, semantic analysis, intermediate code generation and code generation.

Client-Server style

- It is useful for distributed processing, load balancing, separation of concerns and performance analysis.
- In this style there are two types of components, client and server.
- There exists connecting network between the clients and servers.
- The clients request services and the server provides services to the clients.
- The client communicates with servers through protocol and message connectors.

ARCHITECTURAL MAPPING USING DATA FLOW

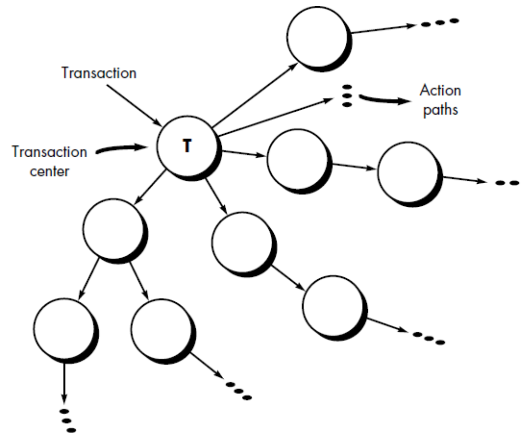
- The architectural styles represent radically different architectures. So it should come as no surprise that a comprehensive mapping that accomplishes the transition from the requirements model to a variety of architectural styles does not exist.
- A mapping technique, called structured design, is often characterized as a data flow-oriented design method because it provides a convenient transition from a data flow diagram to software architecture.
- The transition from information flow (represented as a DFD) to program structure is accomplished as part of a six step process:

(1) the type of information flow is established,
(2) flow boundaries are indicated,
(3) the DFD is mapped into the program structure,
(4) control hierarchy is defined,
(5) the resultant structure is refined using design measures and heuristics, and
(6) the architectural description is refined and elaborated.

TRANSFORM MAPPING

Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style. To map these data flow diagrams into a software architecture, you would initiate the following design steps:

The fundamental system model or context diagram depicts the security function (in example Safe Home securityfunction) as a single transformation, representing the external producers and consumers of data that flow into and out of the function.



Transaction Flow

The fundamental system model implies transform flow; therefore, it is possible to characterize all data flow in this category. However, information flow is often characterized by a single data item, called a *transaction that* triggers other data flow along one of many paths. When a DFD takes the form shown in Figure, *transaction flow* is present. Transaction flow is characterized by data moving along an incoming path that converts external world information into a transaction. The transaction is evaluated and, based on its value, flow along one of many *action paths* is initiated. The hub of information flow from which many action paths emanate is called a *transaction center.* It should be noted that, within a DFD for a large system, both transform and transaction flow may be present. For example, in a transaction-oriented flow, information flow along an action path may have transform flow characteristics.

Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style. The *SafeHome* security system, is representative of many computer-based products and systems in use today. The product monitors the real world and reacts to changes that it encounters. It also interacts with a user through a series of typed inputs and alphanumeric displays. The level 0 data flow diagram for *SafeHome,* is shown in Figure 'a'. During requirements analysis, more detailed flow models would be created for *SafeHome.* In addition, control

and process specifications, a data dictionary, and various behavioral models would also be created.

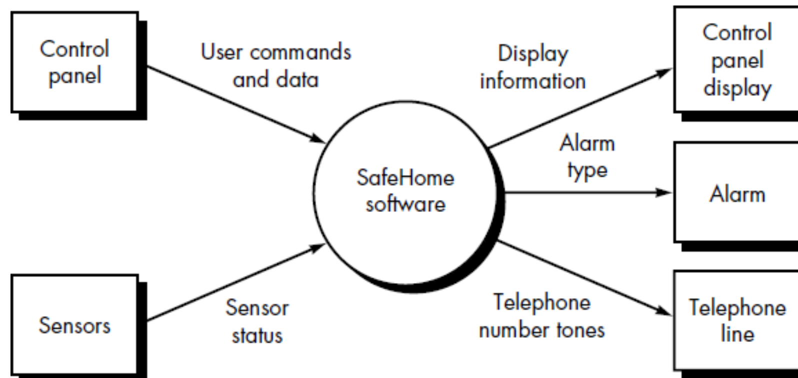


Figure 'a'

Design Steps

Step 1. Review the fundamental system model.

The fundamental system model encompasses the level 0 DFD and supporting information. In actuality, the design step begins with an evaluation of both the *System Specification* and the *Software Requirements Specification.* Both documents describe information flow and structure at the software interface. Figures a and b depict level 0 and level 1 data flow for the *SafeHome* software.

Step 2. Review and refine data flow diagrams for the software. Information obtained from analysis models contained in the *Software Requirements Specification* is refined to produce greater detail. For example, the level 2 DFD for *monitor sensors* is examined, and a level 3 data flow diagram is derived as shown in Figure. At level 3, each transform in the data flow diagram exhibits relatively high Cohesion.

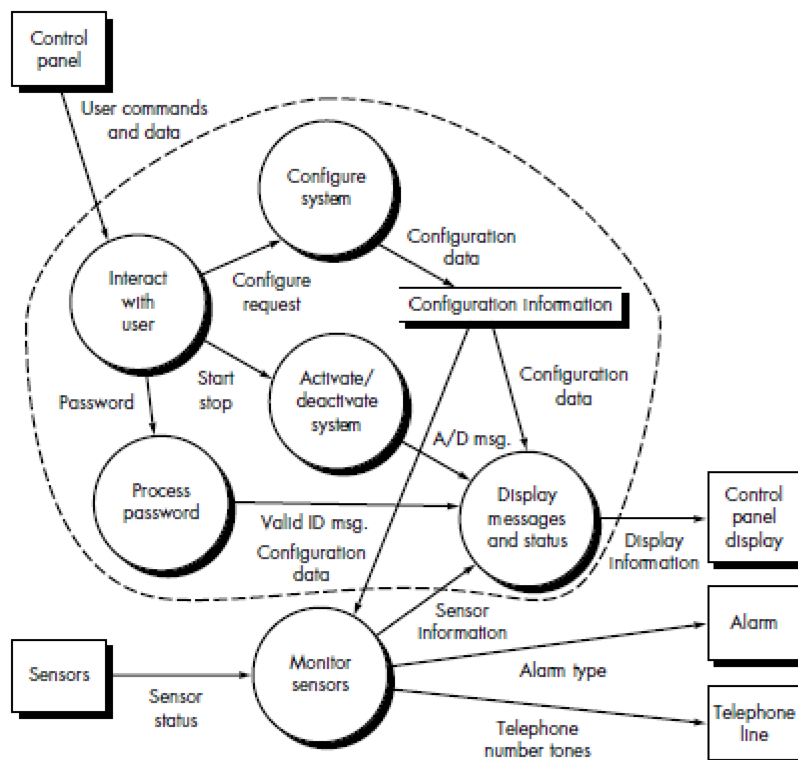


Figure 'b'

Step 3. Determine whether the DFD has transform or transaction flow characteristics.

In general, information flow within a system can always be represented as transform. However, when an obvious transaction characteristic is encountered, a different design mapping is recommended. In this step, the designer selects global (software wide) flow characteristics based on the prevailing nature of the DFD. In addition, local regions of transform or transaction flow are isolated. These *subflows* can be used to refine program architecture derived from a global characteristic described previously. For now, we focus our attention only on the *monitor sensors* subsystem data flow depicted in Figure.

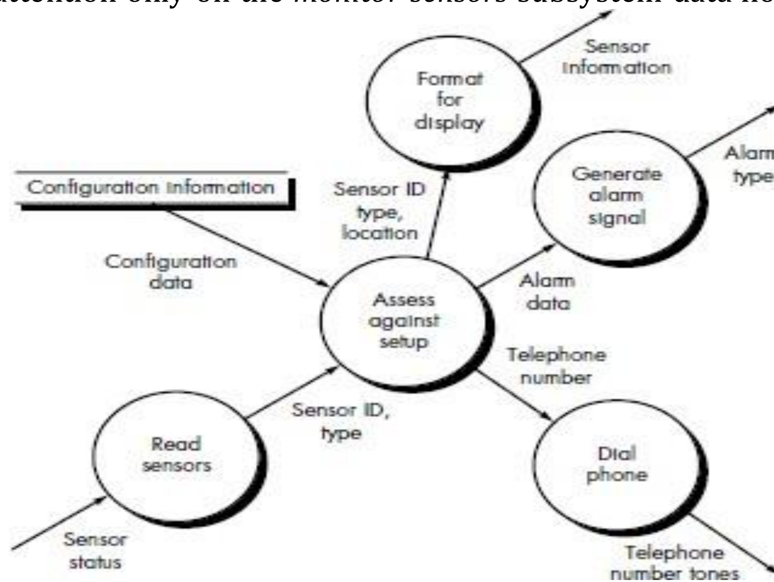


Figure 'c"

Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries. In the preceding section incoming flow was described as a path in which information is converted from external to internal form; outgoing flow converts from internal to external form. Incoming and outgoing flow boundaries are open to interpretation. That is, different designers may select slightly different points in the flow as boundary locations. In fact, alternative design solutions can be derived by varying the placement of flow boundaries. Although care should be taken when boundaries are selected, a variance of one bubble along a flow path will generally have little impact on the final program structure.

Flow boundaries for the example are illustrated as shaded curves running vertically through the flow in Figure. The transforms (bubbles) that constitute the transform center lie within the two shaded boundaries that run from top to bottom in the figure.

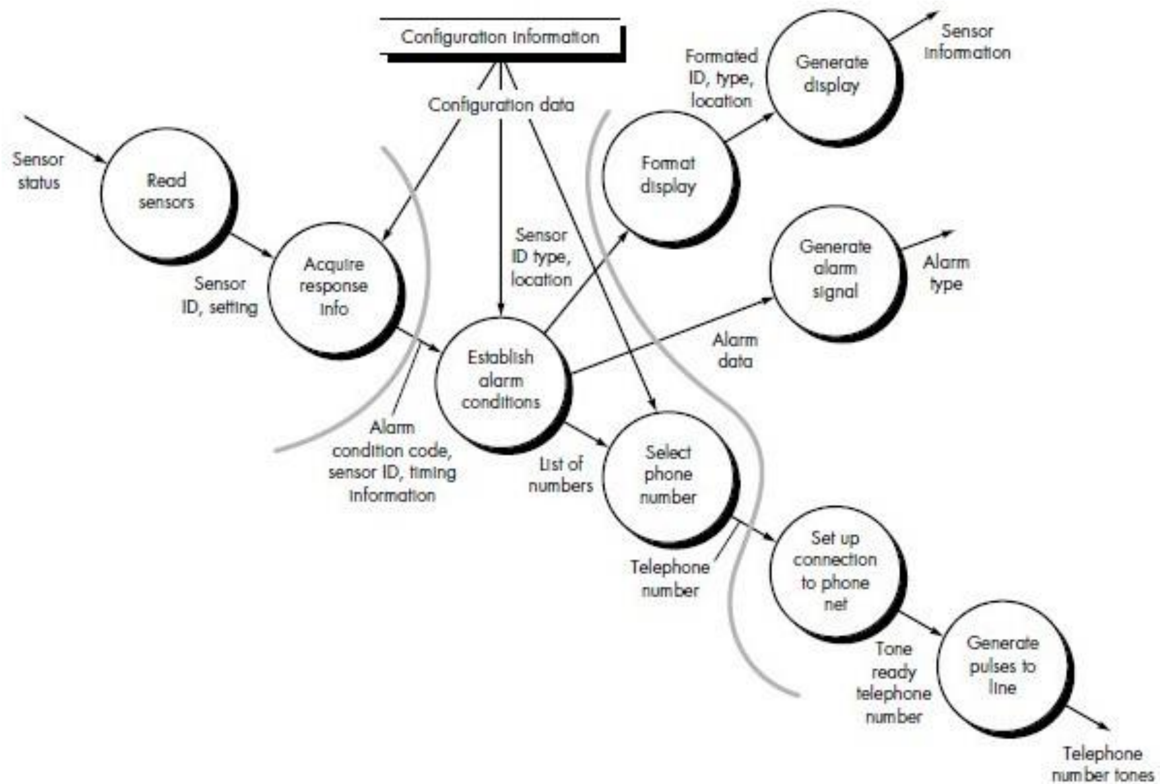


Figure 'd': Level 3 DFD for monitor sensors with flow boundaries

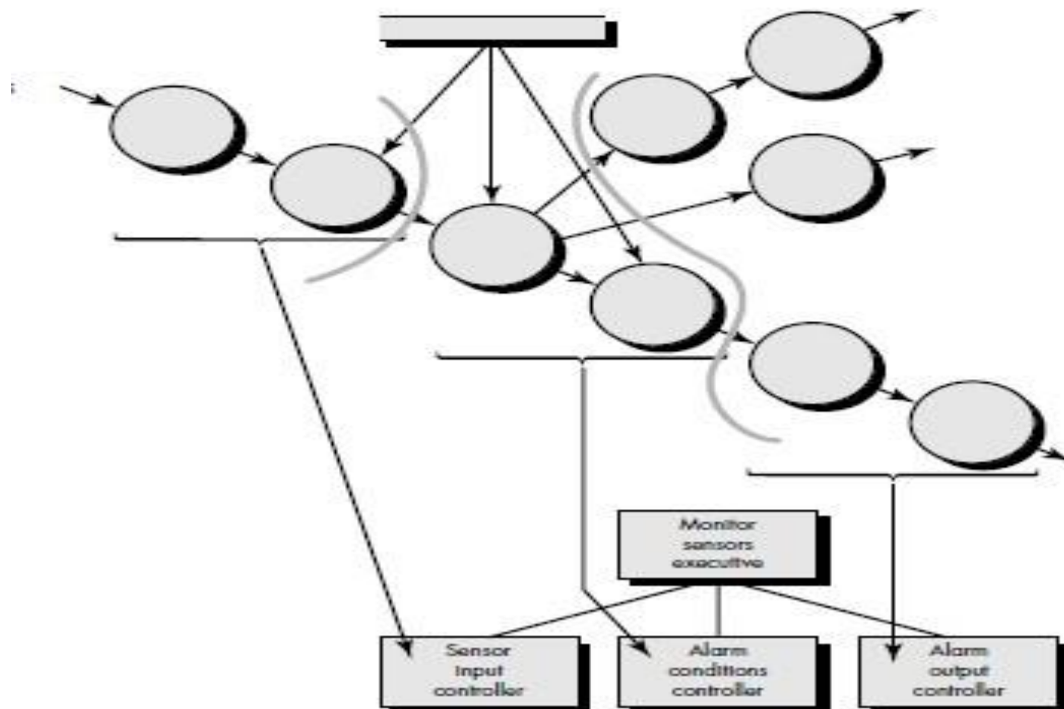


Figure 'e': First-level factoring for monitor sensors

Step 5. Perform "first-level factoring." Program structure represents a top-down distribution of control. Factoring results in a program structure in which top-level modules perform decision making and low-level modules perform most input, computation, and output work. Middle-level modules perform some control and do moderate amounts of work.

When transform flow is encountered, a DFD is mapped to a specific structure (a call and return architecture) that provides control for incoming, transform, and outgoing information processing. This first-level factoring for the *monitor sensors* subsystem is illustrated in Figure 'e'.

Step 6. Perform "second-level factoring." Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture. Beginning at the transform center boundary and moving outward along incoming and then outgoing paths, transforms are mapped into subordinate levels of the software structure. The general approach to second-level factoring for the *SafeHome* data flow is illustrated in Figure 'f'. Although Figure 'f' illustrates a one-to-one mapping between DFD transforms and software modules, different mappings frequently occur. Two or even three bubbles can be combined and represented as one module (recalling potential problems with cohesion) or a single bubble may be expanded to two or more modules. Practical considerations and measures of design quality dictate the outcome of second level factoring. Review and refinement may lead to changes in this structure, but it can serve as a "first-iteration" design.

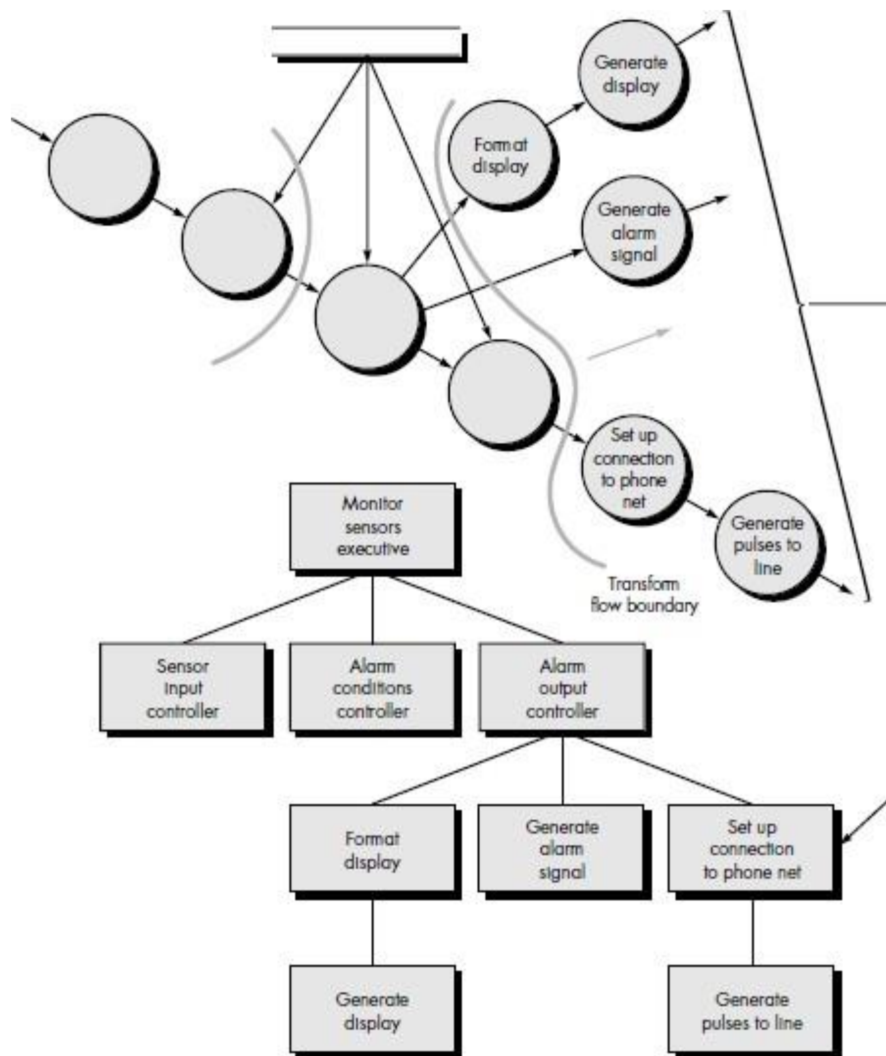


Figure 'f': Second-level factoring for monitor sensors

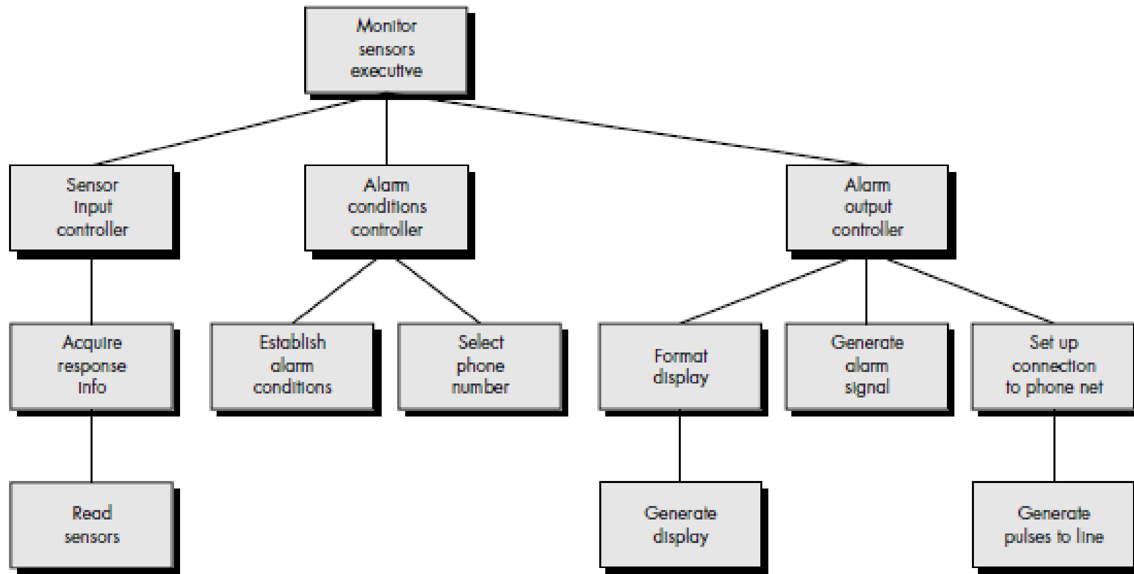


Figure 'g': " First-iteration" program structure for monitor sensors

Step 7. Refine the first-iteration architecture using design heuristics for improved software quality.

A first-iteration architecture can always be refined by applying concepts of module independence. Modules are exploded or imploded to produce sensible factoring, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief. Many modifications can be made to the first iteration architecture developed for the *SafeHome monitor sensors* subsystem. Among many possibilities,

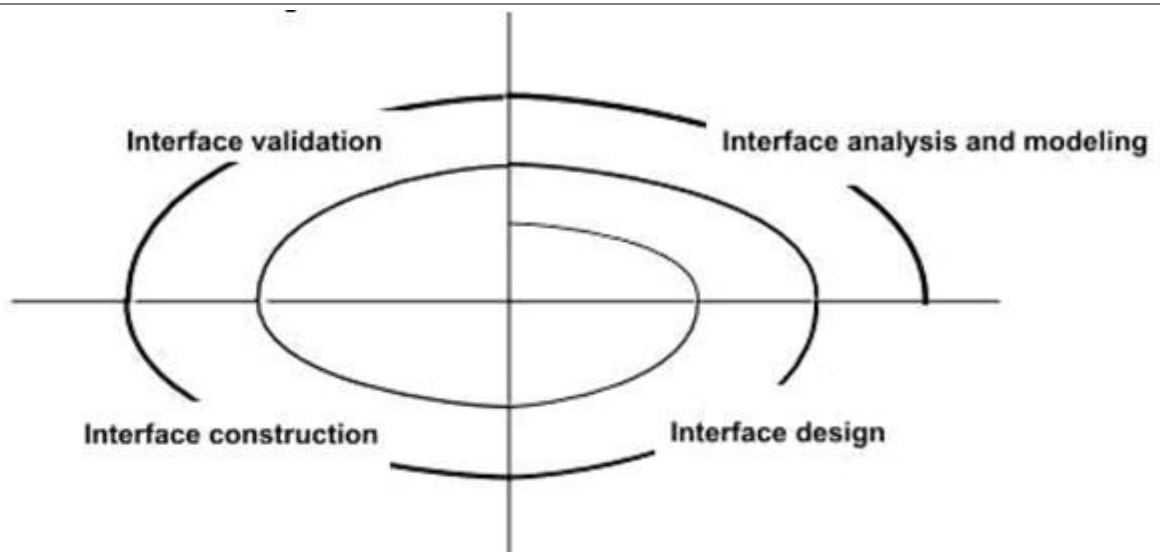
1. The incoming controller can be removed because it is unnecessary when a single incoming flow path is to be managed.

2. The substructure generated from the transform flow can be imploded into the module *establish alarm conditions* (which will now include the processing implied by *select phone number*). The transform controller will not be needed and the small decrease in cohesion is tolerable.

3. The modules *format display* and *generate display* can be imploded into a new module called *produce display.* The refined software structure for the monitor sensors subsystem is shown in Figure. The objective of the preceding seven steps is to develop an architectural representation of software. That is, once structure is defined, we can evaluate and refine software architecture by viewing it as a whole.

USER INTERFACE DESIGN

User interface (UI) design is an iterative process where users interact with designers and interface prototypes to decide on the features, organization and the look and feel of the system user interface. Sometimes the interface is separately prototyped in parallel with other software engineering activities. When iterative development is used, the user interface design proceeds incrementally as the software is developed. In both cases before programming starts, some paper-based designs must be developed and ideally tested. The overall UI design process is shown in the figure.



THE USER INTERFACE DESIGN PROCESS

There are three core activities in this process:

User analysis: In the user analysis process, develop an understanding of the tasks that the user does, their working environment, the other systems that they use, how they interact with other people in their work. For products with a diverse range of users, develop this understanding through focus groups, trails with potential users and similar exercises.

System prototyping: User interface design and development is an iterative process. Although users may talk about the facilities they need from an interface, it is very difficult for them to be specific until something tangible is seen. So prototype systems are developed and exposed to the users, which can guide the evolution of the interface.

Interface evaluation: Even though there are discussions with the users during the prototype process, a more formalized evaluation activity is required, where information about users‘ actual experience with the interface is collected. The scheduling of the UI design within the software process depends to some extent on other activities.

Prototyping may be used as a part of requirements engineering process and also to start the UI design process at this stage. In iterative processes UI design is integrated with the software development. Like the software itself the UI design may have to be refactored and redesigned during development.

User analysis: A critical user interface design activity is the analysis of the user activities that are to be supported by the computer system. To develop these understanding, task analysis, ethnographic studies, user interviews and observations or a mixture of all methods can be used.

The challenge for engineers involved in user analysis is to find a way to describe user analyses so that they communicate the essence of the tasks to other designers and to the users themselves. UML sequence charts can be used but they can be too technical for the users, so a natural language scenario to represent user activities must be developed. Cannot expect users‘ analysis to generate very specific user interface requirements. The analysis helps to understand the needs and concerns of the system users. As more information is obtained how they work, their concerns and their constraints it can be taken into account of the design.

User Interface Prototyping

Because of the dynamic nature of user interfaces, textual description and diagrams are not good enough for expressing user interface requirements. Evolutionary or exploratory prototyping with end-user involvement is the only practical way to design and develop graphical user interfaces for software systems. The aim of prototyping is to gain direct experience with the interface. It is difficult to think abstractly about a user

interface to explain exactly what is required. But when presented with examples it is easy to identify the characteristics that are liked and disliked.

When prototyping a user interface, a two-stage prototyping process is adopted:

- Very early in the process, develop paper prototypes-mock ups of screen designs-and walk through these with end-users.
- Then refine the design and develop increasingly sophisticated automated prototypes,
- then make them available to users for testing and activity simulation.

There are three approaches that can be used for user interface prototyping:

- Script-driven approach
- Visual programming languages
- Internet-based prototyping

Interface Evaluation

Interface evaluation is the process of assessing the usability of an interface and checking that it meets user requirements. It should be part of the normal verification and validation process for software systems.

COMPONENT LEVEL DESIGN

Component-level design, also called procedural design, occurs after data, architectural, and interface designs have been established. The intent is to translate the design model into operational software. But the level of abstraction of the existing design model is relatively high, and the abstraction level of the operational program is low. The translation can be challenging, opening the door to the introduction of subtle errors that are difficult to find and correct in later stages of the software process.

Graphical Design Notation

A flowchart is quite simple pictorially. A box is used to indicate a processing step.

A diamond represents a logical condition, and arrows show the flow of control. Figure illustrates three structured constructs. The sequence is represented as two processing boxes connected by an line (arrow) of control. Condition, also called if then- else, is depicted as a decision diamond that if true, causes then-part processing to occur, and if false, invokes else-part processing. Repetition is presented using two slightly different forms. The do while tests a condition and executes a loop task repetitively as long as the condition holds true. A repeat until executes the loop task first, then tests a condition and repeats the task until the condition fails. The selection (or select-case) construct shown in the figure is actually an extension of the if-then-else. A parameter is tested by successive decisions until a true condition occurs and a case part processing path is executed.

Tabular Design Notation

Condition #2		✓		✓					
Condition #3			✓		✓				
Actions									
Action #1	✓			✓	✓				
Action #2		✓		✓					
Action #3			✓						
Action #4			✓	✓	✓				
Action #5	✓	✓			✓				

Decision tables provide a notation that translates actions and conditions (described in a processing narrative) into a tabular form.

Table is divided into four sections. The upper left-hand quadrant contains a list of all conditions. The lower left-hand quadrant contains a list of all actions that are possible based on combinations of conditions. The right-hand quadrants form a matrix that indicates condition combinations and the corresponding actions that will occur for a specific combination. Therefore, each column of the matrix may be interpreted as a processing rule.

The following steps are applied to develop a decision table:

1. List all actions that can be associated with a specific procedure (or module).
2. List all conditions (or decisions made) during execution of the procedure.
3. Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions.
4. Define rules by indicating what action(s) occurs for a set of conditions.

Program Design Language

Program design language (PDL), also called structured English or pseudocode, is "a pidgin language in that it uses the vocabulary of one language (i.e., English) and the overall syntax of another (i.e., a structured programming language)". In this chapter, PDL is used as a generic reference for a design language.

At first glance PDL looks like a modern programming language. The difference between PDL and a real programming language lies in the use of narrative text (e.g., English) embedded directly within PDL statements.

Variable rate acct.	F	F	T	T	F
Consumption <100 kwh	T	F	T	F	
Consumption ≥100 kwh	F	T	F	T	
Actions					
Min. monthly charge	✓				
Schedule A billing		✓	✓		
Schedule B billing				✓	
Other treatment					✓

A design language should have the following characteristics:

- A fixed syntax of keywords that provide for all structured constructs, data declaration, and modularity characteristics.
- A free syntax of natural language that describes processing features.
- Data declaration facilities that should include both simple (scalar, array) and complex (linked list or tree) data structures.
- Subprogram definition and calling techniques that support various modes of interface description.

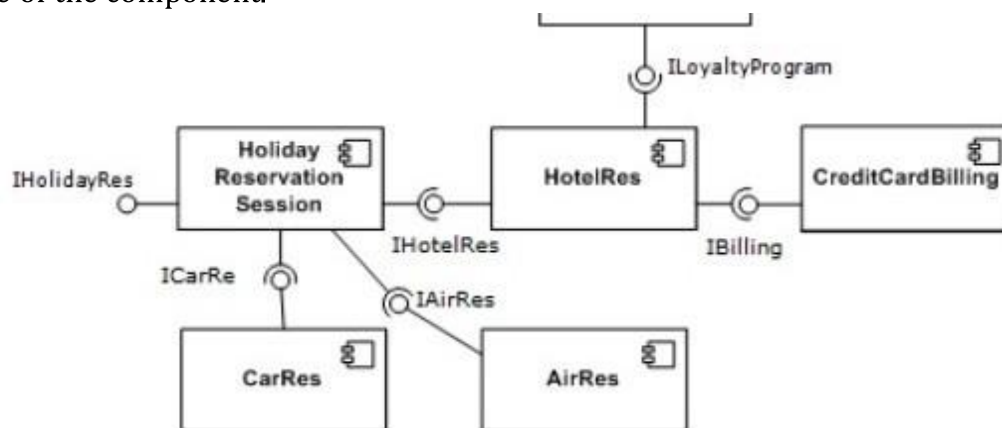
Designing Class based components, traditional Components

An individual **software component** is a software package, a web service, a web resource, or a module that encapsulates a set of related functions (or data).

All system processes are placed into separate components so that all of the data and functions inside each component are semantically related (just as with the contents of classes). Because of this principle, it is often said that components are *modular* and *cohesive*.

With regard to system-wide co-ordination, components communicate with each other via *interfaces*. When a component offers services to the rest of the system, it adopts a *provided* interface that specifies the services that other components can utilize, and how they can do so. This interface can be seen as a signature of the component - the client does not need to know about the inner workings of the component (implementation) in order to make use of it. This principle results in components referred to as *encapsulated*. The UML illustrations within this article represent provided interfaces by a lollipop-symbol attached to the outer edge of the component.

However, when a component needs to use another component in order to function, it adopts a *used* interface that specifies the services that it needs. In the UML illustrations in this article, *used interfaces* are represented by an open socket symbol attached to the outer edge of the component.



A simple example of several software components - pictured within a hypothetical holiday-reservation system represented in UML 2.0.

Another important attribute of components is that they are *substitutable*, so that a component can replace another (at design time or run-time), if the successor component meets the requirements of the initial component (expressed via the interfaces). Consequently, components can be replaced with either an updated version or an alternative without breaking the system in which the component operates.

As a general rule of thumb for engineers substituting components, component B can immediately replace component A, if component B provides at least what component A provided and uses no more than what component A used.

Software components often take the form of objects (not classes) or collections of objects (from object-oriented programming), in some binary or textual form, adhering to some interface description language(IDL) so that the component may exist

autonomously from other components in a computer.

When a component is to be accessed or shared across execution contexts or network links, techniques such as serialization or marshalling are often employed to deliver the component to its destination.

Reusability is an important characteristic of a high-quality software component. Programmers should design and implement software components in such a way that many different programs can reuse them. Furthermore, component-based usability testing should be considered when software components directly interact with users.

It takes significant effort and awareness to write a software component that is effectively reusable. The component needs to be:

- fully documented
- thoroughly tested
  - robust - with comprehensive input-validity checking
  - able to pass back appropriate error messages or return codes
- designed with an awareness that it *will* be put to unforeseen uses

UNIT- 4

SOFTWARE TESTING FUNDAMENTALS

Definition for Testing

Software testing is a process of executing a program or application with the intent of finding the software bugs. It can also be stated as the process of validating and verifying that a software program or application or product.

Testing Fundamental

Software engineer attempts to build software from an abstract concept to a tangible product. Next is Testing.

The engineer creates a series of test cases that are intended to "demolish" the software that has been built.

In fact, testing is the one step in the software process that could be viewed as destructive rather than constructive.

Testing Principles

- All tests should be traceable to customer requirements.
- Tests should be planned long before testing begins.
- The Pareto principle applies to software testing.
- Testing should begin “in the small” and progress toward testing “in the large.”
- Exhaustive testing is not possible.
- To be most effective, testing should be conducted by an independent third party.

Software Testability

S/w testability is simply how easily system or program or product can be tested.

Testing must exhibit set of characteristics that achieve the goal of finding errors with a minimum of effort.

Characteristics of s/w Testability

Operability

- “The better it works, the more efficiently it can be tested”
- Relatively few bugs will block the execution of tests.
- Allowing testing progress without fits and starts

Observability

- “What you see is what you test. “
- Distinct output is generated for each input.
- System states and variables are visible or queriable during execution.
- Incorrect output is easily identified.
- Internal errors are automatically detected & reported.
- Source code is accessible.

Controllability

- "The better we can control the software, the more the testing can be automated and optimized"
- Software and hardware states and variables can be controlled directly by the test engineer.
- Tests can be conveniently specified, automated, and reproduced.

Decomposability

- By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting.
- Independent modules can be tested independently.

Simplicity - The less there is to test, the more quickly we can test it."

- *Functional simplicity* (e.g., the feature set is the minimum necessary to meet requirements).
- *Structural simplicity* (e.g., architecture is modularized to limit the propagation of faults).
- *Code simplicity* (e.g., a coding standard is adopted for ease of inspection and maintenance).

Stability

- "The fewer the changes, the fewer the disruptions to testing."
- Changes to the software are infrequent.
- Changes to the software are controlled.
- Changes to the software do not invalidate existing tests.

Understandability

- "The more information we have, the smarter we will test."
- Dependencies between internal, external, and shared components are well understood.
- Changes to the design are communicated to testers.
- Technical documentation is instantly accessible, well organized, specific and detailed, and accurate.

Testing attributes

1. A good test has a high probability of finding an error.
    - Tester must understand the software and attempt to develop a mental picture of how the software might fail.
2. A good test is not redundant.
    - Testing time and resources are limited.
    - There is no point in conducting a test that has the same purpose as another test.
    - Every test should have a different purpose
    - Ex. Valid/ invalid password.
3. A good test should be "best of breed"
    - In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests.
4. A good test should be neither too simple nor too complex.
    - sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors.
    - Each test should be executed separately.

INTERNAL AND EXTERNAL VIEWS OF TESTING

Objectives of testing are to finding the most errors with a minimum amount of time and effort. Test case design methods provide a mechanism that can help to ensure the

completeness of tests and provide the highest likelihood for uncovering errors in software.

Any product or system can be tested on 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

The first test approach takes an **external view** is called **Black Box testing**
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 effectively exercised.

White box testing

- *White-box testing* of software is predicated on close examination of procedural detail.
- Logical paths through the software are tested by providing test cases that exercise specific sets of conditions and/or loops.
- The "status of the program" may be examined at various points.
- White-box testing, sometimes called *glass-box testing,* is a test case design method that uses the control structure of the procedural design to derive test cases.

Using this method, SE can derive test cases that
1. Guarantee that all independent paths within a module have been exercised at least once

2. Exercise all logical decisions on their true and false sides,

3. Execute all loops at their boundaries and within their operational bounds

4. Exercise internal data structures to ensure their validity.

Basis path testing

Basis path testing is a white-box testing technique used to derive a logical complexity measure of a procedural design. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time.

Methods

1. Flow graph notation

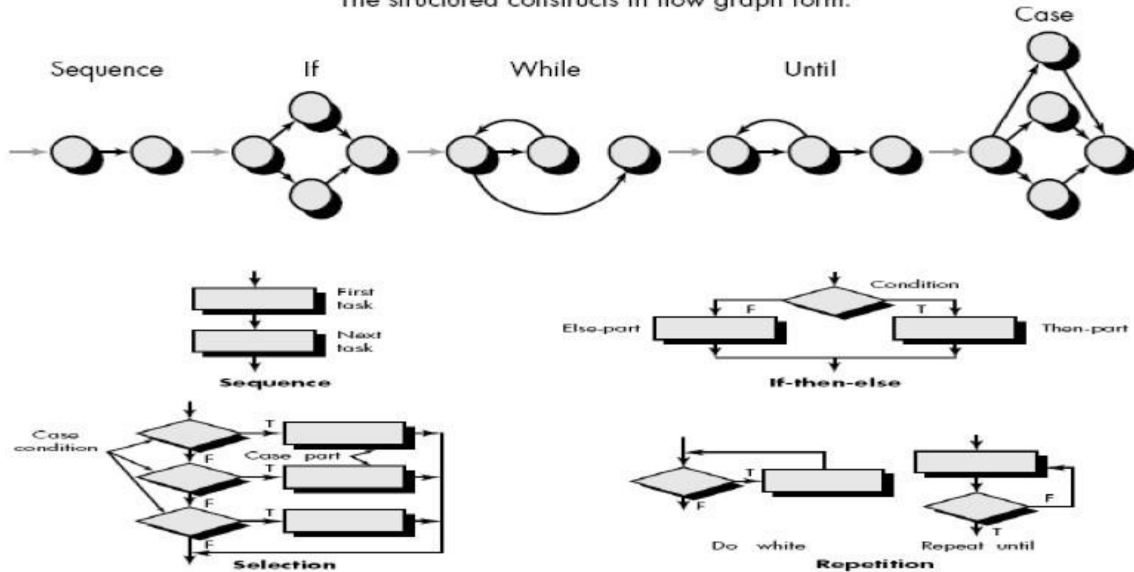
2. Independent program paths or Cyclomatic complexity

3. Deriving test cases

4. Graph Matrices

Flow Graph Notation

Start with simple notation for the representation of *control flow (called flow graph).* It represent logical control flow.



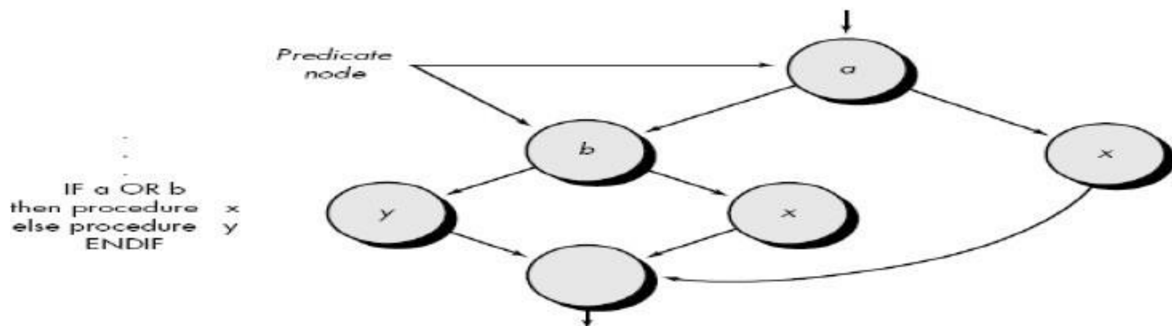
A sequence of process boxes and decision diamond can map into a single node.

The arrows on the flow graph, called edges or links, represent flow of control and are parallel to flowchart arrows.

An edge must terminate at a node, even if the node does not represent any procedural statement.

Areas bounded by edges and nodes are called *regions*. When counting regions, we include the are outside the graph as a region.

When compound condition are encountered in procedural design, flow graph becomes slightly more complicated.



When we translating PDL segment into flow graph, separate node is created for each condition.

Each node that contains a condition is called *predicate node* and is characterized by two or more edges comes from it.

Independent program paths or Cyclomatic complexity

An *independent path* is any path through the program that introduces at least one *new set of processing statement* or *new condition.*

For example, a set of independent paths for flow graph:

Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11 Basis Set

Path 3: 1-2-3-6-8-9-1-11

Path 4: 1-2-3-6-7-9-1-11
Note that each new path introduces a new edge.

The path 1-2-3-4-5-10-1-2-3-6-8-9-1-11 is not considered to e an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Test cases should be designed to force execution of these paths (basis set).

Every statement in the program should be executed at least once and every condition will have been executed on its true and false.

How do we know how many paths to looks for?

Cyclomatic complexity is a software metrics that provides a quantitative measure of the logical complexity of a program.
It defines no. of independent paths in the basis set and also provides number of test that must be conducted.

- One of three ways to compute cyclomatic complexity:
- The *no. of regions* corresponds to the cyclomatic complexity.
- Cyclomatic complexity, V(G), for a flow graph, G, is defined as

$V(G) = E - N + 2$
E is the number of flow graph edges, N is the number of flow graph nodes.
Cyclomatic complexity, V(G), for a flow graph, G, is also defined as

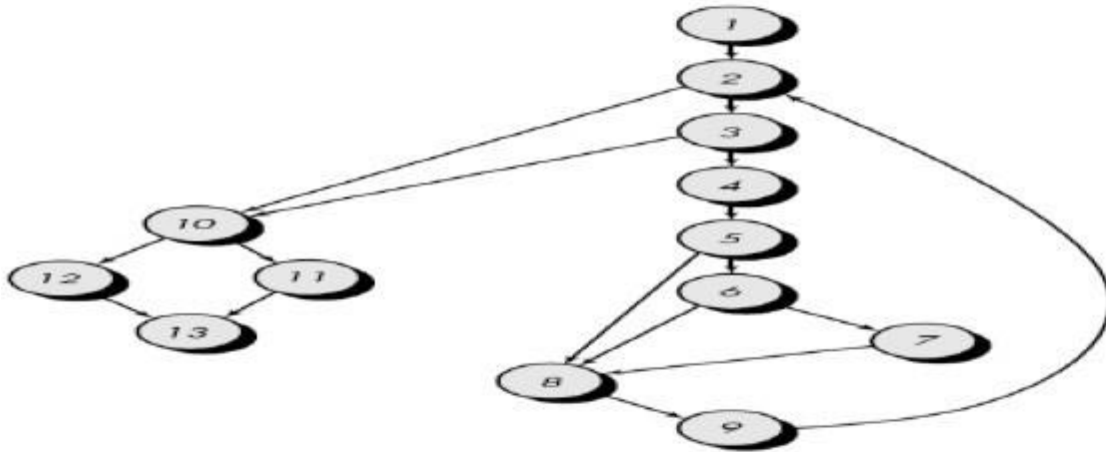
$V(G) = P + 1$
where P is the number of predicate nodes edges
So the value of V(G) provides us with upper bound of test cases.

Deriving Test Cases
It is a series of steps method.

The *procedure average* depicted in PDL.

Average, an extremely simple algorithm, contains compound conditions and loops.

To derive basis set, follow the steps.
1. Using the design or code as a foundation, draw a corresponding flow graph.
A flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes.



Determine the cyclomatic complexity of the resultant flow graph. $V(G)$ can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure *average,* compound conditions count as two) and adding 1
$V(G)$ = 6 regions

$V(G)$ = 17 edges - 13 nodes + 2 = 6

$V(G)$ = 5 predicate nodes + 1 = 6
3. Determine a basis set of linearly independent paths
The value of $V(G)$ provides the number of linearly independent paths through the program control structure.

path 1: 1-2-10-11-13

path 2: 1-2-10-12-13

path 3: 1-2-3-10-11-13

path 4: 1-2-3-4-5-8-9-2-. . .

path 5: 1-2-3-4-5-6-8-9-2-. . .

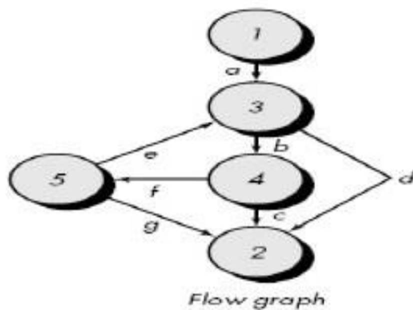
path 6: 1-2-3-4-5-6-7-8-9-2-. . .

The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable.
4. Prepare test cases that will force execution of each path in the basis set.

- Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested.
- Each test case is executed and compared to expected results.
- Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

Graph Matrices
A *graph matrix* is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes. Each node on the flow graph is identify by numbers, while each edge is identify by

letters. The graph matrix is nothing more than a tabular representation of a flow graph. By adding a *link weight* to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing. The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist).



Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

Connection matrix

Node \ Connected to node	1	2	3	4	5
1			1		
2					
3		1		1	
4		1			1
5		1	1		

Graph matrix

Each letter has been replaced with a 1, indicating that a connection exists (this graph matrix is called a *connection matrix).* In fig.(connection matrix) each row with two or more entries represents a predicate node. We can directly measure cyclomatic complexity value by performing arithmetic operations

Connections = Each row Total no. of entries – 1.

V(G)= Sum of all connections + 1

CONROL STRUCTURE TESTING

- It improves the quality of white box testing.
- Conditional testing
- Data flow testing
- Loop testing

Condition Testing

Condition testing is a test construction method that focuses on exercising the logical conditions in a program module.

Errors in conditions can be due to:

- Boolean operator error
- Boolean variable error
- Boolean parenthesis error
- Relational operator error
- Arithmetic expression error

definition: "For a compound condition C, the true and false branches of C *and every simple condition in C* need to be executed at least once." Multiple-condition testing requires that all true-false *combinations* of simple conditions be exercised at least once. Therefore, all statements, branches, and conditions are necessarily covered.

Data flow testing

Selects test paths according to the location of definitions and use of variables. This is a somewhat sophisticated technique and is not practical for extensive use. Its use should be targeted to modules with nested if and loop statements

Loop Testing

• A white-box testing technique that focuses exclusively on the validity of loop constructs

• Four different classes of loops exist

– Simple loops

– Nested loops

– Concatenated loops

– Unstructured loops

• Testing occurs by varying the loop boundary values

– Examples: for(i=0;i<MAX_INDEX;i++) while (currentTemp >= MINIMUM_TEMPERATURE)

Testing of **Simple Loops**

1) Skip the loop entirely

2) Only one pass through the loop

3) Two passes through the loop

4) m passes through the loop, where $m < n$

5) n –1, n, n + 1 passes through the loop

Testing of **Nested Loops**

1) Start at the innermost loop; set all other loops to minimum values

2) Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter values; add other tests for out-of-range or excluded values

3) Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to “typical” values

4) Continue until all loops have been tested

Testing of **Concatenated Loops**

• For independent loops, use the same approach as for simple loops

• Otherwise, use the approach applied for nested loops

Testing of **Unstructured Loops**

• Redesign the code to reflect the use of structured programming practices

• Depending on the resultant design, apply testing for simple loops, nested loops, or concatenated loops

BLACK BOX TESTING

also called behavioral testing, focuses on the functional requirements of the software. That is, black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white box testing.

Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external database access, (4) behavior or performance errors, and (5) initialization and termination errors.

Unlike white-box testing, which is performed early in the testing process, black box testing tends to be applied during later stages of testing .Because black-box testing purposely disregards control structure, attention is focused on the information domain. Tests are designed to answer the following questions:-

- How is functional validity tested?

- How are system behavior and performance tested?

- What classes of input will make good test cases?

- Is the system particularly sensitive to certain input values?

- How are the boundaries of a data class isolated?

- What data rates and data volume can the system tolerate?

- What effect will specific combinations of data have on system operation?
  The concept of Black-box testing can be explained with model-based testing as an example:-.

  Model-based testing (MBT) is a black-box testing technique that uses information contained in the requirements model as the basis for the generation of test cases. In many cases, the model-based testing technique uses UML state diagrams, an element of the behavioral model, as the basis for the design of test cases. The MBT technique requires five steps:
   Analyze an existing behavioral model for the software or create one.

  Recall that a behavioral model indicates how software will respond to external events or stimuli. To create the model, you should perform the steps:-

  - evaluate all use cases to fully understand the sequence of interaction within the system,
  - identify events that drive the interaction sequence and understand how these events relate to specific objects,
  - create a sequence for each use case,
  - build a UML state diagram for the system and
  - review the behavioral model to verify accuracy and consistency.

   Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state. The inputs will trigger events that will cause the transition to occur.
   Review the behavioral model and note the expected outputs as the software makes the transition from state to state. Recall that each state transition is triggered by an event and that as a consequence of the transition, some function is invoked and outputs are created. For each set of inputs (test cases) you specified in step 2, specify the expected outputs as they are characterized in the behavioral model. ―A fundamental assumption of this testing is that there is some mechanism, a test oracle, that will determine whether or not the results of a test

execution are correct‖. In essence, a test oracle establishes the basis for any determination of the correctness of the output. In most cases, the oracle is the requirements model, but it could also be another document or application, data recorded elsewhere, or even a human expert.

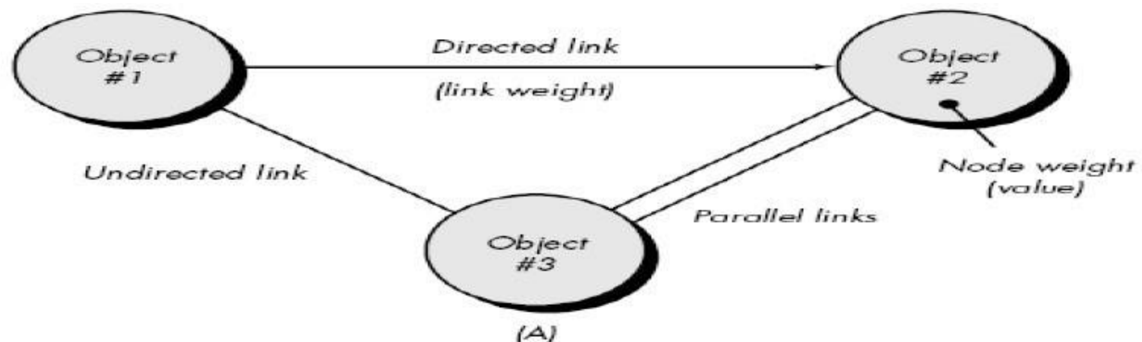
- Execute the test cases. Tests can be executed manually or a test script can be created and executed using a testing tool.
- Compare actual and expected results and take corrective action as required. MBT helps to uncover errors in software behavior, and as a consequence, it is extremely useful when testing event-driven applications.

Graph-Based Testing Methods

- To understand the objects that are modeled in software and the relationships that connect these objects.
- Next step is to define a series of tests that verify "all objects have the expected relationship to one another.

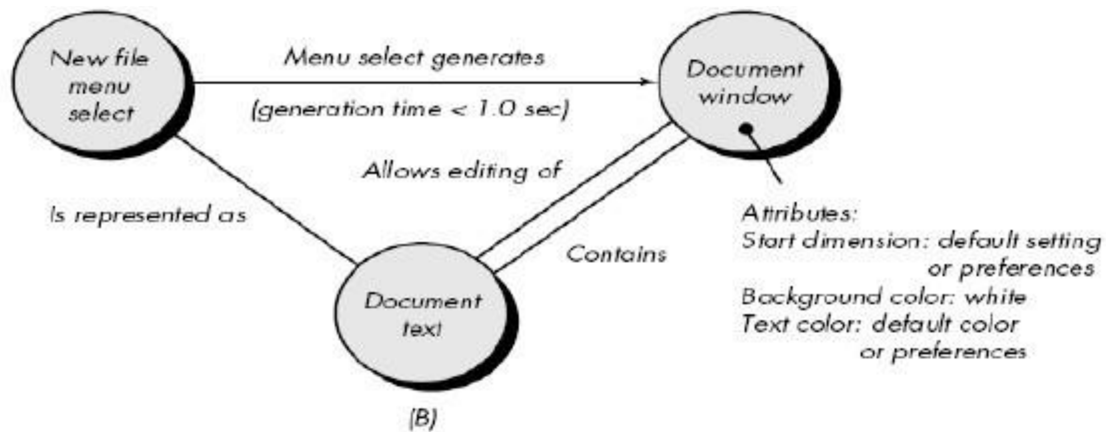
Stated in other way:

- Create a graph of important objects and their relationships
- Develop a series of tests that will cover the graph
- So that each object and relationship is exercised and errors are uncovered.
- Begin by creating graph –
- a collection of *nodes* that represent objects
- *links* that represent the relationships between objects
- *node weights* that describe the properties of a node
- *link weights* that describe some characteristic of a link.



Nodes are represented as circles connected by links that take a number of different forms.

- A *directed link* (represented by an arrow) indicates that a relationship moves in only one direction.
- A *bidirectional link,* also called a *symmetric link*, implies that the relationship applies in both directions.
- *Parallel links* are used when a number of different relationships are established between graph nodes.



(B)

Object #1 = new file menu select
Object #2 = document window
Object #3 = document text
Referring to example figure, a menu select on new file generates a document window.

- The link weight indicates that the window must be generated in less than 1.0 second.

The node weight of document window provides a list of the window attributes that are to be expected when the window is generated.

- An undirected link establishes a symmetric relationship between the new file menu select and document text,
- parallel links indicate relationships between document window and document text
- Number of behavioral testing methods that can make use of graphs:
- Transaction flow modeling.
- The nodes represent steps in some transaction and the links represent the logical connection between steps

Finite state modeling
The nodes represent different user observable states of the software and the links represent the transitions that occur to move from state to state. (Starting point and ending point)
Data flow modeling
The nodes are data objects and the links are the transformations that occur to translate one data object into another.
Timing modeling
The nodes are program objects and the links are the sequential connections between those objects.
Link weights are used to specify the required execution times as the program executes.
Equivalence Partitioning

- *Equivalence partitioning* is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived.
- Test case design for equivalence partitioning is based on an evaluation of *equivalence classes* for an input condition.
- An *equivalence class* represents a set of valid or invalid states for input conditions.

Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition. To define equivalence classes follow the guideline. **1.** If an input condition specifies a range, one valid and two invalid equivalence classes are defined. **2.** If an input condition requires a specific *value,* one valid and two invalid equivalence classes are defined. **3.** If an input condition specifies a member of a *set,* one valid and one invalid equivalence class are defined. **4.** If an input condition is *Boolean,* one valid and one invalid class are defined.

Example

- area code—blank or three-digit number
- prefix—three-digit number not beginning with 0 or 1
- suffix—four-digit number
- password—six digit alphanumeric string
- commands— check, deposit, bill pay, and the like
- area code:
- Input condition, *Boolean*—the area code may or may not be present.
- Input condition, *value*— three digit number
- prefix:
- Input condition, *range*—values defined between 200 and 999, with specific exceptions.
- Suffix:
- Input condition, value—four-digit length
- password:
- Input condition, *Boolean*—a password may or may not be present.
- Input condition, *value*—six-character string.
- command:
- Input condition, *set*— check, deposit, bill pay.

- Input condition, *set*— check, deposit, bill pay.

Boundary Value Analysis (BVA)

- Boundary value analysis is a test case design technique that complements equivalence partitioning.
- Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class.
- In other word, Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.

Guidelines for BVA

1. If an input condition specifies a range bounded by values *a* and *b,* test cases should be designed with values *a* and *b* and just above and just below *a* and *b*.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions.
4. If internal program data structures have prescribed boundaries be certain to design a test case to exercise the data structure at its boundary.

Orthogonal Array Testing

- The number of input parameters is small and the values that each of the parameters may take are clearly bounded.

- When these numbers are very small (e.g., three input parameters taking on three discrete values each), it is possible to consider every input permutation .
- However, as the number of input values grows and the number of discrete values for each data item increases (exhaustive testing occurs)
- *Orthogonal array testing* can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing.
- *Orthogonal Array Testing can be used to reduce the number of combinations and provide maximum coverage with a minimum number of test cases.*

Example

- Consider the *send* function for a fax application.
- Four parameters, P1, P2, P3, and P4, are passed to the *send* function. Each takes on three discrete values.
- P1 takes on values:
- P1 = 1, send it now
- P1 = 2, send it one hour later
- P1 = 3, send it after midnight
- P2, P3, and P4 would also take on values of 1, 2 and 3, signifying other *send* functions.
- OAT is an array of values in which each column represents a Parameter - value that can take a certain set of values called levels.
- Each row represents a test case.
- Parameters are combined pair-wise rather than representing all possible combinations of parameters and levels

A Strategy for Testing Conventional Software/Levels of Testing

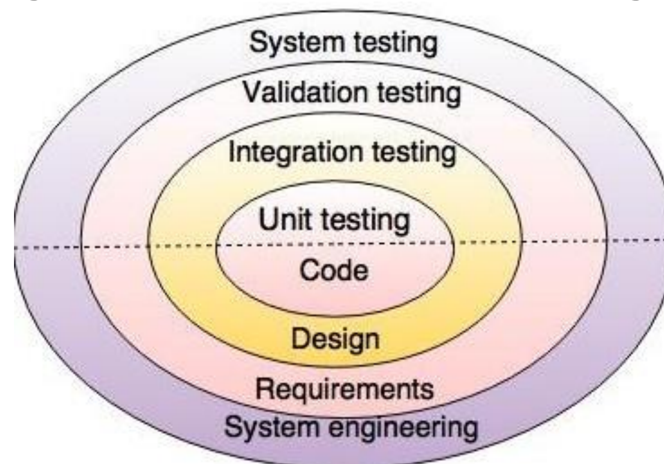


Fig. - Testing Strategy

Regression Testing

Each new addition or change to base lined software may cause problems with functions that previously worked flawlessly. Regression testing re-executes a small subset of tests that have already been conducted. Ensures that changes have not propagated unintended side effects. Helps to ensure that changes do not introduce unintended behavior or additional errors. May be done manually or through the use of automated capture/playback tools. Regression test suite contains three different classes of test cases.

- 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 actual software components that have been changed

Unit testing

– Concentrates on each component/function of the software as implemented in the source code
– Exercises specific paths in a component's control structure to ensure complete coverage and maximum error detection
– Components are then assembled and integrated
– Focuses testing on the function or software module
– Concentrates on the internal processing logic and data structures
– Is simplified when a module is designed with high cohesion
– Reduces the number of test cases
– Allows errors to be more easily predicted and uncovered
– Concentrates on critical modules and those with high cyclomatic complexity when testing resources are limited

Targets for Unit Test Cases

• Module interface
– Ensure that information flows properly into and out of the module
• Local data structures
– Ensure that data stored temporarily maintains its integrity during all steps in an algorithm execution
• Boundary conditions
– Ensure that the module operates properly at boundary values established to limit or restrict processing
• Independent paths (basis paths)
– Paths are exercised to ensure that all statements in a module have been executed at least once Error handling paths
– Ensure that the algorithms respond correctly to specific error conditions

Common Computational Errors in Execution Paths

• Misunderstood or incorrect arithmetic precedence
• Mixed mode operations (e.g., int, float, char)
• Incorrect initialization of values
• Precision inaccuracy and round-off errors
• Incorrect symbolic representation of an expression (int vs. float)

Other Errors to Uncover

• Comparison of different data types
• Incorrect logical operators or precedence
• Expectation of equality when precision error makes equality unlikely (using == with float types)
• Incorrect comparison of variables
• Improper or nonexistent loop termination
• Failure to exit when divergent iteration is encountered
• Improperly modified loop variables
• Boundary value violations

Problems to uncover in Error Handling

• Error description is unintelligible or ambiguous
• Error noted does not correspond to error encountered
• Error condition causes operating system intervention prior to error handling

• Exception condition processing is incorrect
• Error description does not provide enough information to assist in the location of the cause of the error

Drivers and Stubs for Unit Testing

• Driver
A simple main program that accepts test case data, passes such data to the component being tested, and prints the returned results
• Stubs
Serve to replace modules that are subordinate to (called by) the component to be tested
It uses the module's exact interface, may do minimal data manipulation, provides verification of entry, and returns control to the module undergoing testing
• Drivers and stubs both represent overhead
Both must be written but don't constitute part of the installed software product.

Integration Testing

• Defined as a systematic technique for constructing the software architecture
At the same time integration is occurring, conduct tests to uncover errors associated with interfaces
• Objective is to take unit tested modules and build a program structure based on the prescribed design
Two Approaches are
Non-incremental Integration Testing
Incremental Integration Testing

Non-incremental Integration Testing

• Commonly called the "Big Bang" approach
• All components are combined in advance
• The entire program is tested as a whole
• Chaos results
• Many seemingly-unrelated errors are encountered
• Correction is difficult because isolation of causes is complicated
• Once a set of errors are corrected, more errors occur, and testing appears to enter an endless loop

Incremental Integration Testing

Three kinds of integration testing are
Top-down integration
Bottom-up integration
Sandwich integration
• The program is constructed and tested in small increments
• Errors are easier to isolate and correct
• Interfaces are more likely to be tested completely
• A systematic test approach is applied

Top-down Integration

Modules are integrated by moving downward through the control hierarchy, beginning with the main module. Subordinate modules are incorporated in either a depth-first or breadth-first fashion
DF: All modules on a major control path are integrated
BF: All modules directly subordinate at each level are integrated

Advantages

This approach verifies major control or decision points early in the test process

Disadvantages

– Stubs need to be created to substitute for modules that have not been built or tested yet; this code is later discarded
– Because stubs are used to replace lower level modules, no significant data flow can occur until much later in the integration/testing process

Bottom-up Integration

• Integration and testing starts with the most atomic modules in the control hierarchy

• **Advantages**

– This approach verifies low-level data processing early in the testing process
– Need for stubs is eliminated

• **Disadvantages**

– Driver modules need to be built to test the lower-level modules; this code is later discarded or expanded into a full-featured version
– Drivers inherently do not contain the complete algorithms that will eventually use the services of the lower-level modules; consequently, testing may be incomplete or more testing may be needed later when the upper level modules are available

Sandwich Integration

• Consists of a combination of both top-down and bottom-up integration
• Occurs both at the highest level modules and also at the lowest level modules
• Proceeds using functional groups of modules, with each group completed before the next
– High and low-level modules are grouped based on the control and data processing they provide for a specific program feature
– Integration within the group progresses in alternating steps between the high and low level modules of the group
– When integration for a certain functional group is complete, integration and testing moves onto the next group
• Reaps the advantages of both types of integration while minimizing the need for drivers and stubs
• Requires a disciplined approach so that integration doesn't tend towards the "big bang" scenario

Validation Testing

• Validation testing follows integration testing
• The distinction between conventional and object-oriented software disappears
• Focuses on user-visible actions and user-recognizable output from the system
• Demonstrates conformity with requirements
• Designed to ensure that
– All functional requirements are satisfied
– All behavioral characteristics are achieved
– All performance requirements are attained
– Documentation is correct
– Usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability)
• After each validation test
– The function or performance characteristic conforms to specification and is accepted
– A deviation from specification is uncovered and a deficiency list is created
• A configuration review or audit ensures that all elements of the software configuration have been properly developed, cataloged, and have the necessary detail for entering the support phase of the software life cycle

Alpha and Beta Testing

• Alpha testing
– Conducted at the developer's site by end users

– Software is used in a natural setting with developers watching intently
– Testing is conducted in a controlled environment
• Beta testing
– Conducted at end-user sites
– Developer is generally not present
– It serves as a live application of the software in an environment that cannot be controlled by the developer
– The end-user records all problems that are encountered and reports these to the developers at regular intervals
• After beta testing is complete, software engineers make software modifications and prepare for release of the software product to the entire customer base

System Testing

 System Testing (ST) is a black box testing technique performed to evaluate the complete system the system's compliance against specified requirements.
 In System testing, the functionalities of the system are tested from an end-to-end perspective.
 System testing is performed on the entire system in the context of a Functional Requirement Specification(s) (FRS) and/or a System Requirement Specification (SRS).
 System testing tests not only the design, but also the behaviour and even the believed expectations of the customer.
 It is also intended to test up to and beyond the bounds defined in the software/hardware requirements specification(s) .Some of system testing are as follows:

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.

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.

Security Testing

Security testing attempts to verify that protection mechanisms built into a system

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. Essentially, the tester attempts to break the program.

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.

Deployment testing-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

Debugging Process

- Debugging occurs as a consequence of successful testing
- It is still very much an art rather than a science
- Good debugging ability may be an innate human trait
- Large variances in debugging ability exist
- The debugging process begins with the execution of a test case
- Results are assessed and the difference between expected and actual performance is encountered
- This difference is a symptom of an underlying cause that lies hidden

The debugging process attempts to match symptom with cause, thereby leading to error correction

Why is Debugging so Difficult?

- The symptom and the cause may be geographically remote
- The symptom may disappear (temporarily) when another error is corrected
- The symptom may actually be caused by non-errors (e.g., round-off accuracies)
- The symptom may be caused by human error that is not easily traced
- The symptom may be a result of timing problems, rather than processing problems
- It may be difficult to accurately reproduce input conditions, such as asynchronous real-time information
- The symptom may be intermittent such as in embedded systems involving both hardware and software
- The symptom may be due to causes that are distributed across a number of tasks running on different processes

Debugging Strategies

- Objective of debugging is to find and correct the cause of a software error
- Bugs are found by a combination of systematic evaluation, intuition, and luck
- Debugging methods and tools are not a substitute for careful evaluation based on a complete design model and clear source code
- There are three main debugging strategies

– Brute force

– Backtracking

– Cause elimination

Strategy #1: Brute Force

- Most commonly used and least efficient method

• Used when all else fails

• Involves the use of memory dumps, run-time traces, and output statements

• Leads many times to wasted effort and time

Strategy #2: Backtracking
• Can be used successfully in small programs
• The method starts at the location where a symptom has been uncovered

• The source code is then traced backward (manually) until the location of the cause is found

• In large programs, the number of potential backward paths may become unmanageably large

Strategy #3: Cause Elimination
• Involves the use of induction or deduction and introduces the concept of binary partitioning
– Induction (specific to general): Prove that a specific starting value is true; then prove the general case is true

– Deduction (general to specific): Show that a specific conclusion follows from a set of general premises
• Data related to the error occurrence are organized to isolate potential causes

• A cause hypothesis is devised, and the aforementioned data are used to prove or disprove the hypothesis

• Alternatively, a list of all possible causes is developed, and tests are conducted to eliminate each cause

• If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug

Software implementation techniques
Refactoring
 Refactoring is the process of modifying the structure of a program while preserving all of its actual functionality.

 There are various ways of refactoring like renaming a class, changing the method signature, or extracting some code into a method.

 While using every refactoring technique, perform a sequence of steps that keep your code consistent with the original code.

 If we do refactoring manually; there is a large probability of occurrence of errors into your code such as spelling mistakes etc.

 To remove these errors, testing should be done before and after using each refactoring technique.

 Refactoring is made to those programs which are poorly coded

 You should refactor:
Any time that you see a better way to do things
♣ “Better” means making the code easier to understand and to modify in the future
You can do so without breaking the code

♣ Unit tests are essential for this
 You should *not* refactor:
Stable code (code that won't ever need to change)
Someone else's code
♣ Unless you've inherited it (and now it's yours)

Significance of Testing in refactoring
 Testing is the process which we perform on every software to check the software from every perspective, whether it is giving the desired output on giving certain input or not.

 Testing of Java code before doing and after performing refactoring is necessary because refactoring changes the structure of your code.

 If the refactoring is done by hand, then a good suite of tests is a must.

 When using an automated tool to refactor, you should have to still test, but it is less tedious and time consuming in comparison to testing after doing refactoring manually

Types of Refactoring
Type 1 – Physical Structure
- Move
- Rename
- Change Method Signature
- Convert Anonymous Class to Nested
- Convert Nested Type to Top Level (Eclipse 2 only)
- Move Member Type to New File (Eclipse 3 only)

Type 2 – Class Level Structure
- Push Down
- Pull Up
- Extract Interface
- Generalize Type (Eclipse 3 only)
- User Super type Where Possible

Type 3 – Structure inside a Class
- Inline
- Extract Method
- Extract Local Variable
- Extract Constant
- Introduce Parameter (Eclipse 3 only)
- Introduce Factory (Eclipse 3 only)
- Encapsulate Field

Design vs. coding
ν "Design" is the process of determining, in detail, what the finished product will be and how it will be put together

ν "Coding" is following the plan

ν In traditional engineering (building bridges), design is perhaps 15% of the total effort

ν In software engineering, design is 85-90% of the total effort
ν By comparison, coding is cheap

Example 1: switch statements
ν switch statements are very rare in properly designed object-oriented code

ν Therefore, a switch statement is a simple and easily detected “bad smell”

ν Of course, not all uses of switch are bad

ν A switch statement should *not* be used to distinguish between various kinds of object.

BUSINESS PROCESS REENGINEERING

Business Process Reengineering (BPR) extends far beyond the scope of information technologies and software engineering. It defines “the search for, and the implementation of, radical change in business process to achieve breakthrough results.” The overall business is segmented in the following manner:

The business
  business systems
  business process
    business sub processes

Each business system (also called *business function*) is composed of one or more business processes, and each business process is defined by a set of sub processes. BPR can be applied at any level of the hierarchy, but as the scope of BPR broadens (i.e., as we move upward in the hierarchy), the risks associated with BPR grow dramatically. For this reason, most BPR efforts focus on individual processes or sub processes.

Principles of Business Process Reengineering

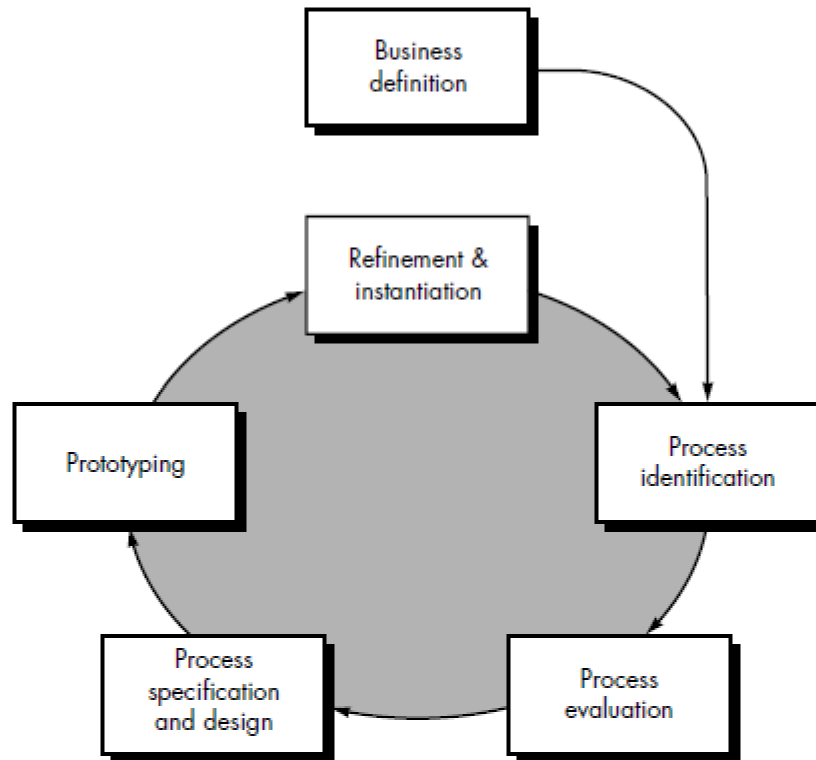
- Organize around outcomes, not tasks.
- Have those who use the output of the process perform the process.
- Incorporate information processing work into the real work that produces the raw information.
- Treat geographically dispersed resources as though they were centralized.
- Link parallel activities instead of integrating their results.
- Put the decision point where the work is performed, and build control into the process.
- Capture data once, at its source.

A BPR Model

Like most engineering activities, business process reengineering is iterative. Business goals and the processes that achieve them must be adapted to a changing business environment.

Business definition. Business goals are identified within the context of four key drivers: *cost reduction, time reduction, quality improvement,* and *personnel development and empowerment.* Goals may be defined at the business level or for a specific component of the business.

Process identification. Processes that are critical to achieving the goals defined in the business definition are identified. They may then be ranked by importance, by need for change, or in any other way that is appropriate for the reengineering activity.



A BPR model

Process evaluation. The existing process is thoroughly analyzed and measured. Process tasks are identified; the costs and time consumed by process tasks are noted; and quality/performance problems are isolated.
Process specification and design. Based on information obtained during the first three BPR activities, use-cases are prepared for each process that is to be redesigned. Within the context of BPR, use-cases identify a scenario that delivers some outcome to a customer. With the use-case as the specification of the process, a new set of tasks are designed for the process.
Prototyping. A redesigned business process must be prototyped before it is fully integrated into the business. This activity "tests" the process so that refinements can be made.
Refinement and instantiation. Based on feedback from the prototype, the business process is refined and then instantiated within a business system. These BPR activities are sometimes used in conjunction with workflow analysis tools. The intent of these tools is to build a model of existing workflow in an effort to better analyze existing processes.

SOFTWARE REENGINEERING

Software Maintenance

The maintenance of existing software can account for over 60 percent of all effort expended by a development organization, and the percentage continues to rise as more software is produced.

A Software Reengineering Process Model

Reengineering takes time; it costs significant amounts of money; and it absorbs resources that might be otherwise occupied on immediate concerns. For all of these

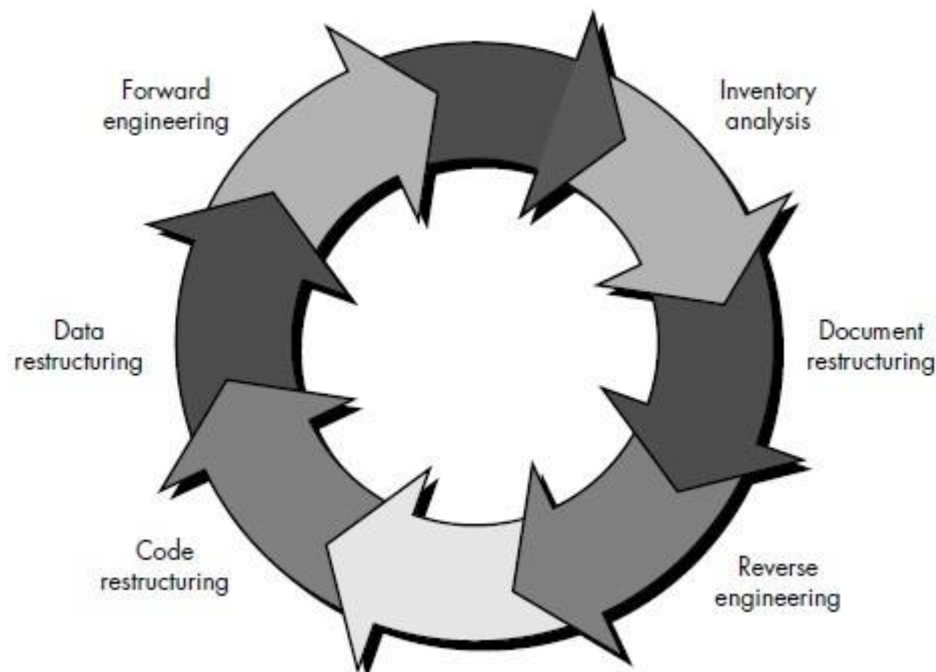
reasons, reengineering is not accomplished in a few months or even a few years. Reengineering of information systems is an activity that will absorb information technology resources for many years. Before you can start rebuilding, it would seem reasonable to inspect the house. To determine whether it is in need of rebuilding, you would create a list of criteria so that your inspection would be systematic.

- Before you tear down and rebuild the entire house, be sure that the structure is weak. If the house is structurally sound, it may be possible to "remodel" without rebuilding (at much lower cost and in much less time).
- Before you start rebuilding be sure you understand how the original was built. Take a peek behind the walls. Understand the wiring, the plumbing, and the structural internals. Even if you trash them all, the insight you'll gain will serve you well when you start construction.
- If you begin to rebuild, use only the most modern, long-lasting materials. This may cost a bit more now, but it will help you to avoid expensive and time-consuming maintenance later.
- If you decide to rebuild, be disciplined about it. Use practices that will result in high quality—today and in the future.

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 other locally important criteria, candidates for reengineering appear. Resources can then be allocated to candidate applications for reengineering work.

It is important to note that the inventory should be revisited on a regular cycle. The status of applications (e.g., business criticality) can change as a function of time, and as a result, priorities for reengineering will shift.



A software Reengineering Process Model

Document Restructuring.

1. Creating documentation is far too time consuming. If the system works, we'll live with what we 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 we have limited resources. We'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.

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.

Code Restructuring

The most common type of reengineering (actually, the use of the term *reengineering* is questionable in this case) is code restructuring. 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). The resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced. Internal code documentation is updated.

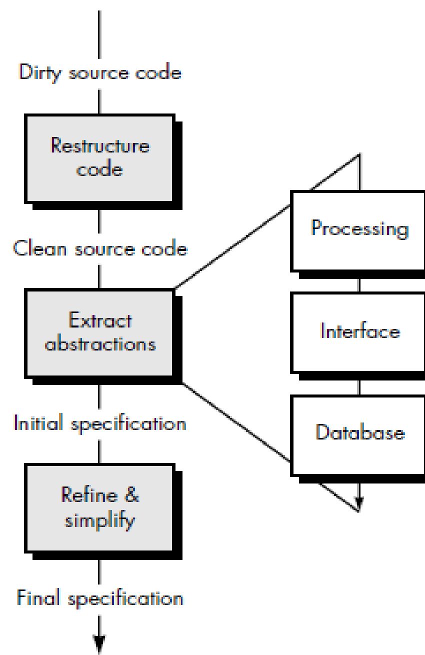
Data Restructuring.

A program with weak data architecture will be difficult to adapt and enhance. In fact, for many applications, data architecture has more to do with the long-term viability of a program that the source code itself. Unlike code restructuring, which occurs at a relatively low level of abstraction, data structuring 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 a 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 CASE 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, also called *renovation* or *reclamation*, 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.

REVERSE ENGINEERING

Reverse engineering can extract design information from source code, but the abstraction level, the completeness of the documentation, the degree to which tools and a human analyst work together, and the directionality of the process are highly variable. The *abstraction level* of a reverse engineering process and the tools used to effect it refers to the sophistication of the design information that can be extracted from source code. Ideally, the abstraction level should be as high as possible. That is, the reverse engineering process should be capable of deriving procedural design representations (a low-level abstraction), program and data structure information (a somewhat higher level of abstraction), data and control flow models (a relatively high level of abstraction), and entity relationship models (a high level of abstraction). As the abstraction level increases, the software engineer is provided with information that will allow easier understanding of the program.
The *completeness* of a reverse engineering process refers to the level of detail that is provided at an abstraction level. In most cases, the completeness decreases as the abstraction level increases. For example, given a source code listing, it is relatively easy to develop a complete procedural design representation. Simple data flow representations may also be derived, but it is far more difficult to develop a complete set of data flow diagrams or entity-relationship models. Completeness improves in direct proportion to the amount of analysis performed by the person doing reverse engineering. *Interactivity* refers to the degree to which the human is "integrated" with automated tools to create an effective reverse engineering process. In most cases, as the abstraction level increases, interactivity must increase or completeness will suffer. If the *directionality* of the reverse engineering process is one way, all information extracted from the source code is provided to the software engineer who can then use it during any maintenance activity. If directionality is two way, the information is fed to a reengineering tool that attempts to restructure or regenerate the old program.



Reverse Engineering to Understand Processing

The first real reverse engineering activity begins with an attempt to understand and then extract procedural abstractions represented by the source code. To understand procedural abstractions, the code is analyzed at varying levels of abstraction: system, program, component, pattern, and statement.

Reverse Engineering to Understand Data

Reverse engineering of data occurs at different levels of abstraction. At the program level, internal program data structures must often be reverse engineered as part of an overall reengineering effort. At the system level, global data structures (e.g., files, databases) are often reengineered to accommodate new database management paradigms.

Internal data structures.

Reverse engineering techniques for internal program data focus on the definition of classes of objects. This is accomplished by examining the program code with the intent of grouping related program variables. In many cases, the data organization within the code identifies abstract data types. For example, record structures, files, lists, and other data structures often provide an initial indicator of classes. Breuer and Lano suggest the following approach for reverse engineering of classes:

1. Identify flags and local data structures within the program that record important information about global data structures (e.g., a file or database).

2. Define the relationship between flags and local data structures and the global data structures. For example, a flag may be set when a file is empty; a local data structure may serve as a buffer that contains the last 100 records acquired from a central database.

3. For every variable (within the program) that represents an array or file, list all other variables that have a logical connection to it. These steps enable a software engineer to identify classes within the program that interact with the global data structures.

Database structure. Regardless of its logical organization and physical structure, a database allows the definition of data objects and supports some method for establishing relationships among the objects. Therefore, reengineering one database

schema into another requires an understanding of existing objects and their relationships.

The following steps may be used to define the existing data model as a precursor to reengineering a new database model:

1. Build an initial object model. The classes defined as part of the model may be acquired by reviewing records in a flat file database or tables in a relational schema. The items contained in records or tables become attributes of a class.

2. Determine candidate keys. The attributes are examined to determine whether they are used to point to another record or table. Those that serve as pointers become candidate keys.

3. Refine the tentative classes. Determine whether similar classes can be combined into a single class.

4. Define generalizations. Examine classes that have many similar attributes to determine whether a class hierarchy should be constructed with a generalization class at its head.

5. Discover associations. Use techniques that are analogous to the CRC approach to establish associations among classes. Once information defined in the preceding steps is known, a series of transformations can be applied to map the old database structure into a new database structure.

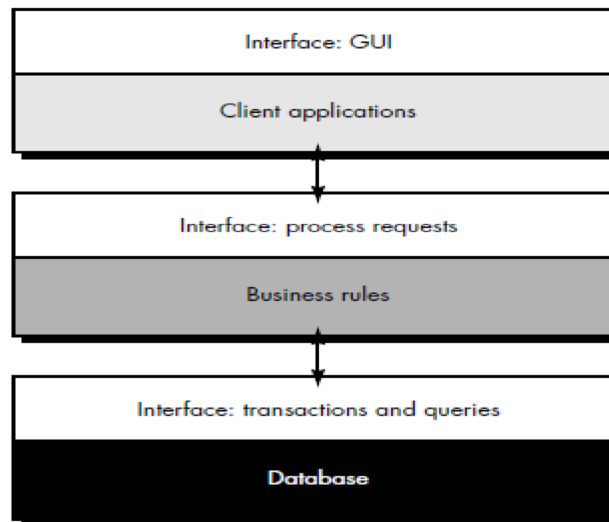
FORWARD ENGINEERING

The forward engineering process applies software engineering principles, concepts, and methods to re-create an existing application. In most cases, forward engineering does not simply create a modern equivalent of an older program. Rather, new user and technology requirements are integrated into the reengineering effort. The redeveloped program extends the capabilities of the older application.

Forward Engineering for Client/Server Architectures

Over the past decade many mainframe applications have been reengineered to accommodate client/server architectures. In essence, centralized computing resources (including software) are distributed among many client platforms. Although a variety of different distributed environments can be designed, the typical mainframe application that is reengineered into a client/server architecture has the following features:

- Application functionality migrates to each client computer.
- New GUI interfaces are implemented at the client sites.
- Database functions are allocated to the server.
- Specialized functionality (e.g., compute-intensive analysis) may remain at the server site.
- New communications, security, archiving, and control requirements must be established at both the client and server sites.



Reengineering mainframe applications to client/server

Forward Engineering for Object-Oriented Architectures

The data models created during reverse engineering are then used in conjunction with CRC modeling to establish the basis for the definition of classes. Class hierarchies, object-relationship models, object-behavior models, and subsystems are defined, and object-oriented design commences.

As object-oriented forward engineering progresses from analysis to design, a CBSE process model can be invoked. If the existing application exists within a domain that is already populated by many object-oriented applications, it is likely that a robust component library exists and can be used during forward engineering. For those classes that must be engineered from scratch, it may be possible to reuse algorithms and data structures from the existing conventional application. However, these must be redesigned to conform to the object-oriented architecture.

Forward Engineering User Interfaces

1. Understand the original interface and the data that move between it and the remainder of the application.
2. Remodel the behavior implied by the existing interface into a series of abstractions that have meaning in the context of a GUI.
3. Introduce improvements that make the mode of interaction more efficient.
4. Build and integrate the new GUI.

Software Project Management

The People
The Product
The Process
The Project

PEOPLE

the players who participate in the software process and the manner in which they are organized to perform effective software engineering.

The Players

The software process (and every software project) is populated by players who can be categorized into one of five constituencies:

1. Senior managers who define the business issues that often have significant influence on the project.

2. Project (technical) managers who must plan, motivate, organize, and control the practitioners who do software work.

3. Practitioners who deliver the technical skills that are necessary to engineer a product or application.

4. Customers who specify the requirements for the software to be engineered and other *stakeholders* who have a peripheral interest in the outcome.

5. End-users who interact with the software once it is released for production use.

Team Leaders

Project management is a people-intensive activity, and for this reason, competent practitioners often make poor team leaders. They simply don't have the right mix of people skills.

Motivation. The ability to encourage (by "push or pull") technical people to produce to their best ability.

Organization. The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.

Ideas or innovation. The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

Another view of the characteristics that define an effective project manager emphasizes four key traits:

- **Problem solving.** An effective software project manager can diagnose the technical and organizational issues that are most relevant, systematically structure a solution or properly motivate other practitioners to develop the solution, apply lessons learned from past projects to new situations, and remain flexible enough to change direction if initial attempts at problem solution are fruitless.

- **Managerial identity.** A good project manager must take charge of the project. He/She must have the confidence to assume control when necessary and the assurance to allow good technical people to follow their instincts.
- **Achievement.** To optimize the productivity of a project team, a manager must reward initiative and accomplishment and demonstrate through his own actionsthat controlled risk taking will not be punished.
- **Influence and team building.** An effective project manager must be able to "read" people; she must be able to understand verbal and nonverbal signals and react to the needs of the people sending these signals. The manager must remain under control in high-stress situations.

The Software Team

A new software project is within the project manager's purview.

The following options are available for applying human resources to a project that will require *n* people working for *k* years:

1. *n* individuals are assigned to *m* different functional tasks, relatively little combined work occurs; coordination is the responsibility of a software manager who may have six other projects to be concerned with.

2. *n* individuals are assigned to *m* different functional tasks ($m < n$) so that informal "teams" are established; an ad hoc team leader may be appointed; coordination among teams is the responsibility of a software manager.

3. *n* individuals are organized into *t* teams; each team is assigned one or more functional tasks; each team has a specific structure that is defined for all teams working on a project; coordination is controlled by both the team and a software project manager.

Three generic team organizations:

Democratic decentralized (DD). This software engineering team has no permanent leader. Rather, "task coordinators are appointed for short durations and then replaced by others who may coordinate different tasks." Decisions on problems and approach are made by group consensus. Communication among team members is horizontal.

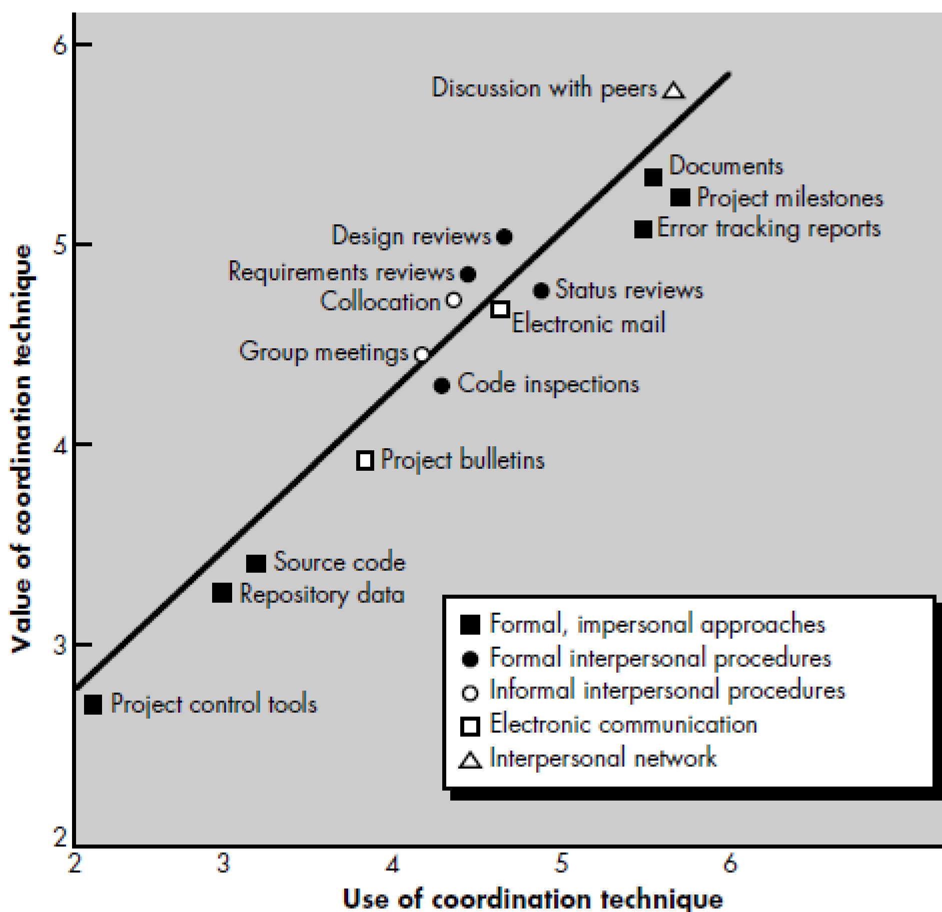
Controlled decentralized (CD). This software engineering team has a defined leader who coordinates specific tasks and secondary leaders that have responsibility for subtasks. Problem solving remains a group activity, but implementation of solutions is partitioned among subgroups by the team leader. Communication among subgroups and individuals is horizontal. Vertical communication along the control hierarchy also occurs.

Controlled Centralized (CC). Top-level problem solving and internal team coordination are managed by a team leader. Communication between the leader and team members is vertical. seven project factors that should be considered when planning the structure of software engineering teams:

- The difficulty of the problem to be solved.
- The size of the resultant program(s) in lines of code or function points
- The time that the team will stay together (team lifetime).
- The degree to which the problem can be modularized.
- The required quality and reliability of the system to be built.
- The rigidity of the delivery date.
- The degree of sociability (communication) required for the project.

Constantine suggests four "organizational paradigms" for software engineering teams:

1. A *closed paradigm* structures a team along a traditional hierarchy of authority (similar to a CC team). Such teams can work well when producing software that is quite similar to past efforts, but they will be less likely to be innovative when working within the closed paradigm.
2. The *random paradigm* structures a team loosely and depends on individual initiative of the team members. When innovation or technological breakthrough is required, teams following the random paradigm will excel. But such teams may struggle when "orderly performance" is required.
3. The *open paradigm* attempts to structure a team in a manner that achieves some of the controls associated with the closed paradigm but also much of the innovation that occurs when using the random paradigm. Work is performed collaboratively, with heavy communication and consensus-based decision making the trademarks of open paradigm teams. Open paradigm team structures are well suited to the solution of complex problems but may not perform as efficiently as other teams.
4. The *synchronous paradigm* relies on the natural compartmentalization of a problem and organizes team members to work on pieces of the problem with little active communication among themselves.
To achieve a high-performance team:
- Team members must have trust in one another.
- The distribution of skills must be appropriate to the problem.
- Mavericks may have to be excluded from the team, if team cohesiveness is to be maintained.



Value and Use of Coordination and Communication Techniques

THE PRODUCT
Software Scope

The first software project management activity is the determination of *software scope.* Scope is defined by answering the following questions:

Context. How does the software to be built fit into a larger system, product, or business context and what constraints are imposed as a result of the context?

Information objectives. What customer-visible data objects are produced as output from the software? What data objects are required for input?

Function and performance. What function does the software perform to transform input data into output? Are any special performance characteristics to be addressed?

THE PROCESS

The generic phases that characterize the software process—definition, development, and support—are applicable to all software. The problem is to select the process model that is appropriate for the software to be engineered by a project team. software engineering paradigms are

- the linear sequential model
- the prototyping model
- the RAD model
- the incremental model
- the spiral model
- the WINWIN spiral model
- the component-based development model
- the concurrent development model
- the formal methods model
- the fourth generation techniques model

Melding the Product and the Process

Customer communication—tasks required to establish effective requirements elicitation between developer and customer.

• **Planning**—tasks required to define resources, timelines, and other projectrelated information.

• **Risk analysis**—tasks required to assess both technical and management risks.

• **Engineering**—tasks required to build one or more representations of the application.

• **Construction and release**—tasks required to construct, test, install, and provide user support (e.g., documentation and training).

• **Customer evaluation**—tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering activity and implemented during the construction activity.

Common process framework activities	Customer communication	Planning	Risk analysis	Engineering	
Software engineering tasks					
Product functions					
Text input					
Editing and formatting					
Automatic copy edit					
Page layout capability					
Automatic indexing and TOC					
File management					
Document production					

Melding the Problem and the Process

Process Decomposition

A software team should have a significant degree of flexibility in choosing the software engineering paradigm that is best for the project and the software engineering tasks that populate the process model once it is chosen. A relatively small project that is similar to past efforts might be best accomplished using the linear sequential approach.
The following work tasks for the *customer communication* activity:
1. Develop list of clarification issues.
2. Meet with customer to address clarification issues.
3. Jointly develop a statement of scope.
4. Review the statement of scope with all concerned.
5. Modify the statement of scope as required.
Such a project might require the following work tasks for the customer communication activity:
1. Review the customer request.
2. Plan and schedule a formal, facilitated meeting with the customer.
3. Conduct research to specify the proposed solution and existing approaches.
4. Prepare a "working document" and an agenda for the formal meeting.
5. Conduct the meeting.
6. Jointly develop mini-specs that reflect data, function, and behavioral features of the software.
7. Review each mini-spec for correctness, consistency, and lack of ambiguity.
8. Assemble the mini-specs into a scoping document.
9. Review the scoping document with all concerned.
10. Modify the scoping document as required.

THE PROJECT
In order to manage a successful software project, we must understand what can gowrong (so that problems can be avoided) and how to do it right.
1.Software people don't understand their customer's needs.
2. The product scope is poorly defined.
3. Changes are managed poorly.
The chosen technology changes.
1. Business needs change [or are ill-defined].
2. Deadlines are unrealistic.
3. Users are resistant.
4. Sponsorship is lost [or was never properly obtained].
5. The project team lacks people with appropriate skills.
6. Managers [and practitioners] avoid best practices and lessons learned.

PROJECT PLANNING PROCESS
The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule.
The plan must be adapted and updated as the project proceeds.
Task Set for Project Planning
1. Establish project scope.
2. Determine feasibility.
3. Analyze risks.
4. Define required resources.
a. Determine required human resources.
b. Define reusable software resources.
c. Identify environmental resources.

5. Estimate cost and effort.
a. Decompose the problem.
b. Develop two or more estimates using size, function points, process tasks, or use cases.
c. Reconcile the estimates.
6. Develop a project schedule
a. Establish a meaningful task set.
b. Define a task network.
c. Use scheduling tools to develop a time-line chart.
d. Define schedule tracking mechanisms.

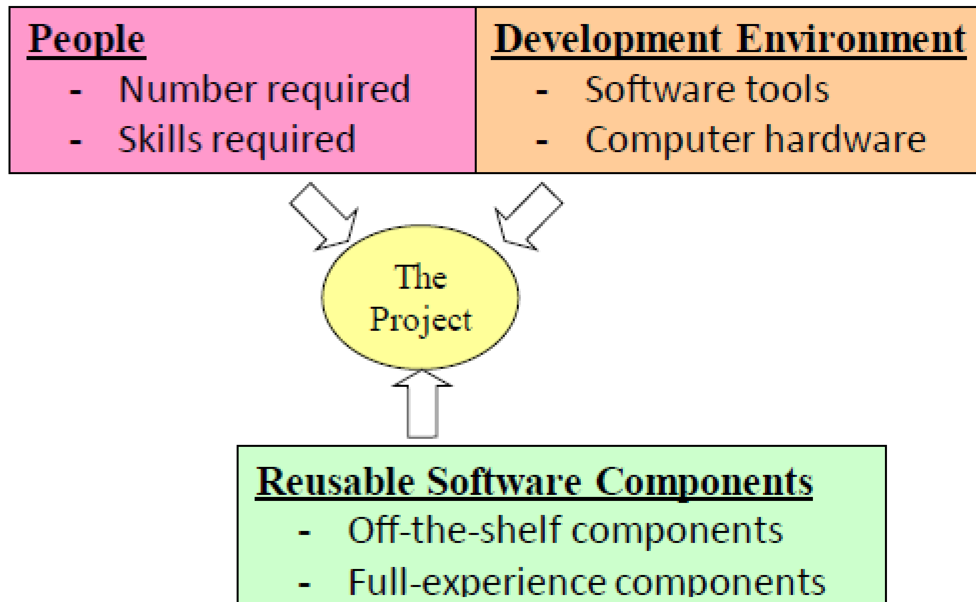
Scope and Feasibility

• Software scope describes
– The functions and features that are to be delivered to end users
– The data that are input to and output from the system
– The "content" that is presented to users as a consequence of using the software
– The performance, constraints, interfaces, and reliability that bound the system
• Scope can be define using two techniques
– A narrative description of software scope is developed after communication with all stakeholders
– A set of use cases is developed by end users
• After the scope has been identified, two questions are asked
– Can we build software to meet this scope?
– Is the project feasible?
• Software engineers too often rush (or are pushed) past these questions
• Later they become mired in a project that is doomed from the onset
• After the scope is resolved, feasibility is addressed
• Software feasibility has four dimensions
– Technology – Is the project technically feasible? Is it within the state of the art? Can defects be reduced to a level matching the application's needs?
– Finance – Is is financially feasible? Can development be completed at a cost that the software organization, its client, or the market can afford?
– Time – Will the project's time-to-market beat the competition?
– Resources – Does the software organization have the resources needed to succeed in doing the project?

Project Resources

• Three major categories of software engineering resources
– People
– Development environment
– Reusable software components
• Often neglected during planning but become a paramount concern during the construction phase of the software process
• Each resource is specified with
– A description of the resource
– A statement of availability
– The time when the resource will be required
– The duration of time that the resource will be applied

Categories of Resources



Human Resources

• Planners need to select the number and the kind of people skills needed to complete the project

• They need to specify the organizational position and job specialty for each person

• Small projects of a few person-months may only need one individual

• Large projects spanning many person-months or years require the location of the person to be specified also

• The number of people required can be determined only after an estimate of the development effort

Development Environment Resources

• A software engineering environment (SEE) incorporates hardware, software, and network resources that provide platforms and tools to develop and test software work products

• Most software organizations have many projects that require access to the SEE provided by the organization

• Planners must identify the time window required for hardware and software and verify that these resources will be available

Reusable Software Resources

Off-the-shelf components

Components are from a third party or were developed for a previous project

Ready to use; fully validated and documented; virtually no risk

Full-experience components

Components are similar to the software that needs to be built

Software team has full experience in the application area of these components

Modification of components will incur relatively low risk

Partial-experience components

Components are related somehow to the software that needs to be built but will require substantial modification
Software team has only limited experience in the application area of these components
Modifications that are required have a fair degree of risk
New components
Components must be built from scratch by the software team specifically for the needs of the current project
Software team has no practical experience in the application area
Software development of components has a high degree of risk.

SCHEDULING

Scheduling is an activity that distributes estimated effort across the planed project duration by allocating the effort to specific software engineering tasks.

First, a macroscopic schedule is developed. a detailed schedule is redefined for each entry in the macroscopic schedule.

A schedule evolves over time.

Basic principles guide software project scheduling:

- Compartmentalization

- Interdependency

- Time allocation

- Effort allocation

- Effort validation

- Defined responsibilities

- Defined outcomes

- Defined milestones

Problem-Based Estimation

LOC and FP data are used in two ways during software project estimation: (1) as estimation variables to ―size‖ each element of the software and (2) as baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

There is often substantial scatter in productivity metrics for an organization, making the use of a single-baseline productivity metric suspect. In general, LOC/pm or FP/pm averages should be computed by project domain. That is, projects should be grouped by team size, application area, complexity, and other relevant parameters. Local domain averages should then be computed. When a new project is estimated, it should first be allocated to a domain, and then the appropriate domain average for past productivity should be used in generating the estimate.

The LOC and FP estimation techniques differ in the level of detail required for decomposition and the target of the partitioning. When LOC is used as the estimation

variable, decomposition is absolutely essential and is often taken to considerable levels of detail. The greater the degree of partitioning, the more likely reasonably accurate estimates of LOC can be developed.

For FP estimates, decomposition works differently. Rather than focusing on function, each of the information domain characteristics—inputs, outputs, data files, inquiries, and external interfaces—as well as the 14 complexity adjustment values are estimated. The resultant estimates can then be used to derive an FP value that can be tied to past data and used to generate an estimate. Regardless of the estimation variable that is used, you should begin by estimating a range of values for each function or information domain value. Using historical data or (when all else fails) intuition, estimate an optimistic, most likely, and pessimistic size value for each function or count for each information domain value. An implicit indication of the degree of uncertainty is

Process-based Estimation

Process-based estimation begins with a delineation of software function obtained from the project scope. A series of framework activities must be performed for each function. Costs and effort for each function and framework activity are computed as the last step. If process-based estimation is performed independently of LOC or FP estimation, two or three estimates for cost and effort are considered that may be compared and reconciled. If both sets of estimates show reasonable agreement, there is good reason to believe that the estimates are reliable. If, on the other hand, the results of these decomposition techniques show little agreement, further investigation and analysis must be conducted.(LOC and FP-based estimations, process and problem-based estimations examples has to be provided.

Estimation with Use Cases

- Use cases are described using many different formats and styles—there is no standard form.
- Use cases represent an external view (the user‘s view) of the software and can therefore be written at many different levels of abstraction.
- Use cases do not address the complexity of the functions and features that are described.
- Use cases can describe complex behavior (e.g., interactions) that involves many functions and features.

Reconciling Estimates

The estimation techniques discussed in the preceding sections result in multiple estimates that must be reconciled to produce a single estimate of effort, project duration, or cost.

- the scope of the project is not adequately understood or has been misinterpreted by the planner, or
- Productivity data used for problem-based estimation techniques is inappropriate for the application, obsolete (in that it no longer accurately reflects the software engineering organization), or has been misapplied.

Empirical Estimation models

An estimation model should be calibrated to reflect local conditions. The model should be tested by applying data collected from completed projects, plugging the data into the model, and then comparing actual to predicted results. If agreement is poor, the model must be tuned and retested before it can be used.

The Structure of Estimation Models

A typical estimation model is derived using regression analysis on data collected from past software projects. The overall structure of such models takes the form

$E=A+B*(ev)^c$

where *A, B,* and *C* are empirically derived constants, *E* is effort in person-months, and *ev* is the estimation variable (either LOC or FP).

Among the many LOC-oriented estimation models proposed in the literature are

E=5.2*(KLOC)0.91	Walston-Felix model
E=5.5+0.73*(KLOC)1.16	Bailey-Basili model
E=3.2*(KLOC)1.05	Boehm simple model
E=5.288*(KLOC)1.047	Doty model for KLOC>9

FP-oriented models have also been proposed. These include

E= -91.4+0.355 FP	Albrecht and Gaffbey model
E=-37+0.96 FP	Kemerer model
E=-12.88+0.405 FP	Small project regression model

A quick examination of these models indicates that each will yield a different result for the same values of LOC or FP. The implication is clear. Estimation models must be calibrated for local needs.

Make/Buy Decision

• It is often more cost effective to acquire rather than develop software

• Managers have many acquisition options

– Software may be purchased (or licensed) off the shelf

– "Full-experience" or "partial-experience" software components may be acquired and integrated to meet specific needs

– Software may be custom built by an outside contractor to meet the purchaser's specifications

• The make/buy decision can be made based on the following conditions

– Will the software product be available sooner than internally developed software?

– Will the cost of acquisition plus the cost of customization be less than the cost of developing the software internally?

– Will the cost of outside support (e.g., a maintenance contract) be less than the cost of internal support?

COst COnstructive Model (COCOMO)

The Constructive Cost Model (COCOMO) is a software cost estimation model developed by Barry W. Boehm.

Basic COCOMO

Basic COCOMO computes software development effort (and cost) as a function of program size.

Program size is expressed in estimated thousands of source lines of code (SLOC, KLOC).

COCOMO applies to three classes of software projects:

- Organic projects - "small" teams with "good" experience working with "less than rigid" requirements
- Semi-detached projects - "medium" teams with mixed experience working with a mix of rigid and less than rigid requirements
- Embedded projects - developed within a set of "tight" constraints. It is also combination of organic and semi-detached projects.(hardware, software, operational, ...)

The basic COCOMO equations take the form

Effort Applied (E) = $a_b(KLOC)^b$

b **[man-months]**

Development Time (D) = $c_b(\text{Effort Applied})^d$

b **[months]**

People required (P) = Effort Applied / Development Time **[count]**

where, **KLOC** is the estimated number of delivered lines (expressed in thousands) of code for project. The coefficients *ab*, *bb*, *cb* and *db* are given in the following table:

Software project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Basic COCOMO is good for quick estimate of software costs. However it does not account for differences in hardware constraints, personnel quality and experience, use of modern tools and techniques, and so on.

Intermediate COCOMO

Intermediate *COCOMO* computes software development effort as function of program size and a set of "cost drivers" that include subjective assessment of product, hardware, personnel and project attributes.

This extension considers a set of four "cost drivers", each with a number of subsidiary attributes:-

- Product attributes
- Required software reliability
- Size of application database
- Complexity of the product

- Hardware attributes
- Run-time performance constraints
- Memory constraints
- Volatility of the virtual machine environment
- Required turnabout time
- Personnel attributes
- Analyst capability
- Software engineering capability
- Applications experience
- Virtual machine experience
- Programming language experience
- Project attributes
- Use of software tools
- Application of software engineering methods
- Required development schedule
- Each of the 15 attributes receives a rating on a six-point scale that ranges from "very low" to "extra high" (in importance or value). An effort multiplier from the table below applies to the rating. The product of all effort multipliers results in an ***effort adjustment factor (EAF)*****.** Typical values for EAF range from 0.9 to1.4.

Cost Drivers	**Ratings**					
	Very Low	**Low**	**Nominal**	**High**	**Very High**	**Extra High**
Product attributes						
Required software reliability	0.75	0.88	1.00	1.15	1.40	
Size of application database		0.94	1.00	1.08	1.16	
Complexity of the product	0.70	0.85	1.00	1.15	1.30	1.65
Hardware attributes						
Run-time performance constraints			1.00	1.11	1.30	1.66
Memory constraints			1.00	1.06	1.21	1.56
Volatility of the virtual machine environment		0.87	1.00	1.15	1.30	
Required turnabout time		0.87	1.00	1.07	1.15	
Personnel attributes						
Analyst capability	1.46	1.19	1.00	0.86	0.71	
Applications experience	1.29	1.13	1.00	0.91	0.82	
Software engineer capability	1.42	1.17	1.00	0.86	0.70	
Virtual machine experience	1.21	1.10	1.00	0.90		
Programming language experience	1.14	1.07	1.00	0.95		
Project attributes						
Application of software engineering methods	1.24	1.10	1.00	0.91	0.82	
Use of software tools	1.24	1.10	1.00	0.91	0.83	
Required development schedule	1.23	1.08	1.00	1.04	1.10	

The Intermediate Cocomo formula now takes the form:

E=*ai* (KLoC)*(bi)*(EAF)

where E is the effort applied in person-months, **KLoC** is the estimated number of thousands of delivered lines of code for the project, and **EAF** is the factor calculated above. The coefficient **ai** and the exponent **bi** are given in the next table.

Software project	a_i	b_i
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

The Development time **D** calculation uses **E** in the same way as in the Basic COCOMO.

Detailed COCOMO

- Detailed COCOMO incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step (analysis, design, etc.) of the software engineering process.
- The detailed model uses different effort multipliers for each cost driver attribute. These **Phase Sensitive** effort multipliers are each to determine the amount of effort required to complete each phase. In detailed cocomo,the whole software is divided in different modules and then we apply COCOMO in different modules to estimate effort and then sum the effort
- In detailed COCOMO, the effort is calculated as function of program size and a set of cost drivers given according to each phase of software life cycle.
- A Detailed project schedule is never static.

The Six phases of detailed COCOMO are:-

- plan and requirement.
- system design.
- detailed design.
- module code and test.
- integration and test.
- Cost Costructive Model

COCOMO II

Barry Boehm introduced COCOMO II (COst COnstructive MOdel) which is an hierarchy of estimation
models that addresses the following areas:

Application composition model. Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of
performance, and evaluation of technology maturity are paramount.

Early design stage model. Used once requirements have been stabilized and basic software
architecture has been established.

Post-architecture-stage model. Used during the construction of the software.

Three different sizing options are available as part of the model hierarchy: object points, function points,
and lines of source code.

OBJECT TYPE	COMPLEXITY WEIGHT		
	SIMPLE	MEDIUM	DIFFICULT
Screen	1	2	3
Report	2	5	8
3GL component			10

The object point is an indirect software measure that is computed using counts of the number of (1) screens (at the user interface), (2) reports, and (3) components likely to be required to build the application.

When component-based development or general software reuse is to be applied, the percent of reuse (%reuse) is estimated and the object point count is adjusted:

NOP=(Object points)*[(100-%reuse)/100]

where NOP is defined as new object points.

To derive an estimate of effort based on the computed NOP value, a ―productivity rate ‖ must be derived.

$$PROD = \frac{NOP}{\text{Person-month}}$$

for different levels of developer experience and development environment maturity.

Once the productivity rate has been determined, an estimate of project effort is computed using

$$\text{Estimated effort} = \frac{NOP}{PROD}$$

In more advanced COCOMO II models, 12 a variety of scale factors, cost drivers, and adjustment procedures are required.

Developer's experience/capability	Very low	Low	Nominal	High	Very high
Environment maturity/capability	Very low	Low	Nominal	High	Very high
PROD	4	7	13	25	50

PROJECT SCHEDULING AND TRACKING

Software project scheduling is an action that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. Scheduling for software engineering projects can be viewed from two rather different perspectives. In the first, an end date for release of a computer-based system has already (and irrevocably) been established. The software organization is constrained to distribute effort within the prescribed time frame. The second view of software scheduling assumes that rough chronological bounds have been discussed but that the end date is set by the software engineering organization.

Basic Principles

Compartmentalization. The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are refined.

Interdependency. The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence, while others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently.

Time allocation. Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.

Effort validation. Every project has a defined number of people on the software team. As time allocation occurs, you must ensure that no more than the allocated number of people has been scheduled at any given time. For example, consider a project that has three assigned software engineers (e.g., three person-days are available per day of assigned effort4). On a given day, seven concurrent tasks must be accomplished. Each task requires 0.50 person-days of effort. More effort has been allocated than there are people to do the work.

Defined responsibilities. Every task that is scheduled should be assigned to a specific team member.

Defined outcomes. Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a component) or a part of a work product. Work products are often combined in deliverables.

Defined milestones. Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality.

The Relationship Between People and Effort

L, is related to effort and development time by the equation:

L=P*E1/3t4/3

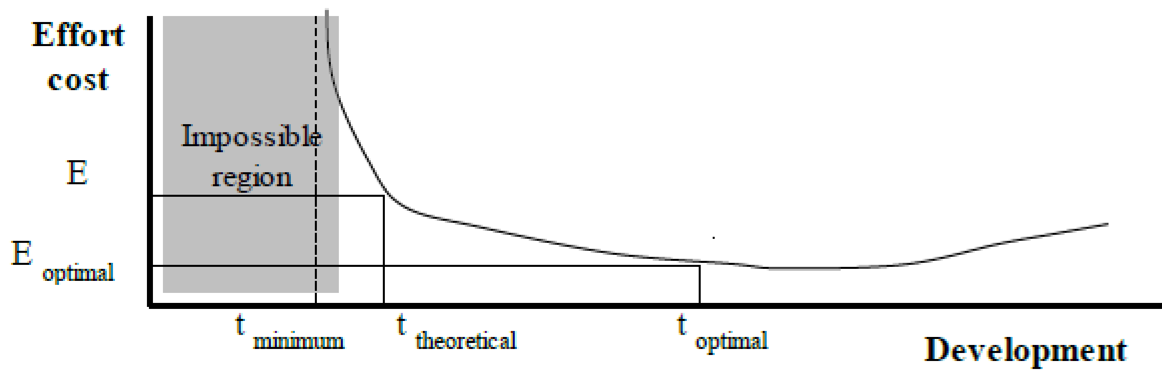
Rearranging this software equation, we can arrive at an expression for development effort *E*:

$$E=\frac{L3}{P3\ t4}$$

Effort Distribution

A recommended distribution of effort across the software process is often referred to as the 40–20–40 rule. Forty percent of all effort is allocated to frontend analysis and design. A similar percentage is applied to back-end testing. 20 percent of effort is deemphasized in the coding.

Because of the effort applied to software design, code should follow with relatively little difficulty. A range of 15 to 20 percent of overall effort can be achieved. Testing and subsequent debugging can account for 30 to 40 percent of software development effort. The criticality of the software often dictates the amount of testing that is required. If software is human rated (i.e., software failure can result in loss of life), even higher percentages are typical.



Task Network
Task Set

- A task set is the work breakdown structure for the project
- No single task set is appropriate for all projects and process models

– It varies depending on the project type and the degree of rigor (based on influential factors) with which the team plans to work

- The task set should provide enough discipline to achieve high software quality

– But it must not burden the project team with unnecessary work

Purpose of a Task Network

- Also called an activity network
- It is a graphic representation of the task flow for a project
- It depicts task length, sequence, concurrency, and dependency
- Points out inter-task dependencies to help the manager ensure continuous progress toward project completion
- The critical path

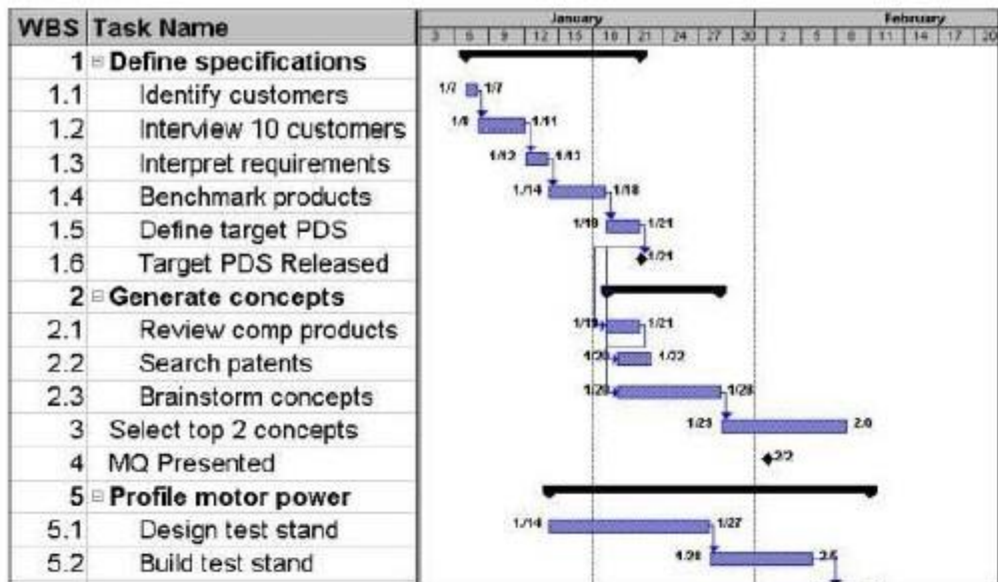
⎫ A single path leading from start to finish in a task network

⎫ It contains the sequence of tasks that must be completed on schedule if the project as a whole is to be completed on schedule

⎫ It also determines the minimum duration of the project

Timeline Chart

- Also called a Gantt chart; invented by Henry Gantt, industrial engineer, 1917
- All project tasks are listed in the far left column
- The next few columns may list the following for each task: projected start date, projected stop date, projected duration, actual start date, actual stop date, actual duration, task inter-dependencies (i.e., predecessors)
- To the far right are columns representing dates on a calendar
- The length of a horizontal bar on the calendar indicates the duration of the task
- When multiple bars occur at the same time interval on the calendar, this implies task concurrency
- A diamond in the calendar area of a specific task indicates that the task is a milestone; a milestone has a time duration of zero



Project Table

Work tasks	Planned start	Actual start	Planned complete	Actual complete	Assigned person	Effort allocated	Notes
I.1.1 Identify needs and benefits							Scoping will require more effort/time
Meet with customers	wk1, d1	wk1, d1	wk1, d2	wk1, d2	BLS	2 p-d	
Identify needs and project constraints	wk1, d2	wk1, d2	wk1, d2	wk1, d2	JPP	1 p-d	
Establish product statement	wk1, d3	wk1, d3	wk1, d3	wk1, d3	BLS/JPP	1 p-d	
Milestone: Product statement defined	wk1, d3	wk1, d3	wk1, d3	wk1, d3			
I.1.2 Define desired output/control/input (OCI)							
Scope keyboard functions	wk1, d4	wk1, d4	wk2, d2		BLS	1.5 p-d	
Scope voice input functions	wk1, d3	wk1, d3	wk2, d2		JPP	2 p-d	
Scope modes of interaction	wk2, d1		wk2, d3		MLL	1 p-d	
Scope document diagnostics	wk2, d1		wk2, d2		BLS	1.5 p-d	
Scope other WP functions	wk1, d4	wk1, d4	wk2, d3		JPP	2 p-d	
Document OCI	wk2, d1		wk2, d3		MLL	3 p-d	
FTR: Review OCI with customer	wk2, d3		wk2, d3		all	3 p-d	
Revise OCI as required	wk2, d4		wk2, d4		all	3 p-d	
Milestone: OCI defined	wk2, d5		wk2, d5				
I.1.3 Define the function/behavior							

Process and Project Metrics

Overview

Software process and project metrics are quantitative measures that enable software engineers to gain insight into the efficiency of the software process and the projects conducted using the process framework. In software project management, we are primarily concerned with productivity and quality metrics. There are four reasons for measuring software processes, products, and resources (to characterize, to evaluate, to predict, and to improve).

Process and Project Metrics

- Metrics should be collected so that process and product indicators can be ascertained
- *Process metrics* used to provide indictors that lead to long term process improvement
- *Project metrics* enable project manager to
  - Assess status of ongoing project
  - Track potential risks
  - Uncover problem are before they go critical
  - Adjust work flow or tasks
  - Evaluate the project team's ability to control quality of software wrok products.

Process Metrics

- Private process metrics (e.g. defect rates by individual or module) are only known to by the individual or team concerned.
- Public process metrics enable organizations to make strategic changes to improve the software process.
- Metrics should not be used to evaluate the performance of individuals.
- Statistical software process improvement helps and organization to discover where they are strong and where are week.

Statistical Process Control

1. Errors are categorized by their origin
2. Record cost to correct each error and defect
3. Count number of errors and defects in each category
4. Overall cost of errors and defects computed for each category
5. Identify category with greatest cost to organization
6. Develop plans to eliminate the most costly class of errors and defects or at least reduce their frequency

Project Metrics

- A software team can use software project metrics to adapt project workflow and technical activities.
- Project metrics are used to avoid development schedule delays, to mitigate potential risks, and to assess product quality on an on-going basis.
- Every project should measure its inputs (resources), outputs (deliverables), and results (effectiveness of deliverables).

Software Measurement

- *Direct process measures* include cost and effort.
- *Direct process measures* include lines of code (LOC), execution speed, memory size, defects rep orted over some time period.
- *Indirect product measures* examine the quality of the software product itself (e.g. functionality, complexity, efficiency, reliability, maintainability).

Size-Oriented Metrics

- Derived by normalizing (dividing) any direct measure (e.g. defects or human effort) associated with the product or project by LOC.
- Size oriented metrics are widely used but their validity and applicability is widely debated.

Function-Oriented Metrics

- Function points are computed from direct measures of the information domain of a business software application and assessment of its complexity.
- Once computed function points are used like LOC to normalize measures for software productivity, quality, and other attributes.

- The relationship of LOC and function points depends on the language used to implement the software.

Reconciling LOC and FP Metrics

- The relationship between lines of code and function points depends upon the programming language that is used to implement the software and the quality of the design
- Function points and LOC-based metrics have been found to be relatively accurate predictors of software development effort and cost
- Using LOC and FP for estimation a historical baseline of information must be established.

Object-Oriented Metrics

- Number of scenario scripts (NSS)
- Number of key classes (NKC)
- Number of support classes (e.g. UI classes, database access classes, computations classes, etc.)
- Average number of support classes per key class
- Number of subsystems (NSUB)

Use Case-Oriented Metrics

- Describe (indirectly) user-visible functions and features in language independent manner
- Number of use case is directly proportional to LOC size of application and number of test cases needed
- However use cases do not come in standard sizes and use as a normalization measure is suspect
- Use case points have been suggested as a mechanism for estimating effort

WebApp Project Metrics

- Number of static Web pages (N_{sp})
- Number of dynamic Web pages (N_{dp})
- Customization index: $C = N_{sp} / (N_{dp} + N_{sp})$
- Number of internal page links
- Number of persistent data objects
- Number of external systems interfaced
- Number of static content objects
- Number of dynamic content objects
- Number of executable functions

Software Quality Metrics

- Factors assessing software quality come from three distinct points of view (product operation, product revision, product modification).
- Software quality factors requiring measures include
  - correctness (defects per KLOC)

- maintainability (mean time to change)
- integrity (threat and security)
- usability (easy to learn, easy to use, productivity increase, user attitude)

- Defect removal efficiency (DRE) is a measure of the filtering ability of the quality assurance and control activities as they are applied through out the process framework

  DRE = E / (E + D)

  E = number of errors found before delivery of work product

  D = number of defects found after work product delivery

Integrating Metrics with Software Process

- Many software developers do not collect measures.
- Without measurement it is impossible to determine whether a process is improving or not.
- Baseline metrics data should be collected from a large, representative sampling of past software projects.
- Getting this historic project data is very difficult, if the previous developers did not collect data in an on-going manner.

Arguments for Software Metrics

- If you don't measure you have no way of determining any improvement
- By requesting and evaluating productivity and quality measures software teams can establish meaningful goals for process improvement
- Software project managers are concerned with developing project estimates, producing high quality systems, and delivering product on time
- Using measurement to establish a project baseline helps to make project managers tasks possible.

Baselines

- Establishing a metrics baseline can benefit portions of the process, project, and product levels
- Baseline data must often be collected by historical investigation of past project (better to collect while projects are on-going)
- To be effective the baseline data needs to have the following attributes:
  - data must be reasonably accurate, not guesstimates
  - data should be collected for as many projects as possible
  - measures must be consistent
  - applications should be similar to work that is to be estimated

Metrics for Small Organizations

- Most software organizations have fewer than 20 software engineers.
- Best advice is to choose simple metrics that provide value to the organization and don't require a lot of effort to collect.

- Even small groups can expect a significant return on the investment required to collect metrics, if this activity leads to process improvement.

Establishing a Software Metrics Program

1. Identify business goal
2. Identify what you want to know
3. Identify subgoals
4. Identify subgoal entities and attributes
5. Formalize measurement goals
6. Identify quantifiable questions and indicators related to sub goals
7. Identify data elements needed to be collected to construct the indicators
8. Define measures to be used and create operational definitions for them
9. Identify actions needed to implement the measures
10. Prepare a plan to implement the measures

Project Scheduling

Software project scheduling is an action that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. Scheduling for software engineering projects can be viewed from two rather different perspectives. In the first, an end date for release of a computer-based system has already (and irrevocably) been established. The software organization is constrained to distribute effort within the prescribed time frame. The second view of software scheduling assumes that rough chronological bounds have been discussed but that the end date is set by the software engineering organization.

Basic Principles

Compartmentalization. The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are refined.

Interdependency. The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence, while others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently.

Time allocation. Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.

Effort validation. Every project has a defined number of people on the software team. As time allocation occurs, you must ensure that no more than the allocated number of people has been scheduled at any given time. For example, consider a project that has three assigned software engineers (e.g., three person-days are available per day of assigned effort4). On a given day, seven concurrent tasks must be accomplished. Each task requires 0.50 person-days of effort. More effort has been allocated than there are people to do the work.

Defined responsibilities. Every task that is scheduled should be assigned to a specific team member.

Defined outcomes. Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a component) or a part of a work product. Work products are often combined in deliverables.

Defined milestones. Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality.

The Relationship Between People and Effort

L, is related to effort and development time by the equation:

$$L=P*E1/3t4/3$$

Rearranging this software equation, we can arrive at an expression for development effort *E*:

$$E=\frac{L3}{P3\ t4}$$

Effort Distribution

A recommended distribution of effort across the software process is often referred to as the 40–20–40 rule. Forty percent of all effort is allocated to frontend analysis and design. A similar percentage is applied to back-end testing. 20 percent of effort is deemphasized in the coding.

Because of the effort applied to software design, code should follow with relatively little difficulty. A range of 15 to 20 percent of overall effort can be achieved. Testing and subsequent debugging can account for 30 to 40 percent of software development effort. The criticality of the software often dictates the amount of testing that is required. If software is human rated (i.e., software failure can result in loss of life), even higher percentages are typical.

Decomposition The decomposition approach was discussed from two different points of view: decomposition of the problem and decomposition of the process.

Software Sizing

The accuracy of a software project estimate is predicated on a number of things:

- the degree to which you have properly estimated the size of the product to be built;
- the ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects);
- the degree to which the project plan reflects the abilities of the software team; and
- the stability of product requirements and the environment that supports the software engineering effort.

Earned Value Analysis

The earned value system provides a common value scale for every [software project] task, regardless of the type of work being performed. The total hours to do the whole project are estimated, and every task is given an earned value based on its estimated percentage of the total.

To determine the earned value, the following steps are performed:

The budgeted cost of work scheduled (BCWS) is determined for each work task represented in the schedule.

The BCWS values for all work tasks are summed to derive the budget at completion (BAC). Hence,

$BAC=(BCWS_k)$ for all task k

Next, the value for budgeted cost of work performed (BCWP) is computed. The value for BCWP is the sum of the BCWS values for all work tasks that have actually been completed by a point in time on the project schedule.

Given values for BCWS, BAC, and BCWP, important progress indicators can be computed:

$$\text{Schedule performance index, SPI} = \frac{BCWP}{BCWS}$$

Schedule Variance, SV = BCWP-BCWS

$$\text{Percent scheduled for completion} = \frac{BCWS}{BAC}$$

$$\text{Percent complete} = \frac{BCWP}{BAC}$$

$$\text{Cost performance index, CPI} = \frac{BCWP}{ACWP}$$

Cost variance, CV=BCWP-ACWP

A CPI value close to 1.0 provides a strong indication that the project is within its defined budget. CV is an absolute indication of cost savings (against planned costs) or shortfall at a particular stage of a project. Like over-the-horizon radar, earned value analysis illuminates scheduling difficulties before they might otherwise be apparent. This enables you to take corrective action before a project crisis develops.

Risk Management

Risk concerns with failure happenings. Risk involves change, such as in changes in mind, opinion, actions, or places.

- uncertainty—the risk may or may not happen;
- loss—if the risk becomes a reality, unwanted consequences or losses will occur

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.

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

- building an excellent product or system that no one really wants (market risk),
- building a product that no longer fits into the overall business strategy for the company (strategic risk),
- building a product that the sales force doesn‘t understand how to sell (sales risk),
- losing the support of senior management due to a change in focus or a change in people (management risk), and
- losing budgetary or personnel commitment (budget risks).

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.

Risk Identification

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.

There are two distinct types of risks for each of the categories: generic risks and product-specific risks.

- Generic risks are a potential threat to every software project.
- Product-specific risks can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built.

One method for identifying risks is to create a risk item checklist. The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:

- Product size—risks associated with the overall size of the software to be built or modified.

- Business impact—risks associated with constraints imposed by management or the marketplace.

 Stakeholder characteristics—risks associated with the sophistication of the stakeholders and the developer's ability to communicate with stakeholders in a timely manner.

 Process definition—risks associated with the degree to which the software process has been defined and is followed by the development organization.

 Development environment—risks associated with the availability and quality of the tools to be used to build the product.

 Technology to be built—risks associated with the complexity of the system to be built and the ―newness‖ of the technology that is packaged by the system.
 Staff size and experience—risks associated with the overall technical and project experience of the software engineers who will do the work.

Risk Projection

Risk projection, also called risk estimation, attempts to rate each risk in two ways— the likelihood or probability that the risk is real and the consequences of the problems associated with the risk, should it occur. You work along with other managers and technical staff to perform four risk projection steps:

 Establish a scale that reflects the perceived likelihood of a risk.
 Delineate the consequences of the risk.
 Estimate the impact of the risk on the project and the product.
 Assess the overall accuracy of the risk projection so that there will be no misunderstandings.
A risk table provides you with a simple technique for risk projection.
The overall risk exposure RE is determined using the following relationship:

$$RE = P * C$$

Risk Refinement

One way to do this is to represent the risk in condition-transition-consequence (CTC) format. That is, the risk is stated in the following form:
Given that <condition> then there is concern that (possibly) <consequence>.

This general condition can be refined in the following manner:

Subcondition 1. Certain reusable components were developed by a third party with no knowledge of internal design standards.

Subcondition 2. The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.

Subcondition 3. Certain reusable components have been implemented in a language that is not supported on the target environment.

Risk Mitigation, Monitoring and Management

An effective strategy must consider three issues: risk avoidance, risk monitoring, and risk management and contingency planning.

To mitigate this risk, a strategy has to be developed for reducing turnover. Among the possible steps to be taken are:

- Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market).
- Mitigate those causes that are under your control before the project starts.
- Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.
- Organize project teams so that information about each development activity is widely dispersed.
- Define work product standards and establish mechanisms to be sure that all models and documents are developed in a timely manner.
- Conduct peer reviews of all work (so that more than one person is ―up to speed).
- Assign a backup staff member for every critical technologist.