SOFTWARE ENGINEERING

UNIT I

SOFTWARE

Software is defined as
Instructions
-Programs that when executed provide desired function
Data structures
-Enable the programs to adequately manipulate information
Documents
-Describe the operation and use of the programs.
Definition of Engineering
-Application of science, tools and methods to find cost effective solution to problems

Definition of SOFTWARE ENGINEERING
Software engineering is the application of engineering to the development of software in a systematic method.

-SE is defined as systematic, disciplined and quantifiable approach for the development, operation and maintenance of software

Software is the set of instructions encompasses programs that execute within a computer of any size and architecture, documents that encompass hard-copy and virtual forms, and data that combine numbers and text. It also includes representations of pictorial, video, and audio information. Software engineers can build the software and virtually everyone in the industrialized world uses it either directly or indirectly. It is so important because it affects nearly every aspect of our lives and has become pervasive in our commerce, our culture, and our everyday activities. The steps to build the computer software is as the user would like to build any successful product, by applying a process that leads to a high-quality result that meets the needs of the people who will use the product. From the software engineer's view, the product is may be the programs, documents, and data that are computer software. But from the user's viewpoint, the product is the resultant information that somehow makes the user's world better. Software's impact on the society and culture continues to be profound. As its importance grows, the software community continually attempts to develop technologies that will make it easier, faster, and less expensive to build high-quality computer programs. Some of these technologies are targeted at a specific application domain like web-site design and implementation; others focus on a technology domain such as object oriented systems and still others are broad-based like operating systems such as LINUX.

However, a software technology has to develop useful information. The technology encompasses a process, a set of methods, and an array of tools called as software engineering.

A GENERIC VIEW OF PROCESS–A LAYERED TECHNOLOGY

Software engineering encompasses a process, the management of activities, technical methods, and use of tools to develop software products.
Fritz Bauer defined Software engineering as the “establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines. “
IEEE definition of software engineering (1) the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).
We need discipline but we also need adaptability and agility.
Software Engineering is a layered technology as shown below. Any engineering approach must rest on an organizational commitment to quality.
The bedrock that supports software engineering is a quality focus.

The foundation for S/W eng is the process layer. It is the glue that holds the technology layers together and enables rational and timely development of computer S/W.

Process defines a framework that must be established for effective delivery of S/W eng technology.

The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products (models, documents, data, reports, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

S/W eng methods provide the technical “how to’s” for building S/W. Methods encompass a broad array of tasks that include communication, req. analysis, design, coding, testing and support.

S/W eng tools provide automated or semi-automated support for the process and the methods.

When tools are integrated so that info. Created by one tool can be used by another, a system for the support of S/W development called computer-aided software engineering is established.

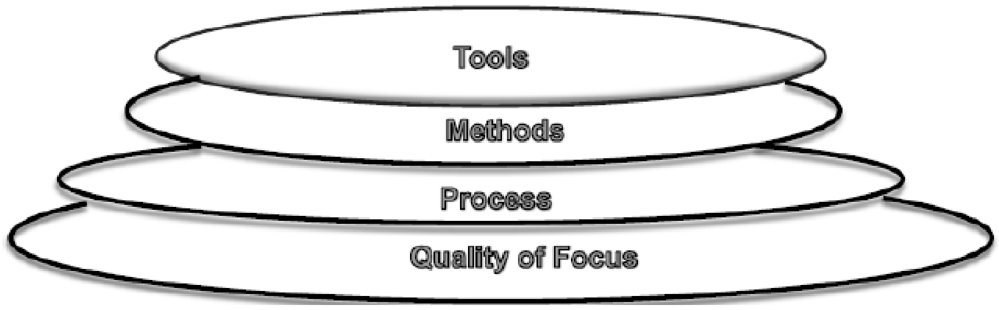


Figure1. Generic View of Process

A PROCESS FRAMEWORK

Software process models can be prescriptive or agile, complex or simple, all-encompassing or targeted, but in every case, five key activities must occur. The framework activities are applicable to all projects and all application domains, and they are a template for every process model.

- **Software process**
  - **Process framework**
    - **Umbrella activities**
      - **Framework activity #1**
        - **Software Engineering action**

Each framework activity is populated by a set of S/W eng actions –a collection of related tasks that produces a major S/W eng work product (design is a S/W eng action). 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 the vast majority of S/W projects.

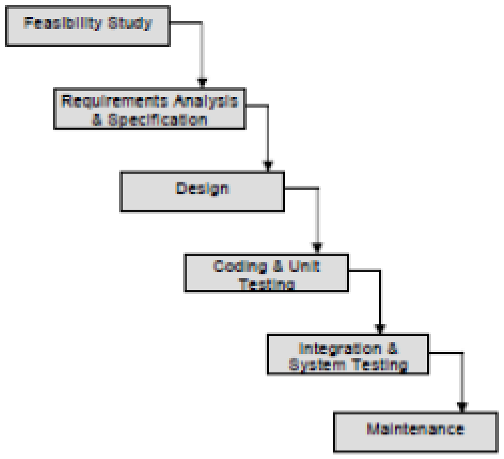
- **Communication:** involves heavy communication with the customer (and other stakeholders) and encompasses requirements gathering.
- **Planning:** Describes the technical tasks to be conducted, the risks that are likely, resources that will be required, the work products to be produced and a work schedule.
- **Modeling:** encompasses the creation of models that allow the developer and customer to better understand S/W req. and the design that will achieve those req.
- **Construction:** combines code generation and the testing required uncovering errors in the code.
- **Deployment:** deliver the product to the customer who evaluates the delivered product and provides feedback.

Each S/W eng action is represented by a number of different task sets –each a collection of S/W eng work tasks, related work products, quality assurance points, and project milestones.
The task set that best accommodates the needs of the project and the characteristics of the team is chosen.
The framework described in the generic view of S/W eng is complemented by a number of **umbrella activities**. Typical activities include:

- **S/W project tracking and control:** allows the team to assess progress against the project plan and take necessary action to maintain schedule.
- Risk Management: Assesses the risks that may affect the outcome of the project or the quality.
- **Software quality assurance:** defines and conducts the activities required to ensure software quality.
- **Formal Technical Review:** uncover and remove errors before they propagate to the next action.
- **Measurement:** defines and collects process, project, and product measures that assist the team in delivering S/W that meets customers' needs.
- **Software configuration management:** Manages the effect of change throughout the S/W process
- **Reusability management:** defines criteria for work product reuse.
- **Work product preparation and production:** encompasses the activities required to create work products such as models, documents, etc.

PROCESS MODELS

 The Waterfall Model


Feasibility Study
Requirements Analysis & Specification
Design
Coding & Unit Testing
Integration & System Testing
Maintenance

The classical waterfall model is intuitively the most obvious way to develop software. Though the classical waterfall model is elegant and intuitively obvious, it is not a practical model in the sense that it can not be used in actual software development projects. Thus, this model can be considered to be a *theoretical way of developing software*. But all other life cycle models are essentially derived from the classical waterfall model. So, in order to be able to appreciate other life cycle models it is necessary to learn the classical waterfall model.
Classical waterfall model divides the life cycle into the following phases as shown in fig.

Feasibility Study

The main aim of feasibility study is to determine whether it would be financially and technically feasible to develop the product.
At first project managers or team leaders try to have a rough understanding of what is required to be done by visiting the client side. They study different input data to the system and output data to be produced by the system. They study what kind of processing is needed to be done on these data and they look at the various constraints on the behavior of the system.

After they have an overall understanding of the problem they investigate the different solutions that are possible. Then they examine each of the solutions in terms of what kind of resources required, what would be the cost of development and what would be the development time for each solution.
Based on this analysis they pick the best solution and determine whether the solution is feasible financially and technically. They check whether the customer budget would meet the cost of the product and whether they have sufficient technical expertise in the area of development.

Requirements Analysis and Specification
The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely
Requirements gathering and analysis, and
Requirements specification
The goal of the requirements gathering activity is to collect all relevant information from the customer regarding the product to be developed. This is done to clearly understand the customer requirements so that incompleteness and inconsistencies are removed.
The requirements analysis activity is begun by collecting all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussions. After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the requirements specification activity can start. During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document.
The customer requirements identified during the requirements gathering and analysis activity are organized into a SRS document. The important components of this document are functional requirements, the nonfunctional requirements, and the goals of implementation.

Design
The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the software architecture is derived from the SRS document. Two distinctly different approaches are available: the traditional design approach and the object-oriented design approach.

Traditional design approach

Traditional design consists of two different activities; first a structured analysis of the requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a structured design activity. During structured design, the results of structured analysis are transformed into the software design.

Object-oriented design approach

In this technique, various objects that occur in the problem domain and the solution domain are first identified, and the different relationships that exist among these objects are identified. The object structure is further refined to obtain the detailed design.

Coding and Unit Testing

The purpose of the coding and unit testing phase (sometimes called the implementation phase) of software development is to translate the software design into source code. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually tested.

During this phase, each module is unit tested to determine the correct working of all the individual modules. It involves testing each module in isolation as this is the most efficient way to debug the errors identified at this stage.

Integration and system testing: -

Integration of different modules is undertaken once they have been coded and unit tested. During the integration and system testing phase, the modules are integrated in a planned manner. The different modules making up a software product are almost never integrated in one shot. Integration is normally carried out incrementally over a number of steps. During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it. Finally, when all the modules have been successfully integrated and tested, system testing is carried out. The goal of system testing is to ensure that the developed system conforms to its requirements laid out in the SRS document. System testing usually consists of three different kinds of testing activities:

α –testing: It is the system testing performed by the development team.

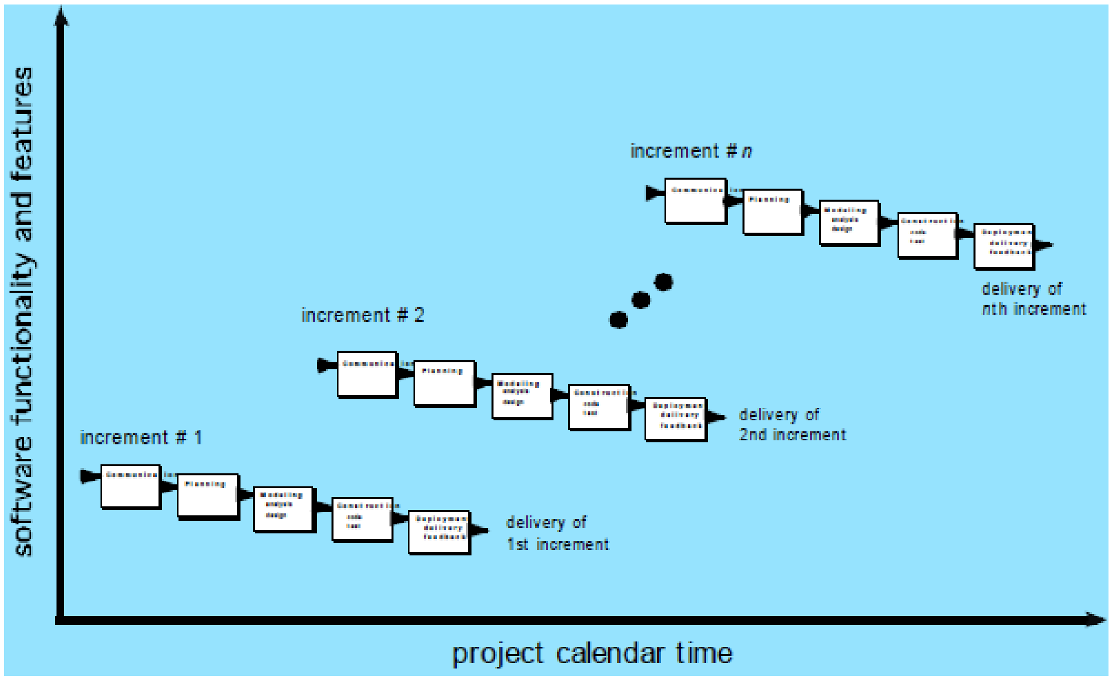
β –testing: It is the system testing performed by a friendly set of customers.

acceptance testing: It is the system testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered product.

Maintenance
Maintenance of a typical software product requires much more than the effort necessary to develop the product itself. Many studies carried out in the past confirm this and indicate that the relative effort of development of a typical software product to its maintenance effort is roughly in the 40:60 ratio.
Maintenance involves performing any one or more of the following three kinds of activities:
Correcting errors that were not discovered during the product development phase. This is called **corrective maintenance.**
Improving the implementation of the system, and enhancing the functionalities of the system according to the customer's requirements. This is called **perfective maintenance**.
Porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system. This is called **adaptive maintenance**.

Incremental Process Models
The process models in this category tend to be among the most widely used (and effective) in the industry.

a. The Incremental Model


software functionality and features
increment # n
increment # 2
increment # 1
delivery of nth increment
delivery of 2nd increment
delivery of 1st increment
project calendar time

incremental model combines elements of the *waterfall* model applied in an iterative fashion. The model applies linear sequences in a staggered fashion as calendar time progresses.

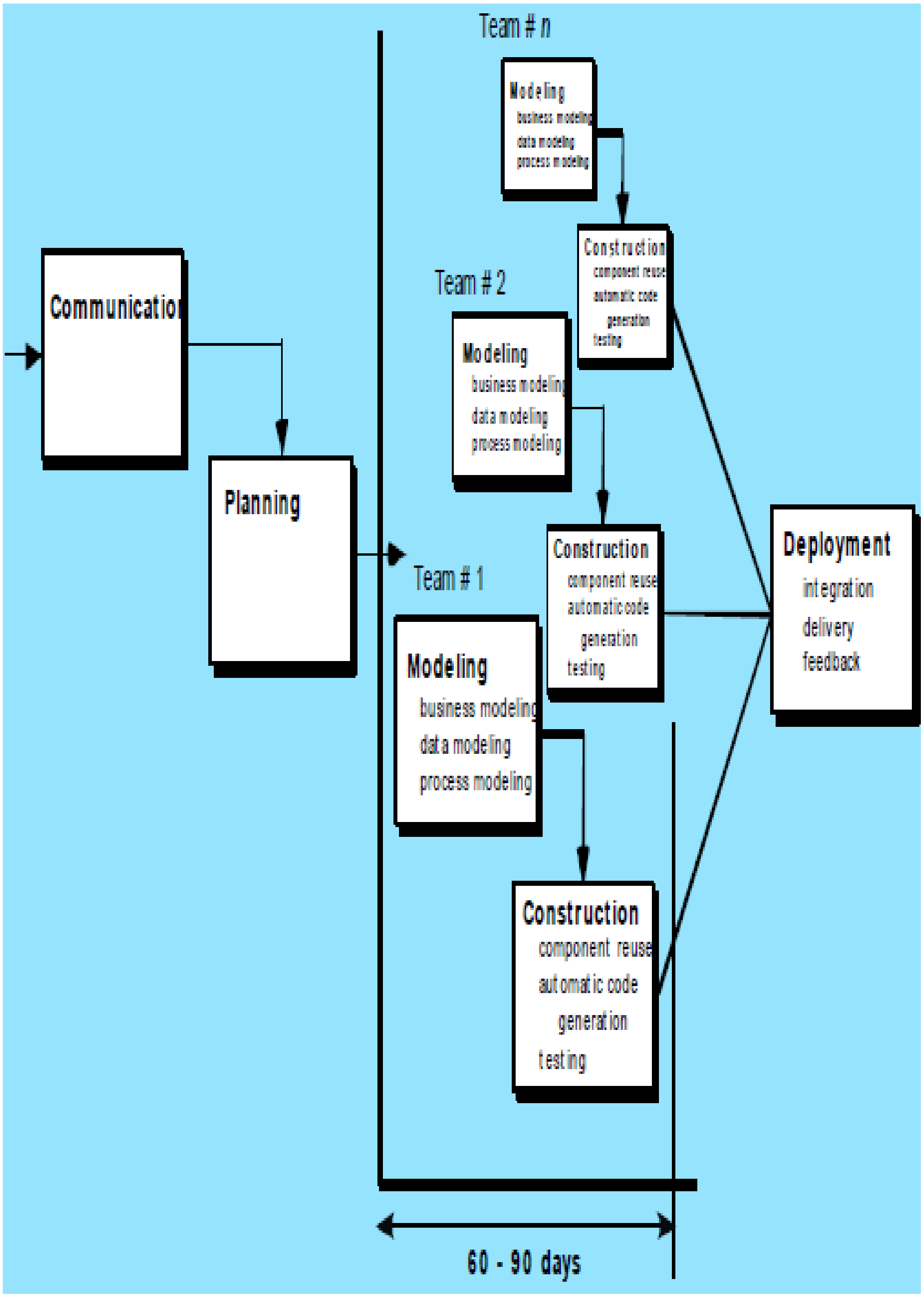
Each linear sequence produces deliverable “increments” of the software. (Ex: a Word Processor delivers basic file mgmt., editing, in the first increment; more sophisticated editing, document production capabilities in the 2ndincrement; spelling and grammar checking in the 3rdincrement.
When an increment model is used, the 1stincrement is often a *core product*. The core product is used by the customer.
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.
The process is repeated following the delivery of each increment, until the complete product is produced.
If the customer demands delivery by a date that is impossible to meet, suggest delivering one or more increments by that date and the rest of the Software later.

b. The RAD Model

Rapid Application Development(*RAD*) is an incremental software process model that emphasizes a short development cycle.
RAD is a “high-speed” adaptation of the waterfall model, in which rapid development is achieved by using a 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 short period of time.

What are the drawbacks of the RAD model?
1. For large, but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.

2. If developers and customers are not committed to the rapid-fire activities necessary to complete the system in a much abbreviated time frame, RAD project will fail.
3. If a system cannot properly be modularized, building the components necessary for RAD will be problematic.

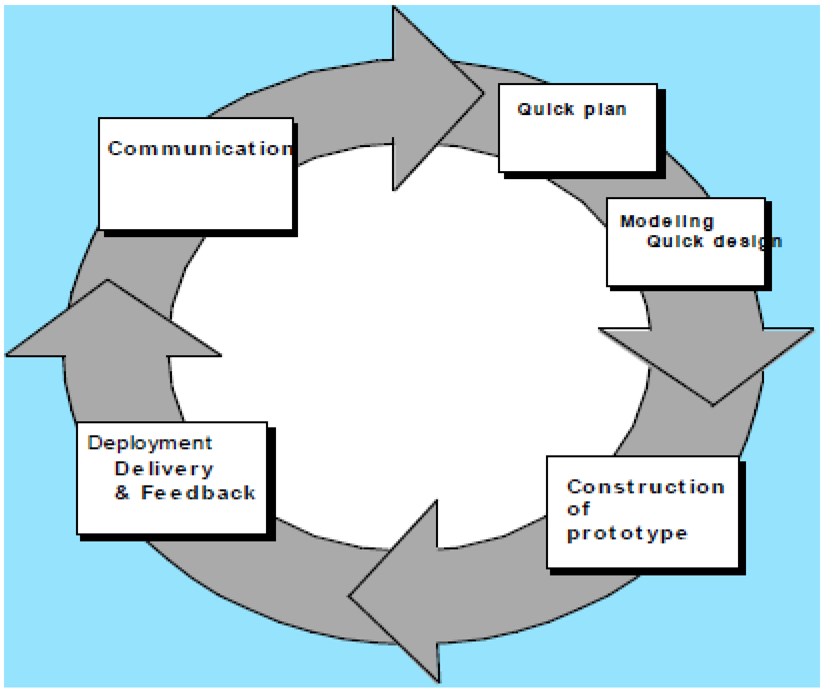

Team # n
Modeling
business modeling
data modeling
process modeling
Construction
component reuse
automatic code generation
testing
Team # 2
Modeling
business modeling
data modeling
process modeling
Construction
component reuse
automatic code generation
testing
Communication
Planning
Team # 1
Modeling
business modeling
data modeling
process modeling
Construction
component reuse
automatic code generation
testing
Deployment
integration
delivery
feedback
60 - 90 days

EVOLUTIONARY PROCESS MODELS

Software evolves over a period of time; business and product requirements often change as development proceeds, making a straight-line path to an end product unrealistic. Software Engineering needs a process model that has been explicitly designed to accommodate a product that evolves over time. Evolutionary process models are iterative. They produce increasingly more complete versions of the Software with each iteration.

a. **Prototyping**

Customers often define a set of general objectives for Software, but doesn't identify detailed input, processing, or input requirements. Prototyping paradigm assists the Software engineering and the customer to better understand what is to be built when requirements are fuzzy.



EVOLUTIONARY PROCESS MODELS

The prototyping paradigm begins with *communication* where requirements and goals of Software are defined. Prototyping iteration is *planned* quickly and modeling in the form of quick design occurs. The *quick design* focuses on a representation of those aspects of the Software that will be visible to the customer "Human interface". The quick design leads to the *Construction of the Prototype.*

The prototype is *deployed* and then *evaluated* by the customer. *Feedback* is used to refine requirements for the Software. Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while enabling the developer to better understand what needs to be done. The prototype can serve as the "first system". Both customers and developers like the prototyping paradigm as users get a feel for the actual system, and developers get to build Software immediately. Yet, prototyping can be problematic:

1.The customer sees what appears to be a working version of the Software, unaware that the prototype is held together “with chewing gum. “Quality, long-term maintainability.” When informed that the product is a prototype, the customer cries foul and demands that few fixes be applied to make it a working product. Too often, Software development management relents.
2.The developer makes implementation compromises in order to get a prototype working quickly. An inappropriate O/S or programming language used simply b/c it’s available and known. After a time, the developer may become comfortable with these choices and forget all the reasons why they were inappropriate.

The key is to define the rules of the game at the beginning. The customer and the developer must both agree that the prototype is built to serve as a mechanism for defining requirements.

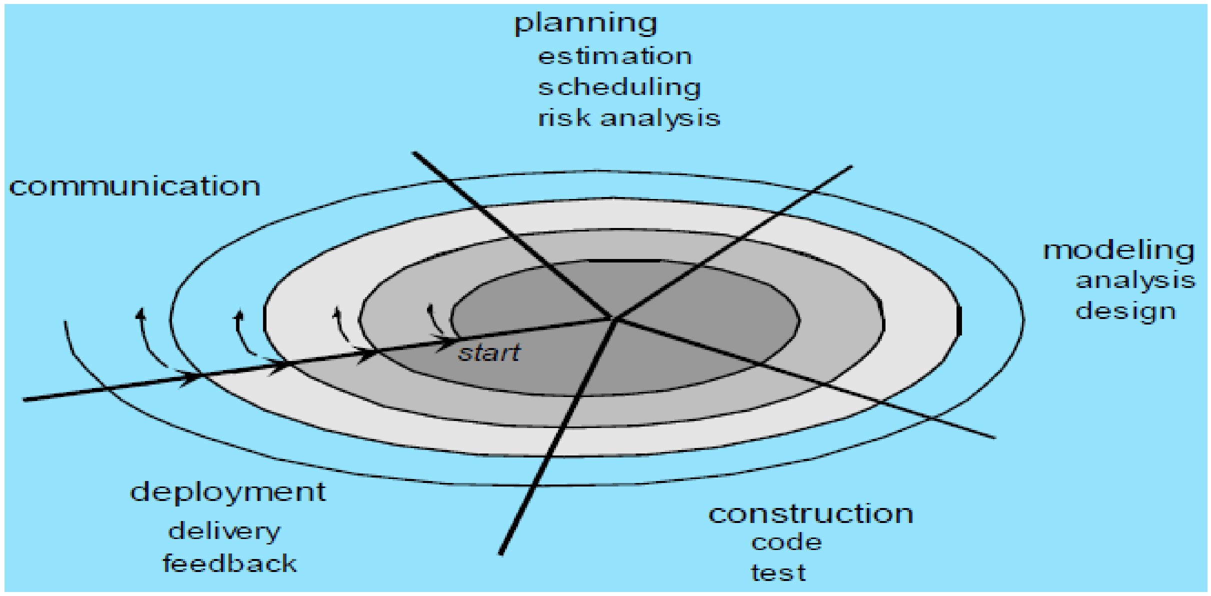
b. **The Spiral Model**

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 has two distinguishing features:

a. A cyclic approach for incrementally growing a system’s degree of definition and implementation while decreasing its degree of risk.

b. A set of *anchor point milestones* for ensuring stakeholder commitment to feasible and mutually satisfactory solutions.

Using the spiral model, Software is developed in a series of evolutionary releases. During early stages, the release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced. A spiral model is divided into a set of framework activities divided by the Software engineering team. As this evolutionary process begins, the Software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center. Risk is considered as each revolution is 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.
The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the Software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. Unlike other process models that end when Software is delivered, the spiral model can be adapted to apply throughout the life of the Software.



The concurrent development model

The *concurrent development model*, sometimes called *concurrent engineering*, can be represented schematically as a series of framework activities, Software engineering actions of tasks, and their associated states. The concurrent model is often more appropriate for system engineering projects where different engineering teams are involved.

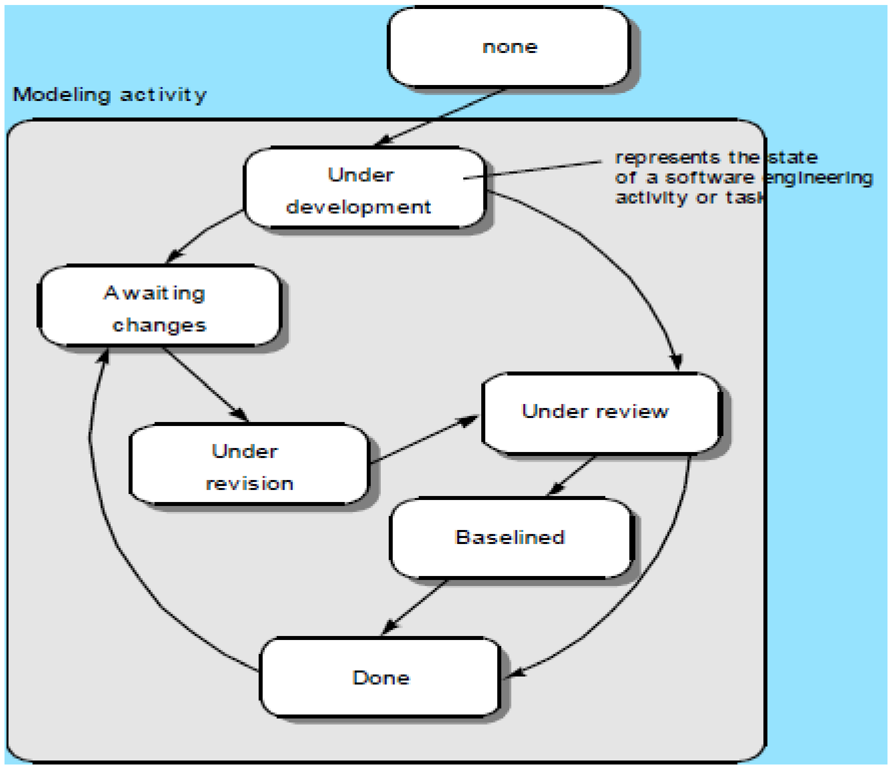

none
Modeling activity
Under development
represents the state of a software engineering activity or task
Awaiting changes
Under review
Under revision
Baselined
Done

Figure above provides a schematic 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 any given time. All activities exist concurrently but reside in different states. For example, early in the project the *communication* activity has completed its first iteration and exists in the **awaiting changes** state. The *modeling* activity which existed in the **none** state while initial communication was completed now makes a transition into **underdevelopment** state. If, however, the customer indicates the changes in requirements must be made, 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.

SPECIALIZED PROCESS MODELS

a. Component Based Development

Commercial off-the-shelf (COTS) Software components, developed by vendors who offer them as products, can be used when Software is to be built. These components provide targeted functionality with well-defined interfaces that enable the component to be integrated into the Software. The *component-based development* model incorporates many of the characteristics of the spiral model.
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.
- Software architecture is designed to accommodate the components.
- Components are integrated into the architecture.
- Comprehensive testing is conducted to ensure proper functionality.

The *component-based development* model leads to Software reuse, and reusability provides Software engineers with a number of measurable benefits.

b. The Formal Methods Model

The Formal Methods Model encompasses a set of activities that leads to formal mathematical specifications of Software. Formal methods enable a Software engineer to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation of this approach, called *clean-room* Software *engineering* is currently applied by some software development organizations.

Although not a mainstream approach, the formal methods model offers the promise of defect-free Software. Yet, concern about its applicability in a business environment has been voiced:

The development of formal models is currently quite time-consuming and expensive.

B/C few software developers have the necessary background to apply formal methods, extensive training is required.

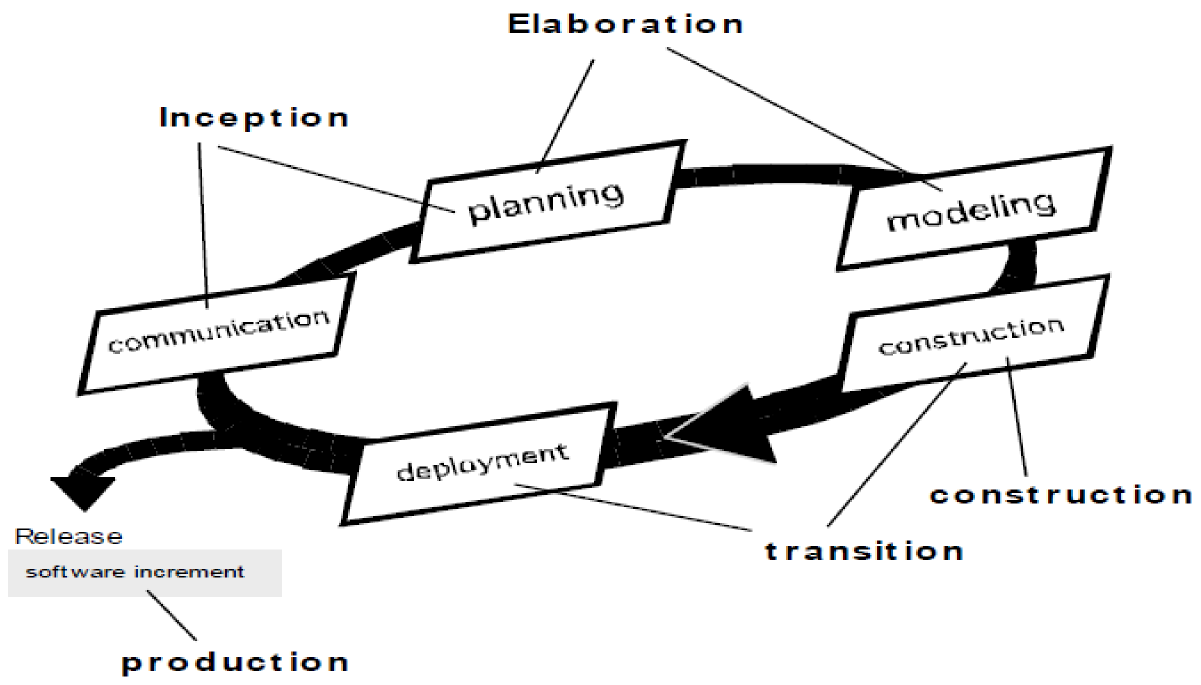
It is difficult to use the methods as a communication mechanism for technically unsophisticated customers.

THE UNIFIED PROCESS

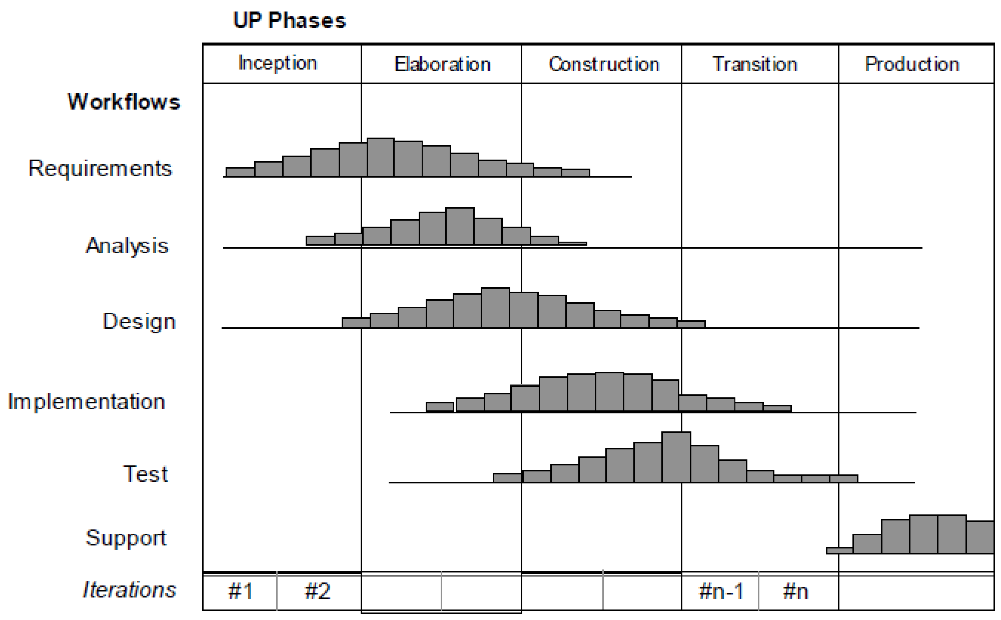
A “use-case driven, architecture-centric, iterative and incremental” software process closely aligned with the Unified Modeling Language (UML).The UP is an attempt to draw on the best features and characteristics of conventional software process models, but characterize them in a way that implements many of the best principles of agile software development. The UP 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.”UML provides the necessary technology to support Object Oriented Software Engineering practice, but it doesn’t provide the process framework to guide project teams in their application of the technology. The UML developers developed the *Unified Process*, a framework Object Oriented Software Engineering using UML.

Phases of the Unified Process

The figure below depicts the phases of the UP and relates them to the generic activities.


Elaboration
Inception
planning
modeling
communication
construction
deployment
construction
transition
Release
software increment
production

The *Inception* phase of the UP encompasses both customer communication and planning activities. By collaborating with the customer and end-users, 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. software increment Release Inception Elaboration construct ion transition production
A use-case describes a sequence of actions that are performed by an *actor* (person, machine, another system) as the actor interacts with the Software. The *elaboration* phase encompasses the customer 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 analysis 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 activity. Software is given to end-users for beta testing, and user feedback reports both defects and necessary changes. At the conclusion of the transition phase, the software increment becomes a usable software release "user manuals, trouble-shooting guides, and installation procedures.) The *production* phase of the UP coincides with the development activity of the generic process. The on-going use of the software is monitored, support for the operating environment is provided and defect reports and requests for changes are submitted and evaluated.
A Software Engineering workflow is distributed across all UP phases.



Agile Processes

Idea of modeling in a light weight fashion to deliver documentation good enough for right now.

-speed up or by pass one or more life cycle phases

-less formal and scope is reduced

-used for time critical applications

-used in organizations that employ disciplined methods

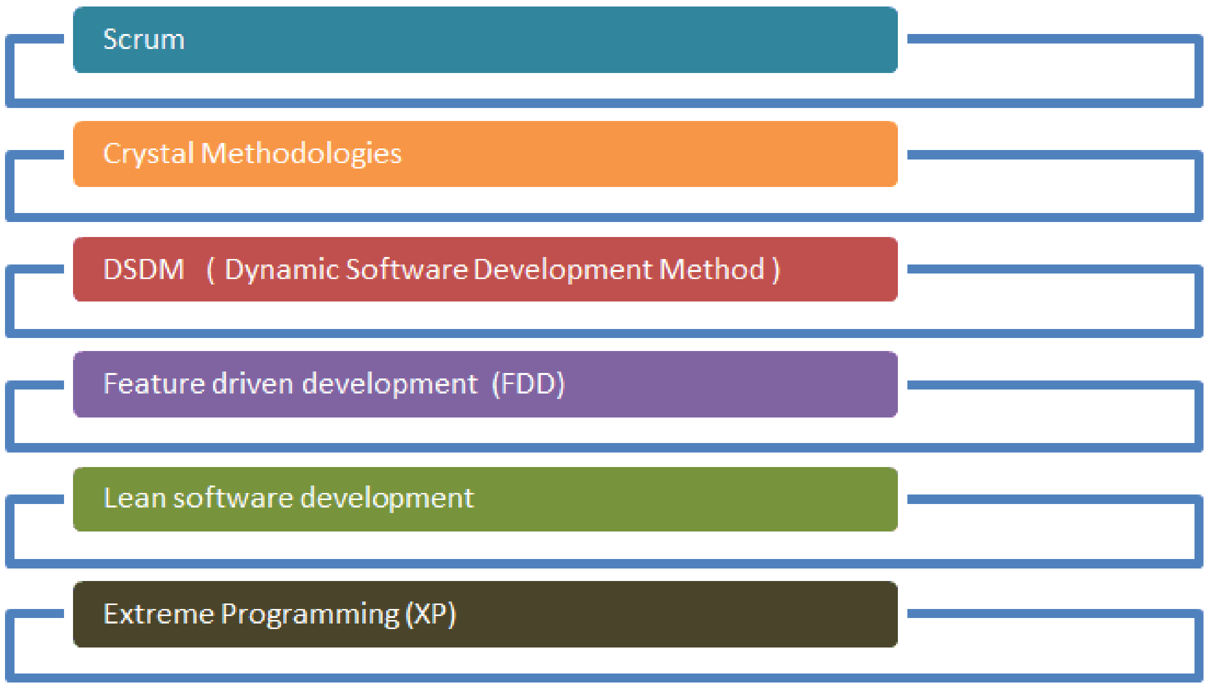
What is Agile Methodology?

AGILE methodology is a practice that promotes **continuous iteration** of development and testing throughout the software development lifecycle of the project. Both development and testing activities are concurrent unlike the Waterfall model

The agile software development emphasizes on four core values.

1. Individual and team interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

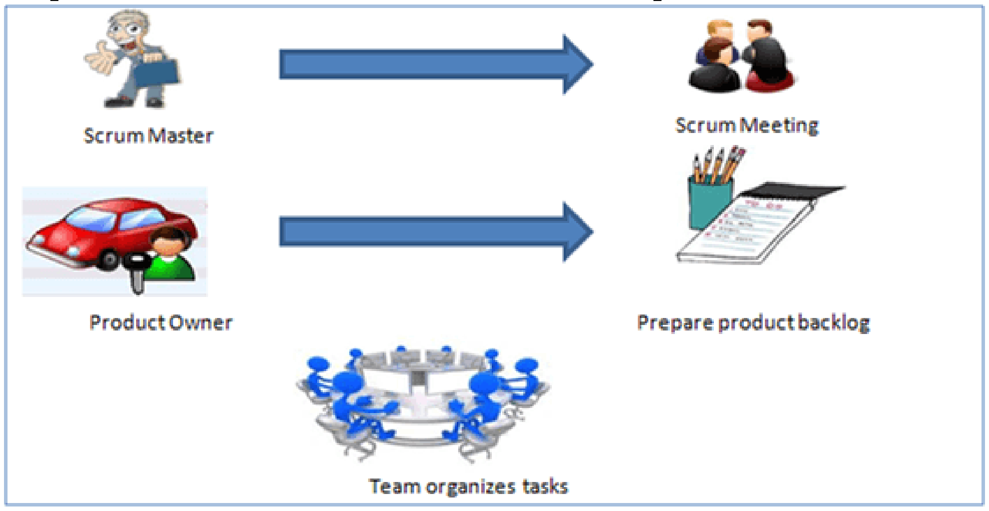
Agile Methodology


Scrum
Crystal Methodologies
DSDM (Dynamic Software Development Method)
Feature driven development (FDD)
Lean software development
Extreme Programming (XP)

There are various **methods** present in agile testing, and those are listed below:

Scrum

SCRUM is an agile development method which concentrates specifically on how to manage tasks within a team-based development environment. Basically, Scrum is derived from activity that occurs during a rugby match. Scrum believes in empowering the development team and advocates working in small teams (say- 7 to 9 members). It consists of three roles, and their responsibilities are explained as follows:



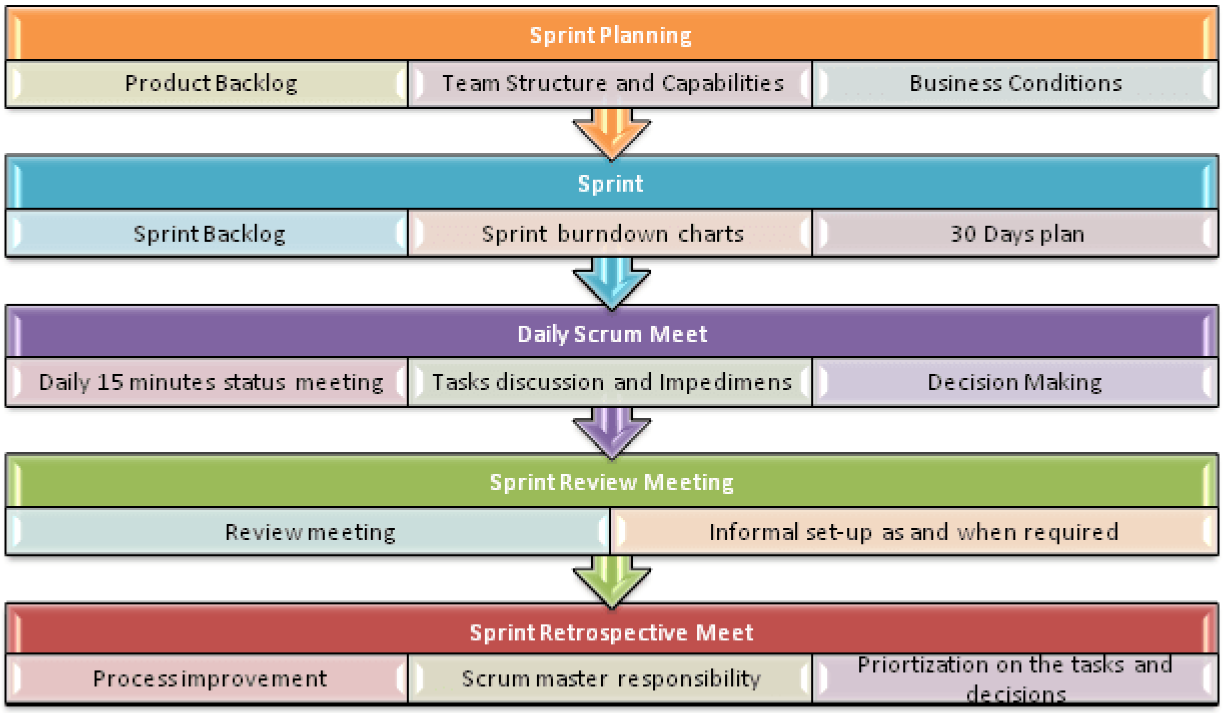
- Scrum Master
    - Master is responsible for setting up the team, sprint meeting and removes obstacles to progress
- Product owner
    - The Product Owner creates product backlog, prioritizes the backlog and is responsible for the delivery of the functionality at each iteration
- Scrum Team
    - Team manages its own work and organizes the work to complete the sprint or cycle

Product Backlog

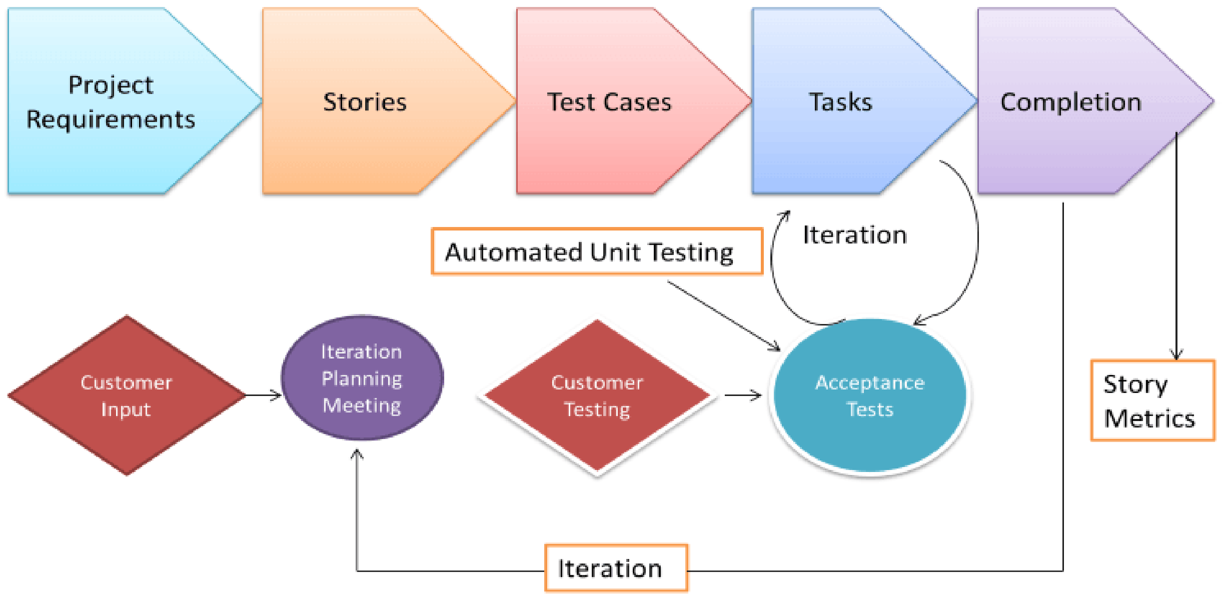
This is a repository where requirements are tracked with details on the no of requirements to be completed for each release. It should be maintained and prioritized by Product Owner, and it should be distributed to the scrum team. Team can also request for a new requirement addition or modification or deletion

Scrum Practices

Practices are described in detailed:


Sprint Planning
Product Backlog
Team Structure and Capabilities
Business Conditions
Sprint
Sprint Backlog
Sprint burndown charts
30 Days plan
Daily Scrum Meet
Daily 15 minutes status meeting
Tasks discussion and Impedimens
Decision Making
Sprint Review Meeting
Review meeting
Informal set-up as and when required
Sprint Retrospective Meet
Process improvement
Scrum master responsibility
Priortization on the tasks and decisions

-
-
-
-
-
-



Extreme Programming (XP)

•
•
•
•

•
•
•
•
•

•
•
•
•
•
•

•
    o
•
    o

1.
2.
    1.
    2.
    3.
    4.
3.

1.
2.
3.

1.
2.
3.
4.
5.
6.
7.

1.
2.
3.
4.
5.
6.
7.
8.

1.
2.
3.
4.
5.
6.
7.

Software Project Estimation

<u>Decomposition Techniques</u>

-Problem based (LOC, Size oriented, Direct method)

-Process based (FP, function oriented, Indirect method)

Productivity Measures

Size related measures based on some output from the software process. This may be lines of delivered source code(LOC), object code instructions, etc.

Function-related measures based on an estimate of the functionality of the delivered software. Function-points(FP) are the best known of this type of measure

Lines of Code (LOC)

What's a line of code?

- The measure was first proposed when programs were typed on cards with one line per card
- How does this correspond to statements as in Java which can span several lines or where there can be several statements on one line

What programs should be counted as part of the system?

Assumes linear relationship between system size and volume of documentation

Function Points

Based on a combination of program characteristics

- external inputs and outputs
- user interactions
- external interfaces
- files used by the system

A weight is associated with each of these

The function point count is computed by multiplying each raw count by the weight and summing all values

Empirical Model

Decomposition Techniques

<u>Problem based decomposition</u>

LOC – Direct measure, White box measure, Internal Specification based

FP - Indirect measure, Black box measure, External Specification based

Empirical Estimation Model

Which uses empirically derived formulas for prediction based on LOC and FP

Cost Constructive Model by Barry Boehm based on LOC

Hierarchy of the model

1.Basic COCOMO

2.Intermediate COCOMO

3.Advanced COCOMO

Basic COCOMO model gives basic idea

Computes software development efforts and duration as a function of program size/profit size which is expressed as LOC

Gives Rough Idea about Estimation

All computation based on size of project

$E = a_b \text{ (KLOC)} b^b$ in Person-Month PM

$D = C_b \text{ (E) } d^b$ months

N (No.of people in team) = E / D persons

Intermediate COCOMO size + cost drivers considered

Computes software development efforts in terms of function of program size with a set of cost drivers which includes subjective assessment of the driver(personal attributes,project,product,hardware)

Cost driver value plays a major role in computation of effort

It ranges from 0.7 to 1.3 & 0.9 (<2)

Provides exact evaluation required for the project

$E = a_i \text{ (KLOC) } b_i \times \text{EAF}$

Advanced COCOMO

Incorporates characteristics of intermediate version with the subjective evaluation of cost drivers impact on the software process activities (Analysis,Design,Coding,Testing)

Risk analysis and management

What is it? Risk analysis and management are a series of steps that help a software team to understand and manage uncertainty.

Establish a contingency plan should the problem actually occur.

Who does it? Everyone involved in the software process—managers, software engineers, and customers— participate in risk analysis and management.

Why is it important? Be Prepared. Understanding the risks and taking proactive measures to avoid or manage them—is a key element of good software project management.

What are the steps? Recognizing what can go wrong is the first step, called “risk identification.”

Next, each risk is analyzed to determine the likelihood that it will occur and the damage that it will do if it does occur. Once this information is established, risks are ranked, by probability and impact. Finally, a plan is developed to manage those risks with high probability and high impact.

What is the work product? A risk mitigation, monitoring, and management (RMMM) plan or a set of risk information sheets is produced.

The RMMM should be revisited as the project proceeds to ensure that risks are kept up to date. Contingency plans for risk management should be realistic.

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 that have been presented : 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 project at hand.

To identify product-specific risks, the project plan and the software statement of scope are examined and an answer to the following question is developed: "What special characteristics of this product may threaten our project plan?"
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.
• *Customer characteristics*—risks associated with the sophistication of the customer and the developer's ability to communicate with the customer 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.

The risk item checklist can be organized in different ways. Questions relevant to each of the topics can be answered for each software project. The answers to these questions allow the planner to estimate the impact of risk. A different risk item checklist format simply lists characteristics that are relevant to each generic subcategory. Finally, a set of "risk components and drivers" are listed along with their probability of occurrence. Drivers for performance, support, cost, and schedule are discussed in answer to later questions.

A number of comprehensive checklists for software project risk have been proposed in the literature. These provide useful insight into generic risks for software projects and should be used whenever risk analysis and management is instituted. However, a relatively short list of questions can be used
to provide a preliminary indication of whether a project is "at risk."

Developing a Risk Table
A risk table provides a project manager with a simple technique for risk projection.

Risks
PS-Product Size
BU-Business Impact
CU – Customer Characteristics
DE - Development Enivronment
ST-Staff size and Experience

Risks	**Category**	**Probability**	**Impact**	**RMMM**
Size estimate may be significantly low	PS	60%	Critical	
Larger number of users than planned	PS	30%	Marginal	
Less reuse than planned	PS	70%	Critical	
End-users resist system	BU	40%	Marginal	
Delivery deadline will be tightened	BU	50%	Critical	
Funding will be lost	CU	40%	Catastrophic	
Customer will change requirements	PS	80%	Critical	
Technology will not meet expectations	TE	30%	Catastrophic	
Lack of training on tools	DE	80%	Marginal	
Staff inexperienced	ST	30%	Critical	
Staff turnover will be high	ST	60%	Critical	

Figure 1. Sample risk table prior to sorting

RMMM PLAN

The RMMM plan documents all work performed as part of risk analysis and is used by the project manager as part of the overall project plan.
Some software teams do not develop a formal RMMM document. Rather, each risk is documented individually using a *risk information sheet* (RIS). In most cases, the RIS is maintained using a database system, so that creation and information entry, priority ordering, searches, and other analysis may be accomplished easily.

PROJECT SCHEDULING

The objective of software project scheduling is to create a set of engineering tasks that will enable to complete the job in time.

- You've selected an appropriate process model.
- You've identified the software engineering tasks that have to be performed.
- You estimated the amount of work and the number of people, you know the deadline, you've even considered the risks.
- Now it's time to connect the dots. That is, you have to create a network of software engineering tasks that will enable you to get the job done on time.
- Once the network is created, you have to assign responsibility for each task, make sure it gets done, and adapt the network as risks become reality.

Basic Concepts.

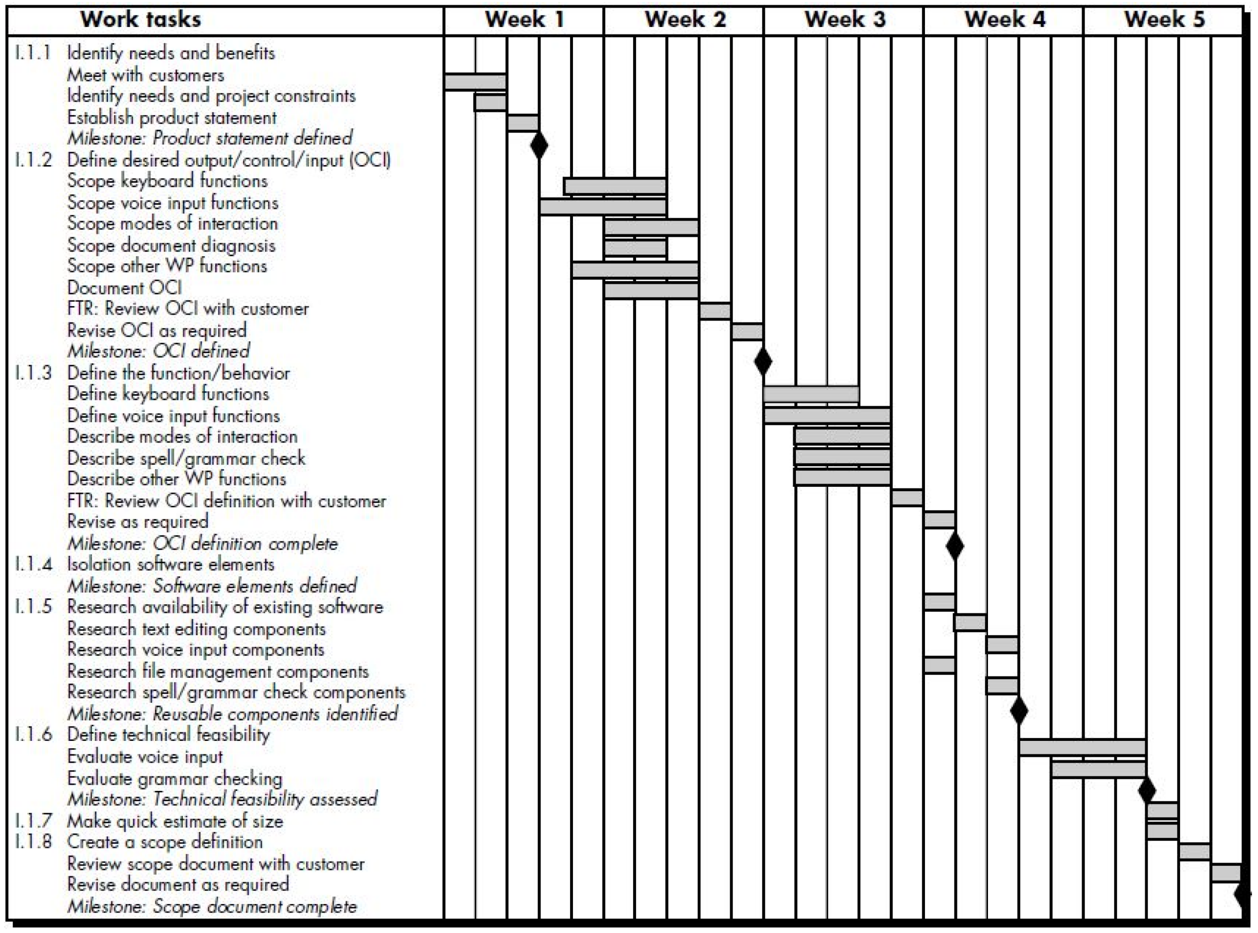
- An unrealistic deadline established by someone outside the software development group and forced on managers and practitioner's within the group.
- Changing customer requirements that are not reflected in schedule changes.
- An honest underestimate of the amount of effort and/or the number of resources that will be required to do the job.
- Predictable and/or unpredictable risks that were not considered when the project commenced.
- Technical difficulties that could not have been foreseen in advance.
- Human difficulties that could not have been foreseen in advance.
- Miscommunication among project staff that results in delays.
- A failure by project management to recognize that the project is falling behind schedule and a lack of action to correct the problem.

Basic Principles

1. **Compartmentalization:** The project must be compartmentalized into a number of manageable activities and tasks.
2. **Interdependency:** The interdependency of each compartmentalized activity or task must be determined.
3. **Time allocation:** Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort).
4. **Effort validation:** The project manager must ensure that no more than the allocated number of people have been scheduled at any given time.
5. **Defined responsibilities:** Every task that is scheduled should be assigned to a specific team member.
6. **Defined outcomes:** Every task that is scheduled should have a defined outcome.
7. **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 and has been approved.

Time-line chart

- When creating a software project schedule, the planner begins with a set of tasks.
- If automated tools are used, the work breakdown is input as a task network or task outline.
- Effort, duration, and start date are then input for each task. In addition, tasks may be assigned to specific individuals.
- As a consequence of this input, a ***timeline chart*** is generated also called ***ganttchart***.
- A timeline chart can be developed for the entire project. Alternatively, separate charts can be developed for each project function or for each individual working on the project.



UNIT - II

System Engineering

Software engineering occurs as a consequence of a process called *system engineering.* Instead of concentrating solely on software, system engineering focuses on a variety of elements, analyzing, designing, and organizing those elements into a system that can be a product, a service, or a technology for the transformation of information or control.

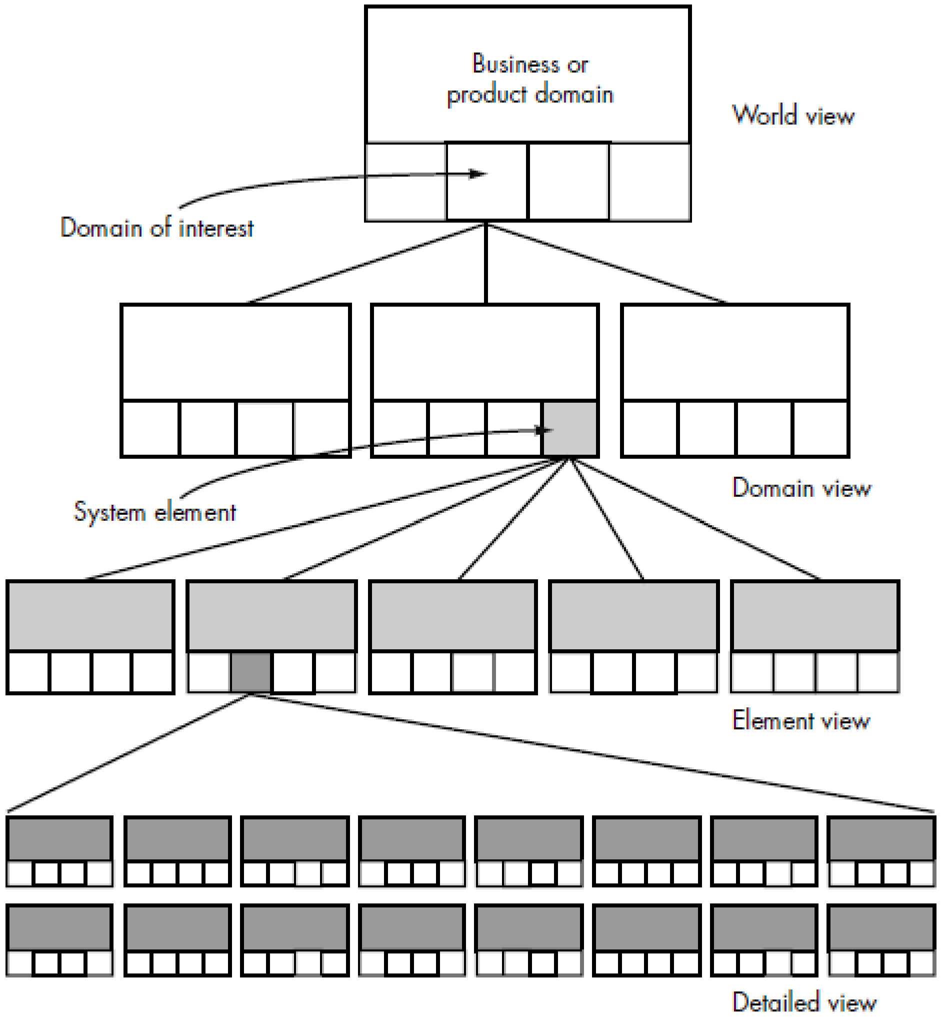
The system engineering process is called *business process engineering* when the context of the engineering work focuses on a business enterprise. When a product (in this context, a product includes everything from a wireless telephone to an air traffic control system) is to be built, the process is called *product engineering.*

COMPUTER-BASED SYSTEMS

The elements combine in a variety of ways to transform information.
Eg. Marketing – transforms raw sales into purchase of product

Computer based system make use of variety of system elements:
Software. Computer programs, data structures, and related documentation that serve to effect the logical method, procedure, or control that is required.
Hardware. Electronic devices that provide computing capability, the interconnectivity devices (e.g., network switches, telecommunications devices) that enable the flow of data, and electromechanical devices (e.g., sensors, motors, pumps) that provide external world function.
People. Users and operators of hardware and software.
Database. A large, organized collection of information that is accessed via software.
Documentation. Descriptive information (e.g., hardcopy manuals, on-line help files, Web sites) that portrays the use and/or operation of the system.
Procedures. The steps that define the specific use of each system element or the procedural context in which the system resides.

Figure1.System Engineering Hierarchy



Stated in a slightly more formal manner, the world view (WV) is composed of a set of domains (*Di*), which can each be a system or system of systems in its own right.

WV = {*D1, D2, D3, . . . , Dn*}

Each domain is composed of specific elements (*Ej*) each of which serves some role in accomplishing the objective and goals of the domain or component:

Di = {*E1, E2, E3, . . . , Em*}

Finally, each element is implemented by specifying the technical components (*Ck*) that achieve the necessary function for an element:

Ej = {*C1, C2, C3, . . . , Ck*}

In the software context, a component could be a computer program, a reusable program component, a module, a class or object, or even a programming language statement.

System Modeling

System engineering is a modeling process. Whether the focus is on the world view or the detailed view, the engineer creates models that
• Define the processes that serve the needs of the view under consideration.
• Represent the behavior of the processes and the assumptions on which the behavior is based.
• Explicitly define both exogenous and endogenous input3 to the model.
• Represent all linkages (including output) that will enable the engineer to better understand the view.

To construct a system model, the engineer should consider a number of restraining factors:

Assumptions that reduce the number of possible permutations and variations, thus enabling a model to reflect the problem in a reasonable manner.

2. *Simplifications* that enable the model to be created in a timely manner.

3. *Limitations* that help to bound the system.

4. *Constraints* that will guide the manner in which the model is created and the approach taken when the model is implemented.

5. *Preferences* that indicate the preferred architecture for all data, functions, and technology.

The system engineer simply modifies the relative influence of different system elements (people, hardware, software) to derive models of each type.

System Simulation

To avoid the reactive system, system simulation is used.

Software tools for system modeling and simulation are being used to help to eliminate surprises when reactive, computer-based systems are built. These tools are applied during the system engineering process, while the role of hardware and software, databases and people is being specified. Modeling and simulation tools enable a system engineer to "test drive" a specification of the system.

BUSINESS PROCESS ENGINEERING:

The goal of business process engineering (BPE) is to define architectures that will enable a business to use information effectively.

Business process engineering is one approach for creating an overall plan for implementing the computing architecture.
Three different architectures must be analyzed and designed within the context of business objectives and goals:

• data architecture
• applications architecture
• technology infrastructure

The *data architecture* provides a framework for the information needs of a business or business function. The individual building blocks of the architecture are the data objects that are used by the business. A data object contains a set of attributes that define some aspect, quality, characteristic, or descriptor of the data that are being described. For example, an information engineer might define the data object **customer.**

To more fully describe **customer,** the following attributes are defined:
Object: Customer
Attributes:
name
company name
job classification and purchase authority
business address and contact information
product interest(s)
past purchase(s)
date of last contact
status of contact

Once a set of data objects is defined, their relationships are identified. A *relationship* indicates how objects are connected to one another.

As an example consider the objects: **customer,** and **product A.** The two objects can be connected by the relationship *purchases;* that is, a customer purchases product A or product A is purchased by a customer.

The *application architecture* encompasses those elements of a system that transform objects within the data architecture for some business purpose. In the context of this book, we consider the application architecture to be the system of programs (software) that performs this transformation. However, in a broader context, the application architecture might incorporate the role of people (who are information transformers and users) and business procedures that have not been automated.

The *technology infrastructure* provides the foundation for the data and application architectures. The infrastructure encompasses the hardware and software that are used to support the application and data. This includes computers, operating systems, networks, telecommunication links, storage technologies, and the architecture (e.g., client/server) that has been designed to implement these technologies.

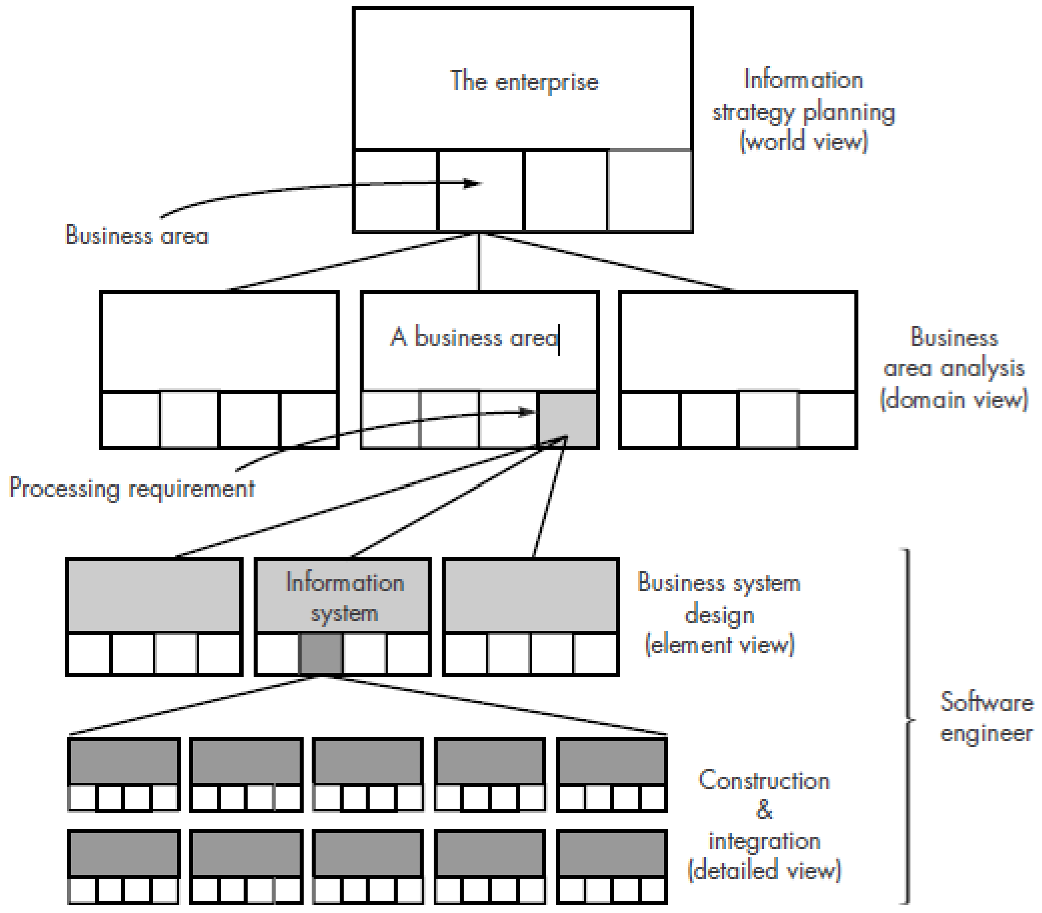

The enterprise
Information strategy planning (world view)
Business area
A business area
Business area analysis (domain view)
Processing requirement
Information system
Business system design (element view)
Software engineer
Construction & integration (detailed view)

Figure. Business Process Engineering Hierarchy

The first step *information strategy planning* (ISP). ISP views the entire business as an entity and isolates the domains of the business that are important to the overall enterprise. ISP defines the data objects that are visible at the enterprise level, their relationships, and how they flow between the business domains.

BAA views the business area as an entity and isolates the business functions and procedures that enable the business area to meet its objectives and goals. BAA, like ISP, defines data objects, their relationships, and how data flow.

A *business system design* (BSD) step, the basic requirements of a specific information system are modeled and these requirements are translated into data architecture, applications architecture, and technology infrastructure.

The final BPE step—*construction and integration* focuses on implementation detail.

PRODUCT ENGINEERING

The goal of product engineering is to translate the customer's desire for a set of defined capabilities into a working product. To achieve this goal, product engineering—like business process engineering—must derive architecture and infrastructure. The architecture encompasses four distinct system components: software, hardware, data (and databases), and people. A support infrastructure is established and includes the technology required to tie the components together and the information (e.g., documents, CD-ROM, video) that is used to support the components.

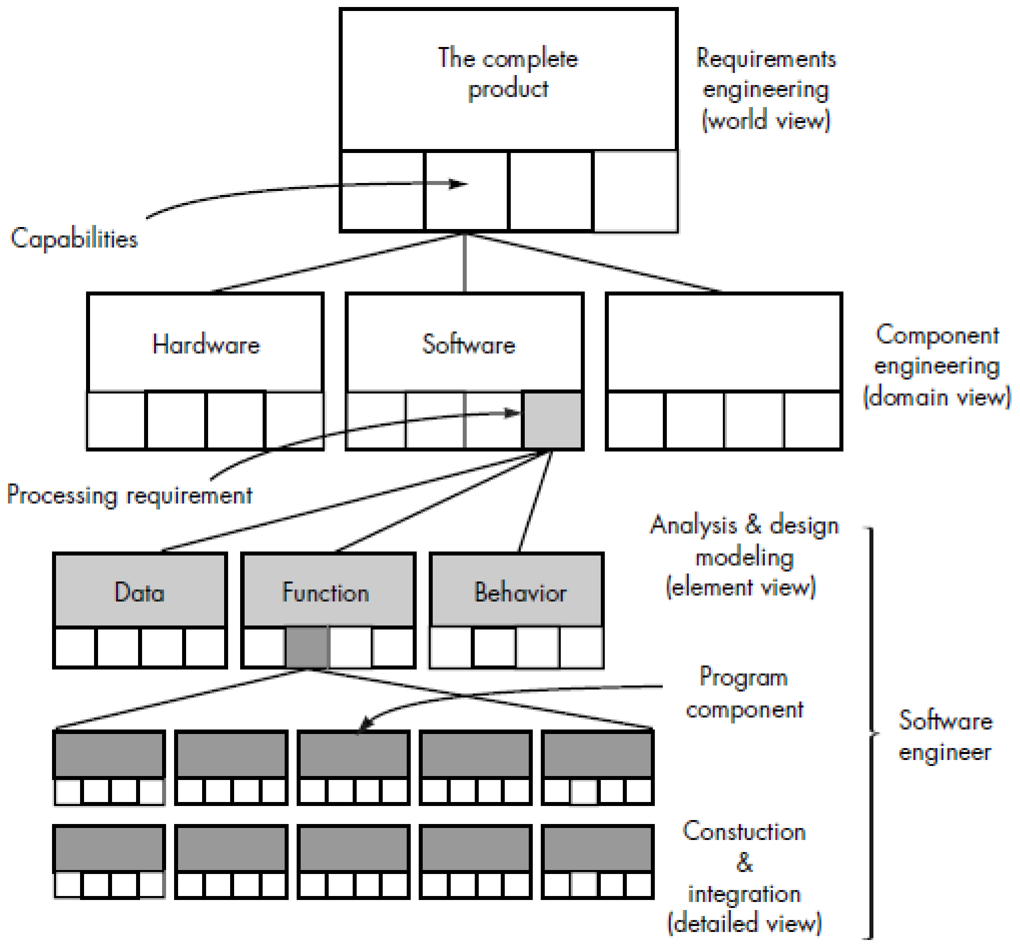

The complete product
Requirements engineering (world view)
Capabilities
Hardware
Software
Component engineering (domain view)
Processing requirement
Analysis & design modeling (element view)
Data
Function
Behavior
Program component
Software engineer
Constuction & integration (detailed view)

Figure. Product Engineering Hierarchy

The world view is achieved through *requirements engineering.*

The overall requirements of the product are elicited from the customer. These requirements encompass information and control needs, product function and behavior, overall product performance, design and interfacing constraints, and other special needs.

Once these requirements are known, the job of requirements engineering is to allocate function and behavior to each of the four components noted earlier. Once allocation has occurred, *system component engineering* commences.

System component engineering is actually a set of concurrent activities that address each of the system components separately: software engineering, hardware engineering, human engineering, and database engineering.

Each of these engineering disciplines takes a domain-specific view, but it is important to note that the engineering disciplines must establish and maintain active communication with one another. Part of the role of requirements engineering is to establish the interfacing mechanisms that will enable this to happen.

The element view for product engineering is the engineering discipline itself applied to the allocated component.

The analysis step models allocated requirements into representations of data, function, and behavior. Design maps the analysis model into data, architectural, interface, and software component-level designs.

REQUIREMENTS ENGINEERING

 Requirement is a condition possessed by the software component in order to solve a real world problems.
 Requirement describe how a system should act, appear or perform.
 IEEE defines a requirement as: “A condition that must be possessed by a system to satisfy a contract specification, standard or other formally imposed document.

Principles of Requirement Engineering

i. Understand the problem before you start to create the analysis model
 There is a tendency to rush to a solution, even before the problem is understood.
 This often leads to elegant software that solves the wrong problem.
ii. Develop prototypes that enable a user to understand how human-machine interaction will occur
Since the perception of the quality of software is often is based on perception of time “friendliness” of the interface, prototyping (and the interaction that results) is highly recommended.
iii. Record the origin and the reason for every document
This is the step in establishing traceability back to the customer.
iv. Use multiple views of requirement
 Building data, functional and behavioral models provides software engineer three different views.
 This reduces the chances of missing errors.
v. Prioritize the requirements
Requirements should be followed for the tight implementation and delivery of the product.
vi. Work to eliminate ambiguity
The use of more technical reviews should be used for no ambiguity.

Requirement Engineering Task

The requirement engineering process tasks are achieved through seven distinct functions:
i. Inception

 Inception is also known as the beginning.
 Inception needs various questions to be answered.
 How does a software project get started?

ii. Elicitation

 This is the process by which users of the system are interviewed in order to reveal
and understand their requirements.
 This sub-phase involves a high level of client input.

iii. Elaboration

The most relevant, mission critical requirements are emphasized and developed first.
 This ensures the final product satisfies all of the important requirements, and does so in the most time and cost efficient manner possible.
 Other "should-have" and "nice-to-have" requirements can be relegated to future phases, should they be necessary.

iv. Negotiation

 A milestone within that phase may consist of implementing Use Case x, y and z .
 By scheduling the project in this iterative way, our client has a good understanding of when certain things will be delivered and when their input will be required.
 Additionally, features are generally developed in a "vertical" fashion; once Use Case x has been developed, its unabridged functionality can be demonstrated to the client.
 In this way, our client has a clear understanding of where development time is being spent, and potential issues can be caught early and dealt with efficiently.

v. Specification

 This is the process by which requirements and their analyses are committed to some formal media.
 For this project, we would generate four documents during this sub-phase:
 Use Case document: a use case is a description of the system's response as a result of a specific request from a user. Each use case will cover certain requirements discovered during elicitation and analysis. For example, a use case may cover "Adding a New Employee", or "Generating Site Report".
 Interface prototypes: an interface prototype is a rough representation of certain
critical functionality, recorded as anything from a hand-drawn image to an elaborate HTML mock-up. The prototypes are completely horizontal in nature; they roughly illustrate how the interface for a certain feature will look and what information is accessible but they will have no back end (vertical) functionality.
 Architecture Diagram: an architecture diagram will be generated for developer use as it is a high level view of how the software will be structured. Only if we feel it will help in our understanding of the system will this document be created.

 Database Diagram: as with the architecture diagram, a database diagram is generally created for developer use. Through these final two media, we are beginning
the shift from words (requirements) to code (software).

vi. Validation

 This is the final phase of the requirements engineering process.
 It involves scrutinizing the documents generated during the specification sub-phase and ensuring their relevance and validity.
 This must be done by all interested parties so they are in agreement with the proposed system's behavior, especially in the case of the Use Case document and prototypes.

vii. Management

 As requirements are elicited, analyzed, specified and validated, we will be estimating most features.
 This is an on-going process that will culminate in the delivery of the Project Estimation and Timeline document.
 This will outline the duration of work required to develop and deliver Phase 2.
 As indicated in the Requirements Analysis sub-phase, Phase 2 will likely consist of the majority of the "must-have" features; however it is ultimately up to the client to decide what is most important (even within the "must-have" category), and this will be developed first.

Initiating Requirements Engineering Process

• Identify stakeholders
• Recognize the existence of multiple stakeholder viewpoints
• Work toward collaboration among stakeholders
• These context-free questions focus on customer, stakeholders, overall goals, and benefits of the system
o Who is behind the request for work?
o Who will use the solution?
o What will be the economic benefit of a successful solution?
o Is there another source for the solution needed?
• The next set of questions enable developer to better understand the problem and the customer's perceptions of the solution
o How would you characterize good output form a successful solution?
o What problem(s) will this solution address?
o Can you describe the business environment in which the solution will be used?
o Will special performance constraints affect the way the solution is approached?
• The final set of questions focuses on communication effectiveness
o Are you the best person to give "official" answers to these questions?
o Are my questions relevant to your problem?
o Am I asking too many questions?
o Can anyone else provide additional information?
o Should I be asking you anything else?

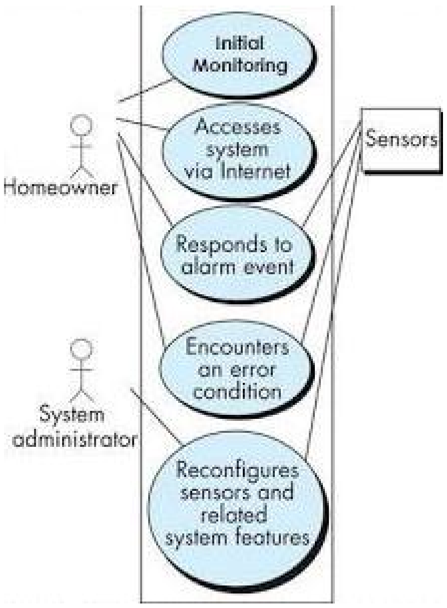
Eliciting Requirements

• Collaborative requirements gathering
o Meetings attended by both developers and customers
o Rules for preparation and participation are established
o Flexible agenda is used
o Facilitator controls the meeting
o Definition mechanism (e.g., stickers, flip sheets, electronic bulletin board) used to gauge group consensus
o Goal is to identify the problem, propose solution elements, negotiate approaches,
and specify preliminary set of solutions requirements
• Quality function deployment (QFD)
o Identifies three types of requirements (normal, expected, exciting)
o In customer meetings **function deployment** is used to determine value of each function that is required for the system
o **Information deployment** identifies both data objects and events that the system must consume or produce (these are linked to functions)
o **Task deployment** examines the system behavior in the context of its environment
o **Value analysis** is conducted to determine relative priority of each requirement generated by the deployment activities
• User-scenarios
o Also known as use-cases, describe how the system will be used
o Developers and users create a set of usage threads for the system to be constructed

Developing Use-Cases

• Each use-case tells stylized story about how end-users interact with the system under a specific set of circumstances
• First step is to identify **actors** (people or devices) that use the system in the context of the function and behavior of the system to be described
o Who are the primary or secondary actors?
o What preconditions must exist before story begins?
o What are the main tasks or functions performed by each actor?
o What extensions might be considered as the story is described?
o What variations in actor interactions are possible?
o What system information will the actor acquire, produce, or change?
o Will the actor need to inform the system about external environment changes?
o What information does the actor desire from the system?
o Does the actor need to be informed about unexpected changes?

Example : Use Case Diagram for Safe home system


Initial Monitoring
Accesses system via Internet
Responds to alarm event
Encounters an error condition
Reconfigures sensors and related system features
Homeowner
System administrator
Sensors

Negotiating Requirements

• Negotiation activities
o Identification of system key stakeholders
o Determination of stakeholders' "win conditions"
o Negotiate to reconcile stakeholders' win conditions into "win-win" result for all stakeholders (including developers)
• Key points
o It's not a competition
o Map out a strategy
o Listen actively
o Focus on other party's interests
o Don't let it get personal
o Be creative
o Be ready to commit

Requirement Validation

• Is each requirement consistent with overall project or system objective?
• Are all requirements specified at the appropriate level off abstraction?
• Is each requirement essential to system objective or is it an add-on feature?
• Is each requirement bounded and unambiguous?
• Do you know the source for each requirement?
• Do requirements conflict with one another?
• Is the requirement achievable in the proposed technical environment for the system or product?

• Is each requirement testable?
• Does the requirements model reflect the information, function, and behavior of the system to be built?
• Has the requirements model been partitioned in a way that exposes more detailed system information progressively?
• Have all the requirements patterns been properly validated and are they consistent with customer requirements?

UNIT –III Software Design

Design Concepts

What is it? Design is a meaningful engineering representation of something that is to be built. It can be traced to a customer's requirements and at the same time assessed for quality against a set of predefined criteria for "good" design. In the software engineering context, design focuses on four major areas of concern: data, architecture, interfaces, and components.

Who does it? Software engineers design computer based systems, but the skills required at each level of design work are different. At the data and architectural level, design focuses on patterns as they apply to the application to be built. At the interface level, human ergonomics often dictate our design approach. At the component level, a "programming approach" leads us to effective data and procedural designs.

Why is it important? You wouldn't attempt to build a house without a blueprint, would you? You'd risk confusion, errors, a floor plan that didn't make sense, windows and doors in the wrong place.

Computer software is considerably more complex than a house; hence, we need a blueprint— the design.

What are the steps? Design begins with the requirements model. We work to transform this model into four levels of design detail: the data structure, the system architecture, the interface representation, and the component level detail. During each design activity, we apply basic concepts and principles
that lead to high quality.

What is the work product? Ultimately, a Design Specification is produced. The specification is composed of the design models that describe data, architecture, interfaces, and components. Each is a work product of the design process.

SOFTWARE DESIGN AND SOFTWARE ENGINEERING

Each of the elements of the analysis model provides information that
is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in below Figure.

Software requirements, manifested by the data, functional, and behavioral models, feed the design task. Using one of a number of design methods, the design task produces a ***data design, an architectural design, an interface design, and a component design.***

The ***data design*** transforms the information domain model created during analysis into the data structures that will be required to implement the software. The data objects and relationships defined in the entity relationship diagram and the detailed data content depicted in the data dictionary provide the basis for the data design activity.

Part of data design may occur in conjunction with the design of software architecture. More detailed data design occurs as each software component is designed.

The ***architectural design*** defines the relationship between major structural elements of the software, the “design patterns” that can be used to achieve the requirements that have been defined for the system, and the constraints that affect the way in which architectural design patterns can be applied.

The architectural design representation—the framework of a computer-based system—can be derived from the system specification, the analysis model, and the interaction of subsystems defined within the analysis model.

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

The ***component-level design*** transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the PSPEC, CSPEC, and STD serve as the basis for component design.

During design we make decisions that will ultimately affect the success of software construction and, as important, the ease with which software can be maintained.

But why is design so important?
The importance of software design can be stated with a single word—*quality.*

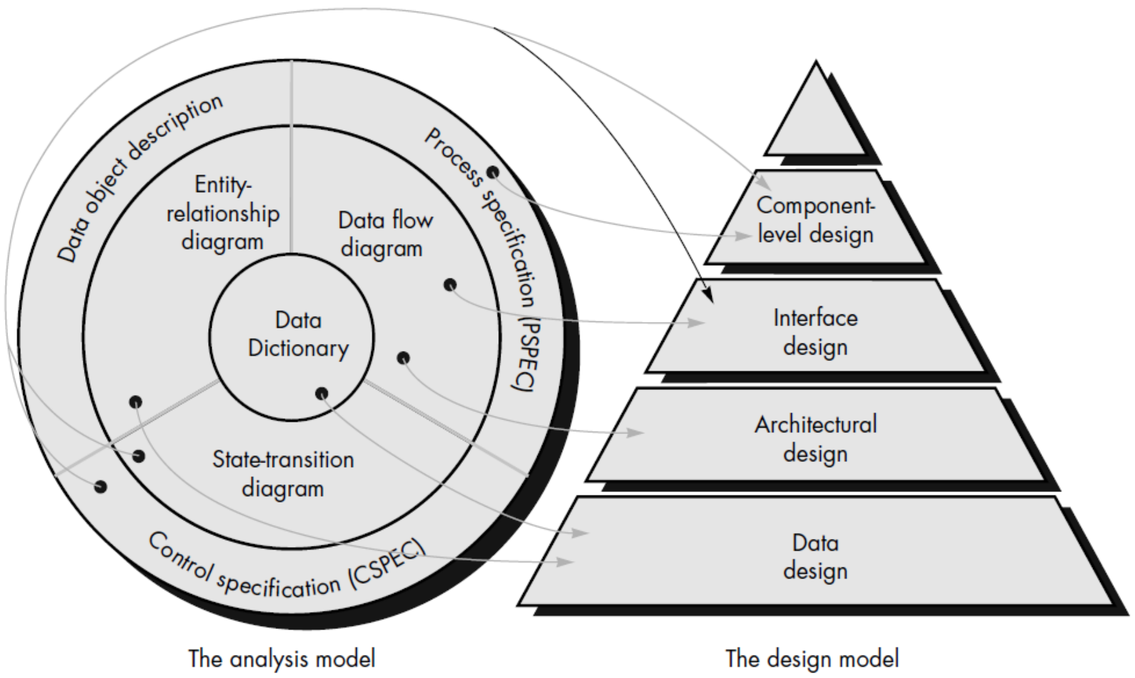


Figure. Translating Analysis Model to Software Design

THE DESIGN PROCESS

• The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.

• The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
• The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

DESIGN PRINCIPLES

Software design is both a process and a model.

The **design *process*** is a sequence of steps that enable the designer to describe all aspects of the software to be built.

The ***design model*** that is created for software provides a variety of different views of the computer software.

It begins by representing the totality of the thing to be built and slowly refines the thing to provide guidance for build each detail.

• **The design process should not suffer from "tunnel vision."** A good designer should consider alternative approaches, judging each based on the requirements of the problem, the resources available to do the job, and the design concepts.
• **The design should be traceable to the analysis model.** Because a single element of the design model often traces to multiple requirements, it is necessary to have a means for tracking how requirements have been satisfied by the design model.
• **The design should not reinvent the wheel.** Systems are constructed using a set of design patterns, many of which have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited! Design time should be invested in representing truly new ideas and integrating those patterns that already exist.
• **The design should "minimize the intellectual distance" between the software and the problem as it exists in the real world.**
That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.
• **The design should exhibit uniformity and integration.** A design is uniform if it appears that one person developed the entire thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.
• **The design should be structured to accommodate change.** The design concepts should enable a design to achieve this principle.

• **The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.** Well designed software should never “bomb.” It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.

• **Design is not coding, coding is not design.** Even when detailed procedural designs are created for program components, ***the level of abstraction of the design model is higher than source code***. The only design decisions made at the coding level address the small implementation details that enable the procedural design to be coded.

• **The design should be assessed for quality as it is being created, not after the fact.** A variety of design concepts and design measures are available to assist the designer in assessing quality.

• **The design should be reviewed to minimize conceptual (semantic) errors.** There is sometimes a tendency to focus on minutiae when the design is reviewed, missing the forest for the trees. A design team should ensure that major conceptual elements of the design (omissions, ambiguity, inconsistency) have been addressed before worrying about the syntax of the design model.

DESIGN CONCEPTS

1) Abstraction

When we consider a modular solution to any problem, many *levels of abstraction* can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more procedural orientation is taken. Problem-oriented terminology is coupled with implementation- oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

A ***procedural abstraction*** is a named sequence of instructions that has a specific and limited function. An example of a procedural abstraction would be the word *open* for a door. *Open* implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).

A ***data abstraction*** is a named collection of data that describes a data object. In the context of the procedural abstraction *open*, we can define a data abstraction called **door**. Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction *open* would make use of information contained in the attributes of the data abstraction **door**.

Control abstraction is the third form of abstraction used in software design. Like procedural and data abstraction, control abstraction implies a program control mechanism without specifying internal details. An example of a control abstraction is the *synchronization semaphore* used to coordinate activities in an operating system.

2) Refinement

Stepwise refinement is a top-down design strategy. A program is developed by successively refining levels of procedural detail.
A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

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

3) Modularity

Software architecture embodies modularity; that is, software is divided into separately named and addressable components, often called *modules,* that are integrated to satisfy problem requirements.

This is a "divide and conquer" conclusion—it's easier to solve a complex problem when you break it into manageable pieces. The result expressed in the last Expression has important implications with regard to modularity and software. It is, in fact, an argument for modularity.

Another important question arises when modularity is considered. How do we define an appropriate module of a given size? The answer lies in the method(s) used to define modules within a system.

1. to evaluate a design method with respect to its ability to define an effective modular system:

Modular decomposability. If a design method provides a systematic mechanism for decomposing the problem into sub-problems, it will reduce the complexity of the overall problem, thereby achieving an effective modular solution.
Modular composability. If a design method enables existing (reusable) design components to be assembled into a new system, it will yield a modular solution that does not reinvent the wheel.
Modular understandability. If a module can be understood as a standalone unit (without reference to other modules), it will be easier to build and easier to change.
Modular continuity. If small changes to the system requirements result in changes to individual modules, rather than system wide changes, the impact of change-induced side effects will be minimized.
Modular protection. If an aberrant condition occurs within a module and its effects are constrained within that module, the impact of error-induced side effects will be minimized.

4) Software Architecture

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system".

In its simplest form, architecture is the hierarchical structure of program components (modules), the manner in which these components interact and the structure of data that are used by the components. In a broader sense, however, *components* can be generalized to represent major system elements and their interactions.

One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enable a software engineer to reuse design level concepts.

Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.

Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models. *Structural models* represent architecture as an organized collection of program components.

Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications.

Dynamic models address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.

Process models focus on the design of the business or technical process that the system must accommodate.

Finally, *functional models* can be used to represent the functional hierarchy of a system.

5) Control Hierarchy

Control hierarchy, also called ***program structure***, represents the organization of program components (modules) and implies a hierarchy of control. It does not represent procedural aspects of software such as sequence of processes, occurrence or order of decisions, or repetition of operations; nor is it necessarily applicable to all architectural styles.

Different notations are used to represent control hierarchy for those architectural styles that are amenable to this representation. The most common is the treelike diagram (See Figure) that represents hierarchical control for call and return architectures.

Referring to Figure, *depth* and *width* provide an indication of the number of levels of control and overall *span of control*, respectively.

Fan-out is a measure of the number of modules that are directly controlled by another module.
Fan-in indicates how many modules directly control a given module.

The control relationship among modules is expressed in the following way: A module that controls another module is said to be *superordinate* to it, and conversely, a module controlled by another is said to be *subordinate* to the controller. For example, referring to Figure below, module *M* is superordinate to modules *a, b,* and *c.* Module *h* is subordinate to module *e* and is ultimately subordinate to module *M.*

Width-oriented relationships (e.g., between modules *d* and *e*) although possible to express in practice, need not be defined with explicit terminology.

The control hierarchy also represents two subtly different characteristics of the software architecture: visibility and connectivity. *Visibility* indicates the set of program components that may be invoked or used as data by a given component, even when this is accomplished indirectly. All of the objects are visible to the module.

Connectivity indicates the set of components that are directly invoked or used as data by a given component. For example, a module that directly causes another module to begin execution is connected to it.

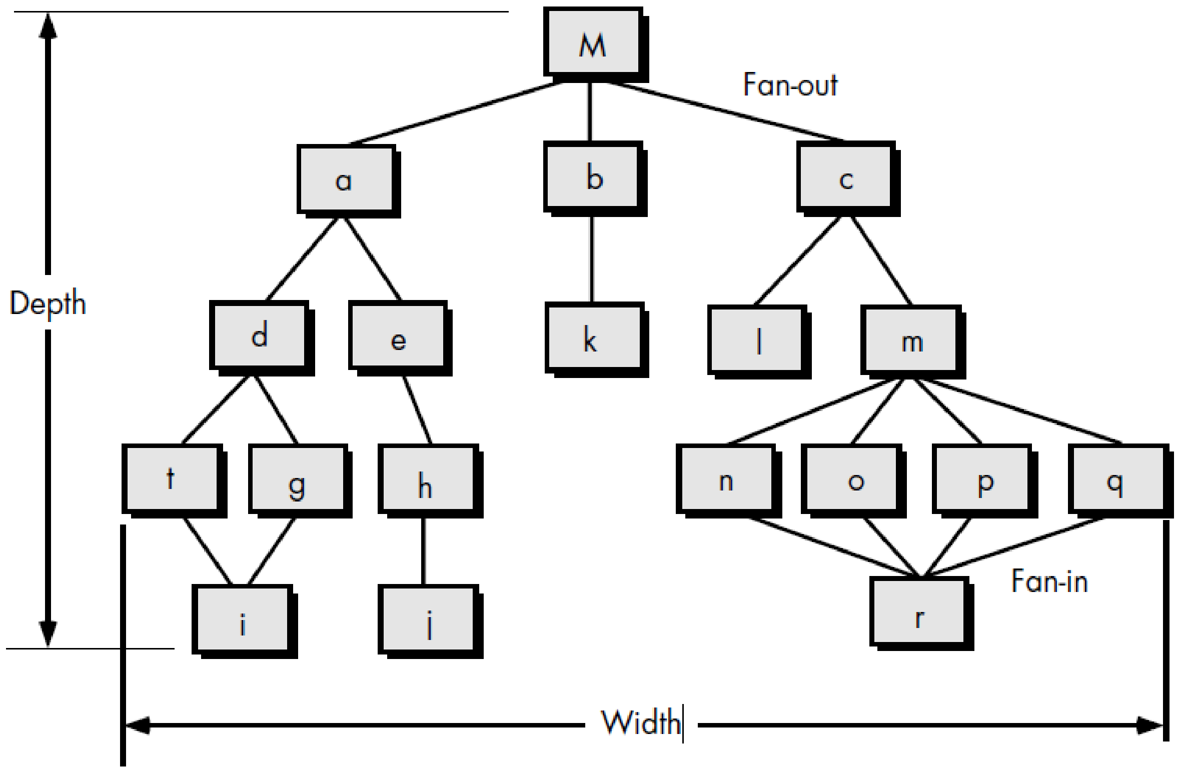


Figure. Control Hierarchy

6) **Structural Partitioning**

If the architectural style of a system is hierarchical, the program structure can be partitioned both horizontally and vertically. Referring to following Figure,

Horizontal partitioning defines separate branches of the modular hierarchy for each major program function.

Control modules, represented in a darker shade are used to coordinate communication between and execution of the functions. The simplest approach to horizontal partitioning defines three partitions—input, data transformation (often called *processing*) and output. Partitioning the architecture horizontally provides a number of distinct benefits:

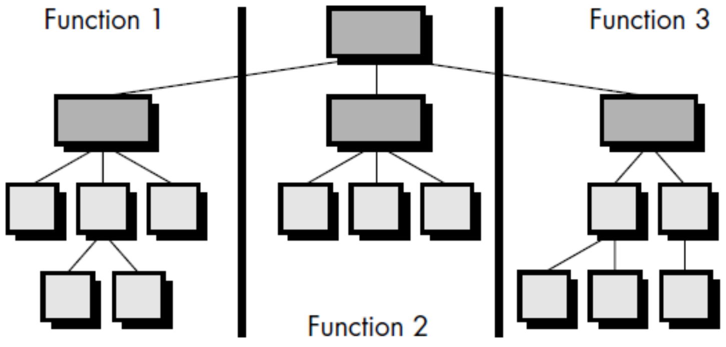
• software that is easier to test
• software that is easier to maintain
• propagation of fewer side effects
• software that is easier to extend

Because major functions are decoupled from one another, change tends to be less complex and extensions to the system (a common occurrence) tend to be easier to accomplish without side effects. On the negative side, horizontal partitioning often causes more data to be passed across module interfaces and can complicate the overall control of program flow (if processing requires rapid movement from one function to another).

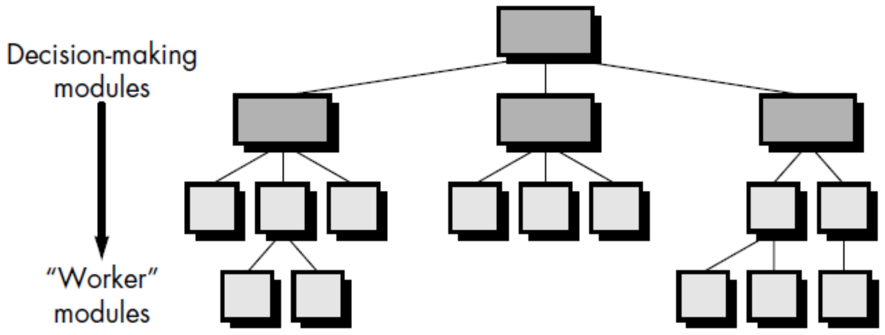
Vertical partitioning (See below Figure), often called ***factoring****,* suggests that control (decision making) and work should be distributed top-down in the program structure.

Top level modules should perform control functions and do little actual processing work. Modules that reside low in the structure should be the workers, performing all input, computation, and output tasks.

The nature of change in program structures justifies the need for vertical partitioning. Referring to Figure, it can be seen that a change in a control module (high in the structure) will have a higher probability of propagating side effects to modules that are subordinate to it. A change to a worker module, given its low level in the structure, is less likely to cause the propagation of side effects. In general, changes to computer programs revolve around changes to input, computation or transformation, and output. The overall control structure of the program (i.e., its basic behavior is far less likely to change). For this reason vertically partitioned structures are less likely to be susceptible to side effects when changes are made and will therefore be more maintainable—a key quality factor.



(a) Horizontal partitioning



(b) Vertical partitioning

7) Data Structure

Data structure is a representation of the logical relationship among individual elements of data. Because the structure of information will invariably affect the final procedural design, data structure is as important as program structure to the representation of software architecture.

The organization and complexity of a data structure are limited only by the ingenuity of the designer. There are, however, a limited number of classic data structures that form the building blocks for more sophisticated structures.

A *scalar item* is the simplest of all data structures. As its name implies, a scalar item represents a single element of information that may be addressed by an identifier; that is, access may be achieved by specifying a single address in memory. The size and format of a scalar item may vary within bounds that are dictated by a programming language. For example, a scalar item may be a logical entity one bit long, an integer or floating point number that is 8 to 64 bits long, or a character string that is hundreds or thousands of bytes long.

When scalar items are organized as a list or contiguous group, a *sequential vector* is formed. Vectors are the most common of all data structures and open the door to variable indexing of information.

When the sequential vector is extended to two, three, and ultimately, an arbitrary number of dimensions, an *n-dimensional space* is created. The most common *n*-dimensional space is the two-dimensional matrix. In many programming languages, an *n* dimensional space is called an *array*.

Items, vectors, and spaces may be organized in a variety of formats. A *linked list* is a data structure that organizes noncontiguous scalar items, vectors, or spaces in a manner (called *nodes*) that enables them to be processed as a list. Each node contains the appropriate data organization (e.g., a vector) and one or more pointers that indicate the address in storage of the next node in the list. Nodes may be added at any point in the list by redefining pointers to accommodate the new list entry.

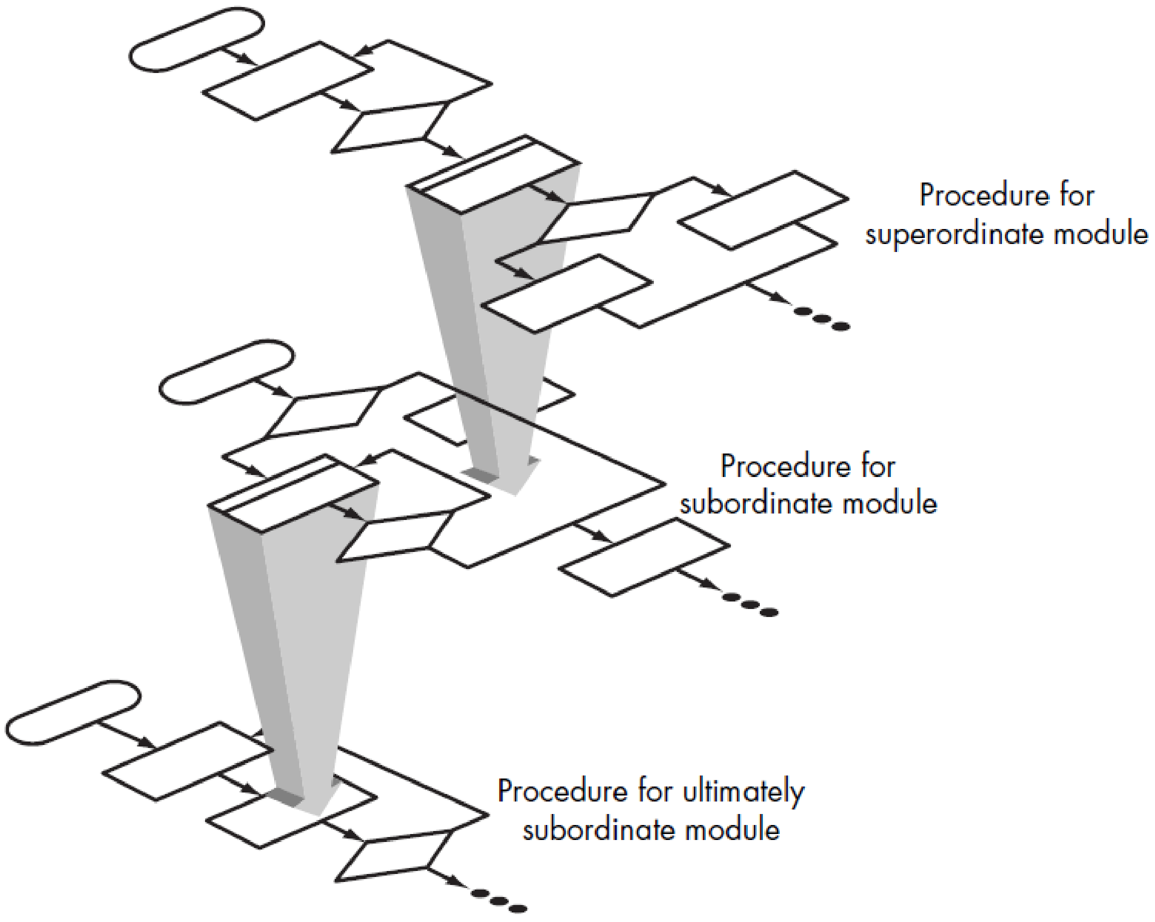
Other data structures incorporate or are constructed using the fundamental data structures just described. For example, a *hierarchical data structure* is implemented using multilinked lists that contain scalar items, vectors, and possibly, *n*-dimensional spaces. A hierarchical structure is commonly encountered in applications that require information categorization and associativity.

It is important to note that data structures, like program structure, can be represented at different levels of abstraction. For example, a stack is a conceptual model of a data structure that can be implemented as a vector or a linked list.
Depending on the level of design detail, the internal workings of a stack may or may not be specified.

8) Software Procedure

Program structure defines control hierarchy without regard to the sequence of processing and decisions. Software procedure focuses on the processing details of each module individually. Procedure must provide a precise specification of processing, including sequence of events, exact decision points, repetitive operations, and even data organization and structure.

There is, of course, a relationship between structure and procedure. The processing indicated for each module must include a reference to all modules subordinate to the module being described. That is, a procedural representation of software is layered as illustrated in the below Figure



Procedure for superordinate module
Procedure for subordinate module
Procedure for ultimately subordinate module

Figure. Procedure is Layered

9) Information Hiding

The concept of modularity leads every software designer to a fundamental question: "How do we decompose a software solution to obtain the best set of modules?"

The principle of *information hiding* suggests that modules be "characterized by design decisions that (each) hides from all others." In other words, modules should be specified and designed so that information (procedure and data) contained within a module is inaccessible to other modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module.

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedure are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

EFFECTIVE MODULAR DESIGN

Functional Independence

The concept of *functional independence* is a direct outgrowth of modularity and the concepts of abstraction and information hiding.

Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules. functional independence is a key to good design, and design is the key to software quality.

Independence is measured using two qualitative criteria: ***cohesion and coupling.***

Cohesion is a measure of the relative functional strength of a module.

Coupling is a measure of the relative interdependence among modules.

<u>Cohesion</u>

Cohesion is a natural extension of the information hiding concept. A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program.
Stated simply, a cohesive module should (ideally) do just one thing.

At the low (undesirable) end of the spectrum, we encounter a module that performs a set of tasks that relate to each other loosely, if at all. Such modules are termed ***coincidentally cohesive.***

A module that performs tasks that are related logically (e.g., a module that produces all output regardless of type) is ***logically cohesive.***

When a module contains tasks that are related by the fact that all must be executed with the same span of time, the module exhibits ***temporal cohesion.***

As an example of low cohesion, consider a module that performs error processing for an engineering analysis package. The module is called when computed data exceed pre-specified bounds. It performs the following tasks: (1) computes supplementary data based on original computed data, (2) produces an error report (with graphical content) on the user's workstation, (3) performs follow-up calculations requested by the user, (4) updates a database, and (5) enables menu selection for subsequent processing. Although the preceding tasks are loosely related, each is an independent functional entity that might best be performed as a separate module. Combining the functions into a single module can serve only to increase the likelihood of error propagation when a modification is made to one of its processing tasks.

Moderate levels of cohesion are relatively close to one another in the degree of module independence. When processing elements of a module are related and must be executed in a specific order, ***procedural cohesion*** exists.

When all processing elements concentrate on one area of a data structure, ***communicational cohesion*** is present.

High cohesion is characterized by a module that performs one distinct procedural task.
As we have already noted, it is unnecessary to determine the precise level of cohesion.

Rather it is important to strive for high cohesion and recognize low cohesion so that software design can be modified to achieve greater functional independence.

Coupling

Coupling is a measure of interconnection among modules in a software structure.

Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

In software design, we strive for lowest possible coupling.

Figure below provides examples of different types of module coupling.

Modules *a* and *d* are subordinate to different modules. Each is unrelated and therefore **no direct coupling** occurs.

Module *c* is subordinate to module *a* and is accessed via a conventional argument list, through which data are passed. As long as a simple argument list is present (i.e., simple data are passed; a one-to-one correspondence of items exists), low coupling (called ***data coupling***) is exhibited in this portion of structure.

A variation of data coupling, called ***stamp coupling****,* is found when a portion of a data structure (rather than simple arguments) is passed via a module interface. This occurs between modules *b* and *a*.

At moderate levels, coupling is characterized by passage of control between modules. ***Control coupling*** is very common in most software designs and is shown in Figure below where a "control flag" (a variable that controls decisions in a subordinate or superordinate module) is passed between modules *d* and *e*.

Relatively high levels of coupling occur when modules are tied to an environment external to software. For example, I/O couples a module to specific devices, formats, and communication protocols.

External coupling is essential, but should be limited to a small number of modules with a structure. High coupling also occurs when a number of modules reference a global data area.

Common coupling*,* as this mode is called, is shown in Figure below. Modules *c, g,* and *k* each access a data item in a global data area (e.g., a disk file or a globally accessible memory area). Module *c* initializes the item. Later module *g* re-computes and updates the item. Let's assume that an error occurs and *g* updates the item incorrectly. Much later in processing module, *k* reads the item, attempts to process it, and fails, causing the software to abort. The apparent cause of abort is module *k*; the actual cause, module *g.* Diagnosing problems in structures with considerable common coupling is time consuming and difficult. However, this does not mean that the use of global data is necessarily "bad." It does mean that a software designer must be aware of potential consequences of common coupling and take special care to guard against them.

The highest degree of coupling, ***content coupling***, occurs when one module makes use of data or control information maintained within the boundary of another module.

Secondarily, content coupling occurs when branches are made into the middle of a module. This mode of coupling can and should be avoided.

The coupling modes just discussed occur because of design decisions made when structure was developed. Variants of external coupling, however, may be introduced during coding. For example, *compiler coupling* ties source code to specific (and often nonstandard) attributes of a compiler; *operating system* (OS) *coupling* ties design and resultant code to operating system "hooks" that can create havoc when OS changes occur.

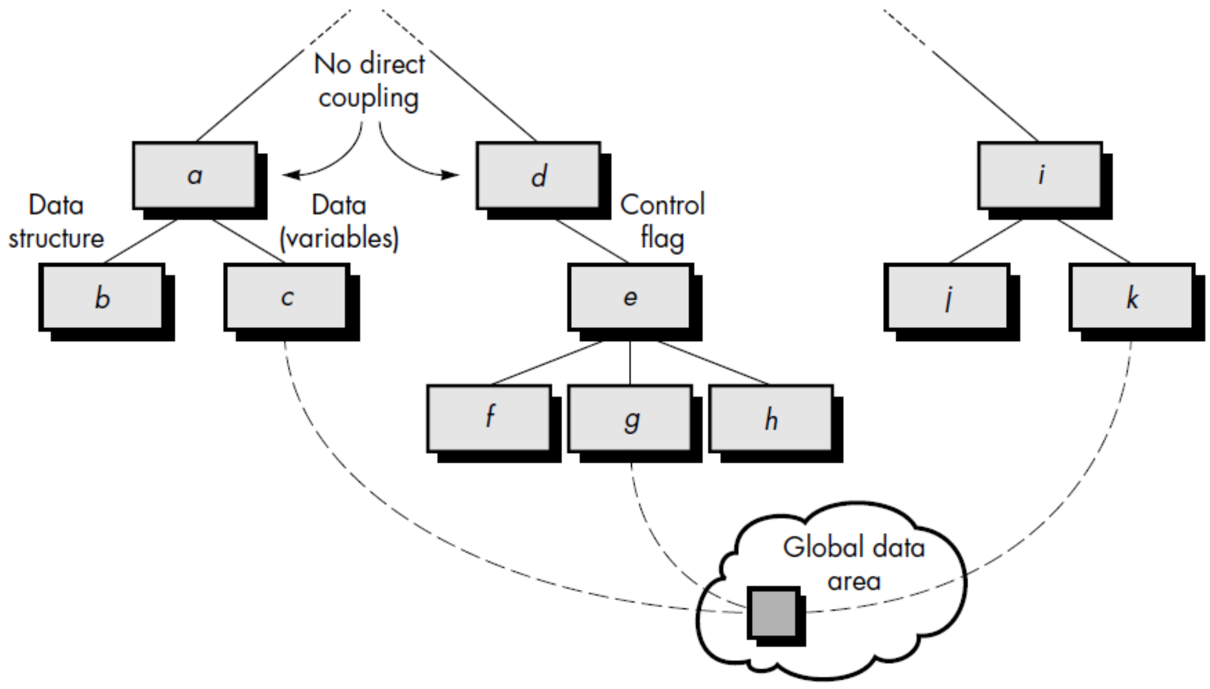


Figure. Types of Coupling

DATA DESIGN

It creates a model of data and/or information that is represented at a high level of abstraction (the customer/user’s view of data).

The structure of data has always been an important part of software design. The translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system.

Data Modeling, Data Structures, Databases, and the Data warehouse

The data design activity translates these elements of the requirements model into data structures at the software component level and, when necessary, a database architecture at the application level. The challenge for a business has been to extract useful information from this data environment, particularly when the information desired is cross-functional (e.g., information that can be obtained only if specific marketing data are cross-correlated with product engineering data).

To solve this challenge, the business IT community has developed *data mining* techniques, also called *knowledge discovery in databases* (KDD), that navigate through existing databases in an attempt to extract appropriate business-level information. However, the existence of multiple databases, their different structures, the degree of detail contained with the databases, and many other factors make data mining difficult within an existing database environment. An alternative solution, called a *data warehouse,* adds an additional layer to the data architecture.

A data warehouse is a separate data environment that is not directly integrated with day-to-day applications but encompasses all data used by a business. In a sense, a data warehouse is a large, independent database that encompasses some, but not all, of the data that are stored in databases that serve the set of applications required by a business. But many characteristics differentiate a data ware-house from the typical database.

Data Design at the Component Level

Data design at the component level focuses on the representation of data structures that are directly accessed by one or more software components.

The systematic analysis principles applied to function and behavior should also be applied to data.
All data structures and the operations to be performed on each should be identified.
A data dictionary should be established and used to define both data and pro-gram design.
Low-level data design decisions should be deferred until late in the design process.

The representation of data structure should be known only to those modules that must make direct use of the data contained within the structure.
A library of useful data structures and the operations that may be applied to them should be developed.
A software design and programming language should support the specification and realization of abstract data types.

ARCHITECTURAL DESIGN

What is it? Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the inter-relationships that occur among all architectural components of a system.

Who does it? Although a software engineer can design both data and architecture, the job is often allocated to specialists when large, complex systems are to be built. A database or data ware-house designer creates the data architecture for a system. The "system architect" selects an appropriate architectural style for the requirements derived during system engineering and software requirements analysis.

Why is it important? In the *Quick Look* for the last chapter, we asked: "You wouldn't attempt to build a house without a blueprint, would you?" You also wouldn't begin drawing blueprints by sketching the plumbing layout for the house. You'd need to look at the big picture—the house itself—before you worry about details. That's what architectural design does—it provides you with the big picture and ensures that you've got it right.

What are the steps? Architectural design begins with data design and then proceeds to the derivation of one or more representations of the system. Alternative architectural styles or patterns analyzed to derive the structure that is best suited to the customer requirements and quality attributes. Once alternative has been selected the architecture is elaborated using an architectural design method.

What is the work-product? Architectural design composed of Data Architecture and Program Structure. In addition component properties and relationships.

what is software architecture? The architecture is not the operational software. Rather, it is a representation that enables a software engineer to (1) analyze the effectiveness of the design in meeting its stated requirements, (2) consider architectural alternatives at a stage when making design changes is still relatively easy, and (3) reducing the risks associated with the construction of the software.

Why software architecture is important? Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer based system.

Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together".

ARCHITECTURAL STYLES

The software that is built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses (1) a set of *components* (e.g., a database, computational modules) that perform a function required by a system; (2) a set of *connectors* that enable "communication, co-ordinations and cooperation" among components; (3) *constraints* that define how components can be integrated to form the system; and (4) *semantic models* that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

COMMONLY USED ARCHITECTURAL STYLES AND PATTERNS

Data-centered architectures. A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Below Figure illustrates a typical data-centered style. Promotes integrability. Client components independently execute process.

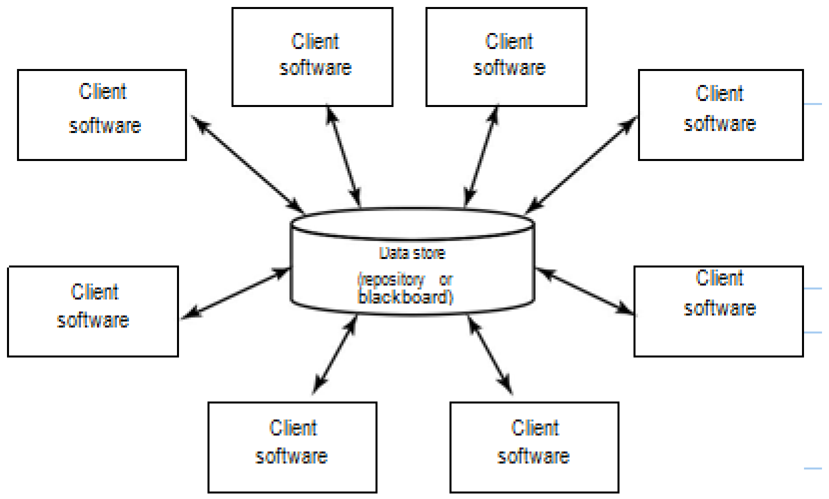
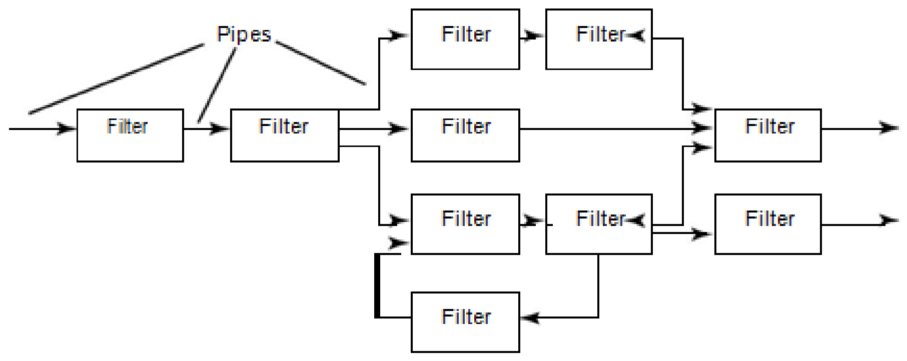
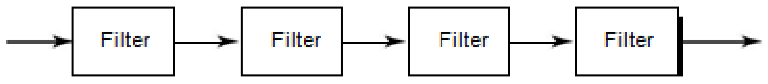


Figure. Data Centered Architecture

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



(a) Pipes and filters



Batch sequential

If the data flow degenerates into a single line of transforms, it is termed *batch sequential.* This pattern (Refer Figure) accepts a batch of data and then applies a series of sequential components (filters) to transform it.

Call and return architectures. This architectural style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale.

A number of sub styles exist within this category:
Main program/subprogram architectures. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components. ***(Refer previous Figure of Control Hierarchy)***

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

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

Layered architectures. The basic structure of a layered architecture is illus-trated in the following Figure. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.

These architectural styles are only a small subset of those available to the software designer. Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural pattern (style) or combination of patterns (styles) that best fits those characteristics and constraints can be chosen. In many cases, more than one pattern might be appropriate and alternative architectural styles might be designed and evaluated.

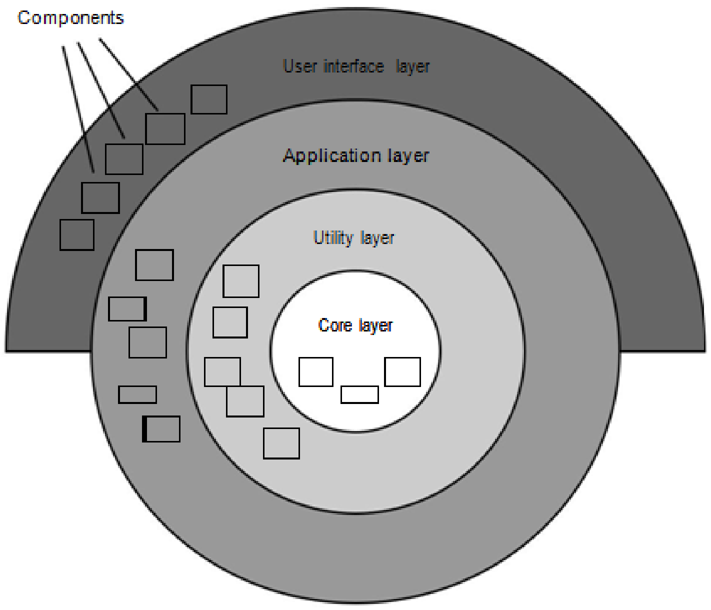


Figure. Layered Architecture

MAPPING REQUIREMENTS INTO A SOFTWARE ARCHITECTURE

The software requirements can be mapped into various representations of the design model. The transition from the requirements model to a variety of architectural styles. To illustrate one approach to architectural mapping, we consider the call and return architecture—an extremely common structure for many types of systems.

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.7 The transition from information flow (represented as a DFD) to pro-gram 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 program structure; (4) control hierarchy is defined; (5) resultant structure is refined using design measures and heuristics; and (6) the architectural description is refined and elaborated.
The type of information flow is the driver for the mapping approach required in step 3. In the following sections we examine two flow types.

Transform Flow

Recalling the fundamental system model (level 0 data flow diagram), information must enter and exit software in an "external world" form. For example, data typed on a keyboard, tones on a telephone line, and video images in a multimedia application are all forms of external world information. Such externalized data must be converted into an internal form for processing. Information enters the system along paths that transform external data into an internal form. These paths are identified as *incoming flow.* At the kernel of the software, a transition occurs. Incoming data are passed through a *transform center* and begin to move along paths that now lead "out" of the software. Data moving along these paths are called *outgoing flow.* The overall flow of data occurs in a sequential manner and follows one, or only a few, "straight line" paths. When a segment of a data flow diagram exhibits these characteristics, *transform flow* is present.

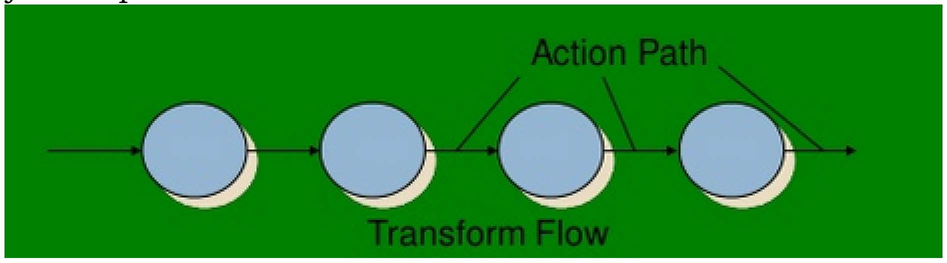


Figure. Transform Flow

Incoming Flow -> Transform Center -> Outgoing Flow

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 below 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 trans-action flow may be present. For example, in a transaction-oriented flow, information flow along an action path may have transform flow characteristics.

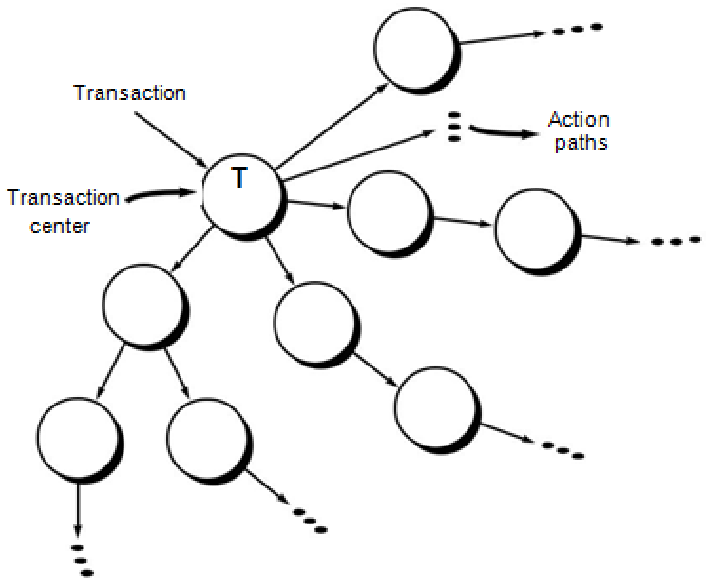
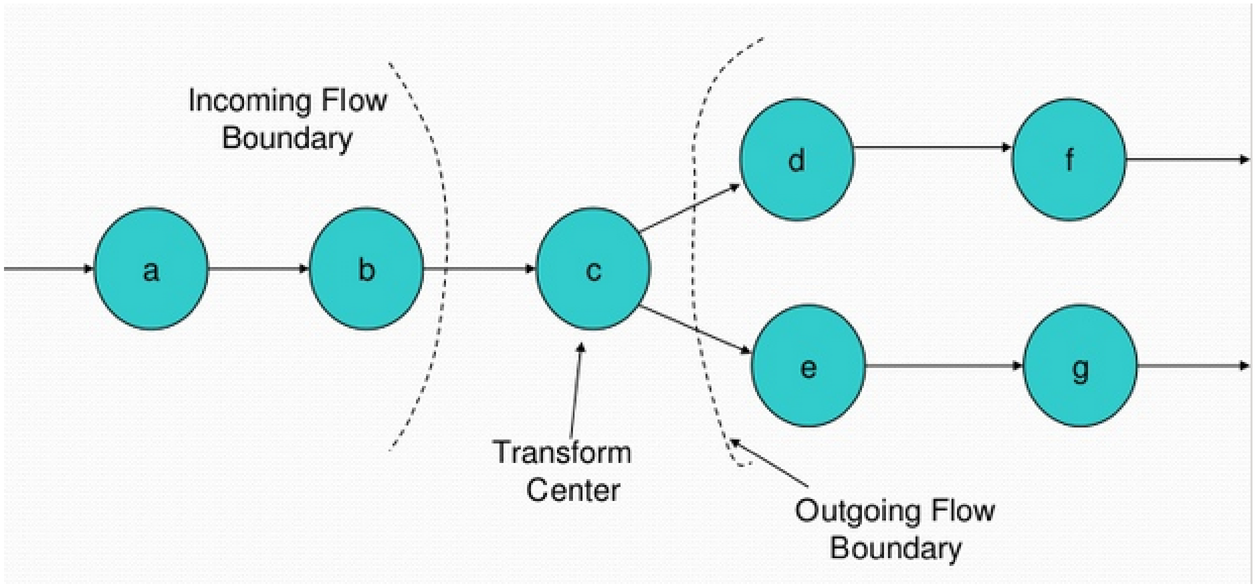


Figure. Transaction Flow
Incoming flow -> Transaction Centre -> Action Paths

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. In this section trans-form mapping is described by applying design steps to an example system.

After identifying Transform Flow, identify the boundaries of incoming and outgoing flow. In between the boundaries the central transform is located.


Incoming Flow Boundary
a
b
c
d
e
f
g
Transform Center
Outgoing Flow Boundary

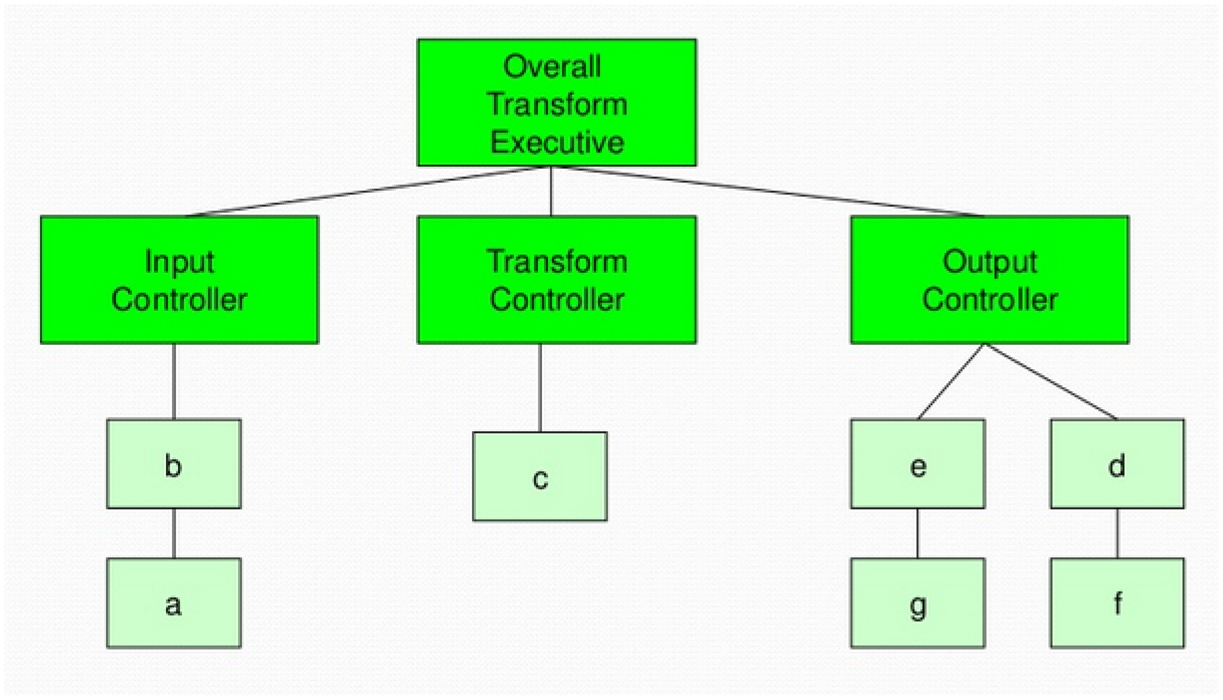


Figure. Mapping the Transform Flow to Call and Return Architecture

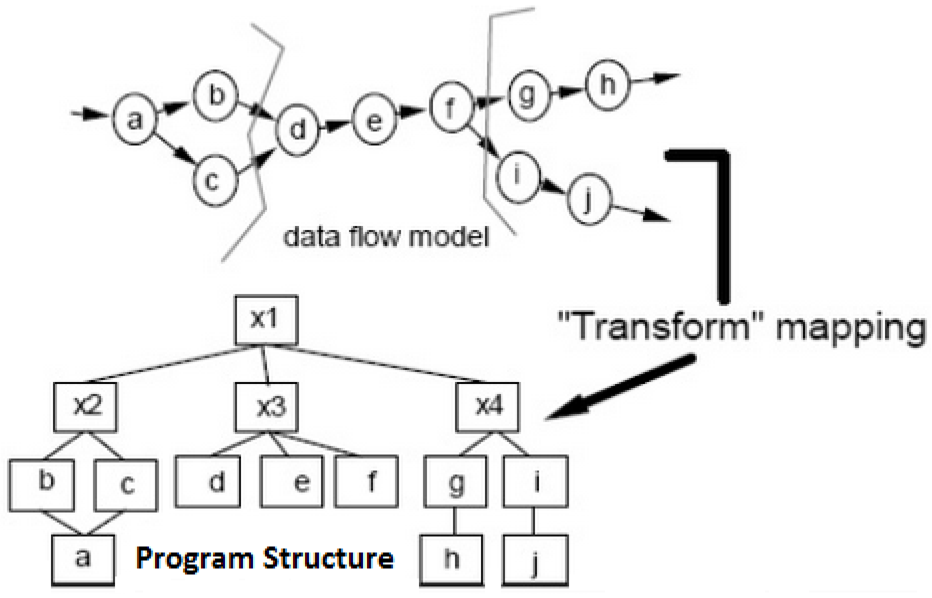


Figure. Transform Mapping

TRANSACTION MAPPING

In many software applications, a single data item triggers one or a number of information flows that effect a function implied by the triggering data item. The data item, called a *transaction,* and its corresponding flow characteristics are discussed earlier. In this section we consider design steps used to treat transaction flow.

After identifying the Transaction Flow, identify the Transaction Centre and flow characteristics of each action path. Similar to transform flow , boundaries are identified for incoming flow and the separate action paths. Each action path is a transform flow or transaction flow.

Design Steps:

Step 1. Review the fundamental system model.

Step 2. Review and refine data flow diagrams for the software.

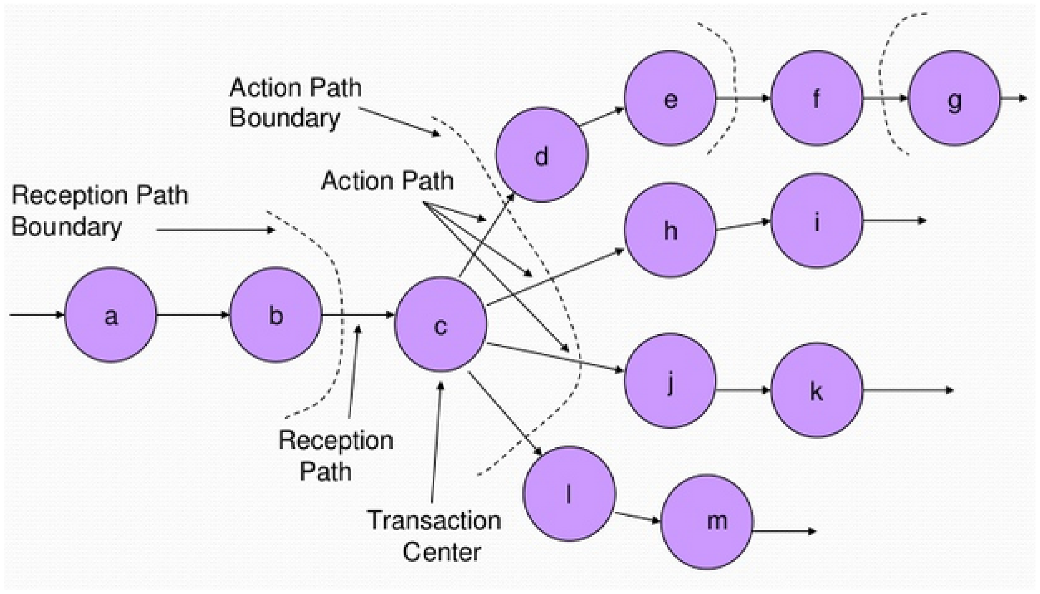
Step 3. Determine whether the DFD has transform or transaction flow characteristics.

Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries.
(Identify the Transaction Center and flow characteristics along each action path)

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.)

Step 6. Perform "second-level factoring."
(Mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture)

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(good cohesion, minimal coupling))


Action Path
Boundary
Reception Path
Boundary
Action Path
Reception
Path
Transaction
Center
a
b
c
d
e
f
g
h
i
j
k
l
m

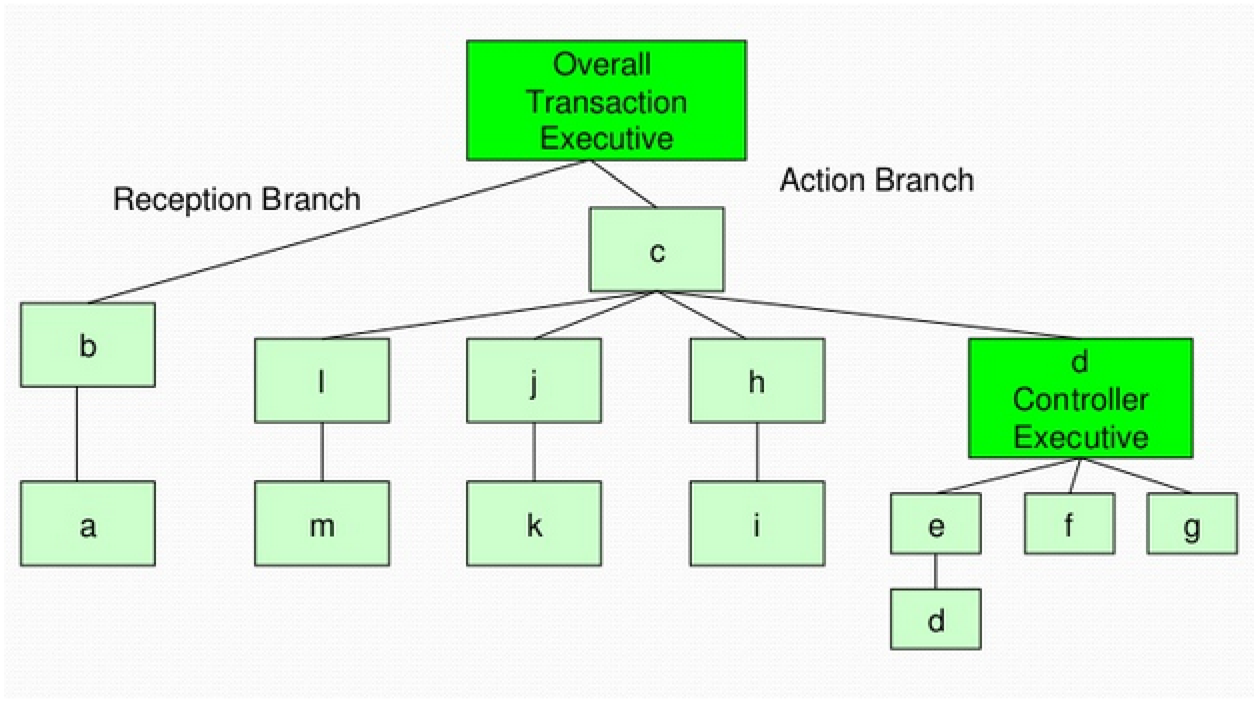


Figure. Mapping the Transaction Flow to Call and Return Architecture

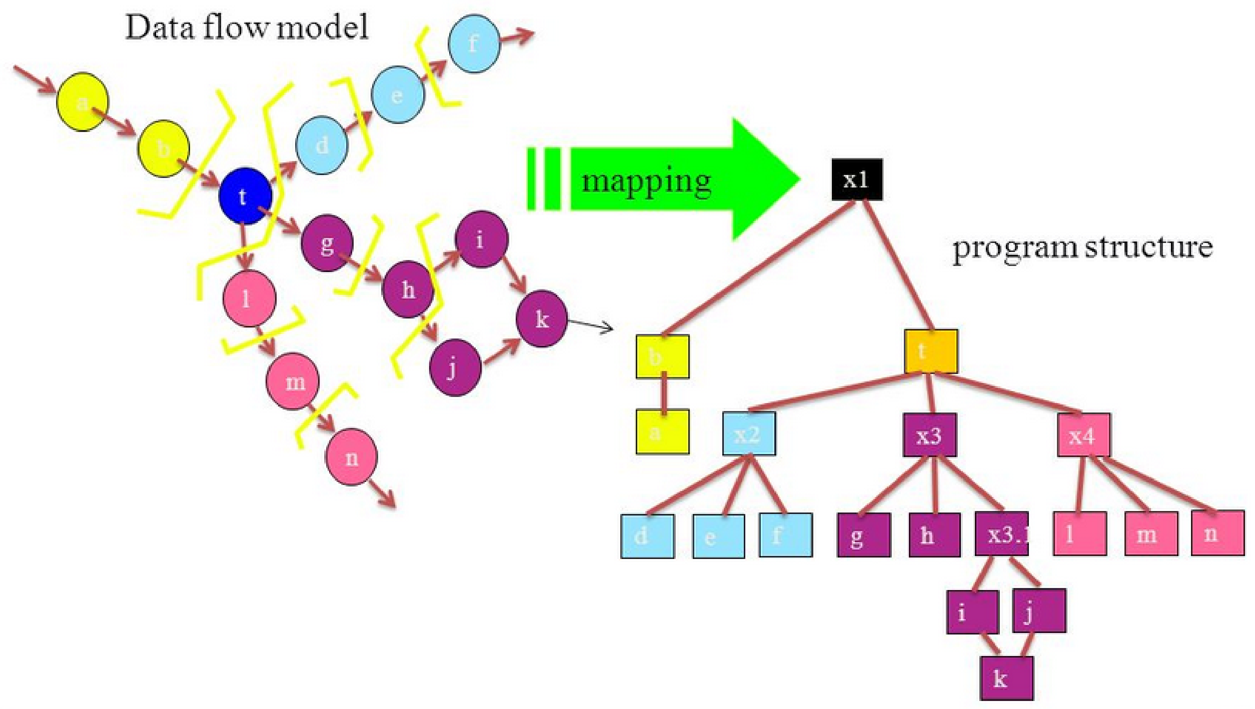


Figure. Transaction Mapping

USER INTERFACE DESIGN

Interface design focuses on three areas of concern:
(1) the design of interfaces between software components,
(2) the design of interfaces between the software and other nonhuman producers and consumers of information (i.e., other external entities), and
(3) the design of the interface between a human (i.e., the user) and the computer.

What is it? User interface design creates an effective communication medium between a human and a computer.
Following a set of interface design principles, design identifies interface objects

and actions and then creates a screen layout that forms the basis for a user interface prototype.

Who does it? A software engineer designs the user interface by applying an iterative process that draws on predefined design principles.

Why is it important? If software is difficult to use, if it forces you into mistakes, or if it frustrates your efforts to accomplish your goals, you won't like it, regardless of the computational power it exhibits or the functionality it offers. Because it molds a user's perception of the software, the interface has to be right.

What are the steps? User interface design begins with the identification of user, task, and environmental requirements. Once user tasks have been identified, user scenarios are created and analyzed to define a set of interface objects and actions. These form the basis for the creation of screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items. Tools are used to prototype and ultimately implement the design model, and the result is evaluated for quality.

What is the work product? User scenarios are created and screen layouts are generated. An interface prototype is developed and modified in an iterative fashion.

These golden rules actually form the basis for a set of user interface design principles that guide this important software design activity.

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

o **<u>Place the User in Control</u>**

design principles that allow the user to maintain control:

Define interaction modes in a way that does not force a user into unnecessary or undesired actions. An interaction mode is the current state of the interface. For example, if *spell check* is selected in a word-processor menu, the software moves to a spell checking mode. There is no reason to force the user to remain in spell checking mode if the user desires to make a small text edit along the way. The user should be able to enter and exit the mode with little or no effort.

Provide for flexible interaction. Because different users have different interaction preferences, choices should be provided. For example, software

might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, or voice recognition commands. But every action is not amenable to every interaction mechanism.
Consider, for example, the difficulty of using keyboard command (or voice input) to draw a complex shape.
Allow user interaction to be interruptible and undoable. Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to "undo" any action.
Streamline interaction as skill levels advance and allow the interaction to be customized. Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.
Hide technical internals from the casual user. The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology.
In essence, the interface should never require that the user interact at a level that is "inside" the machine (e.g., a user should never be required to type operating system commands from within application software).
Design for direct interaction with objects that appear on the screen. The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing. For example, an application interface that allows a user to "stretch" an object (scale it in size) is an implementation of direct manipulation.

- **Reduce the User's Memory Load**

The more a user has to remember, the more error-prone will be the interaction with the system.

design principles that enable an interface to reduce the user's memory load:
Reduce demand on short-term memory. When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions and results. This can be accomplished by providing visual cues that enable a user to recognize past actions, rather than having to recall them.
Establish meaningful defaults. The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences.
However, a "reset" option should be available, enabling the redefinition of original default values.
Define shortcuts that are intuitive. When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the print function), the

mnemonic should be tied to the action in a way that is easy to remember (e.g., first letter of the task to be invoked).
The visual layout of the interface should be based on a real world metaphor. For example, a bill payment system should use a check book and check register metaphor to guide the user through the bill paying process. This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.
Disclose information in a progressive fashion. The interface should be organized hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick. An example, common to many word-processing applications, is the underlining function. The function itself is one of a number of of functions under a text style menu. However, every underlining capability is not listed. The user must pick underlining, then all underlining options (e.g., single underline, double underline, dashed underline) are presented.

- **Make the Interface Consistent**

The interface should present and acquire information in a consistent fashion. This implies that (1) all visual information is organized according to a design standard that is maintained throughout all screen displays, (2) input mechanisms are constrained to a limited set that are used consistently throughout the application, and (3) mechanisms for navigating from task to task are consistently defined and implemented.

Design principles that help make the interface consistent:
Allow the user to put the current task into a meaningful context. Many interfaces implement complex layers of interactions with dozens of screen images.
It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand.
In addition, the user should be able to determine where he has come from and what alternatives exist for a transition to a new task.

Maintain consistency across a family of applications. A set of applications (or products) should all implement the same design rules so that consistency is maintained for all interaction.

If past interactive models have created user expectations, do not make

changes unless there is a compelling reason to do so. Once a particular interactive sequence has become a de facto standard (e.g., the use of alt-S to save a file), the user expects this in every application he encounters. A change (e.g., using alt-S to invoke scaling) will cause confusion.

USER INTERFACE DESIGN

The software engineer creates a *design model,* establishes a *user model,* the end-user develops a *user's model* or the *system perception,* and the implementers of the system create a *system image*

USERS

users can be categorized as:

• **Novices.** No *syntactic knowledge* of the system and little *semantic knowledge* of the application or computer usage in general.

• **Knowledgeable, intermittent users.** Reasonable semantic knowledge of the application but relatively low recall of syntactic information necessary to use the interface.

• **Knowledgeable, frequent users.** Good semantic and syntactic knowledge that often leads to the "power-user syndrome"; that is, individuals who look for shortcuts and abbreviated modes of interaction.

The information gathered as part of the analysis activity is used to create an analysis model for the interface. Using this model as a basis, the design activity commences.

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

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

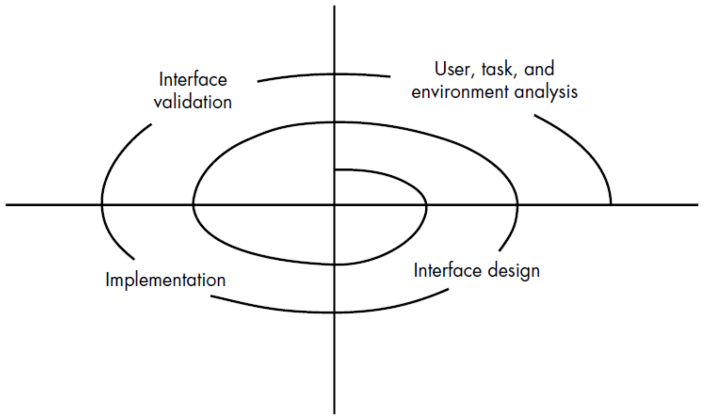


Figure. The Users Interface Design Process

the user interface design process encompasses four distinct framework activities:

1. User, task, and environment analysis and modeling
2. Interface design
3. Interface construction
4. Interface validation

The spiral shown in Figure implies that each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration of requirements and the resultant design.

In most cases, the implementation activity involves prototyping—the only practical way to validate what has been designed.

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

In essence, the software engineer attempts to understand the system perception for each class of users.

Once general requirements have been defined, a more detailed task analysis is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated (over a number of iterative passes through the spiral).

Validation focuses on (1) the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements;

(2) the degree to which the interface is easy to use and easy to learn; and
(3) the users' acceptance of the interface as a useful tool in their work.

COMPONENT LEVEL DESIGN

What is it? Data, architectural, and interface design must be translated into operational software. To accomplish this, the design must be represented at a level of abstraction that is close to code. Component-level design establishes the algorithmic detail required to manipulate data structures, effect communication between software components via their interfaces, and implement the processing algorithms allocated to each component.

Who does it? A software engineer performs component-level design.

Why is it important? You have to be able to determine whether the program will work before you build it. The component-level design represents the software in a way that allows you to review the details of the design for correctness and consistency with earlier design representations (i.e., the data, architectural, and interface designs). It provides a means for assessing whether data structures, interfaces, and algorithms will work.

What are the steps? Design representations of data, architecture, and interfaces form the foundation for component-level design. The processing narrative for each component is translated into a procedural design model using a set of structured programming constructs. Graphical, tabular, or text-based notation is used to represent the design.

What is the work product? The procedural design for each component, represented in graphical, tabular, or text-based notation, is the primary work product produced during component-level design.

The design process encompasses a sequence of activities that slowly reduces the level of abstraction with which software is represented. Component-level design depicts the software at a level of abstraction that is very close to code.

At the component level, the software engineer must represent data structures, interfaces, and algorithms in sufficient detail to guide in the generation of programming language source code. To accomplish this, the designer uses one of a number of design notations that represent component-level detail in either graphical, tabular, or text-based formats.

It is possible to represent the component-level design using a programming language. In essence, the program is created using the design model as a guide. An alter-native approach is to represent the procedural design using some intermediate (e.g., graphical, tabular, or text-based) representation that can be translated easily into source code. Regardless of the mechanism that is used to represent the component-level design, the data structures, interfaces, and algorithms defined should conform to a variety of well-established procedural design guidelines that help us to avoid errors as the procedural design evolves.

The foundations of component-level design were formed in the early 1960s. In the late 1960s, Dijkstra and others proposed the use of a set of constrained logical constructs from which any program could be formed. The constructs emphasized "maintenance of functional domain." That is, each construct had a predictable logical structure, was entered at the top and exited at the bottom, enabling a reader to follow procedural flow more easily.

The constructs are sequence, condition, and repetition. *Sequence* implements processing steps that are essential in the specification of any algorithm. *Condition* pro-vides the facility for selected processing based on some logical occurrence, and *repetition* allows for looping. These three constructs are fundamental to *structured programming*—an important component-level design technique.

The structured constructs were proposed to limit the procedural design of soft-ware to a small number of predictable operations. Complexity metrics indicate that the use of the structured constructs reduces program complexity and thereby enhances readability, testability, and maintainability. The use of a limited number of logical constructs also contributes to a human understanding process that psychologists call *chunking.* To understand this process, consider the way in which you are reading this page. You do not read individual letters but rather recognize pat-terns or chunks of letters that form words or phrases. The structured constructs are logical chunks that allow a reader to recognize procedural elements of a module, rather than reading the design or code line by line. Understanding is enhanced when readily recognizable logical patterns are encountered.

Any program, regardless of application area or technical complexity, can be designed and implemented using only the three structured constructs. It should be noted, however, that dogmatic use of only these constructs can sometimes cause practical difficulties.

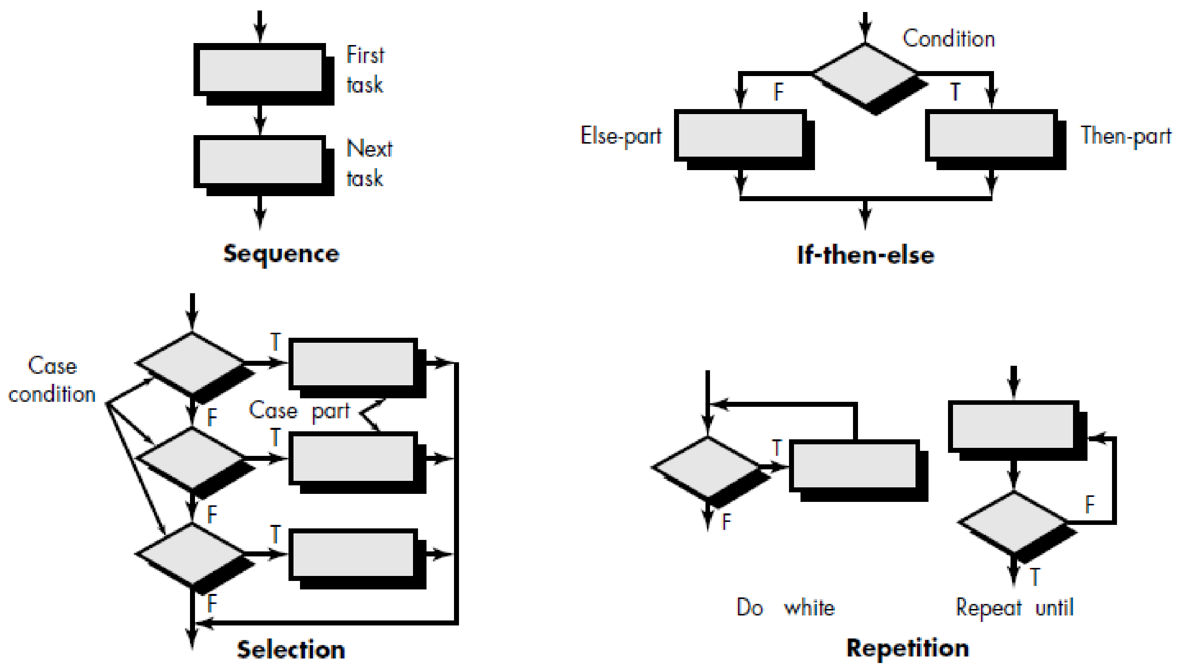
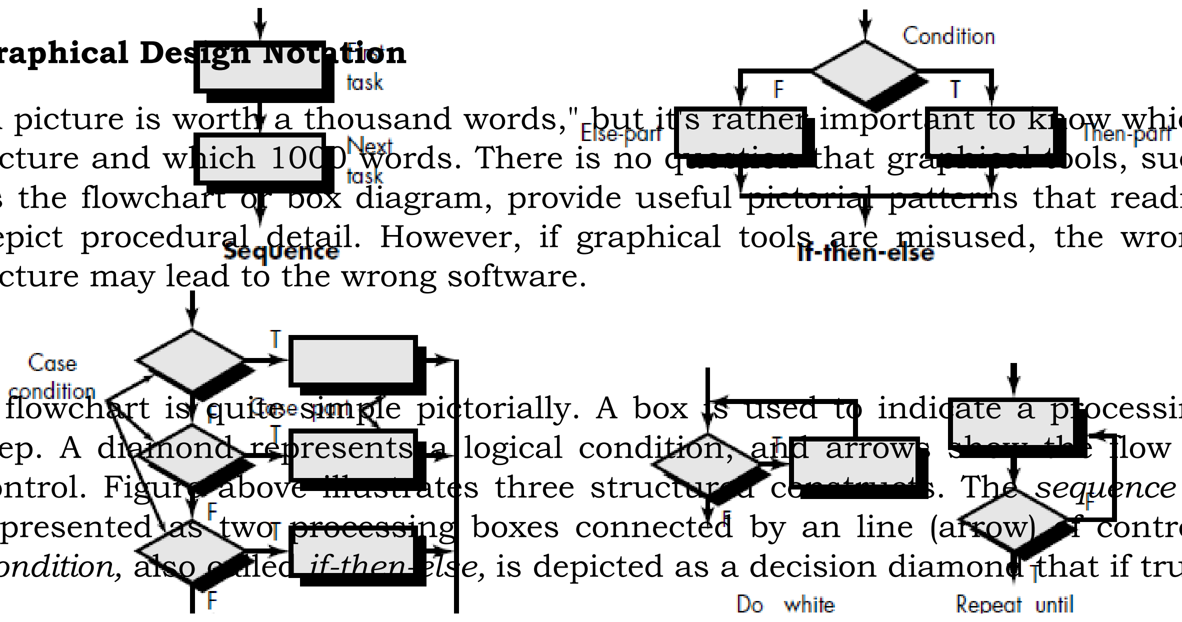


Figure. Flow Chart Constructs

Graphical Design Notation

"A picture is worth a thousand words," but it's rather important to know which picture and which 1000 words. There is no question that graphical tools, such as the flowchart or box diagram, provide useful pictorial patterns that readily depict procedural detail. However, if graphical tools are misused, the wrong picture may lead to the wrong software.



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 above 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 represented 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.

The structured constructs may be nested within one another as shown in Figure. Referring to the figure, repeat-until forms the then part of if-then-else (shown enclosed by the outer dashed boundary). Another if-then-else forms the else part of the larger condition. Finally, the condition itself becomes a second block in a sequence. By nesting constructs in this manner, a complex logical schema may be developed. It should be noted that any one of the blocks in Figure could reference another module, thereby accomplishing procedural layering implied by program structure.

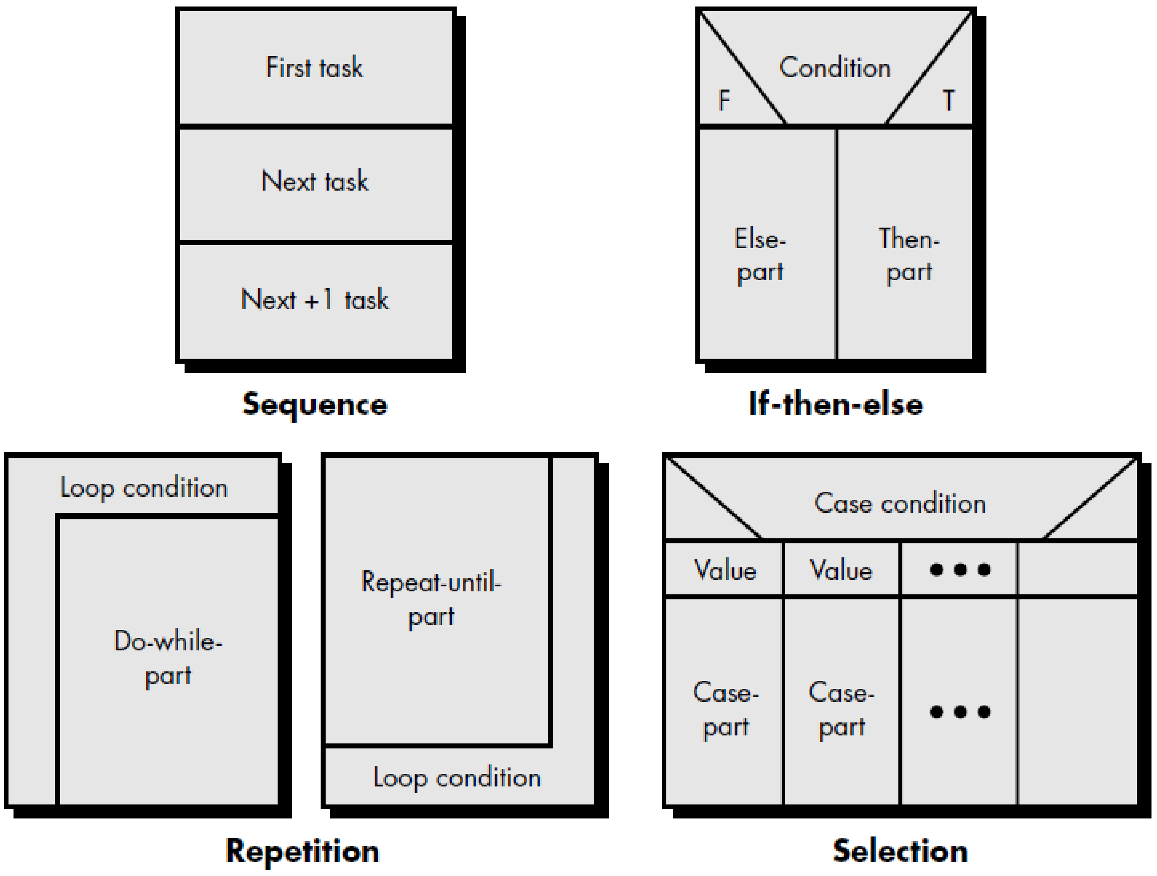


Figure. Box Diagram Constructs

Tabular Design Notation

In many software applications, a module may be required to evaluate a complex combination of conditions and select appropriate actions based on these conditions. Decision tables provide a notation that translates actions and conditions (described in a processing narrative) into a tabular form. The table is difficult to misinterpret and may even be used as a machine readable input to a table driven algorithm. In a comprehensive treatment of this design tool.

Some old software tools and techniques mesh well with new tools and techniques of soft-ware engineering. Decision tables are an excellent example. Decision tables preceded soft-ware engineering by nearly a decade, but fit so well with software engineering that they might have been designed for that purpose.

Decision table organization is illustrated in Figure above Referring to the figure, the 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:

List all actions that can be associated with a specific procedure (or module).
List all conditions (or decisions made) during execution of the procedure.

Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible per-mutation of conditions.

Define rules by indicating what action(s) occurs for a set of conditions.

To illustrate the use of a decision table, consider the following excerpt from a processing narrative for a public utility billing system:

If the customer account is billed using a fixed rate method, a minimum monthly charge is assessed for consumption of less than 100 KWH (kilowatt-hours). Otherwise, computer billing applies a Schedule A rate structure. However, if the account is billed using a variable rate method, a Schedule A rate structure will apply to consumption below 100 KWH, with additional consumption billed according to Schedule B.

Figure below illustrates a decision table representation of the preceding narrative. Each of the five rules indicates one of five viable conditions (i.e., a T (true) in both fixed rate and variable rate account makes no sense in the context of this procedure; therefore, this condition is omitted). As a general rule, the decision table can be effectively used to supplement other procedural design notation.

	Rules								
Conditions	**1**	**2**	**3**	**4**					n
Condition #1	✓			✓	✓				
Condition #2		✓		✓					
Condition #3			✓		✓				
Actions									
Action #1	✓			✓	✓				
Action #2		✓		✓					
Action #3			✓						
Action #4			✓	✓	✓				
Action #5	✓	✓			✓				

Figure. Tabular Design Notation

Program Design Language

Program design language (PDL), also called *structured English* or *pseudo-code,* 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)".

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. Given the use of narrative text embedded directly into a syntactical structure, PDL cannot be compiled (at least not yet). However, PDL tools currently exist to translate PDL into a programming language “skeleton” and/or a graphical representation (e.g., a flowchart) of design. These tools also produce nesting maps, a design operation index, cross-reference tables, and a variety of other information.

A program design language may be a simple transposition of a language such as Ada or C. Alternatively, it may be a product purchased specifically for procedural design. Regardless of origin, 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.

A basic PDL syntax should include constructs for subprogram definition, interface description, data declaration, techniques for block structuring, condition constructs, repetition constructs, and I/O constructs. The format and semantics for some of these PDL constructs are presented in the section that follows.

It should be noted that PDL can be extended to include keywords for multitasking and/or concurrent processing, interrupt handling, inter process synchronization, and many other features. The application design for which PDL is to be used should dictate the final form for the design language.

	Rules				
Conditions	**1**	**2**	**3**	**4**	**5**
Fixed rate acct.	T	T	F	F	F
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					✓

Figure. The Resultant Decision Table

A PDL Example

Sample PDL for select

```
SET prompt
SELECT from list of characters
          CHOOSE from value returned
                    CHOICE Bart
                PUT message
        END CHOICE
                    CHOICE Homer
                               PUT message
                    END CHOICE
                    DEFAULT CHOICE
                               BREAK out of loop
                    END CHOICE
          END CHOOSE
END SELECT
```

SOFTWARE TESTING

Testing Objectives

Testing is a process of executing a program with the intent of finding an error.

A good test case is one that has a high probability of finding an as-yet-undiscovered error.

A successful test is one that uncovers an as-yet-undiscovered error.

Objective is to Find maximum number of errors in the source code within minimum amount of time and minimum amount of effort.

Once source code has been generated, software must be tested to uncover (and correct) as many errors as possible before delivery to the customer. The goal is to design a series of test cases that have a high likelihood of finding errors.

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.

Testability

Software testability is simply how easily [a computer program] can be tested.

Operability. "The better it works, the more efficiently it can be tested."

Observability. "What you see is what you test."

Controllability. "The better we can control the software, the more the testing can be automated and optimized."

Decomposability. "By controlling the scope of testing, we can more quickly iso-late problems and perform smarter retesting."

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

Stability. "The fewer the changes, the fewer the disruptions to testing."

Understandability. "The more information we have, the smarter we will test."

TEST CASE DESIGN

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

(1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function;
(2) Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised.

The first test approach is called black-box testing and the second, white-box testing.

SOFTWARE TESTING TECHNIQUES

Black-box and White-box testing

When computer software is considered, ***Black-box testing*** alludes to tests that are conducted at the software interface. Although they are designed to uncover errors, black-box tests are used to demonstrate that software functions are operational, that input is properly accepted and output is correctly produced, and that the integrity of external information (e.g., a database) is maintained. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software.

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 to determine if the expected or asserted status corresponds to the actual status.

White-box Testing

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 white-box testing methods, the software engineer can derive test cases that (1) guarantee that all independent paths within a module have been exercised at least once, exercise all logical decisions on their true and false sides, (3) execute all loops at their boundaries and within their operational bounds, and (4) exercise internal data structures to ensure their validity.

The goal is to remove the errors of the following kinds:

- ***Logic errors*** and incorrect assumptions are inversely proportional to the probability that a program path will be executed.
- We often believe that a ***logical path*** is not likely to be executed when, in fact, it may be executed on a regular basis.
- ***Typographical errors*** are random.

1) BASIS PATH TESTING

Basis path testing is a white-box testing technique. The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

1).1) Flow Graph Notation

Before the basis path method can be introduced, a simple notation for the representation of control flow, called a *flow graph* (or *program graph*) must be introduced. The flow graph depicts logical control flow using the notation illustrated in Figure. Each structured construct has a corresponding flow graph symbol.

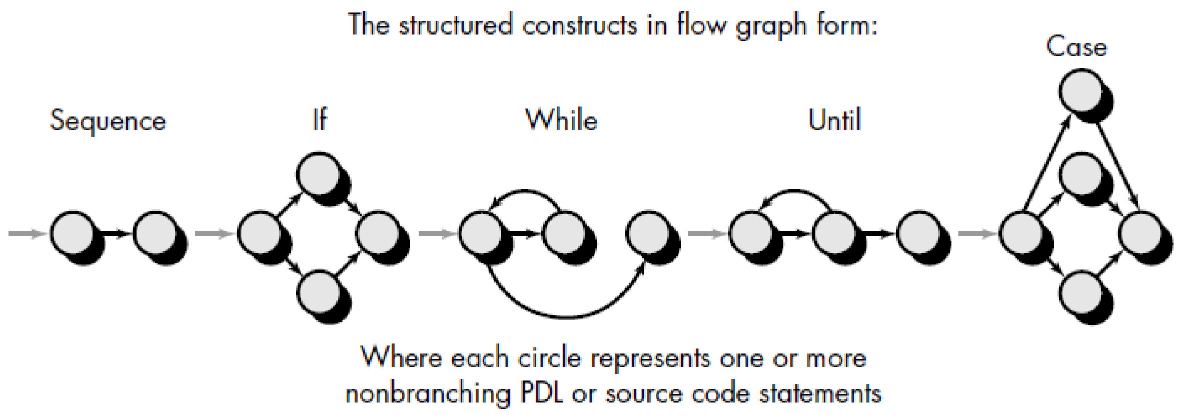

The structured constructs in flow graph form:
Sequence
If
While
Until
Case
Where each circle represents one or more nonbranching PDL or source code statements

Figure. Flow Graph Notation

Each circle, called a *flow graph* ***node***, represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called ***edges*** or *links,* represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the symbol for the if-then-else construct). Areas bounded by edges and nodes are called ***regions***. When counting regions, we include the area outside the graph as a region.

When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated. A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement. The PDL segment translates into the flow graph. Note that a separate node is created for each of the conditions, each node that contains a condition is called a ***predicate node*** and is characterized by two or more edges emanating from it.

STEPS:

1. Using Design or Code draw the corresponding Flow Graph
2. Draw the Decision to Decision Path Graph
3. Determine the Cyclometric Complexity
4. Determine a basis set of independent paths
5. Prepare test cases that will force execution of each path in the basis set.

Control Flow Graph:

Describes how the control flows through the programs.

This is analyzed using a graphical representation known as a flow graph. The flow graph is a directed graph in which node represents the statement, edges represent the flow of control.

Decision to Decision Path Graph:

Draw a Decision to Decision Path Graph from flow graph.

Concentrate only on decision nodes.

The nodes of a flow graph, which are in a sequence are combined in to a single node.

Decision to Decision Path Graph is a Directed Graph in which the nodes represent the sequence of statements and the edges represents the control flow between the nodes.

Independent Path Graph:

To find the Independent paths, execute all independent paths atleast once during path testing.

It is to ensure that, i) every program statement is executed atleast once ii) every branch has been exercised for True and False condition.

Cyclomatic Complexity

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined.

How do we know how many paths to look for? The computation of cyclomatic complexity provides the answer.

Cyclomatic complexity has a foundation in graph theory and provides us with an extremely useful software metric. Complexity is computed in one of three ways:

Cyclomatic complexity, $V(G)$, for a flow graph, G, is defined as

$V(G) = E - N + 2$

(where E is the number of flow graph edges, N is the number of flow graph nodes.)

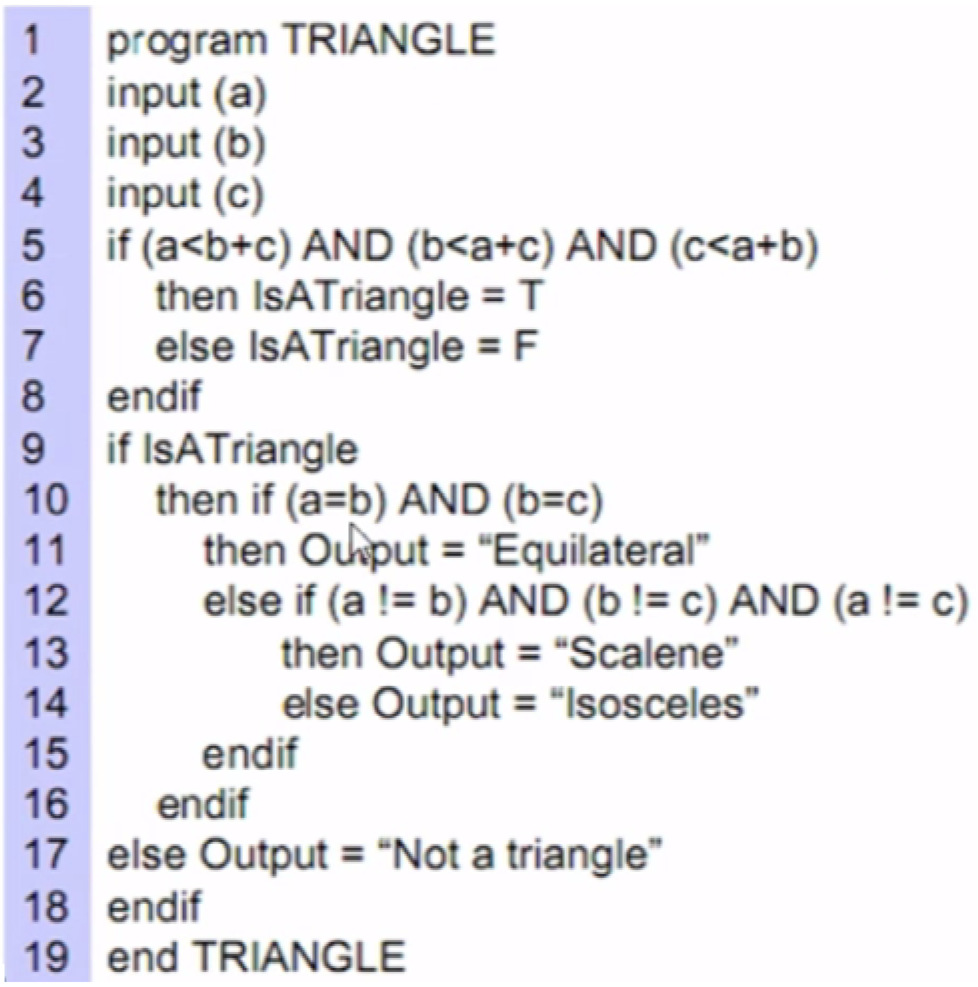
Cyclomatic complexity, $V(G)$, for a flow graph, G, is also defined

as $V(G) = P + 1$

where P is the number of predicate nodes contained in the flow graph G.

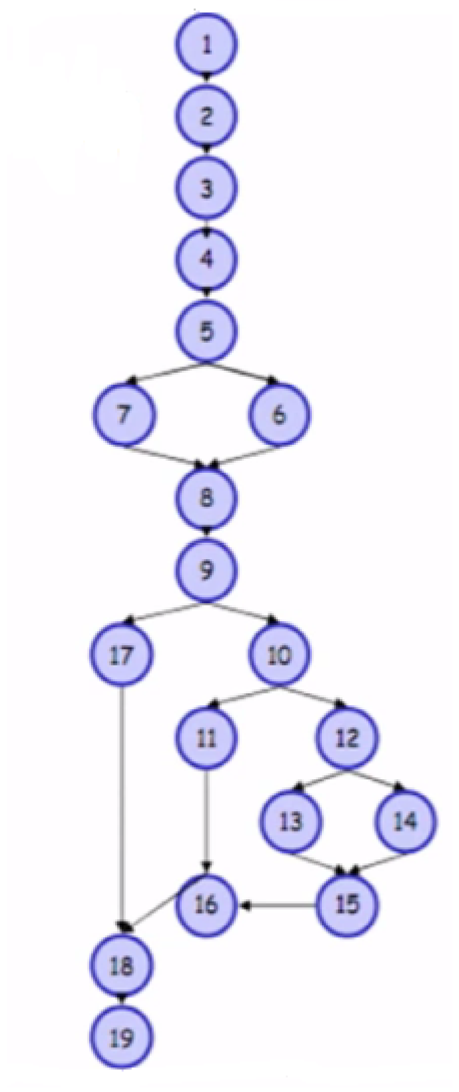
Cyclomatic complexity, $V(G)$, for a flow graph, G, is also defined as $V(G)$ = *Total number of Regions*

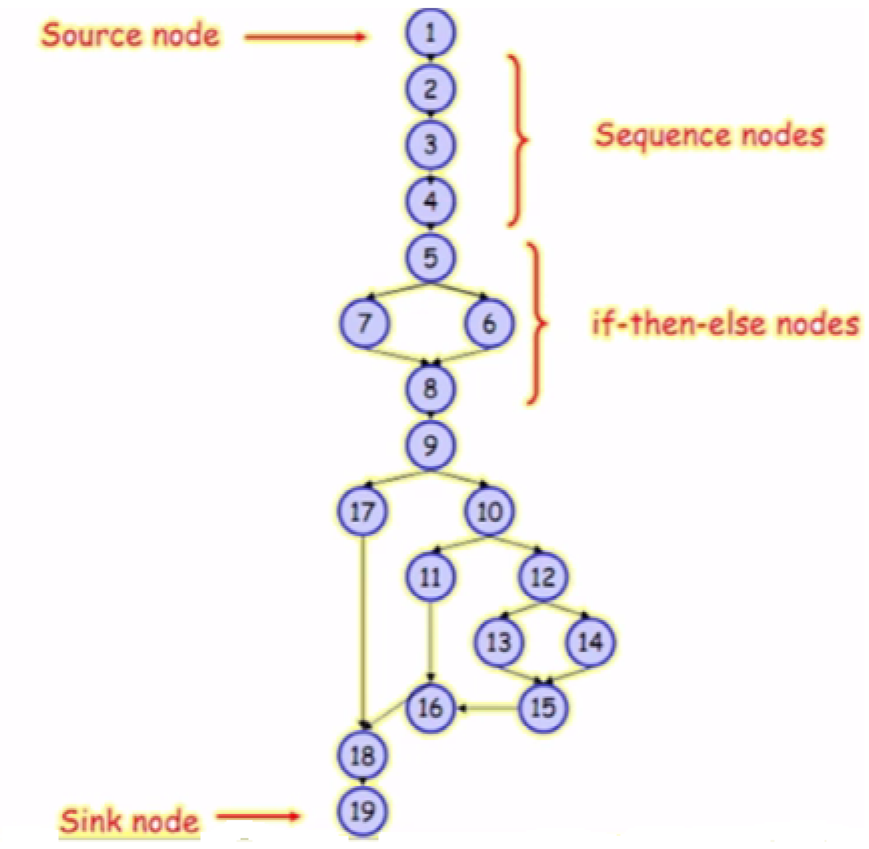
Example



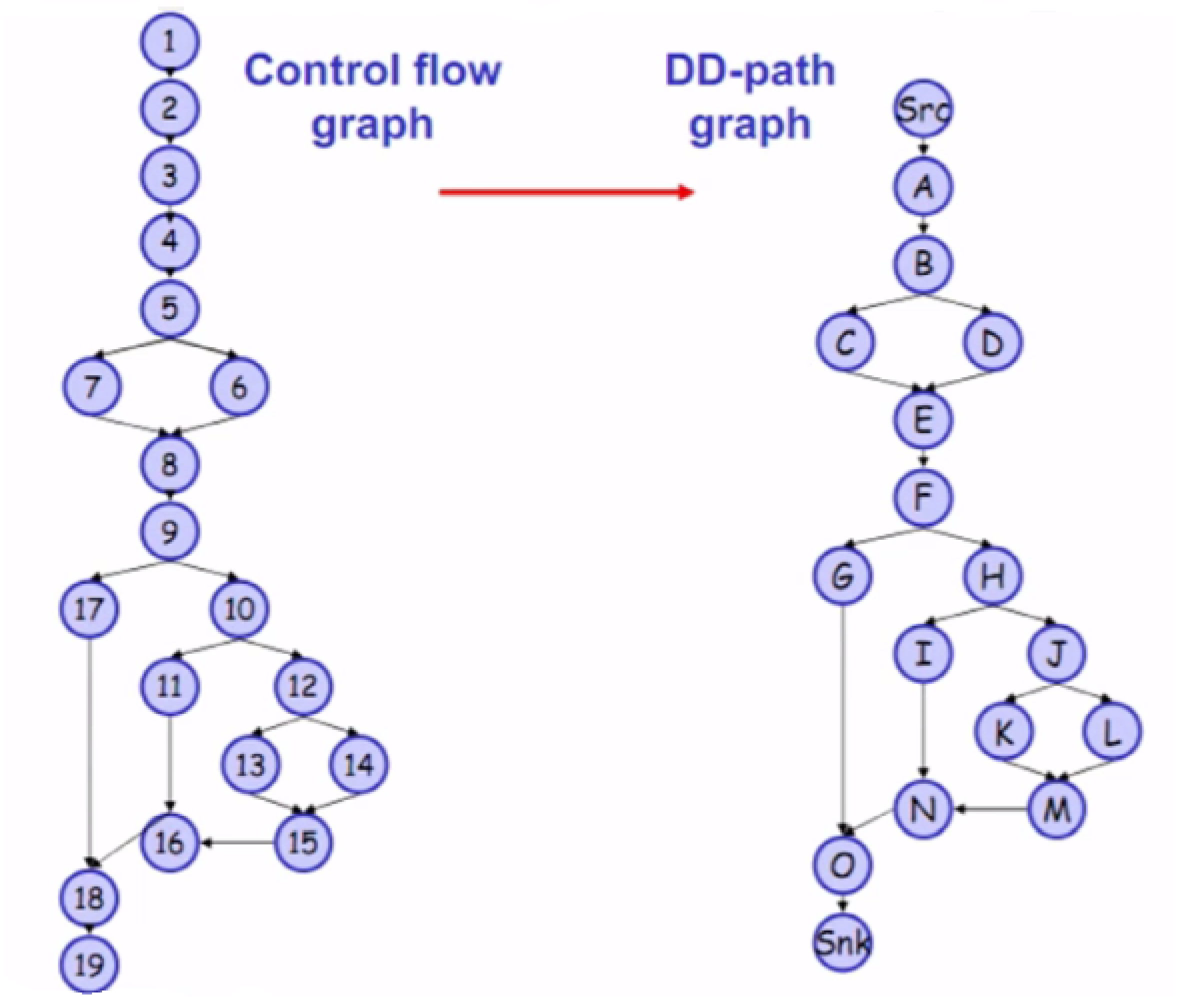
```
1   program TRIANGLE
2   input (a)
3   input (b)
4   input (c)
5   if (a<b+c) AND (b<a+c) AND (c<a+b)
6       then IsATriangle = T
7       else IsATriangle = F
8   endif
9   if IsATriangle
10      then if (a=b) AND (b=c)
11          then Output = "Equilateral"
12          else if (a != b) AND (b != c) AND (a != c)
13                then Output = "Scalene"
14                else Output = "Isosceles"
15          endif
16      endif
17  else Output = "Not a triangle"
18  endif
19  end TRIANGLE
```

The above Triangle program has 19 statements.

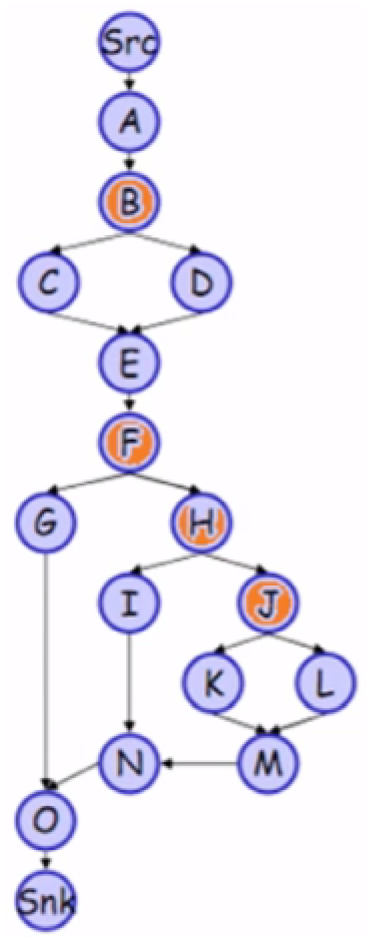


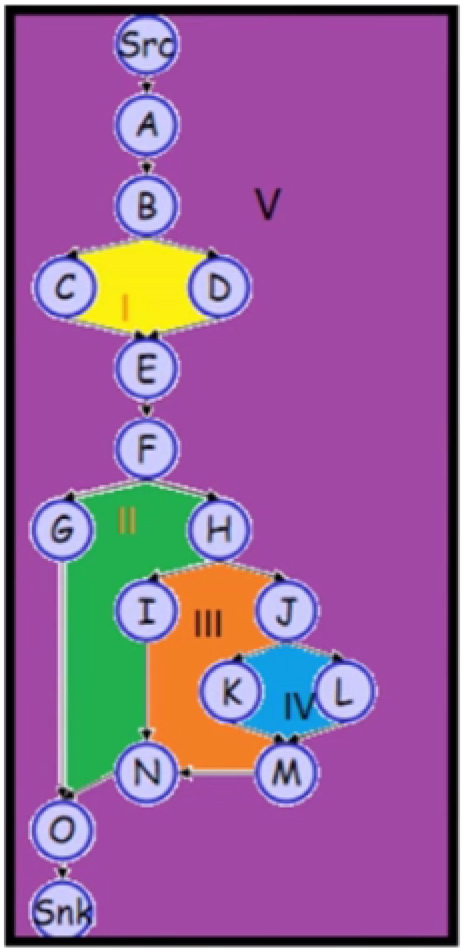


Above Diagram is the Control Flow Graph for the Triangle Program, which has 20 edges and 17 nodes (except source and sink nodes).



This above Diagram converts Control Flow Graph to Decision to Decision Path Graph.





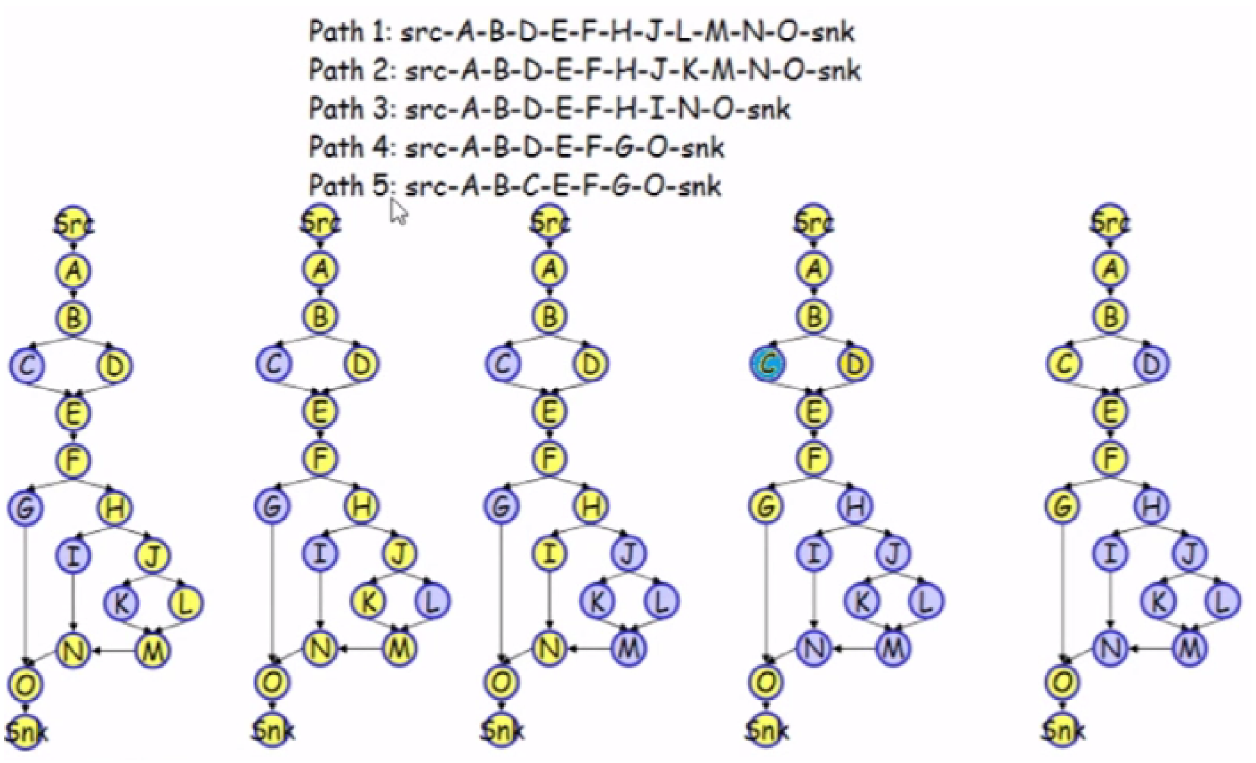
The above Diagram states that B,F,H,J are the Predicate Nodes. The next diagram shows the total Number of Regions.

Cyclomatic complexity for our Triangle Program is,

Method 1: Cyclomatic complexity V(G) = e-n+2 = 20-17+2 = 5

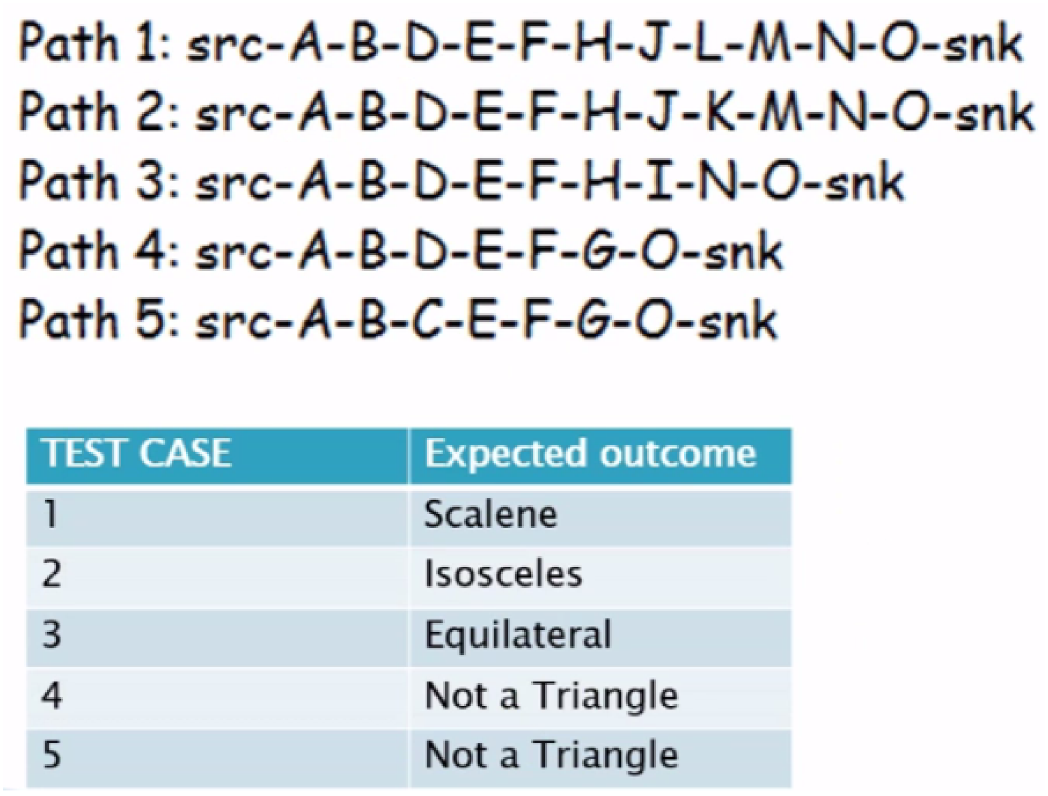
Method 2: Cyclomatic complexity V(G) = P + 1 = 4 + 1 = 5

Method 3: Cyclomatic complexity V(G) = No. of Regions = 5



Path 1: src-A-B-D-E-F-H-J-L-M-N-O-snk
Path 2: src-A-B-D-E-F-H-J-K-M-N-O-snk
Path 3: src-A-B-D-E-F-H-I-N-O-snk
Path 4: src-A-B-D-E-F-G-O-snk
Path 5: src-A-B-C-E-F-G-O-snk

The Above Diagram shows the Independent paths.


Path 1: src-A-B-D-E-F-H-J-L-M-N-O-snk
Path 2: src-A-B-D-E-F-H-J-K-M-N-O-snk
Path 3: src-A-B-D-E-F-H-I-N-O-snk
Path 4: src-A-B-D-E-F-G-O-snk
Path 5: src-A-B-C-E-F-G-O-snk

TEST CASE	Expected outcome
1	Scalene
2	Isosceles
3	Equilateral
4	Not a Triangle
5	Not a Triangle

The above Diagram shows the Test Cases for the Triangle Program.

CONTROL STRUCTURE TESTING

The basis path testing technique described in Section 17.4 is one of a number of techniques for control structure testing. Although basis path testing is simple and highly effective, it is not sufficient in itself. In this section, other variations on control structure testing are discussed. These broaden testing coverage and improve quality of white-box testing.

Condition Testing

Condition testing is a test case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator. A relational expression takes the form

E_1 <relational-operator> E_2

where E_1 and E_2 are arithmetic expressions and <relational-operator> is one of the following: $<$, $\leq$, $=$, $\neq$ (non equality), $>$, or $\geq$. A *compound condition* is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (|), AND (&) and NOT (¬). A condition without relational expressions is referred to as a *Boolean expression.* Therefore, the possible types of elements in a condition include a Boolean operator, a Boolean variable, a pair of Boolean parentheses (surrounding a simple or compound condition), a relational operator, or an arithmetic expression.

Branch testing is probably the simplest condition testing strategy. 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.

Domain testing requires three or four tests to be derived for a relational expression. For a relational expression of the form

E_1 <relational-operator> E_2

three tests are required to make the value of E_1 greater than, equal to, or less than that of E_2]. If <relational-operator> is incorrect and E_1 and E_2 are correct, then these three tests guarantee the detection of the relational operator error. To detect errors in E_1 and E_2, a test that makes the value of E_1 greater or less than that of E_2 should make the difference between these two values as small as possible.

For a Boolean expression with n variables, all of 2^n possible tests are required ($n > 0$). This strategy can detect Boolean operator, variable, and parenthesis errors, but it is practical only if n is small.

Data Flow Testing

The *data flow testing* method selects test paths of a program according to the locations of definitions and uses of variables in the program.

To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with S as its statement number,

$$\text{DEF}(S) = \{X \mid \text{statement } S \text{ contains a definition of } X\}$$
$$\text{USE}(S) = \{X \mid \text{statement } S \text{ contains a use of } X\}$$

If statement S is an *if* or *loop* statement, its DEF set is empty and its USE set is based on the condition of statement S. The definition of variable X at statement S is said to be *live* at statement S' if there exists a path from statement S to statement S' that contains no other definition of X.

A *definition-use* (DU) *chain* of variable X is of the form $[X, S, S']$, where S and S' are statement numbers, X is in DEF(S) and USE(S'), and the definition of X in statement S is live at statement S'.

One simple data flow testing strategy is to require that every DU chain be covered at least once. We refer to this strategy as the *DU testing strategy.* It has been shown that DU testing does not guarantee the coverage of all branches of a program. How-ever, a branch is not guaranteed to be covered by DU testing only in rare situations such as if-then-else constructs in which the *then part* has no definition of any variable and the *else part* does not exist. In this situation, the else branch of the *if* statement is not necessarily covered by DU testing.

Loop Testing

Loops are the cornerstone for the vast majority of all algorithms implemented in software. And yet, we often pay them little heed while conducting software tests.

Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined: simple loops, concatenated loops, nested loops, and unstructured loops.

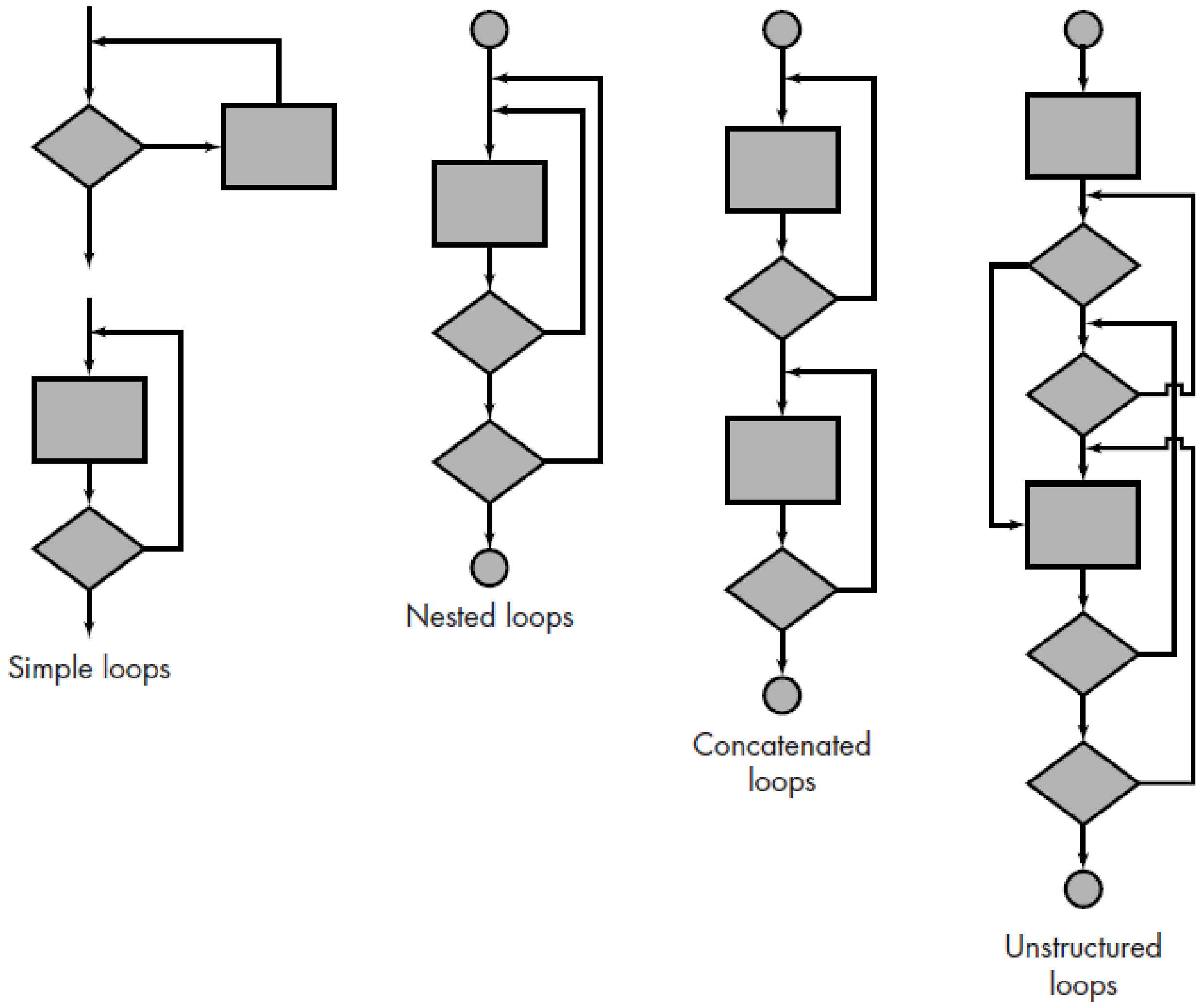


Figure. Classes of Loops

BLACK BOX TESTING

Black-box testing, also called behavioral testing, focuses on the functional requirements of the software. That is, black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program.

Black box testing derive test cases from functional specification of the software. Focus is on behavior of the software and efficiency of the performance. Concentrates only on input and output of the software functionality. This can be applied for both functional and non functional mode of the software testing.

BLACK BOX TESTING TECHNIQUES

1. **Equivalence Partitioning**

   It is a black box testing technique Where the set of test case applications divided into logical groups called partitions which exhibits similar behavior when processed. Each partition covers specific aspect of the application. We are not testing all conditions but one condition from each partition is tested.

   For Example, In an application, *User name allows numeric* means we are not testing all numeric values instead we are testing one or two numeric values which means it will allow all the numeric values.

   User name allows Alphabets means we are not testing all alphabets instead we are testing one or two alphabets which means it will allow all the alphabets.

2. **Boundary Value Analysis**

   To assist the input at boundary of the each equivalence partition uses boundary value analysis black box testing technique. Behavior of the test input is incorrect when the partition change from one to another, which leads to find most defects of the application.

   For Example, *Input between 0 and 100* means we are testing the boundary values of 0 and 100.

   ie., (-1 and 0 and +1) and (99 and 100 and 101)

3. **Comparison Testing**

   There are some situations (e.g., aircraft avionics, automobile braking systems) in which the reliability of software is absolutely critical. In such applications redundant hardware and software are often used to minimize the possibility of error. When redundant software is developed, separate software engineering teams develop independent versions of an application using the same specification. In such situations, each version can be tested with the same test data to ensure that all provide identical output. Then all versions are executed in parallel with real-time comparison of results to ensure consistency.

   To manage redundant systems, independent versions of software be developed for critical applications, even when only a single version will be used in the delivered computer-based system. These independent versions form the basis of a black-box testing technique called comparison testing or back-to-back testing.

SOFTWARE TESTING STRATEGIES

The strategy provides a road map that describes the steps to be conducted as part of testing, when these steps are planned and then undertaken, and how much effort, time, and resources will be required.

Therefore, any testing strategy must incorporate test planning, test case design, test execution, and resultant data collection and evaluation.

Verification and Validation Testing

Verification refers to the set of activities that ensure that software correctly implements a specific function.

Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements.

A Software Testing Strategy

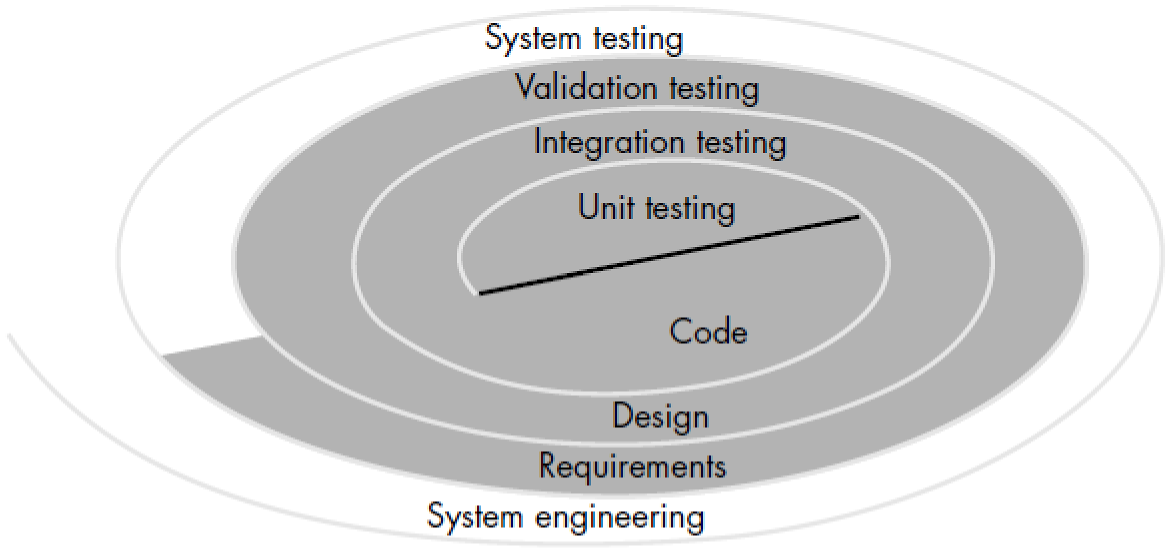


Figure. Software Testing Strategy

Initially, system engineering defines the role of software and leads to software

requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established. Moving inward along the spiral, we come to design and finally to coding. To develop computer software, we spiral inward along streamlines that decrease the level of abstraction on each turn.

A strategy for software testing may also be viewed in the context of the spiral (Refer above Figure). *Unit testing* begins at the vortex of the spiral and concentrates on each unit (i.e., component) of the software as implemented in source code. Testing progresses by moving outward along the spiral to *integration testing,* where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, we encounter *validation testing,* where requirements established as part of software requirements analysis are validated against the software that has been constructed.

Finally, we arrive at *system testing,* where the software and other system elements are tested as a whole. To test computer software, we spiral out along streamlines that broaden the scope of testing with each turn.

I) UNIT TESTING

Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The relative complexity of tests and uncovered errors is limited by the constrained scope established for unit testing. ***The unit test is white-box oriented, and the step can be conducted in parallel for multiple components.***

The tests that occur as part of unit tests are illustrated schematically in the below Figure. *The module **interface** is tested to ensure that information properly flows into and out of the program unit under test. The **local data structure** is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.*

***Boundary conditions** are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. All **independent paths** (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once. And finally, all **error handling paths** are tested.*

A component is not a stand-alone program, driver and/or stub software must be developed for each unit test. The unit test environment is illustrated in the below Figure.

In most applications *a **driver** is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results. **Stubs** serve to replace modules that are subordinate (called by) the component to be tested.* **A stub or "dummy subprogram"** uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

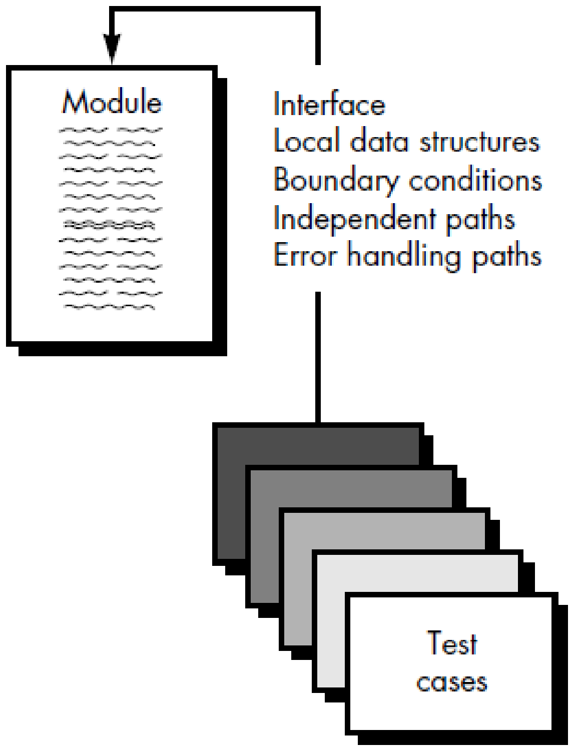


Figure. Unit Testing

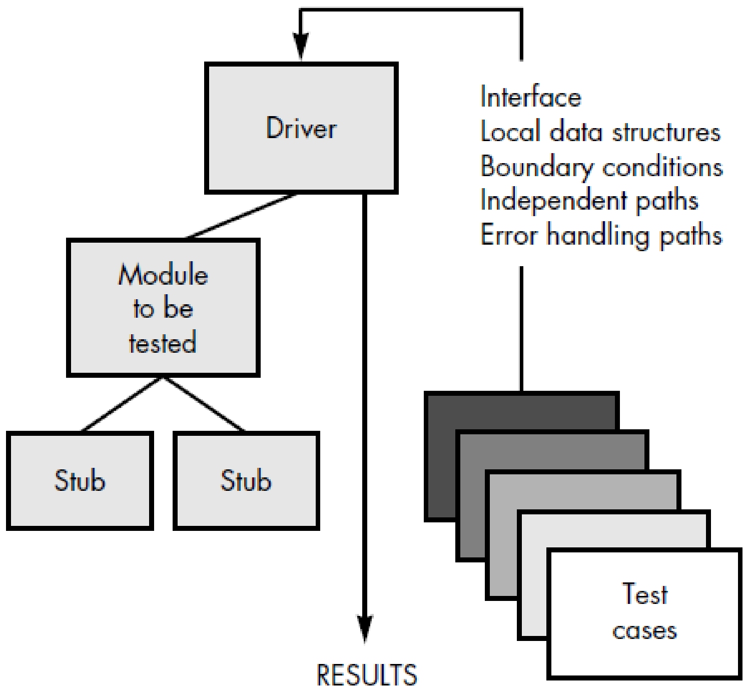


Figure. Unit Test Environment

II) INTEGRATION TESTING

The problem of putting units together causes ***interfacing***. Data can be lost across an interface; one module can have an inadvertent, adverse affect on another; sub-functions, when combined, may not produce the desired major function;

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

2.1) TOP-DOWN INTEGRATION TESTING

Top-down integration testing is an **incremental approach** to construction of *program structure*. **Modules are integrated by moving downward through the**

control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a ***depth-first or breadth-first*** manner.

Referring to Figure below, ***depth-first integration*** **would integrate all components on a major control path of the structure**. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics. For example, selecting the left-hand path, components M_1, M_2 , M_5 would be integrated first. Next, M_8 or (if necessary for proper functioning of M2) M6 would be integrated. Then, the central and right hand control paths are built. ***Breadth-first integration*** **incorporates all components directly subordinate at each level, moving across the structure horizontally**. From the below figure, components M2, M3, and M4 (a replacement for stub S4) would be integrated first. The next control level, M5, M6, and so on, follows.

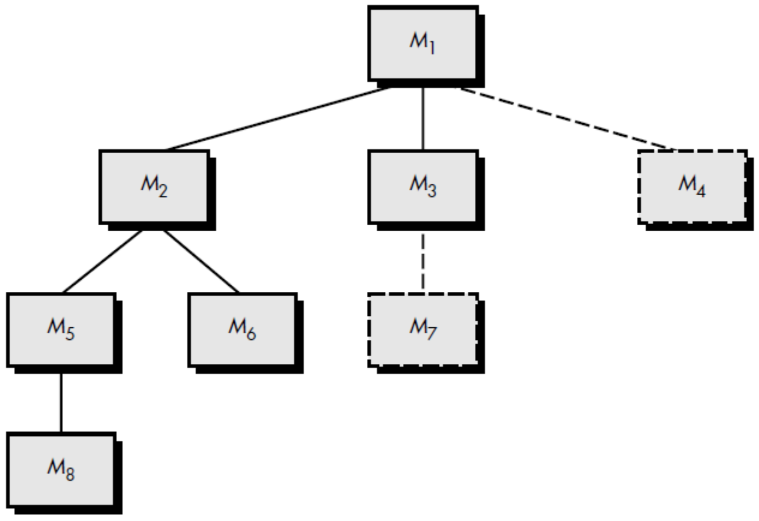


Figure. Top down integration testing

The integration process is performed in a series of five steps:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
3. Tests are conducted as each component is integrated.

4. On completion of each set of tests, another stub is replaced with the real component.
5. Regression testing may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built.

2.2) BOTTOM-UP INTEGRATION TESTING

Bottom-up integration testing, as its name implies, begins construction and testing with *atomic modules* (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.

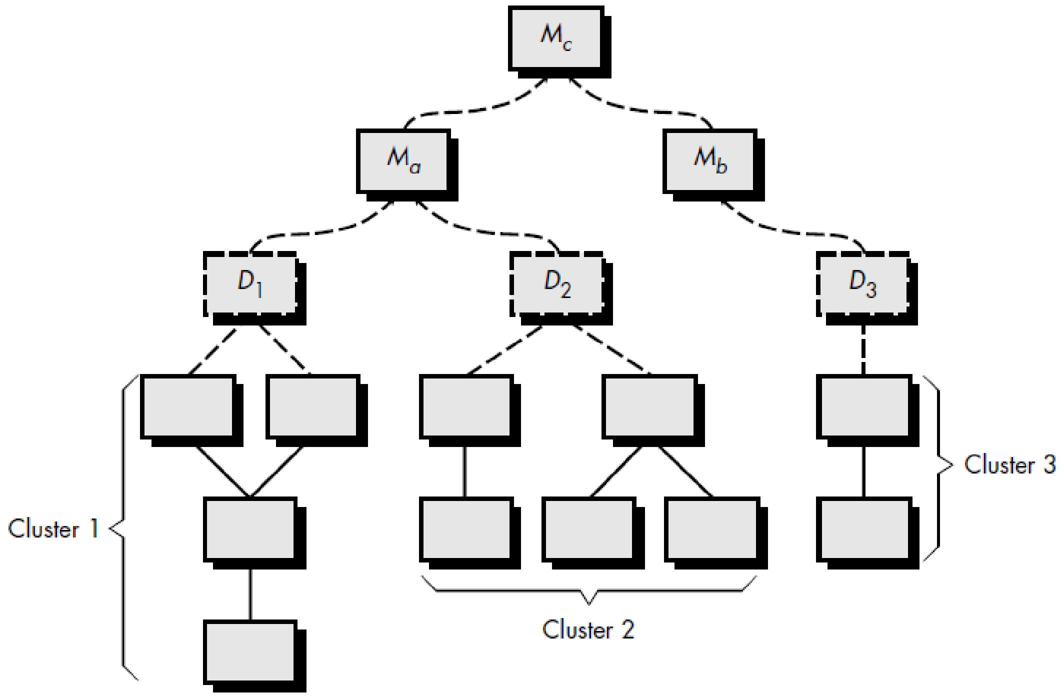


Figure. Bottom up integration testing

A bottom-up integration strategy may be implemented with the following steps:

1)Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software sub-function.

2)A driver (a control program for testing) is written to coordinate test case input and output.
3)The cluster is tested.
4)Drivers are removed and clusters are combined moving upward in the program structure.

Integration follows the pattern illustrated in the above Figure. Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to M_a. Drivers D_1 and D_2 are removed and the clusters are interfaced directly to M_a. Similarly, driver D_3 for cluster 3 is removed prior to integration with module M_b. Both M_a and M_b will ultimately be integrated with component M_c, and so forth.

Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly.

2.3) REGRESSION TESTING

In the context of an integration test strategy, ***regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.***

Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated *capture/playback tools.* Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison.

The regression test suite (the subset of tests to be executed) 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 software components that have been changed.

2.4) SMOKE TESTING

Smoke testing is an integration testing approach that is commonly used when "shrink wrapped" software products are being developed. It is designed as a pacing mechanism **for time-critical projects, allowing the software team to assess its project on a frequent basis**.

The smoke testing approach encompasses the following activities:

1) Software components that have been translated into code are integrated into a "build."

2) A series of tests is designed to expose errors that will keep the build from properly performing its function.

3) The build is integrated with other builds and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

III) VALIDATION TESTING

Validation can be defined in many ways, but a simple definition is that validation succeeds when software functions in a manner that can be **reasonably expected** by the customer.

Reasonable expectations are defined in the *Software Requirements Specification*—a document that describes all user-visible attributes of the software. The specification contains a section called *Validation Criteria.*

Information contained in that section forms the basis for a validation testing approach.

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

Alpha and Beta Testing

3.1) Acceptance Testing

When custom software is built for one customer, a series of *acceptance tests* are conducted to enable the customer to validate all requirements. Conducted by the end user rather than software engineers, an acceptance test can range from an informal "test drive" to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.

3.2) Alpha Testing

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

3.3) Beta Testing

The *beta test* is conducted at one or more customer sites by the end-user of the

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

IV) SYSTEM TESTING

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

Types of system tests :

4.1) 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.

4.2) Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration.

4.3) Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.

A variation of stress testing is a technique called *sensitivity testing.*

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

4.5) Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process.

Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion.

Testing Strategies for Object Oriented Software

I) Unit Testing in the Object Oriented Context.
With Object Oriented the concept of unit changes.

Because of the dependence of sub classes, unit testing is bit more complicated.

In the Object Oriented Context, unit testing is basically Class Testing.

II) Integration Testing in the Object Oriented Context.
Object Oriented programming does not have an hierarchical control structure.

Integrating one object at a time is also difficult because of the direct and indirect interactions of the components that make up the class.

There are two basic strategies:

1) Thread based
   - Integrates the set of classes required to respond to one input or event for the system
   - Each thread is integrated and tested individually
   - Regression testing is applied to ensure no side effects occur.

2) Use based
   - Testing begins the construction of system by testing those classes that use very few server classes.
   - After independent classes, the next layer of classes, called dependent classes are tested.
   - This continues until the entire system is constructed.

3) Drivers and Stubs

   Drivers can be used to test operations at the lowest level and for testing group of classes.

   A Driver may be used to replace the user interface.

   Stubs may be used where collaboration between classes is required but one or more of the collaborating classes is not fully implemented.

4) Cluster Testing

   It is one step in integration testing of Object Oriented software.

   Here, a cluster of collaborating classes are exercised by designing test cases that attempt to uncover errors in the collaborations.

The Art of Debugging

Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error.

In general, **three categories in approach** for debugging are : (1) brute force, (2) backtracking, and (3) cause elimination.

We apply ***brute force*** debugging methods when all else fails. Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with WRITE statements.
CLUE for Errors are Find.

Backtracking, Beginning at the site where a symptom has been uncovered,

the source code is traced backward (manually) until the site of the cause is found.

cause elimination forms a *cause hypothesis*. Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each.

UNIT - V

SOFTWARE CONFIGURATION MANAGEMENT

SOFTWARE CONFIGURATION

The output of a software process is information that may be divided in to three broad categories:

1) Computer programs (both source level and executable)

2) Work products that describe the computer programs (targeted at both technical and end users)
3) Data (contained within the program or external to it)

The items that comprise all the information produced as part of software process are collectively called as software configuration.

CONFIGURATIONMANAGEMENT

Definition:

The set of activities that have been developed to manage change throughout the software lifecycle.

Purpose:

Systematically control changes to the configuration and maintain the integrity and traceability of the configuration throughout the system's lifecycle.

Software Configuration Management (SCM):

It is an umbrella activity that is applied throughout the software process.

Because change can occur anytime, SCM activities are developed to:

1) Identify change
2) Control change
3) Ensure that change is being properly implemented
4) Report changed to others who have interest.

ORIGIN OF THESE CHANGES:

New business or market conditions bring changes in product requirements.

New customer needs demand modification of functionality delivered by the products or services delivered by the computer based system.

Reorganization or business growth causes changes in project priorities or software engineering team structure.

Budgetary constraints cause a change in the definition of the system.

WHY CHANGE HAPPENS?

Change is a fact of life in software development.

Customers want to modify requirements.

Developers want to modify the technical approach.

Managers want to modify the project strategy

Why all these modifications???

As time passes while building the software, all the people involved in it come to know more about what they need, which approach will be the best, how to get it done within time constraints, etc.

This additional knowledge is the driving force behind most changes in software development.

SOFTWARECONFIGURATIONITEMS (SCI S)

Definition: Information that is created as part of the software engineering process.

Examples:

Software Project Plan

Software Requirements Specification

Models, Prototypes, Requirements

Design document

Protocols, Hierarchy Graphs

Source code

Modules

Test suite

Software tools(e.g., compilers)

BASELINES

A Baseline is a software configuration management concept that helps us to control change.

Specification or product that has been formally reviewed and agreed upon, Serves as the basis for further development, and can be changed only through formal change control procedures.

Signals a point of departure from one activity to the start of another activity.

Helps control change without impeding justifiable change.

PROJECT BASELINE

Central repository of reviewed and approved artifacts that represent a given stable point in overall system development.

Shared Database for project and kept in consistent state.

Policies allow the team to achieve consistent state and manage the project.

BASELINE PROCESS

1. A series of software engineering tasks produces an SCI

2. The SCI is reviewed and possibly approved.

3. The approved SCI is given a new version number and placed in a project

   database (i.e., software repository)

4. A copy of the SCI is taken from the project data base and examined/modified

   by a software engineer

5. The base line of the modified SCI goes back to Step 2

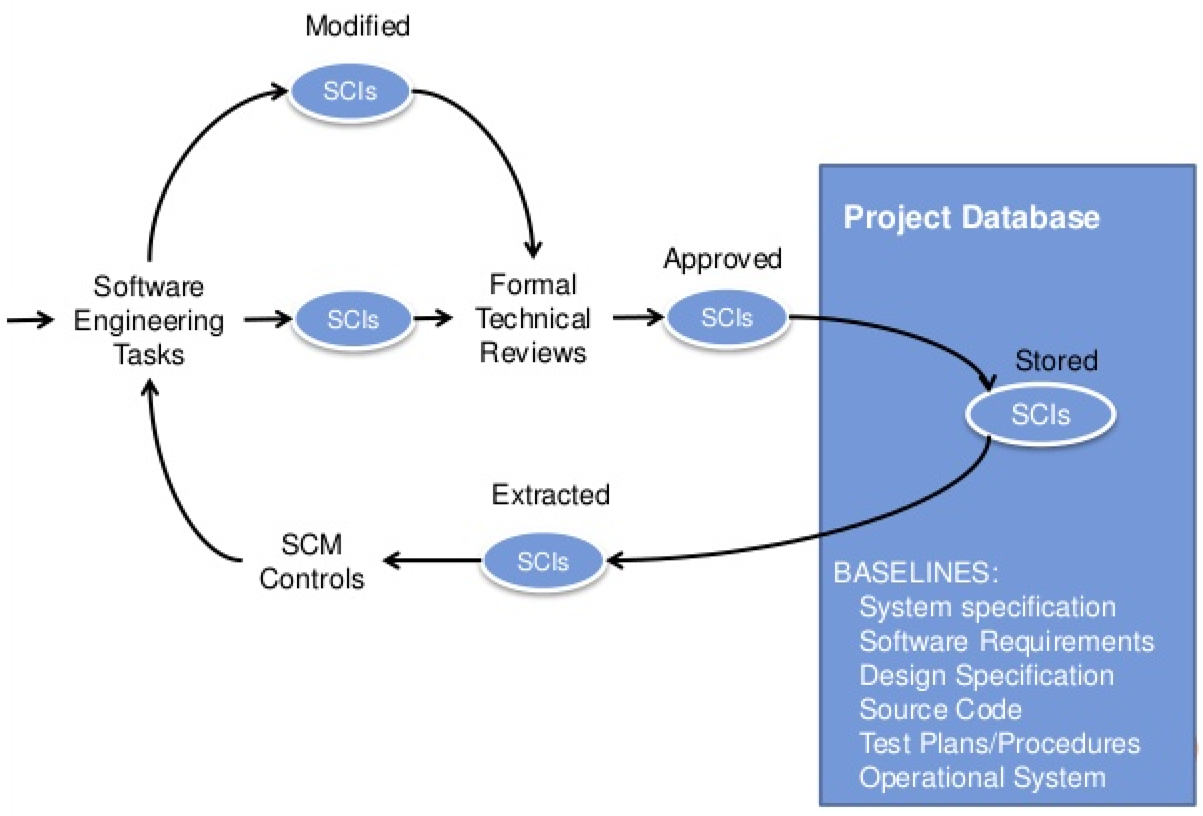


Figure. The Baseline Process

ELEMENTS OF SCM

The Four Elements of SCM are,

Software configuration Identification

Software configuration Control

Software configuration Auditing

Software configuration Status Reporting

THE SCM PROCESS

The SCM process defines a series of tasks:

Identification of objects in the software configuration Version Control

Change Control

Configuration Audit, and Reporting

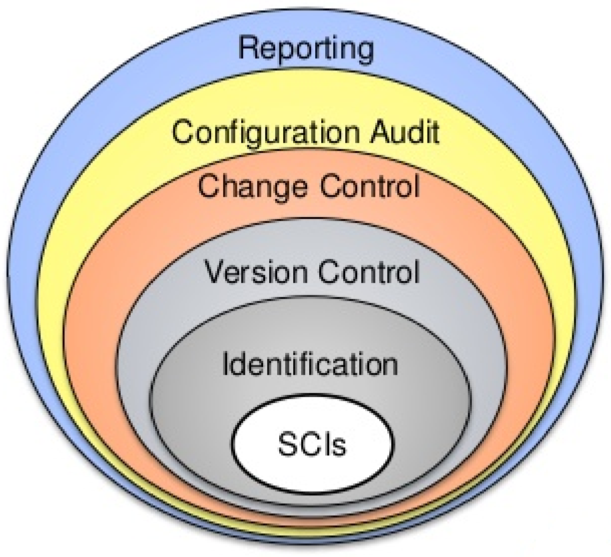


Figure. The SCM Process

SCM PROCESS:

IDENTIFICATION

Provides labels for the base lines and their updates.

Evolution graph : depicts versions/variants.

An object may be represented by variant, versions, and components.

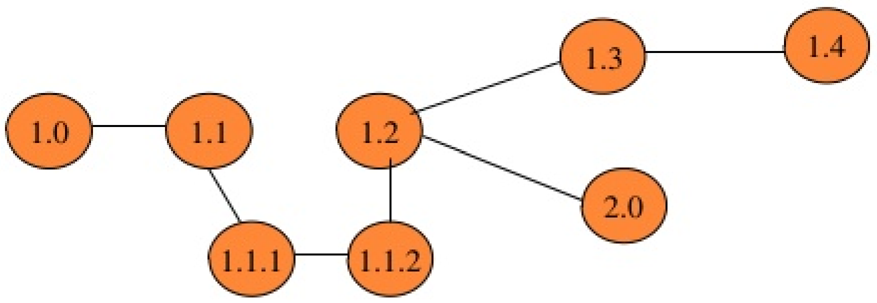


Figure. Evolution Graph - Object versions

Two types of objects can be identified as,

- Basic objects, and
- Aggregate objects

A basic object is a unit of information created by a software engineer during analysis, design, code, or test.

For example, a basic object might be a section of requirement specification, part of design model, source code for a component, etc.

An aggregate object is a collection of basic objects and other aggregate objects.

Each object has a set of distinct features that identify it:

A **name** that is unambiguous to all other objects

A **description** that contains the CSCI type, a project identifier ,and change and/or version information

List of resources needed by the object

The **object realization** (i.e., the document, the file, the model, etc.)

SOFTWARE CONFIGURATION CONTROL

Three basic things to SCC

• Documentation for formally precipitating and defining a proposed change to a software system.

- An organizational body (Configuration Control Board) for formally evaluating and approving or disapproving a proposed change to a software system.
- Procedures for controlling changes to a software system.

Why is this needed?

Not all possible changes are beneficial.

Need a mechanism control to access different items (who can access what).

CHANGE CONTROL - ACCESS CONTROL AND SYNCHRONIZATION CONTROL

The "check-in" and "check-out" process implements two important elements of change control—access control and synchronization control. Access control governs which software engineers have the authority to access and modify a particular configuration object. Synchronization control helps to ensure that parallel changes, performed by two different people, don't overwrite one another.

Access and synchronization control flow are illustrated schematically in Figure below. Based on an approved change request and ECO, a software engineer checks out a configuration object. An access control function ensures that the software engineer has authority to check out the object, and synchronization control locks the object in the project database so that no updates can be made to it until the currently checked out version has been replaced. Note that other copies can be checked-out, but other updates cannot be made. A copy of the base lined object, called the extracted version, is modified by the software engineer. After appropriate SQA and testing, the modified version of the object is checked in and the new baseline object is unlocked.

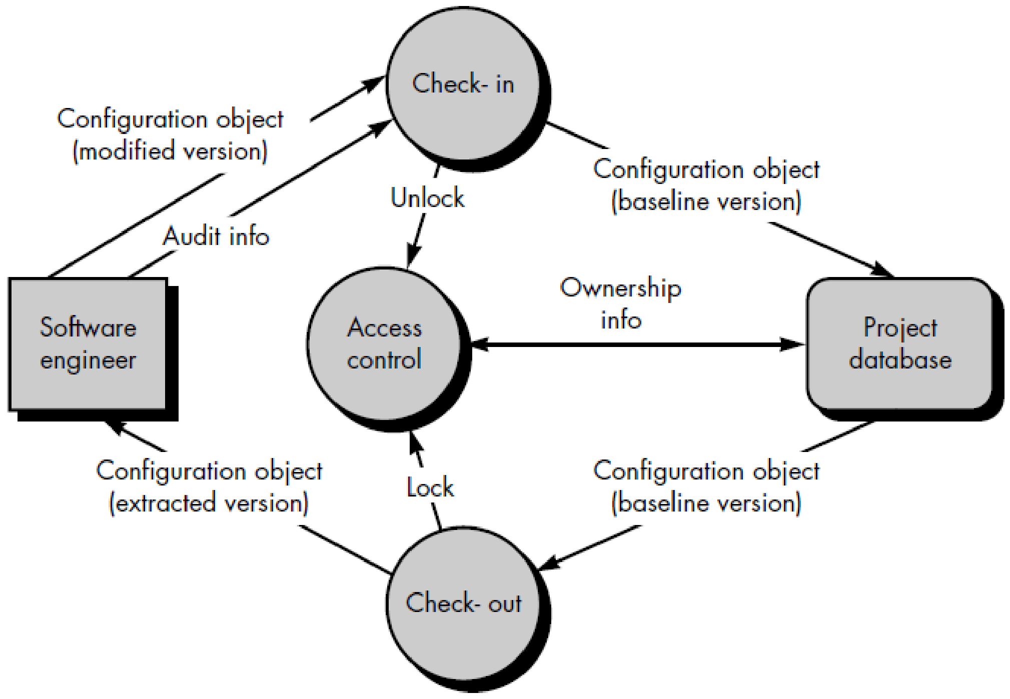

Check- in
Configuration object (modified version)
Audit info
Unlock
Configuration object (baseline version)
Software engineer
Access control
Ownership info
Project database
Configuration object (extracted version)
Lock
Configuration object (baseline version)
Check- out

Figure. Access and Synchronization control

Configuration Management Cycle

Configuration Manager – is the in charge of administrating project database and providing access control to engineers.

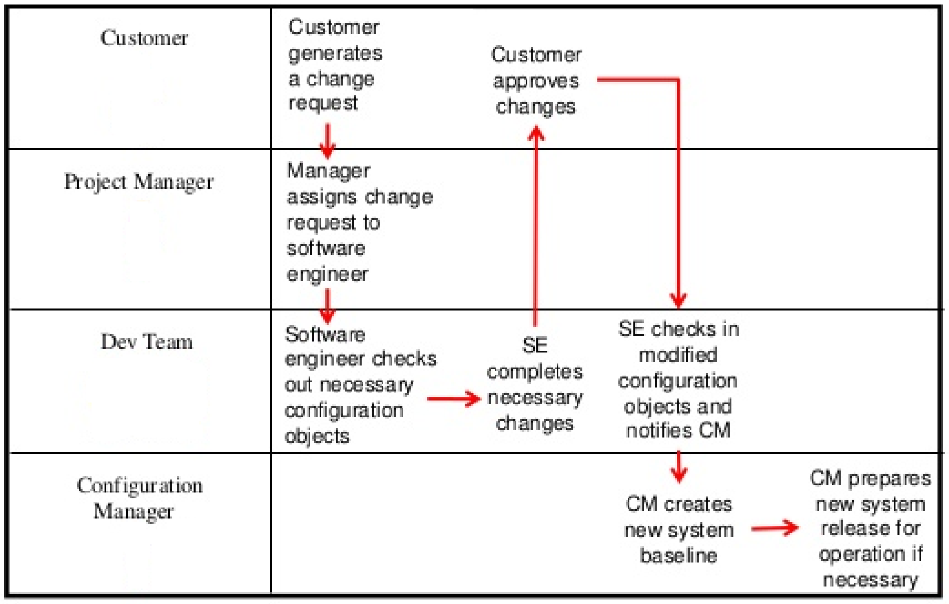

Customer
Customer generates a change request
Customer approves changes
Project Manager
Manager assigns change request to software engineer
Dev Team
Software engineer checks out necessary configuration objects
SE completes necessary changes
SE checks in modified configuration objects and notifies CM
Configuration Manager
CM creates new system baseline
CM prepares new system release for operation if necessary

Figure. Configuration Management Cycle

SOFTWARE CONFIGURATION AUDITING

Provides mechanism for determining the degree to which the current configuration of the software system mirrors the software system pictured in the baseline and the requirements documentation.

Ask the following questions:

- Has the specified change been made?
- Has a formal technical review been conducted to assess technical correctness?
- Has the software process been followed and standards been applied?
- Have the SCM procedures for noting the change, recording it, and reporting it been followed?
- Have all related SCIs been properly updated?

SOFTWARE CONFIGURATION STATUS REPORTING

Provides a mechanism for maintaining a record of where the system is at any point with respect to what appears in published baseline documentation.

When a change proposal is approved it may take some time before the change is initiated or completed.

VERSION CONTROL

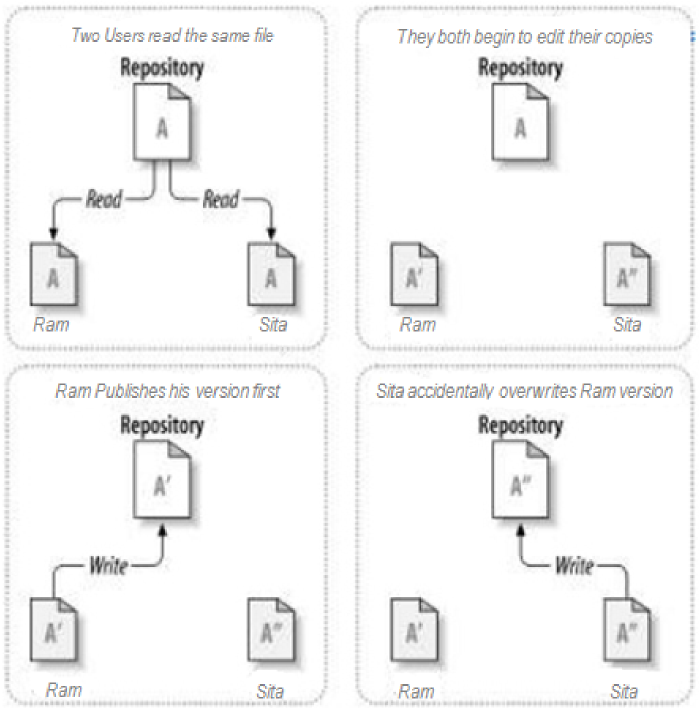
The core mission of a version control system is to enable collaborative editing and sharing of data.

File sharing is the most common problem faced by all version control systems, so most of the systems use Version Control with Subversion.

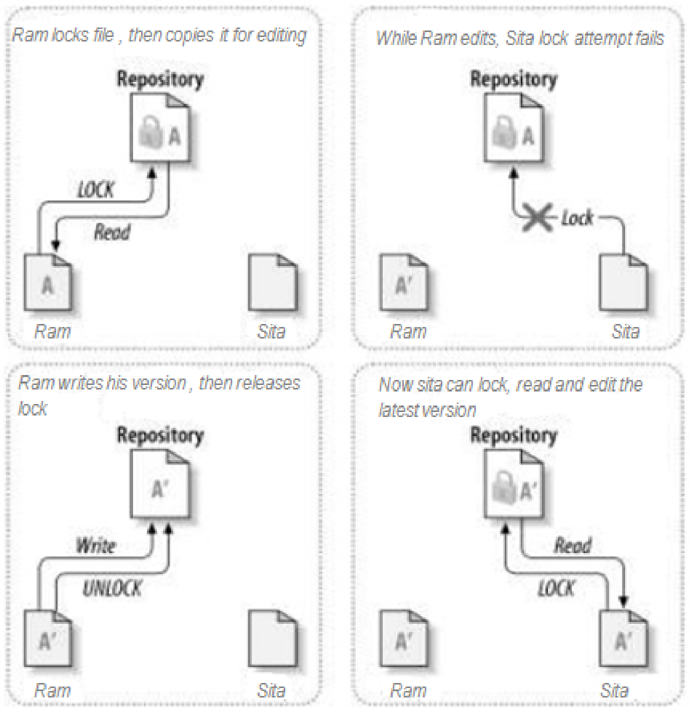
For this the solutions can be:

- Lock-Modify-Unlock Solution
- Copy-Modify-Merge Solution

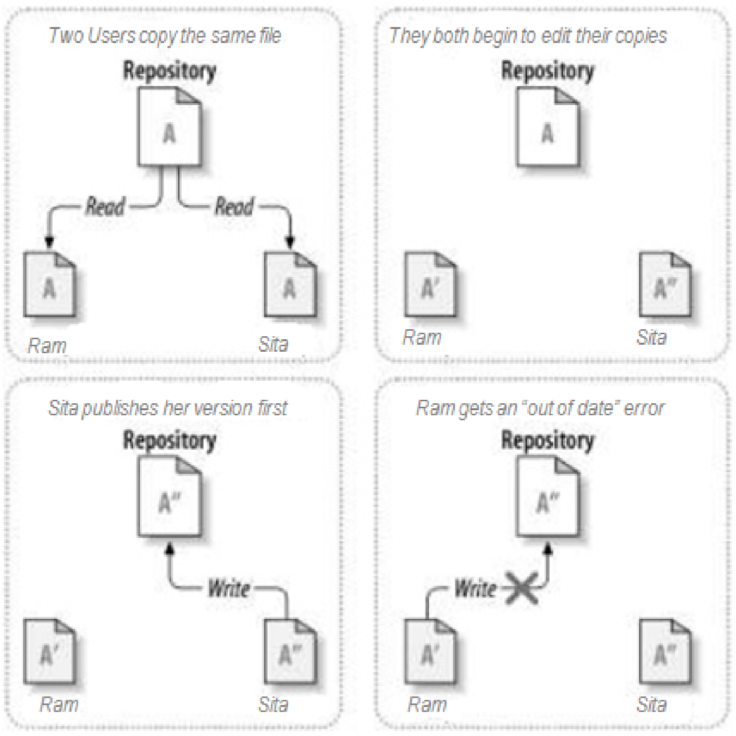
The Problem of File Sharing


Two Users read the same file
Repository
A
Read
Read
A
A
Ram
Sita
They both begin to edit their copies
Repository
A
A′
A″
Ram
Sita
Ram Publishes his version first
Repository
A′
Write
A′
A″
Ram
Sita
Sita accidentally overwrites Ram version
Repository
A″
Write
A′
A″
Ram
Sita

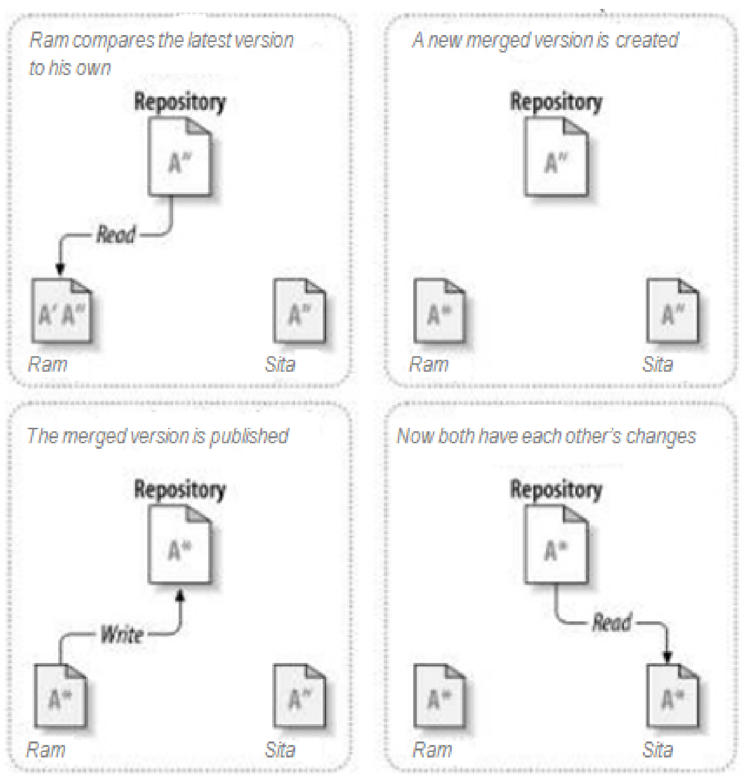
Lock – Modify – Unlock Solution


Ram locks file , then copies it for editing
Repository
A
LOCK
Read
A
Ram
Sita
While Ram edits, Sita lock attempt fails
Repository
A
Lock
A'
Ram
Sita
Ram writes his version , then releases lock
Repository
A'
Write
UNLOCK
A'
Ram
Sita
Now sita can lock, read and edit the latest version
Repository
A'
Read
LOCK
A'
A'
Ram
Sita

Copy – Modify – Merge Solution


Two Users copy the same file
Repository
A
Read
Read
A
A
Ram
Sita
They both begin to edit their copies
Repository
A
A′
A″
Ram
Sita
Sita publishes her version first
Repository
A″
Write
A′
A″
Ram
Sita
Ram gets an "out of date" error
Repository
A″
Write
A′
A″
Ram
Sita

Copy – Modify – Merge Solution


Ram compares the latest version to his own
Repository
A″
Read
A′ A″
A″
Ram
Sita
A new merged version is created
Repository
A″
A*
A″
Ram
Sita
The merged version is published
Repository
A*
Write
A*
A″
Ram
Sita
Now both have each other's changes
Repository
A*
Read
A*
A*
Ram
Sita

SOFTWARE QUALITY ASSURANCE (SQA)

SOFTWARE QUALITY ASSURANCE

1. Software requirements are the foundation from which quality is measured.

2. Specified standards define a set of development criteria that guide the manner in which software is engineered.

3. A set of implicit requirements often goes unmentioned (e.g., the desire for ease of use and good maintainability).

The SQA group serves as the customer's in-house representative. That is, the people who perform SQA must look at the software from the customer's point of view.

SQA Activities

SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting.

1) Prepares an SQA plan for a project.
   The plan identifies
   • evaluations to be performed
   • audits and reviews to be performed
   • standards that are applicable to the project
   • procedures for error reporting and tracking
   • documents to be produced by the SQA group
   • amount of feedback provided to the software project team

2) SQA group Participates in the development of the project's software process description.
3) Reviews software engineering activities to verify compliance with the defined software process.
4) Audits designated software work products to verify compliance with those defined as part of the software process.
5) Ensures that deviations in software work and work products are documented and handled according to a documented procedure.
6) Records any noncompliance and reports to senior management.

Software reviews:

Software reviews are a "filter" for the software engineering process. That is, reviews are applied at various points during software development and **serve to uncover errors and defects that can then be removed**. Software reviews "purify" the software engineering activities that we have called *analysis, design,* and *coding.*

A review—any review—is a way of using the diversity of a group of people to:
1. Point out needed improvements in the product of a single person or team;
2. Confirm those parts of a product in which improvement is either not desired or not needed;
3. Achieve technical work of more uniform, or at least more predictable, quality than can be achieved without reviews, in order to make technical work more manageable.

Defect Amplification and Removal
A defect amplification model can be used to illustrate the generation and

detection of errors during the preliminary design, detail design, and coding steps of the software engineering process.

Formal Technical Review (FTR) :
A formal technical review is a software quality assurance activity performed by software engineers (and others).
The objectives of the FTR are,
(1) to uncover errors in function, logic, or implementation for any representation of the software;
(2) to verify that the software under review meets its requirements;
(3) to ensure that the software has been represented according to predefined standards;
(4) to achieve software that is developed in a uniform manner; and
(5) to make projects more manageable.

In addition, the FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation.

Review Meeting:
Regardless of the FTR format that is chosen, every review meeting should abide by the following constraints:
• Between three and five people (typically) should be involved in the review.
• Advance preparation should occur but should require no more than two hours of work for each person.
• The duration of the review meeting should be less than two hours.

The individual who has developed the work product—the *producer*—informs the project leader that the work product is complete and that a review is required.

The project leader contacts a *review leader,* who evaluates the product for readiness, generates copies of product materials, and distributes them to two or three reviewers for advance preparation.

Each reviewer is expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the work.

Concurrently, the review leader also reviews the product and establishes an agenda for the review meeting, which is typically scheduled for the next day.

The review meeting is attended by the review leader, all reviewers, and the producer.

One of the reviewers takes on the role of the *recorder;* that is, the individual who records (in writing) all important issues raised during the review.

The FTR begins with an introduction of the agenda and a brief introduction by the producer. The producer then proceeds to "walk through" the work product, explaining the material, while reviewers raise issues based on their advance preparation. When valid problems or errors are discovered, the recorder notes each.

At the end of the review, all attendees of the FTR must decide whether to (1) accept the product without further modification, (2) reject the product due to severe errors (once corrected, another review must be performed), or (3) accept the product provisionally (minor errors have been encountered and must be corrected, but no additional review will be required). The decision made, all FTR attendees complete a sign-off, indicating their participation in the review and their concurrence with the review team's findings.

Review Reporting and Record Keeping
During the FTR, a reviewer (the recorder) actively records all issues that have been raised. These are summarized at the end of the review meeting and a review issues list is produced. In addition, a formal technical review summary report is completed.

A *review summary report* answers three questions:
1. What was reviewed?
2. Who reviewed it?
3. What were the findings and conclusions?

The *review issues list* serves two purposes: (1) to identify problem areas within the product and (2) to serve as an action item checklist that guides the producer as corrections are made. An issues list is normally attached to the summary report.

Review Guidelines
1) Review the product, not the producer.
2) Set an agenda and maintain it.
3) the issues should be recorded for further discussion
4) Express problem areas, but don't attempt to solve every problem noted.
5) Take written notes.
6) Limit the number of participants and insist upon advance preparation.
7) Develop a checklist for each product that is likely to be reviewed.
8) Allocate resources and schedule time for FTRs.
9) Conduct meaningful training for all reviewers.

10) Review your early reviews.

Statistical quality assurance:

Statistical quality assurance reflects a growing trend throughout industry to become more quantitative about quality. For software, statistical quality assurance implies the following steps:
1. Information about software defects is collected and categorized.
2. An attempt is made to trace each defect to its underlying cause (e.g., nonconformance to specifications, design error, violation of standards, poor communication with the customer).
3. Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the "vital few").
4. Once the vital few causes have been identified, move to correct the problems that have caused the defects.

The number of serious errors, number of moderate errors, number of minor errors and size of the product (LOC, design statements, pages of documentation) taken for consideration in the data collection of statistical SQA.

The application of the statistical SQA and the **Pareto principle** can be summarized in a single sentence: *Spend your time focusing on things that really matter, but first be sure that you understand what really matters!*

Software Reliability :
Software reliability, unlike many other quality factors, can be measured directed and estimated using historical and developmental data.
Software reliability is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time"

THE ISO 9000 QUALITY STANDARDS

A *quality assurance system* may be defined as the organizational structure, responsibilities, procedures, processes, and resources for implementing quality management. It covers product's entire life cycle including planning, controlling, measuring, testing and reporting, and improving quality levels throughout the development and manufacturing process.

The ISO 9000 quality assurance models treat an enterprise as a network of interconnected processes. For a quality system to be ISO compliant, these processes must address the areas identified in the standard and must be documented and practiced as described.

The ISO 9001 Standard

ISO 9001 is the quality assurance standard that applies to software engineering. The standard contains 20 requirements that must be present for an effective quality assurance system. Because the ISO 9001 standard is applicable to all engineering disciplines, a special set of ISO guidelines (ISO 9000-3) have been developed to help interpret the standard for use in the software process.

SQA PLAN

The *SQA Plan* provides a road map for instituting software quality assurance. Developed by the SQA group, the plan serves as a template for SQA activities that are instituted for each software project.

The documentation section describes each of the work products produced as part of the software process. These include

• project documents (e.g., project plan)
• models (e.g., ERDs, class hierarchies)
• technical documents (e.g., specifications, test plans)
• user documents (e.g., help files)

In addition, this section defines the minimum set of work products that are acceptable to achieve high quality.

The reviews and audits section of the plan identifies the reviews and audits

The test section references the *Software Test Plan and Procedure*. It also defines test record-keeping requirements.

The remainder of the *SQA Plan* identifies the tools and methods that support SQA activities and tasks and references software configuration management procedures for controlling change.