Maintenance

What is Software Maintenance?

Software Maintenance is a very broad activity that includes error corrections, enhancements of capabilities, deletion of obsolete capabilities, and optimization.

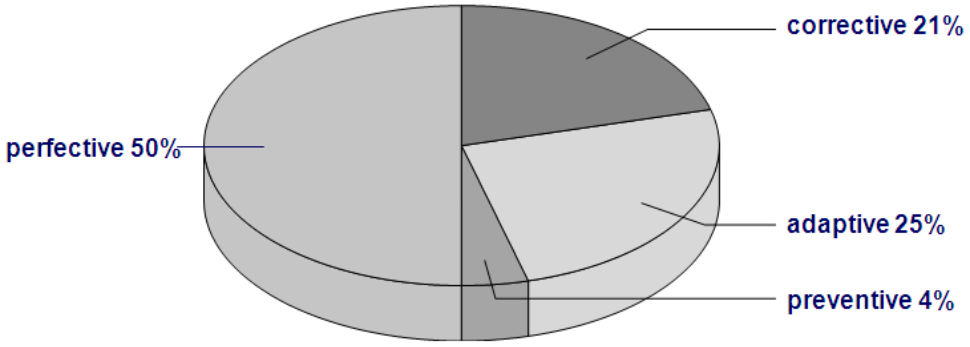
Need of Maintenance

- **Maintenance to repair software faults**
  - Changing a system to correct deficiencies in the way meets its requirements
- **Maintenance to adapt software to a different operating environment**
  - Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation
- **Maintenance to add to or modify the system's functionality**
  - Modifying the system to satisfy new requirements

Types of Maintenance

1. ***corrective maintenance*:** correcting errors (fixing errors)
2. ***adaptive maintenance*:** adapting to changes in the environment (both hardware and software) (accommodating changes in the environment or user needs)
3. ***perfective maintenance*:** adapting to changing user requirements (reengineering the application to improve performance or make the software product easier to maintain)
4. ***preventive maintenance*:** increasing the system's maintainability (modifying software to avoid anticipated future problems)
   - Higher quality $\Rightarrow$ less (corrective) maintenance
   - Anticipating changes $\Rightarrow$ less (adaptive and perfective) maintenance
   - Better tuning to user needs $\Rightarrow$ less (perfective) maintenance
   - Less code $\Rightarrow$ less maintenance

Distribution of maintenance activities



Major causes of maintenance problems

- Unstructured code
- Insufficient domain knowledge
- Insufficient documentation

Maintenance Prediction

- Maintenance prediction is concerned with assessing which parts of the system may cause problems and have high maintenance costs
  - **Change acceptance** depends on the maintainability of the components affected by the change
  - **Implementing changes** degrades the system and reduces its maintainability
  - **Maintenance costs** depend on the number of changes and costs of change depend on maintainability
- Predicting the number of changes requires and understanding of the relationships between a system and its environment
- Tightly coupled systems require changes whenever the environment is changed
- Factors influencing this relationship are
  - Number and complexity of system interfaces
  - Number of inherently volatile system requirements
  - The business processes where the system is used
- Predictions of maintainability can be made by assessing the complexity of system components
- Complexity depends on
  - Complexity of control structures
  - Complexity of data structures
  - Procedure and module size
- Process measurements may be used to assess maintainability
  - Number of requests for corrective maintenance
  - Average time required for impact analysis
  - Average time taken to implement a change request
  - Number of outstanding change requests
- If any or all of these is increasing, this may indicate a decline in maintainability

Problems during Maintenance

- Often the program is written by another person or group of persons.
- Often the program is changed by person who did not understand it clearly.
- Program listings are not structured.
- High staff turnover.
- Information gap.
- Systems are not designed for change

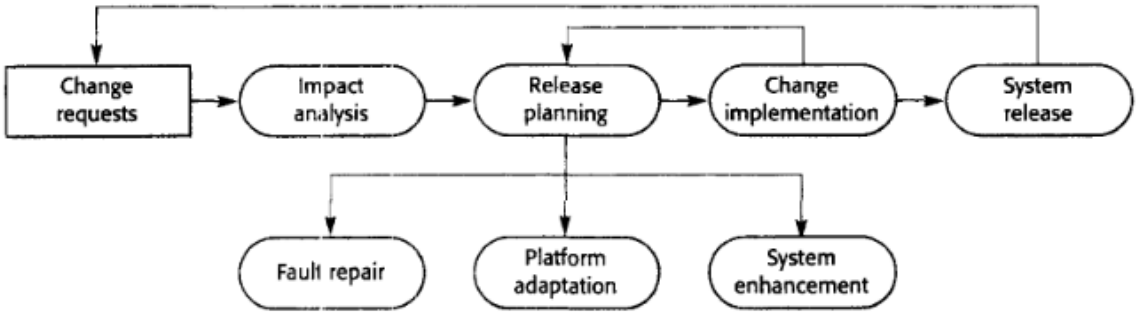
Overview of maintenance process

The process of modifying a software system or component after delivery to correct faults, improve Performance or other attributes, or adapt to a changed environment.
Maintenance is thus concerned with:

- correcting errors found after the software has been delivered
- adapting the software to changing requirements, changing environments, ...

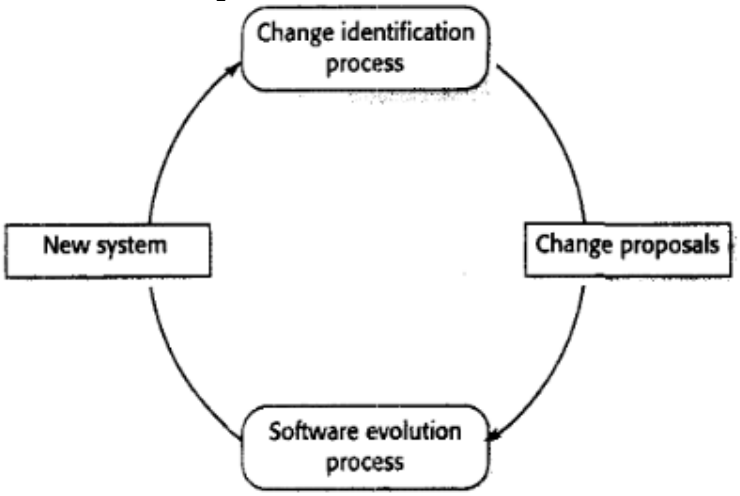
Maintenance Process

The evolution process includes the fundamental activities of change analysis, release planning, system implementation and releasing a system to customers.



The maintenance evolution process includes the fundamental activities of change analysis, release planning, system implementation and releasing a system to customers. The cost and impact of these changes are assessed to see how much of the system is affected by the change and how much it might cost to implement the change. If the proposed changes are accepted, a new release of the system is planned. During release planning, all proposed changes (fault repair, adaptation and new functionality) are considered. A decision is then made on which changes to implement in the next version of the system. The changes are implemented and validated, and a new version of the system is released. The process then iterates with a new set of changes proposed for the next release.

Change identification and evolution process



Ideally, the change implementation stage of this process should modify the system specification; design and implementation to reflect the changes to the system new requirements that reflect the system changes are proposed, analyzed and validated. System components are redesigned and implemented and the system is retested.

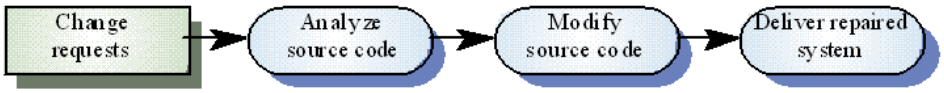
- Maintenance is triggered by change requests from customers or marketing requirements
- Changes are normally batched and implemented in a new release of the system
- Programs sometimes need to be repaired without a complete process iteration but this is dangerous as it leads to documentation and programs getting out of step

Change Process

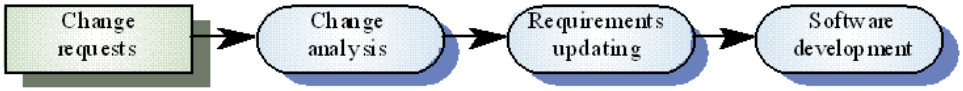
During the evolution process, the requirements are analyzed in detail and, frequently. Implications of the changes emerge that were not apparent in the earlier change analysis process. This means that the proposed changes may be modified and further customer discussions may be required before they are implemented.

Three reasons for urgent change in software

- If a serious system fault occurs
- If changes to system operating environment that affect normal operation
- Emergence of new competitors or the introduction of new rules



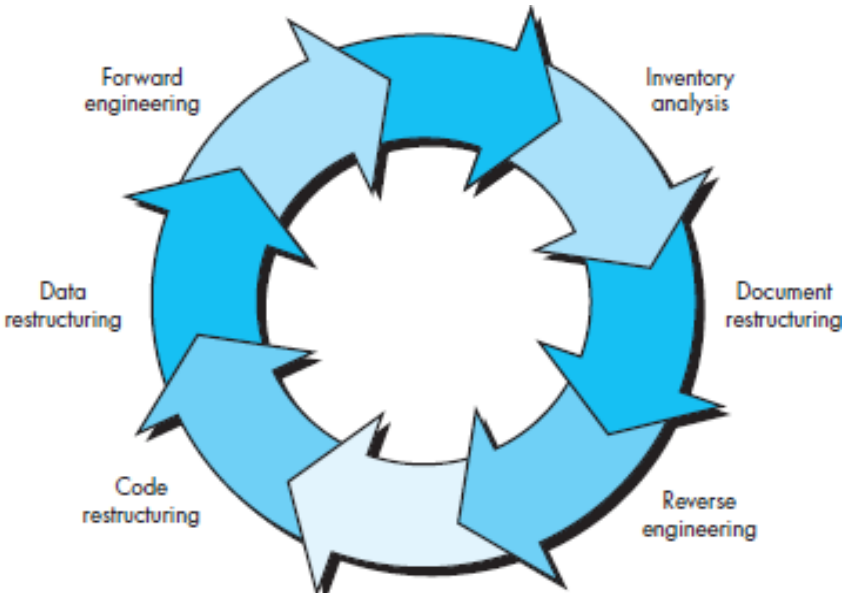
Fault repair process



Iterative development process

Software reengineering

Business process reengineering (BPR) defines business goals, evaluates existing business processes, and creates revised business processes that better meet current goals. Software reengineering involves inventory analysis, document restructuring, reverse engineering, program and data restructuring, and forward engineering.

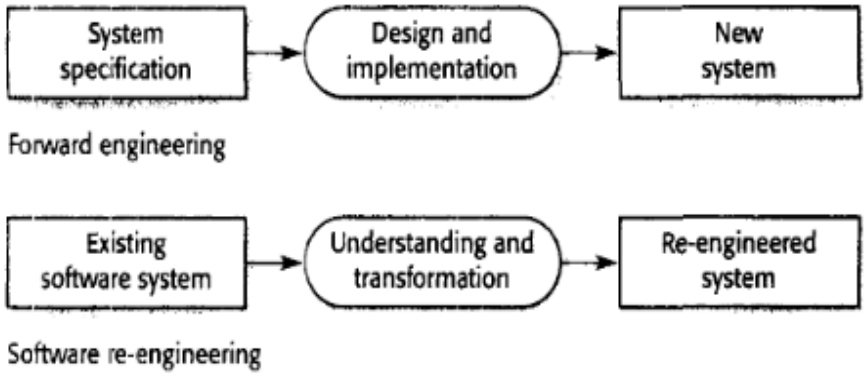


- ***Inventory analysis*** - sorting active software applications by business criticality, longevity, current maintainability, and other local criteria helps to identify reengineering candidates

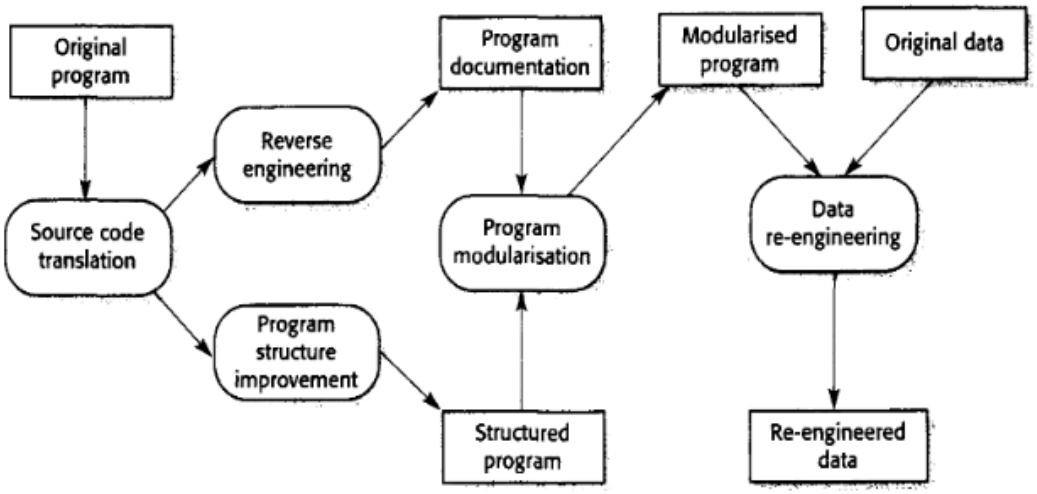
- ***Document restructuring*** - need to decide to live with weak documentation, update poor documents if they are used, or fully rewrite the documentation for critical systems focusing on the "essential minimum"
- ***Reverse engineering*** - process of design recovery - analyzing a program in an effort to create a representation of the program at some abstraction level higher than source code
- ***Code restructuring*** - source code is analyzed and violations of structured programming practices are noted and repaired, the revised code also needs to be reviewed and tested
- ***Data restructuring*** - usually requires full reverse engineering, current data architecture is dissected and data models are defined, existing data structures are reviewed for quality
- ***Forward engineering*** - also called reclamation or renovation, recovers design information from existing source code and uses this information to reconstitute the existing system to improve its overall quality and/or performance

Compare Forward and reengineering

Forward engineering starts with a system specification and involves the design and implementation of a new system. Re-engineering starts with an existing system and the development process for the replacement is based on understanding and transforming the original system.



Reverse engineering and Reengineering



1. **Source code translation** Using a translation tool, the program is converted from an old programming language to a more modern version of the same language or to a different language.
2. **Reverse engineering** The program is analyzed and information extracted from it. This helps to document its organization and functionality. Again, this process is usually completely automated.
3. **Program structure improvement** The control structure of the program is analyzed and modified to make it easier to read and understand. This can be partially automated but some manual intervention is usually required.
4. **Program modularization** Related parts of the program are grouped together and, where appropriate, redundancy is removed. In some cases, this stage may involve architectural refactoring (e.g., a system that uses several different data stores may be refactored to use a single repository). This is a manual process.
5. **Data reengineering** The data processed by the program is changed to reflect program changes. This may mean redefining database schemas and converting existing databases to the new structure.

Risk Management

What is Risk?

- A risk is a potential problem – it might happen and it might not, this is uncertainty.
- We don't know whether a particular event will occur or no but if it does has a negative impact on a project.
- A possibility of suffering from loss in software development process is called a software risk.
- Loss can be anything, increase in production cost, development of poor quality software, not being able to complete the project on time.

Types of software risks

- Software risk exists because the future is uncertain and there are many known and unknown things that cannot be incorporated in the project plan.
- A software risk can be of two types

  (1) Internal risks that are within the control of the project manager and
  (2) External risks that are beyond the control of project manager.

Definitions of Risks

- Risk is the probability of suffering loss.
- Risk provides an opportunity to develop the project better.
- Risk exposure= Size (loss)* probability of (loss)
- There is a difference between a Problem and Risk
- Problem is some event which has already occurred but risk is something that is unpredictable.

Two characteristics of risk

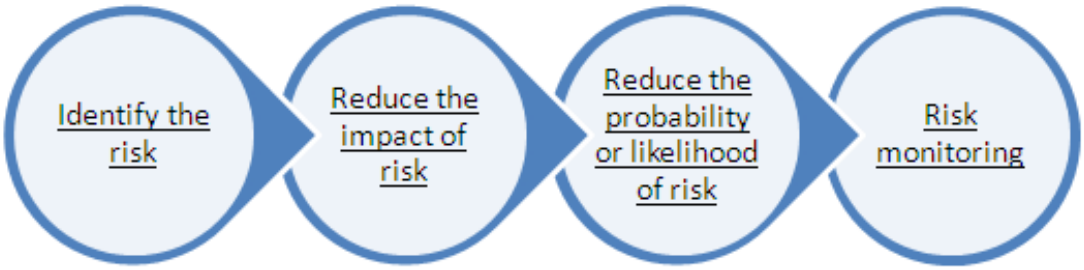
- Uncertainty – the risk may or may not happen, that is, there are no 100% risks (those, instead, are called constraints)
- Loss – the risk becomes a reality and unwanted consequences or losses occur

Risk Categorization

- **Project risks**
  - They threaten the project plan
  - If they become real, it is likely that the project schedule will slip and that costs will increase
- **Technical risks**
  - They threaten the quality and timeliness of the software to be produced
  - If they become real, implementation may become difficult or impossible
- **Business risks**
  - They threaten the viability of the software to be built
  - If they become real, they jeopardize the project or the product
  - Sub-categories of Business risks
    - Market risk – building an excellent product or system that no one really wants
    - Strategic risk – building a product that no longer fits into the overall business strategy for the company
    - Sales risk – building a product that the sales force doesn't understand how to sell
    - Management risk – losing the support of senior management due to a change in focus or a change in people
    - Budget risk – losing budgetary or personnel commitment
- **Known risks**
  - Those risks that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date)
- **Predictable risks**
  - Those risks that are extrapolated from past project experience (e.g., past turnover)
- **Unpredictable risks**
  - Those risks that can and do occur, but are extremely difficult to identify in advance

Risk Management

- Risk management is carried out to:
  - Identify the risk
  - Reduce the impact of risk
  - Reduce the probability or likelihood of risk
  - Risk monitoring

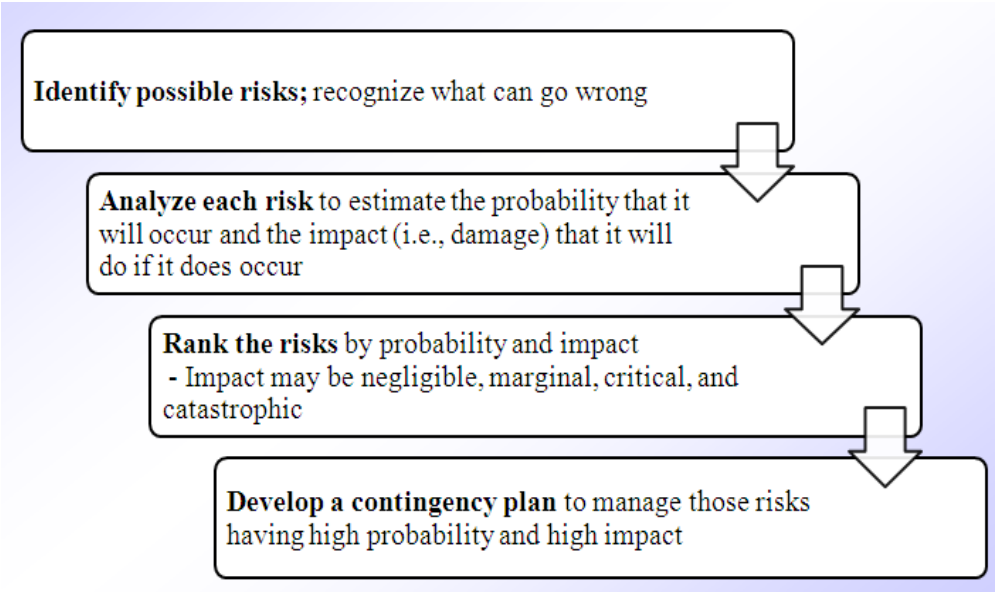


Risk Management Paradigm

Reactive vs. Proactive Risk Strategies

- Reactive risk strategies
  - "Don't worry, I'll think of something"
  - The majority of software teams and managers rely on this approach
  - Nothing is done about risks until something goes wrong
    - The team then flies into action in an attempt to correct the problem rapidly (fire fighting)
  - Crisis management is the choice of management techniques
- Proactive risk strategies
  - Steps for risk management are followed (see next slide)
  - Primary objective is to avoid risk and to have a contingency plan in place to handle unavoidable risks in a controlled and effective manner

Steps for Risk Management



Risk Identification

- Risk identification is a systematic attempt to specify threats to the project plan
- By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary
- Generic risks
  - Risks that are a potential threat to every software project
- Product-specific risks
  - Risks that can be identified only by those a with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built
  - This requires examination of the project plan and the statement of scope
  - "What special characteristics of this product may threaten our project plan?"

a) Risk Item Checklist

Used as one way to identify risks

Focuses on known and predictable risks in specific subcategories (see next slide)

Can be organized in several ways

- A list of characteristics relevant to each risk subcategory
- Questionnaire that leads to an estimate on the impact of each risk
- A list containing a set of risk component and drivers and their probability of occurrence

b) Known and Predictable Risk Categories

There are seven categories of predictable risks

1-Product size	risks associated with overall size of the software to be built
2-Business impact	risks associated with constraints imposed by management or the marketplace
3-Customer characteristics	risks associated with sophistication of the customer and the developer's ability to communicate with the customer in a timely manner
4-Process definition	risks associated with the degree to which the software process has been defined and is followed
5-Development environment	risks associated with availability and quality of the tools to be used to build the project
6-Technology to be built	risks associated with complexity of the system to be built and the "newness" of the technology in the system
7-Staff size and experience	risks associated with overall technical and project experience of the software engineers who will do the work

c) Assessing Project Risk

1) Have ***top software and customer managers*** formally committed to support the project?
2) Are ***end-users enthusiastically*** committed to the project and the system/product to be built?
3) Are requirements ***fully understood*** by the software engineering team and its customers?
4) Have customers been ***involved in the definition*** of requirements?
5) Do ***end-users have realistic*** expectations?
6) Is the **project scope stable**?
7) Does the software engineering team have the ***right mix of skills***?
8) Are project ***requirements stable***?
9) Does the project team have ***experience with the technology*** to be implemented?
10) Is the number of people on the project ***team adequate to do the job***?
11) Do all customers agree on the ***importance of the project*** and on the requirements for the product to be built?

d) Risk Components and Drivers

- The project manager identifies the <u>risk drivers</u> that affect the following risk components
  - **Performance risk** - the degree of uncertainty that the product will meet its requirements and be fit for its intended use
  - **Cost risk** - the degree of uncertainty that the project budget will be maintained
  - **Support risk** - the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance
  - **Schedule risk** - the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time

- The impact of each risk driver on the risk component is divided into one of four impact levels
  - Negligible, marginal, critical, and catastrophic
- Risk drivers can be assessed as impossible, improbable, probable, and frequent

Risk Projection (Estimation)

- Risk projection (or estimation) attempts to rate each risk in two ways
  - The probability that the risk is real
  - The consequence of the problems associated with the risk, should it occur
- The project planner, managers, and technical staff perform four risk projection steps (see next slide)
- The intent of these steps is to consider risks in a manner that leads to prioritization
- By prioritizing risks, the software team can allocate limited resources where they will have the most impact

a) Risk Projection/Estimation Steps

1) Establish a scale that reflects the perceived likelihood of a risk (e.g., 1-low, 10-high)
2) Delineate the consequences of the risk
3) Estimate the impact of the risk on the project and product
4) Note the overall accuracy of the risk projection so that there will be no misunderstandings

b) Contents of a Risk Table

- A risk table provides a project manager with a simple technique for risk projection
- It consists of five columns
  - Risk Summary – short description of the risk
  - Risk Category – one of seven risk categories (slide 12)
  - Probability – estimation of risk occurrence based on group input
  - Impact – (1) catastrophic (2) critical (3) marginal (4) negligible
  - RMMM – Pointer to a paragraph in the Risk Mitigation, Monitoring, and Management Plan

Risk Summary	**Risk Category**	**Probability**	**Impact (1-4)**	**RMMM**

c) Developing a Risk Table

- List all risks in the first column (by way of the help of the risk item checklists)
- Mark the category of each risk

- Estimate the probability of each risk occurring
- Assess the impact of each risk based on an averaging of the four risk components to determine an overall impact value (See next slide)
- Sort the rows by probability and impact in descending order
- Draw a horizontal cutoff line in the table that indicates the risks that will be given further attention

d) Assessing Risk Impact

- Three factors affect the consequences that are likely if a risk does occur
  - Its nature – This indicates the problems that are likely if the risk occurs
  - Its scope – This combines the severity of the risk (how serious was it) with its overall distribution (how much was affected)
  - Its timing – This considers when and for how long the impact will be felt
- The overall risk exposure formula is RE = P x C
  - P = the probability of occurrence for a risk
  - C = the cost to the project should the risk actually occur
- Example
  - P = 80% probability that 18 of 60 software components will have to be developed
  - C = Total cost of developing 18 components is $25,000
  - RE = .80 x $25,000 = $20,000

Risk Mitigation, Monitoring, and Management (RMMM)

- **Mitigation**—how can we avoid the risk?
- **Monitoring**—what factors can we track that will enable us to determine if the risk is becoming more or less likely?
- **Management**—what contingency plans do we have if the risk becomes a reality?

- An effective strategy for dealing with risk must consider three issues (Note: these are not mutually exclusive)
  - Risk mitigation (i.e., avoidance)
  - Risk monitoring
  - Risk management and contingency planning
- Risk mitigation (avoidance) is the primary strategy and is achieved through a plan
  - Example: Risk of high staff turnover

Strategy for Reducing Staff Turnover

- Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market)
- Mitigate those causes that are under our control before the project starts
- Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave
- Organize project teams so that information about each development activity is widely dispersed
- Define documentation standards and establish mechanisms to ensure that documents are developed in a timely manner

- Conduct peer reviews of all work (so that more than one person is "up to speed")
- Assign a backup staff member for every critical technologist
- During risk monitoring, the project manager monitors factors that may provide an indication of whether a risk is becoming more or less likely
- Risk management and contingency planning assume that mitigation efforts have failed and that the risk has become a reality
- RMMM steps incur additional project cost
    - Large projects may have identified 30 – 40 risks
- Risk is not limited to the software project itself
    - Risks can occur after the software has been delivered to the user

Software safety and hazard analysis

- These are software quality assurance activities that focus on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail
- If hazards can be identified early in the software process, software design features can be specified that will either eliminate or control potential hazards

The RMMM Plan

- The RMMM plan may be a part of the software development plan or may be a separate document
- Once RMMM has been documented and the project has begun, the risk mitigation, and monitoring steps begin
    - Risk mitigation is a problem avoidance activity
    - Risk monitoring is a project tracking activity
- Risk monitoring has three objectives
    - To assess whether predicted risks do, in fact, occur
    - To ensure that risk aversion steps defined for the risk are being properly applied
    - To collect information that can be used for future risk analysis
- The findings from risk monitoring may allow the project manager to ascertain what risks caused which problems throughout the project

Risk Monitoring

- Assess each identified risks regularly to decide whether or not it is becoming less or more probable.
- Also assess whether the effects of the risk have changed.
- Each key risk should be discussed at management progress meetings

Purpose of risk monitoring

- Risk responses have been implemented as planned.
- Risk response actions are as effective as expected or if new responses should be developed.
- Project assumptions are still valid.
- Risk exposure has changed from its prior state, with analysis of trends.
- A risk trigger has occurred.
- Proper policies and procedures are followed.
- New risks have occurred that were not previously identified.

Risk information sheet.

- In most cases, RIS is maintained using a database system.
- So Creation and information entry, priority ordering ,searches and other analysis may be accomplished easily.
- The format of RIS is describe in diagram

Risk information sheet			
Risk ID: P02-4-32	Date: 5/9/04	Prob: 80%	Impact: high
Description: Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.			
Refinement/context: Subcondition 1: Certain reusable components were developed by a third party with no knowledge of internal design standards. Subcondition 2: The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components. Subcondition 3: Certain reusable components have been implemented in a language that is not supported on the target environment.			
Mitigation/monitoring: 1. Contact third party to determine conformance with design standards. 2. Press for interface standards completion; consider component structure when deciding on interface protocol. 3. Check to determine number of components in subcondition 3 category; check to determine if language support can be acquired.			
Management/contingency plan/trigger: *RE* computed to be $20,200. Allocate this amount within project contingency cost. Develop revised schedule assuming that 18 additional components will have to be custom built; allocate staff accordingly. Trigger: Mitigation steps unproductive as of 7/1/04			
Current status: 5/12/04: Mitigation steps initiated.			
Originator: D. Gagne		Assigned: B. Laster	

Seven Principles of Risk Management

Maintain a global perspective	View software risks within the context of a system and the business problem that is intended to solve
Take a forward-looking view	Think about risks that may arise in the future; establish contingency plans
Encourage open communication	Encourage all stakeholders and users to point out risks at any time
Integrate risk management	Integrate the consideration of risk into the software process
Emphasize a continuous process of risk management	Modify identified risks as more becomes known and add new risks as better insight is achieved
Develop a shared product vision	A shared vision by all stakeholders facilitates better risk identification and assessment
Encourage teamwork when managing risk	Pool the skills and experience of all stakeholders when conducting risk management activities

Project Management concept: People – Product-Process-Project

Management techniques required to plan, organize, monitor and control software projects

Effective software project management focuses on the four P's:

- People — the most important element of a successful project
- Product — the software to be built
- Process — the set of framework activities and software engineering tasks to get the job done
- Project — all work required to make the product a reality

The People

- must be organized into effective teams
- motivated to do high-quality work
- coordinated to achieve effective communication and results

The people management maturity model defines: recruiting, selection, performance management, training, compensation, career development, organization and work design, and team/culture development.

The Product

Before a project can be planned:

- Product objectives and scope should be established
- Alternative solutions should be considered
- Technical and management constraints should be identified

Estimates of cost, effective assessment of risk, realistic breakdown of project tasks, or manageable project schedule

The Process

A software process provides the framework for which a comprehensive plan for software development can be established.

- Task sets – tasks, milestones, work products, and quality assurance points
- Umbrella activities – software quality assurance, software configuration management, and measurement

The Project

- To manage complexity
- To avoid failure
- To develop a common sense approach for planning, monitoring, and controlling the project.

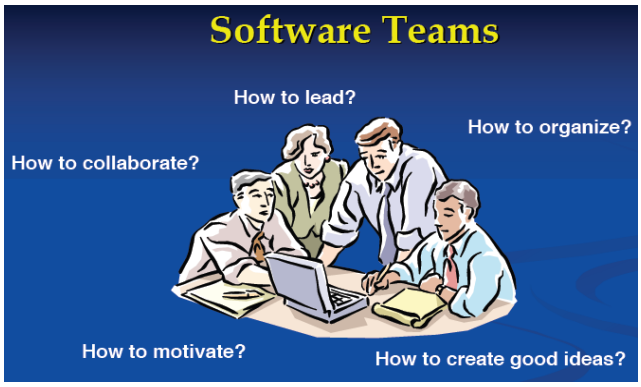
1. **People**

People build computer software, and projects succeed because well-trained, motivated people get things done.

a) The Stakeholders

- *Senior managers* who define the business issues that often have significant influence on the project.
- *Project (technical) managers* who must plan, motivate, organize, and control the practitioners who do software work.
- *Practitioners* who deliver the technical skills that are necessary to engineer a product or application.
- *Customers* who specify the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome.
- *End-users* who interact with the software once it is released for production use.

b) Team Leaders



- Project management is a people-intensive activity → need "people skill"
- MOI model for Leadership:
    - ✓ **Motivation:** The ability to encourage (by "push or pull") technical people to produce to their best ability.
    - ✓ **Organization:** The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product
    - ✓ **Ideas for innovation***:* The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

c) The Software Team

- *N individuals vs. m tasks*
- Team organizations
    - – Democratic decentralized (DD): no permanent leader, rather "task coordinator", decision made by group consensus.
    - – Controlled decentralized (CD): has defined leader, decision remains group activity, works partitioned
    - – Controlled centralized (CC): Top-level problem solving, internal coordination

Seven project factors when planning the structure of software engineering team:

- The difficulty of the problem
- The size of the resultant program
- The time
- The degree of problem to be modularized
- The required quality and reliability
- The rigidity of the delivery date
- Degree of sociability (communication)

Organizational Paradigms

- closed paradigm—structures a team along a traditional hierarchy of authority
- random paradigm—structures a team loosely and depends on individual initiative of the team members
- open paradigm—attempts to structure a team in a manner that achieves some of the controls associated with the closed paradigm but also much of the innovation that occurs when using the random paradigm
- synchronous paradigm—relies on the natural compartmentalization of a problem and organizes team members to work on pieces of the problem with little active communication among themselves

Avoid Team "Toxicity"

- High frustration caused by personal, business, or technological factors that cause friction among team members.
- Fragmented or poorly coordinated procedures" or a poorly defined or improperly chosen process model that becomes a roadblock to accomplishment.
- Unclear definition of roles resulting in a lack of accountability and resultant finger-pointing.
- "Continuous and repeated exposure to failure" that leads to a loss of confidence and a lowering of morale.

d) Agile Teams

- Team members must have trust in one another.
- The distribution of skills must be appropriate to the problem.
- Mavericks may have to be excluded from the team, if team cohesiveness is to be maintained.
- Team is "self-organizing"
    - An adaptive team structure
    - Uses elements of Constantine's random, open, and synchronous paradigms
    - Significant autonomy

e) Coordination and Communication Issues

Many reasons that software projects get into trouble:
- Scale
- Uncertainty
- Interoperability

Therefore, must establish methods for coordinating the people.

Hence, establish formal and informal communication among team members:

- Formal, impersonal approaches: SE docs and deliverables, tech memo.
- Formal, interpersonal procedures: QA activities, status review meetings and design
- Informal, interpersonal procedures: group meeting
- Electronic communication: email
- Interpersonal networking: interpersonal discussion with outsiders.

2. The Product

a) Scope

- **Context.** How does the software to be built fit into a larger system, product, or business context and what constraints are imposed as a result of the context?
- **Information objectives**. What customer-visible data objects are produced as output from the software? What data objects are required for input?
- **Function and performance**. What function does the software perform to transform input data into output? Are any special performance characteristics to be addressed?
- Software project scope must be unambiguous and understandable at the management and technical levels.

b) Problem Decomposition

- Sometimes called partitioning or problem elaboration
- Once scope is defined …
    - It is decomposed into constituent functions
    - It is decomposed into user-visible data objects

      or
    - It is decomposed into a set of problem classes
    - Decomposition process continues until all functions or problem classes have been defined

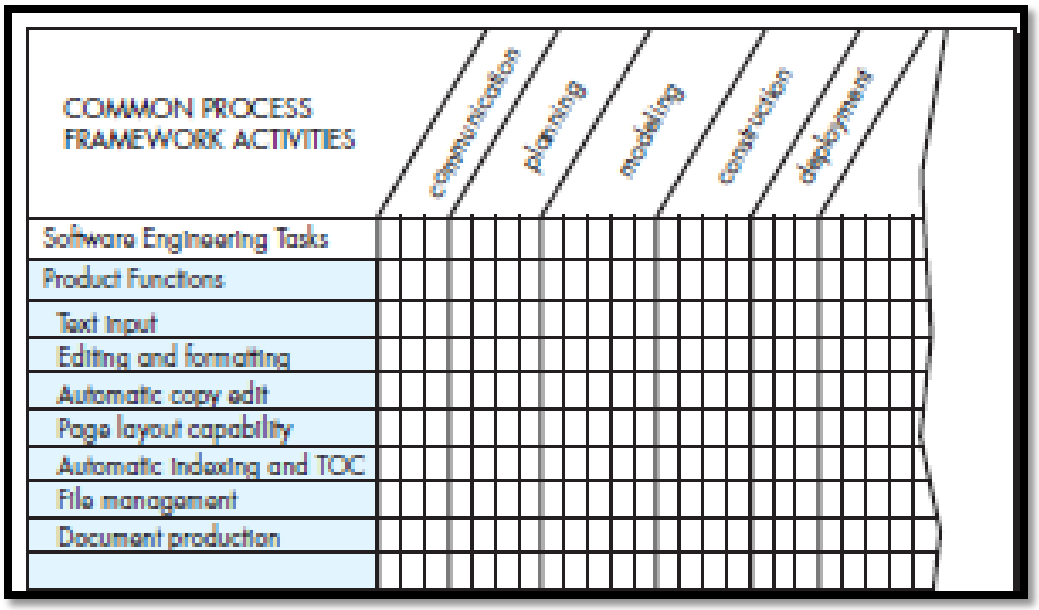
3. The Process

Once a process framework has been established,

- Consider project characteristics
- Determine the degree of rigor required
- Define a task set for each software engineering activity
    - Task set =
        - Software engineering tasks
        - Work products
        - Quality assurance points
        - Milestones

a) Melding the Problem and the Process

- The generic phases that characterize the software process – definition, development, and support – are applicable to all software.



b) Process decomposition

➢ The problem is to select the process that is appropriate for the software to be engineered by a project team.

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

Must decide which model is most appropriate for

- the customers
- The characteristics of the product
- The project environment

4. The Project

➢ Must understand what can go wrong (so that problems can be avoided)

➢ Ten signs that indicate that an information systems project is in jeopardy:

1. Software people don't understand their customer's needs
2. The product scope is poorly defined
3. Changes are managed poorly
4. The chosen technology changes
5. Business needs change (or ill-defined)
6. Deadlines are unrealistic
7. Users are resistant
8. Sponsorship is lost (or was never properly obtained)

9. The project team lacks people with appropriate skills
10. Managers (and practitioners) avoid best practices and lessons learned

Five-part commonsense approach to software project:

1. Start on the right foot: working hard to understand the problem
2. Maintain momentum: provide incentives
3. Track progress: track work products
4. Make smart decisions: decisions should be "keep it simple"
5. Conduct a postmortem analysis: lessons learned and evaluation of project

The W^5HH Principle

Barry Boehm suggests an approach that addresses project objectives, milestones and schedules, responsibilities, management and technical approaches, and requires resources:

- Why is the system being developed?
- What will be done, by when?
- When will it be done?
- Who is responsible for a function?
- Where they are organizationally located?
- How will the job be done technically and managerially?
- How much of each resource is needed?

Project scheduling and Tracking

- ***Software project scheduling*** is an activity that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks.
- During early stages of project planning, a *macroscopic schedule* is developed.
- This type of schedule identifies all major software engineering activities and the product functions to which they are applied.
- As the project gets under way, each entry on the macroscopic schedule is refined into a *detailed schedule*.

Basic Concept

Why software is delivered late?

- An unrealistic deadline established
- Changing customer requirements that are not reflected in schedule changes.
- 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.

What should we do when management demands that make a deadline that is impossible?

- ✓ Perform a detailed estimate using historical data from past projects. Determine the estimated effort and duration for the project.
- ✓ Using an incremental process model that will deliver critical functionality by the imposed deadline. Document the plan.
- ✓ Meet with the customer and (using the detailed estimate), explain why the imposed deadline is unrealistic.
- ✓ Offer the incremental development strategy as an alternative

Basic Principles

- Compartmentalization:
  - The project must be compartmentalized into a number of manageable activities and tasks.
  - To accomplish compartmentalization, both the product and the process are decomposed.
- Interdependency.
  - The interdependency of each compartmentalized activity or task must be determined.
  - Some tasks must occur in sequence while others can occur in parallel.

- Some activities cannot commence until the work product produced by another is available.

- Time allocation.
  - Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort).
  - In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies
- Effort validation.
  - As time allocation occurs, the project manager must ensure that no more than the allocated number of people has been scheduled at any given time.
- Defined responsibilities.
  - Every task that is scheduled should be assigned to a specific team member.
- Defined outcomes.
  - Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product or deliverable.
- Defined milestones.
  - A milestone is accomplished when one or more work products has been reviewed for quality and has been approved.

Relationship between People and Effort- Putnam Norden Rayleigh (PNR) Curve

- Common management myth: *If we fall behind schedule, we can always add more programmers and catch up later in the project*
  - This practice actually has a disruptive effect and causes the schedule to slip even further
  - The added people must learn the system
  - The people who teach them are the same people who were earlier doing the work
  - During teaching, no work is being accomplished
  - Lines of communication increases for each new person added

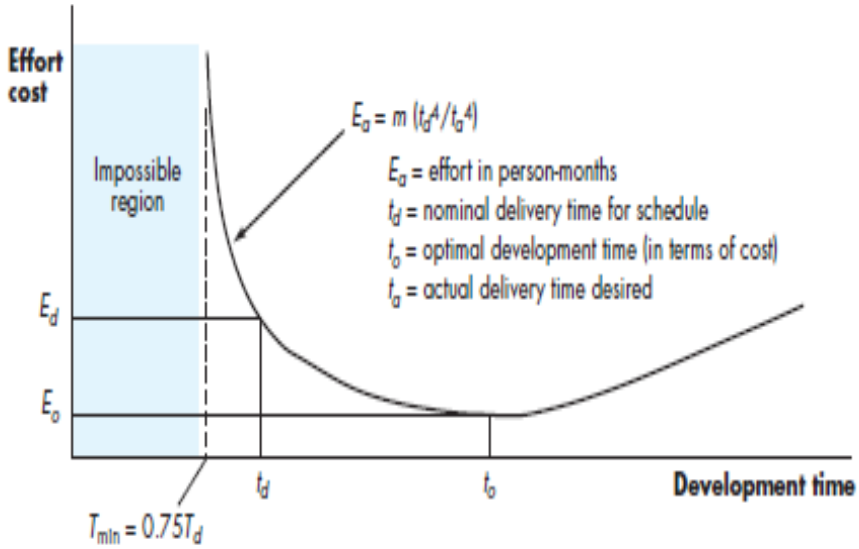
Software engineering handles the relationship between people and effort management for product development phase. These some points are as follows:-

1. When software size is small single person can handle same project by performing steps like requirement analysis, designing, code generation, and testing etc.
2. If the project is large additional people are required to complete. the project in stipulated in time it become easy to complete project by distributing work among people and get it done as early as possible.
3. The communication path of new comer also increase as time increase and day by day the project become extra complicated. And new customer gets confusion become more after the days by days.
4. It is possible to reduce a desire project completion date by getting more people to same point. It also possible to expand the completion date by reducing number of resources. And you can mention for the date of completion.
5. The **Putnam Norden Rayleigh (PNR)** curve is an indication of relationship which exists between effort applied and delivery time for software project.

6. The curve indicate a minimum time value at to which indicates test cost time for delivery as use move to left to right. It is observed that curved raised non-linearly.
7. It is possible to make delivery fast; the curve rises very sharply to left of td. The PNR curve indicates that project delivery time should not be compressed much behind on td.
8. The number of delivery lines of code are also known as source statements L. Relationship of L with effort & development time by equation can be described as

L= P * E1/3 T ¾

Here 'E' represents development effort in person months, P is productivity



After rearranging the last equation can arrive at an expansion for development effort e.

E = L3/(P3 T4)

E is called as effort expanded over entire life cycle for software development and maintenance T is the development period in years. And this equation is lead to

E = L3 / (P3 T4) ~ 3.8 Person years.

This shows that by extending last date of project with six month e.g. we can reduce the no of people from eight to four. The outcome benefit can be gained by using less number of people over longer time to achieve the same objective.

Project Effort Distribution

- The 40-20-40 rule:
  - 40% front-end analysis and design
  - 20% coding
  - 40% back-end testing
- Generally accepted guidelines are:
  - 02-03 % planning
  - 10-25 % requirements analysis
  - 20-25 % design
  - 15-20 % coding
  - 30-40 % testing and debugging

Defining Task Set for the Software Project

- A task set is the work breakdown structure for the project
- No single task set is appropriate for all projects and process models
  - It varies depending on the project type and the degree of rigor
- The task set should provide enough discipline to achieve high software quality
  - But it must not burden the project team with unnecessary work
- To define a Task set
  1) Determine Project Type
  2) Identify adaptation criteria
  3) Assess the degree of rigor required
  4) Select appropriate software engineering tasks.

1) Determine the Project Type

- ☐ **Concept development projects** that are initiated to explore some new business concept or application of some new technology.
- ☐ **New application development projects** that are undertaken as a consequence of a specific customer request.
- ☐ **Application enhancement projects** that occur when existing software undergoes major modifications to function, performance, or interfaces that are observable by the end-user.
- ☐ **Application maintenance projects** that correct, adapt, or extend existing software in ways that may not be immediately obvious to the end-user.
- ☐ **Reengineering projects** that are undertaken with the intent of rebuilding an existing (legacy) system in whole or in part.

2) Identify adaptation criteria

Each of the adaptation criteria is assigned a grade that ranges between 1 and 5, where 1 represents a project in which a small subset of process tasks are required and 5 represents a project in which a complete set of process tasks should be applied.

- Size of project
- Number of potential users.
- Mission criticality
- Application longevity
- Stability of requirements
- Ease of customer/developer communication
- Maturity of applicable technology
- Performance constraints,
- Project staff
- Reengineering.

3) ***Degree of rigor***

- ☐ *Adaptation criteria* are used to determine the recommended degree of rigor with which the software process should be applied on a project.
- ☐ The *degree of rigor* is a function of many project characteristics. As an example, small, non-business-critical projects can generally be addressed with somewhat less rigor than large, complex business-critical applications.
- ☐ Finally, apply software engineering task.

4) Selecting Software Engineering Tasks

- In order to develop a project schedule, a task set must be distributed on the project time line.
- Major software engineering tasks are applicable to all process model flows.
- As an example, we consider the software engineering tasks for a concept development project.

a) Major Task Set for concept development project are:

- **Concept scoping** determines the overall scope of the project.
- **Preliminary concept planning** establishes the organization's ability to undertake the work implied by the project scope.
- **Technology risk assessment** evaluates the risk associated with the technology to be implemented as part of project scope.
- **Proof of concept** demonstrates the feasibility of a new technology in the software context.
- **Concept implementation** implements the concept representation in a manner that can be reviewed by a customer and is used for "marketing" purposes when a concept must be sold to other customers or management.
- **Customer reaction** to the concept asks for feedback on a new technology concept and targets specific customer applications.

  - ✓ The software team must understand what must be done **(scoping);**
  - ✓ Then the team (or manager) must determine whether anyone is available to do it **(planning),**
  - ✓ Consider the risks associated with the work **(risk assessment).**
  - ✓ Prove the technology in some way **(proof of concept)**
  - ✓ Implement it in a prototypical manner so that the customer can evaluate it **(concept implementation and customer evaluation).**
  - ✓ Finally, if the concept is viable, a production version **(translation)** must be produced.

b) Refinement of Major Task

- ■ Refinement begins by taking each major task and decomposing it into a set of subtasks (with related work products and milestones)
- ■ As an example of task decomposition, consider *concept scoping* for a development Project
- ■ Task refinement can be accomplished using an outline format a process design language approach is used to illustrate the flow of the concept scoping activity

Task definition: Task 1.1 Concept Scoping

1.1.1 Identify need, benefits and potential customers;

1.1.2 Define desired output/control and input events that drive the application;

Begin Task 1.1.2

1.1.2.1 FTR: Review written description of need[7]

1.1.2.2 Derive a list of customer visible outputs/inputs

1.1.2.3 FTR: Review outputs/inputs with customer and revise as required;

endtask Task 1.1.2

1.1.3 Define the functionality/behavior for each major function;

Begin Task 1.1.3

1.1.3.1 FTR: Review output and input data objects derived in task 1.1.2;

1.1.3.2 Derive a model of functions/behaviors;

1.1.3.3 FTR: Review functions/behaviors with customer and revise as required;

endtask Task 1.1.3

1.1.4 Isolate those elements of the technology to be implemented in software;

1.1.5 Research availability of existing software;

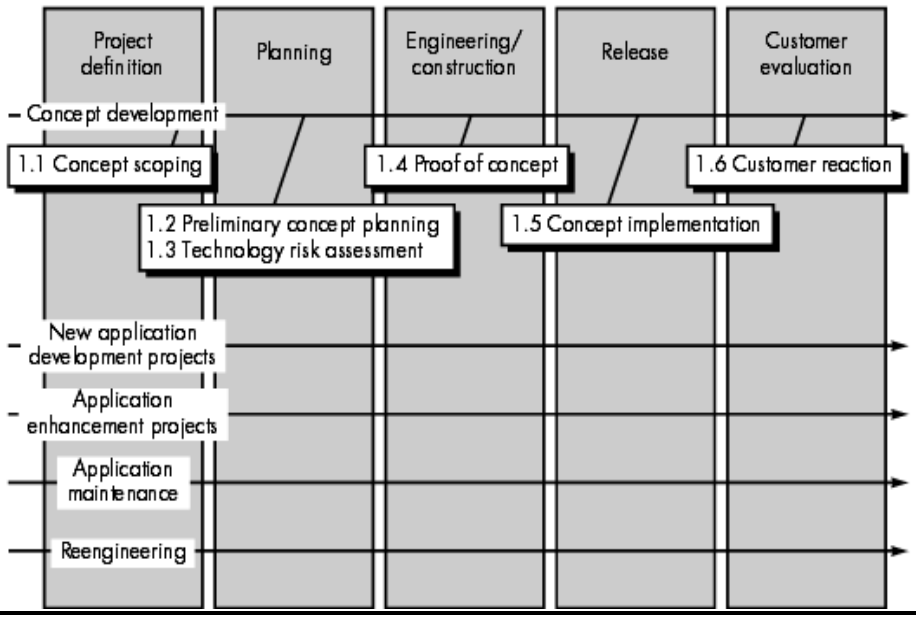
1.1.6 Define technical feasibility;

1.1.7 Make quick estimate of size;

1.1.8 Create a Scope Definition;

endTask definition: Task 1.1

FIGURE 7.1 Concept development tasks in a linear sequential model



Software configuration management: Basics and standards

Definition:

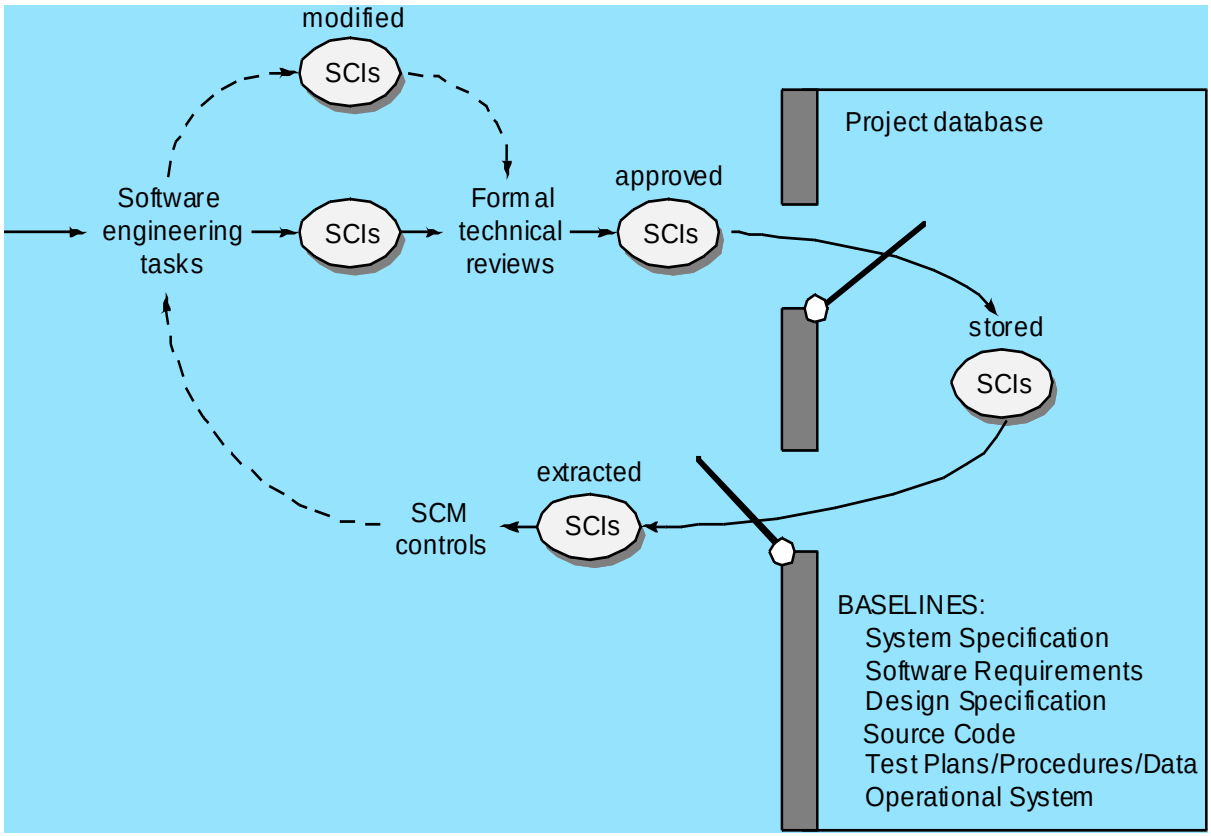
The set of activities that have been developed to manage change throughout the software life cycle.

Purpose:

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

Baselines

- Definition: 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.

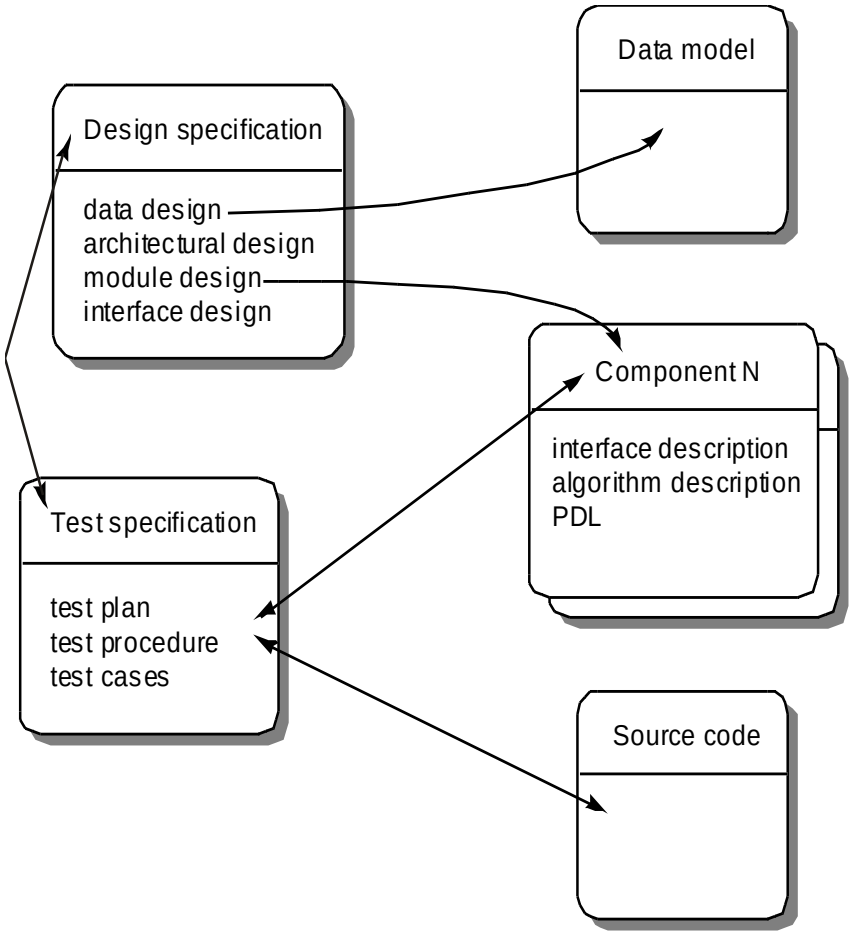


Project Baseline

- Central repository of reviewed and approved artifacts that represent a given stable point in overall system development.
- Shared DB for project and kept in consistent state.
- Policies allow the team to achieve consistent state and manage the project.

Software Configuration Item (SCI)

- 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)



A configuration object has a name, attributes,
and is "connected" to other objects by relationships. Referring to Figure 9.2, the configuration objects, **Design Specification, data model, component N, source code** and **Test Specification** are each defined separately. However, each of the objects is related to the others as shown by the arrows. A curved arrow indicates a *compositional relation.* That is, **data model** and **component N** are part of the object **Design Specification.** A double-headed straight arrow indicates an interrelationship.

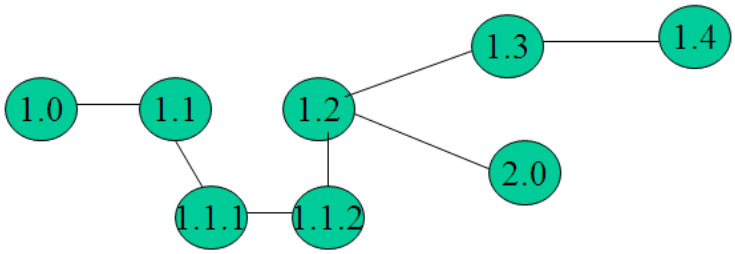
Elements of SCM

There are four elements of SCM:

1. Software Configuration Identification
2. Software Configuration Control
3. Software Configuration Auditing
4. Software Configuration Status Accounting

1. Software Configuration Identification

- Provides labels for the baselines and their updates.
- Evolution graph: depicts versions/variants.



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

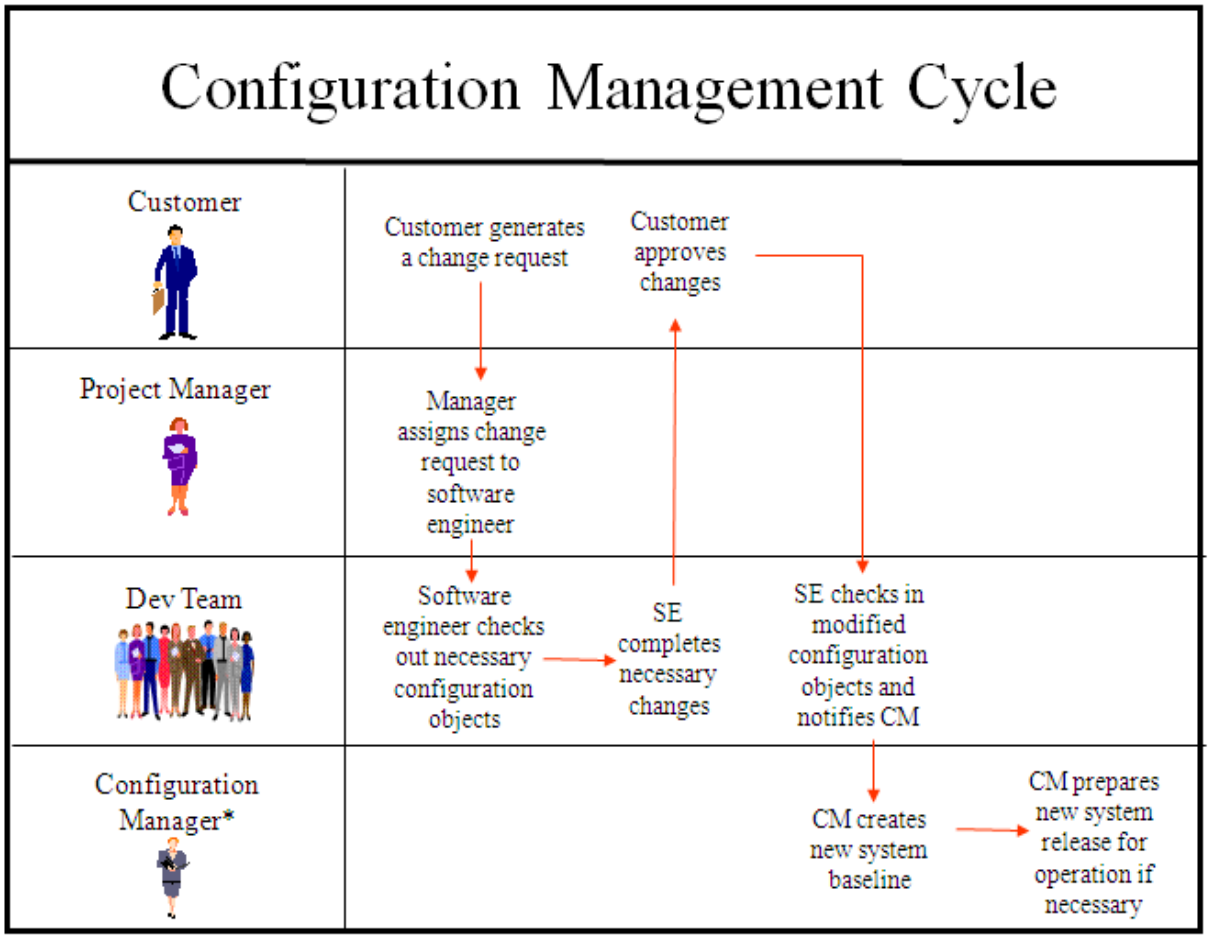
2. Software Configuration Control

Three basic ingredients to SCC

1. Documentation for formally precipitating and defining a proposed change to a software system.
2. An organizational body (Configuration Control Board) for formally evaluating and approving or disapproving a proposed change to a software system.
3. Procedures for controlling changes to a software system.

Why needed?

1. Not all possible changes are beneficial.
2. Need a mechanism to control access to different items of the configuration (who can access what).



3. 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.
- Asks 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?

4. Software Configuration Status Accounting

- 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.
- Why needed?
  - Ensure that there is progress within the development of the project.

- Track updates to baselines.

User interface design - Golden Rules

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

1. Place the User in Control

- **Define interaction modes in a way that does not force a user into unnecessary or undesired actions.**
- **Provide for flexible interaction.**
- **Allow user interaction to be interruptible and undoable.**
- **Streamline interaction as skill levels advance and allow the interaction to be customized.**
- **Hide technical internals from the casual user.**
- **Design for direct interaction with objects that appear on the screen.**

2. Reduce the User's Memory Load

- **Reduce demand on short-term memory.**
- **Establish meaningful defaults.**
- **Define shortcuts that are intuitive.**
- **The visual layout of the interface should be based on a real world metaphor.**
- **Disclose information in a progressive fashion.**

3. Make the Interface Consistent

- **Allow the user to put the current task into a meaningful context.**
- **Maintain consistency across a family of applications.**
- **If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.**

Computer aided software engineering tools (CASE)

Computer-aided software engineering (CASE) tools assist software engineering managers and practitioners in every activity associated with the software process.

What is CASE?

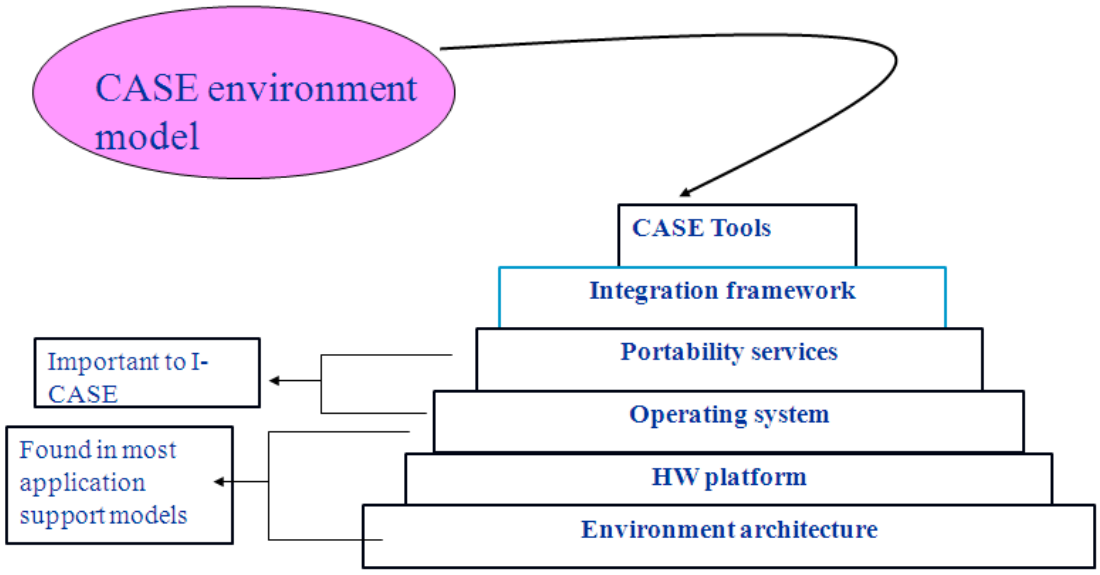
- **CASE is the use of IT in SW development activities, techniques and methodology**
- **CASE tools are programs that automate or support one or more phases in a SW development life cycle**

Purpose of CASE tools

- **increase the speed of SW development activities**
- **increase the SW productivity**
- **improve the quality of the SW developed**

CASE building blocks

The environment architecture, composed of the hardware platform and system support (including networking software, database management, and object management services), lays the ground work for CASE. The building blocks for CASE are illustrated in the following Figure.



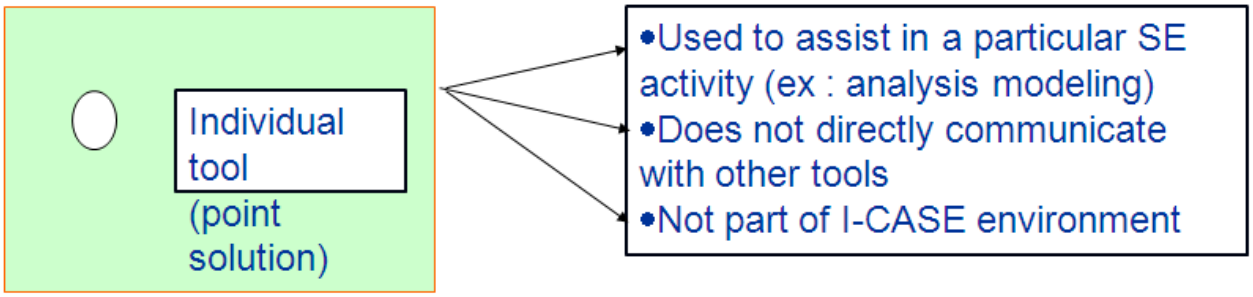
- <u>portability services</u> = as a bridge between CASE tools + integration framework + environment architecture
- <u>integration framework</u> = collection of specialized programs that enables individual CASE tools to :
  - ✓ communicate with one another
  - ✓ create a project data base
  - ✓ exhibit the same look and feel to the SW engineer
- integrated tools help project team develop, organize and control work products

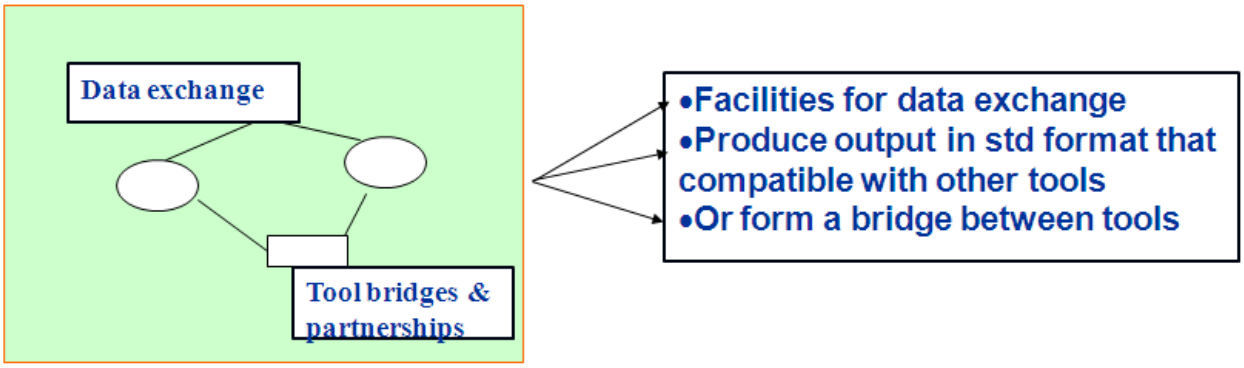
- The building blocks depicted in Figure represent a foundation for the integration of CASE tools. However, most CASE tools in use today have not been constructed using all these building blocks.

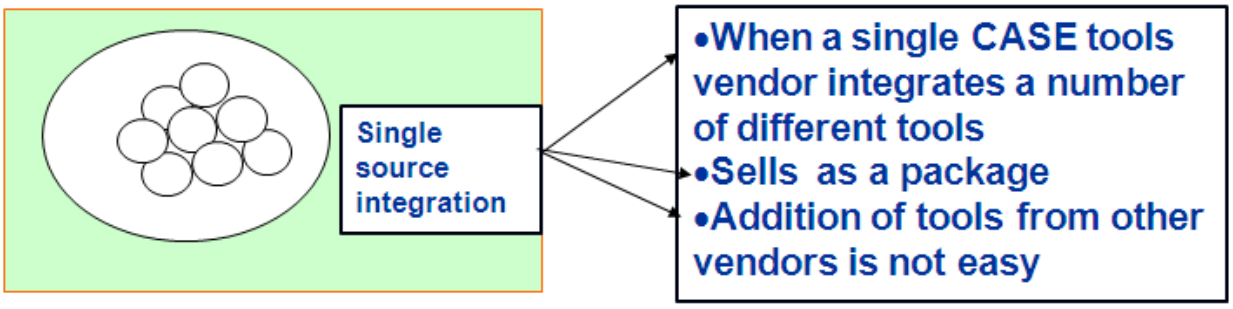
- In fact, some CASE tools remain "point solutions." That is, a tool is used to assist in a particular software engineering activity (e.g., analysis modeling) but does not directly communicate with other tools, is not tied into a project database, is not part of an integrated CASE environment (ICASE). Although this situation is not ideal, a CASE tool can be used quite effectively, even if it is a point solution.

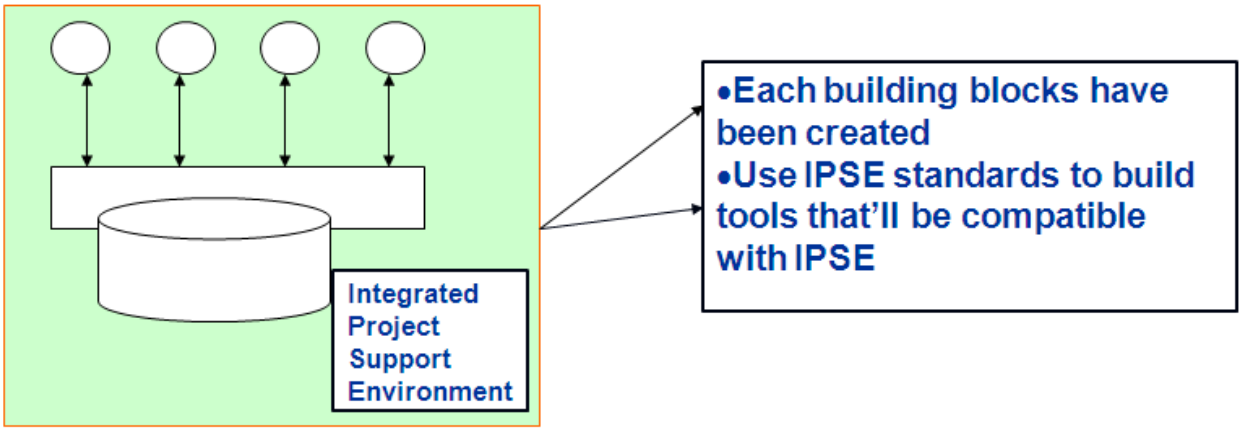
- CASE Integration options are shown below:

  - Point solution
  - Data Exchange , Tool bridges and partnerships
  - Single source integration
  - Integrated project support environment









Taxonomy of CASE tools

- CASE tools do not have to be part of an integrated environment to be useful to SW engineers but the impact on product quality will be greater if they are [pressman]
- CASE tools can be classified by function, role, use in SE process, environment architecture, etc.

1. **Business process engineering tools.**
2. **Process modeling and management tools.**
3. **Project planning tools**
4. **Risk analysis tools**
5. **Project management tools**
6. **Requirements tracing tools**
7. **Metrics and management tools.**
8. **Documentation tools.**
9. **System software tools**
10. **Quality assurance tools.**
11. **Database management tools**
12. **Software configuration management tools**
13. **Analysis and design tools**
14. **PRO/SIM tools.** PRO/SIM (prototyping and simulation) tools
15. **Interface design and development tools**
16. **Prototyping tools**
17. **Programming tools**
18. **Web development tools**
19. **Integration and testing tools**
    - *Data acquisition*—tools that acquire data to be used during testing.
    - *Static measurement*—tools that analyze source code without executing testcases.
    - *Dynamic measurement*—tools that analyze source code during execution.
    - *Simulation*—tools that simulate function of hardware or other externals.
    - *Test management*—tools that assist in the planning, development, and control of testing.

- *Cross-functional tools*—tools that cross the bounds of the preceding categories.

20. Static analysis tools
21. Dynamic analysis tools
22. Test management tools
23. Client/server testing tools
24. Reengineering tools

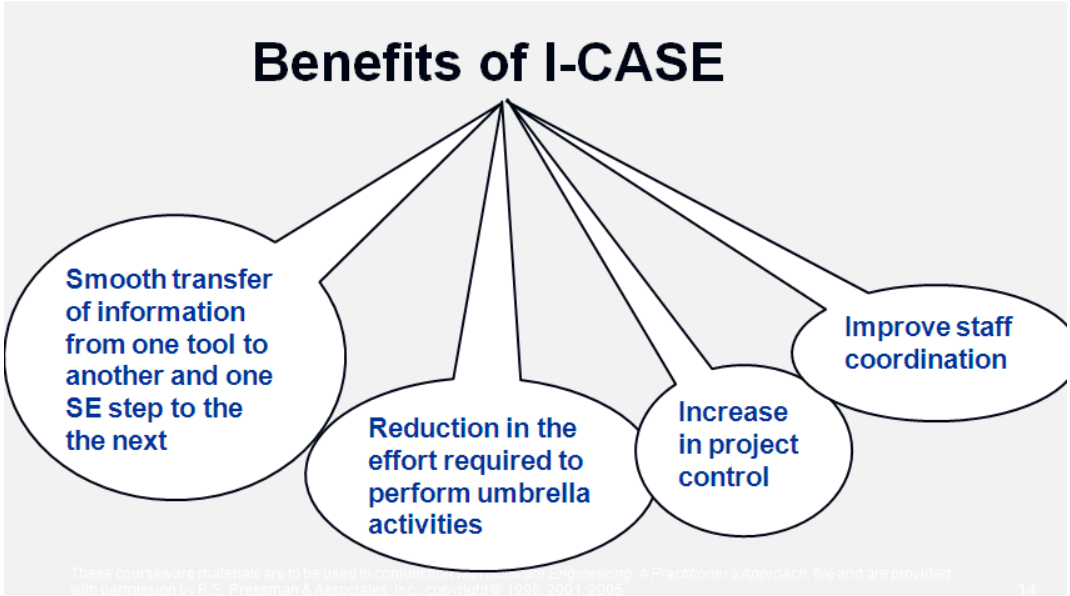
- *Reverse engineering to specification tools* take source code as input and generate graphical structured analysis and design models, where-used lists, and other design information.
- *Code restructuring and analysis tools* analyze program syntax, generate a control flow graph, and automatically generate a structured program.
- *On-line system reengineering tools* are used to modify on-line database systems (e.g., convert IDMS or DB2 files into entity-relationship format).

Integrated CASE environment

- Integration = combination and closure
- Combines a variety of different tools and a spectrum of information ➔ enables closure of communication among tools, between people and across the SW process
- Tools are integrated ➔ SE information is available to each tool that needs it
- Usage is integrated ➔ common look and feel is provided for all tools
- Development philosophy is integrated ➔ standards SE approach

An integrated CASE environment should

- Provide a mechanism for **sharing software engineering information** among all tools contained in the environment.
- Enable a change to **one item of information to be tracked to other related information** items.
- Provide **version control and overall configuration management** for all software engineering information.
- Allow **direct, non sequential access** to any tool contained in the environment.
- Establish **automated support for the software process model** that has been chosen, integrating CASE tools and software configuration items **(SCIs)** into a standard work breakdown structure.
- Enable the users of each tool to experience a consistent **look and feel at the human**/computer interface.
- Support **communication among software engineers**.
- Collect both **management and technical metrics** that can be used to improve the process and the product.



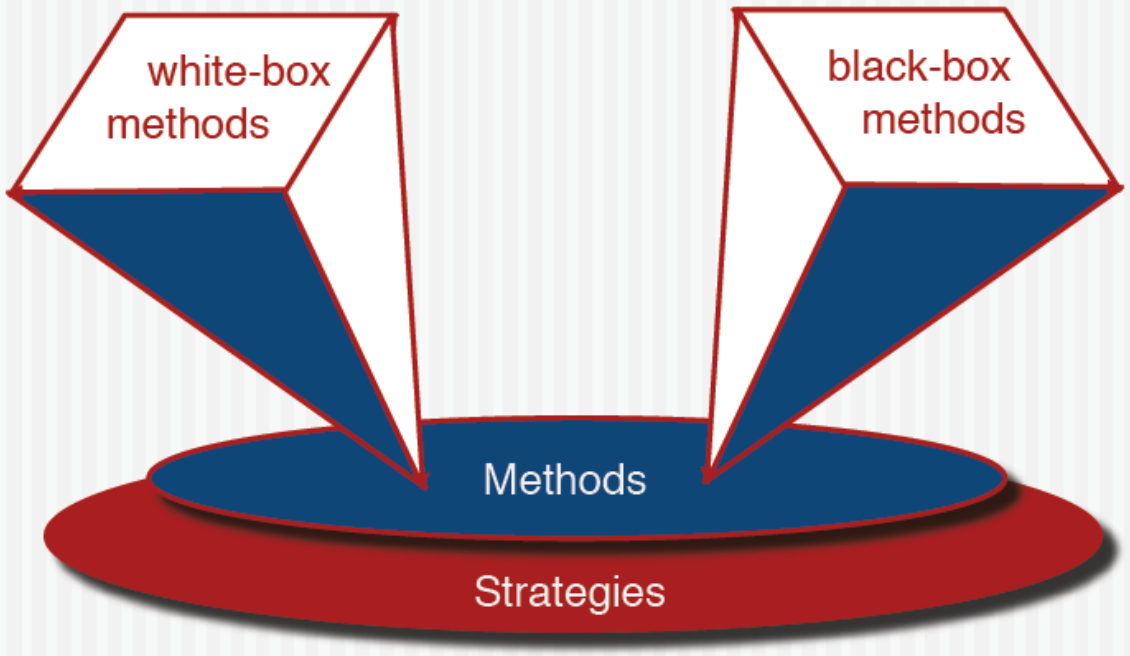
Challenges of I-CASE

- Consistent representations of SE information
- Standardized interfaces between tools
- Homogeneous mechanism for communication between SW engineer and each tool
- An effective approach that will enable I-CASE to move among various HW platforms and OS.

Software Testing Techniques

- Testing fundamentals
- White-box testing
- Black-box testing
- Object-oriented testing methods

Software Testing



Some Terminologies

- **Error, Mistake, Bug, Fault and Failure**

People make **errors**. A good synonym is **mistake**. This may be a syntax error or misunderstanding of specifications. Sometimes, there are logical errors.

When developers make mistakes while coding, we call these mistakes "**bugs**".

A **fault** is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, ER diagrams, source code etc. Defect is a good synonym for fault.

A **failure** occurs when a fault executes. A particular fault may cause different failures, depending on how it has been exercised.

Software Testing Fundamentals

Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design, and coding.

Software testing demonstrates that software function appear to be working according to specifications and performance requirements.

Testing Objectives:

Myers [MYE79] states a number of rules that can serve well as 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 high probability of finding an undiscovered error.
- A successful test is one that uncovers an as-yet undiscovered error.

The major testing objective is to design tests that systematically uncover types of errors with minimum time and effort.

Characteristics of Testable Software

- Operable
  - The better it works (i.e., better quality), the easier it is to test
- Observable
  - Incorrect output is easily identified; internal errors are automatically detected
- Controllable
  - The states and variables of the software can be controlled directly by the tester
- Decomposable
  - The software is built from independent modules that can be tested independently

Characteristics of Testable Software (continued)

- Simple
  - The program should exhibit functional, structural, and code simplicity
- Stable
  - Changes to the software during testing are infrequent and do not invalidate existing tests
- Understandable
  - The architectural design is well understood; documentation is available and organized

Test Characteristics

- A good test has a high probability of finding an error
  - The tester must understand the software and how it might fail
- A good test is not redundant
  - Testing time is limited; one test should not serve the same purpose as another test
- A good test should be “best of breed”
  - Tests that have the highest likelihood of uncovering a whole class of errors should be used
- A good test should be neither too simple nor too complex
  - Each test should be executed separately; combining a series of tests could cause side effects and mask certain errors

Two Unit Testing Techniques

- Black-box testing
  - Knowing the specified function that a product has been designed to perform, test to see if that function is fully operational and error free
  - Includes tests that are conducted at the software interface
  - Not concerned with internal logical structure of the software
- White-box testing
  - Knowing the internal workings of a product, test that all internal operations are performed according to specifications and all internal components have been exercised
  - Involves tests that concentrate on close examination of procedural detail
  - Logical paths through the software are tested
  - Test cases exercise specific sets of conditions and loops

Test Case Design

Two general software testing approaches:

Black-Box Testing and White-Box Testing

Black-box testing:

knowing the specific functions of a software,
design tests to demonstrate each function and check its errors.

Major focus:

functions, operations, external interfaces,
external data and information

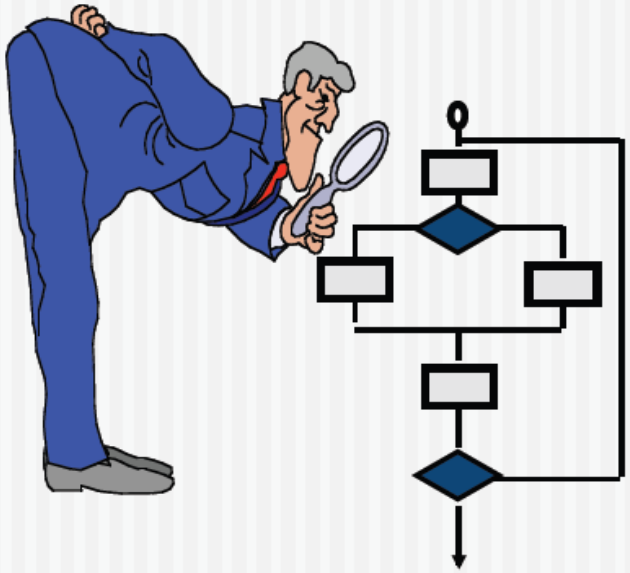
White-box testing:

knowing the internals of a software,
design tests to exercise all internals of a software to make sure
they operates according to specifications and designs

Major focus: *internal structures, logic paths, control flows, data flows*
internal data structures, conditions, loops, etc.

White-box Testing

White-Box Testing



... our goal is to ensure that all statements and conditions have been executed at least once ...

Why Cover?

- logic errors and incorrect assumptions are inversely proportional to a path's execution probability
- we often <u>believe</u> that a path is not likely to be executed; in fact, reality is often counter intuitive
- typographical errors are random; it's likely that untested paths will contain some

White-box Testing

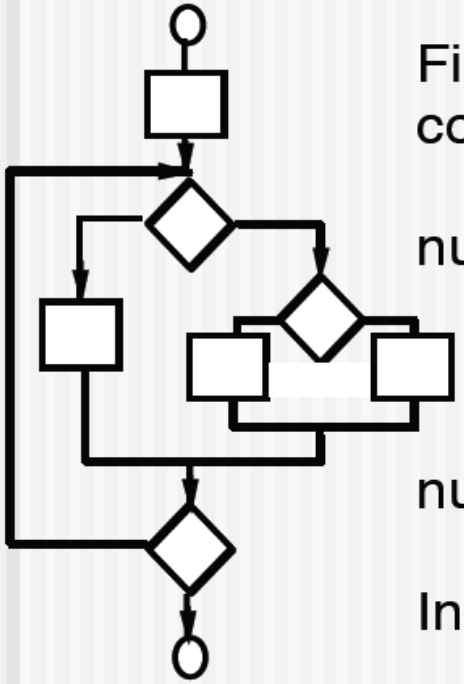
- Uses the control structure part of component-level design to derive the test cases
- These test cases
  - Guarantee that all independent paths within a module have been exercised at least once
  - Exercise all logical decisions on their true and false sides
  - Execute all loops at their boundaries and within their operational bounds
  - Exercise internal data structures to ensure their validity

"Bugs lurk in corners and congregate at boundaries"

Basis Path Testing

- White-box testing technique proposed by Tom McCabe
- Enables the test case designer to derive a logical complexity measure of a procedural design
- Uses 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 <u>every statement</u> in the program <u>at least one time</u> during testing

Basis Path Testing



First, we compute the cyclomatic complexity:

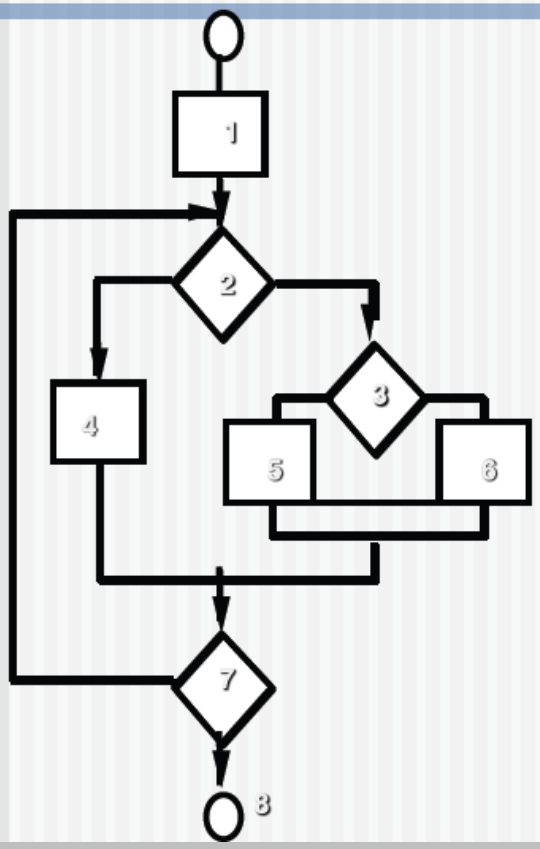
number of simple decisions + 1

or

number of enclosed areas + 1

In this case, V(G) = 4

Basis Path Testing



Next, we derive the independent paths:

Since V(G) = 4, there are four paths

Path 1: 1,2,3,6,7,8

Path 2: 1,2,3,5,7,8

Path 3: 1,2,4,7,8

Path 4: 1,2,4,7,2,4,...7,8

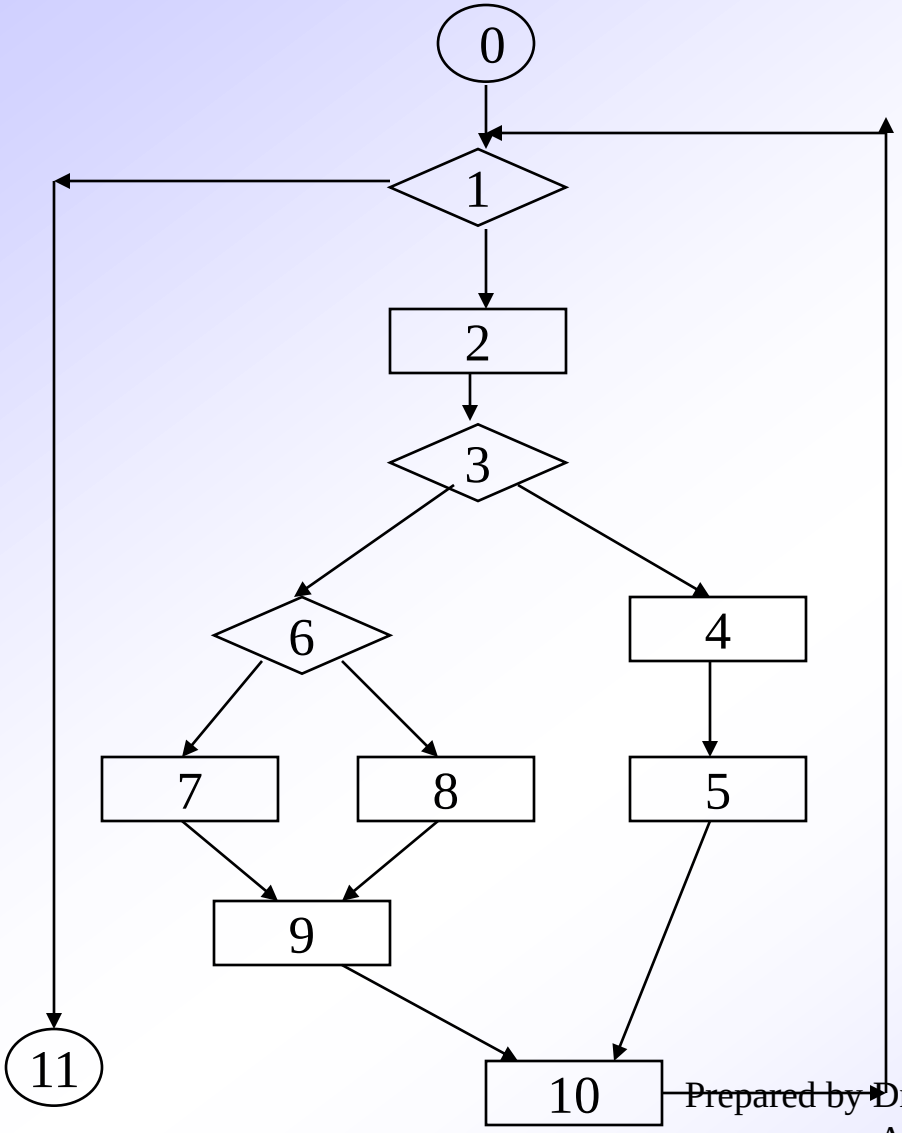
Finally, we derive test cases to exercise these paths.

Flow Graph Notation

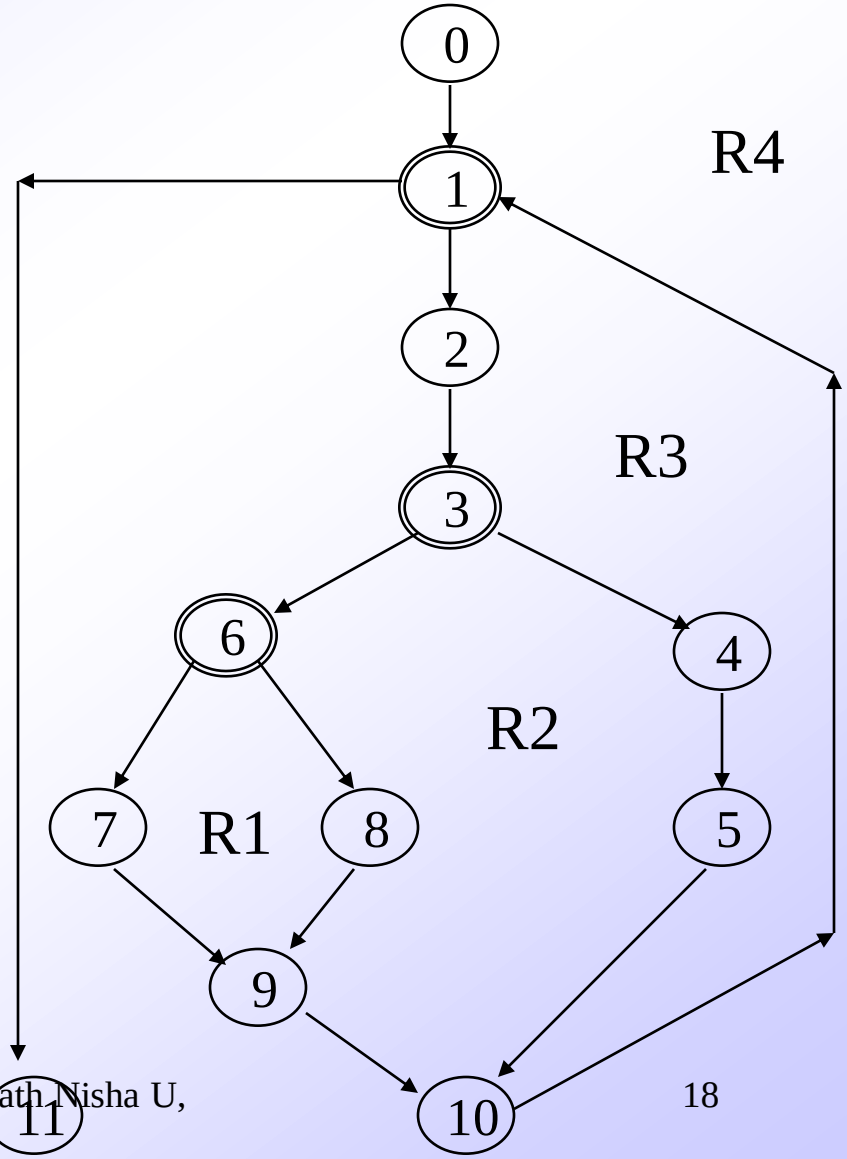
- A circle in a graph represents a node, which stands for a sequence of one or more procedural statements
- A node containing a simple conditional expression is referred to as a predicate node
  - Each compound condition in a conditional expression containing one or more Boolean operators (e.g., and, or) is represented by a separate predicate node
  - A predicate node has two edges leading out from it (True and False)
- An edge, or a link, is a an arrow representing flow of control in a specific direction
  - An edge must start and terminate at a node
  - An edge does not intersect or cross over another edge
- Areas bounded by a set of edges and nodes are called regions
- When counting regions, include the area outside the graph as a region, too

Flow Graph Example

FLOW CHART



FLOW GRAPH



Independent Program Paths

- Defined as a path through the program from the start node until the end node that introduces at least one new set of processing statements or a new condition (i.e., new nodes)
- Must move along <u>at least one</u> edge that has not been traversed before by a previous path
- Basis set for flow graph on previous slide
  - Path 1: 0-1-11
  - Path 2: 0-1-2-3-4-5-10-1-11
  - Path 3: 0-1-2-3-6-8-9-10-1-11
  - Path 4: 0-1-2-3-6-7-9-10-1-11
- The <u>number of paths</u> in the basis set is determined by the <u>cyclomatic complexity</u>

Cyclomatic Complexity

- Provides a quantitative measure of the logical complexity of a program
- Defines the number of independent paths in the basis set
- Provides an upper bound for the number of tests that must be conducted to ensure all statements have been executed at least once
- Can be computed three ways
  - The number of regions
  - V(G) = E – N + 2, where E is the number of edges and N is the number of nodes in graph G
  - V(G) = P + 1, where P is the number of predicate nodes in the flow graph G
- Results in the following equations for the example flow graph
  - Number of regions = 4
  - V(G) = 14 edges – 12 nodes + 2 = 4
  - V(G) = 3 predicate nodes + 1 = 4

Deriving the Basis Set and Test Cases

1) Using the design or code as a foundation, draw a corresponding flow graph
2) Determine the cyclomatic complexity of the resultant flow graph
3) Determine a basis set of linearly independent paths
4) Prepare test cases that will force execution of each path in the basis set

Deriving Test Cases

Step 1 : *Using the design or code as a foundation, draw a corresponding flow graph.*

Step 2: *Determine the cyclomatic complexity of the resultant flow graph.*

Step 3: *Determine a basis set of linearly independent paths.*

For example,

path 1: 1-2-10-11-13
path 2: 1-2-10-12-13
path 3: 1-2-3-10-11-13
path 4: 1-2-3-4-5-8-9-2-…
path 5: 1-2-3-4-5-6-8-9-2-..
Path 6: 1-2-3-4-5-6-7-8-9-2-..

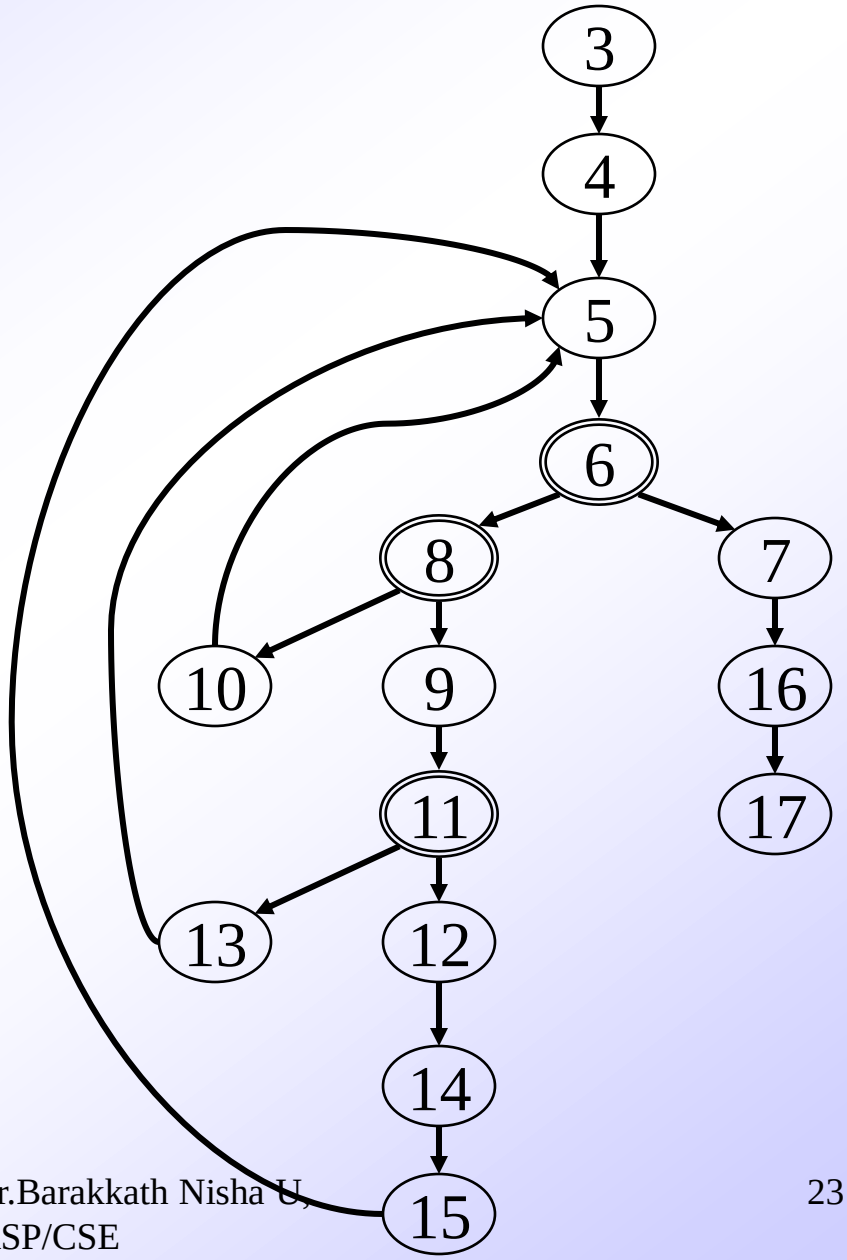
Step 4: *Prepare test cases that will force execution of each path in the basis set.*

Path 1: test case:

value (k) = valid input, where k < i defined below.
value (i) = -999, where 2 <= I <= 100
expected results: correct average based on k values and proper totals.

A Second Flow Graph Example

```
 1  int functionY(void)
 2  {
 3     int x = 0;
 4     int y = 19;

 5  A: x++;
 6     if (x > 999)
 7        goto D;
 8     if (x % 11 == 0)
 9        goto B;
10     else goto A;

11  B: if (x % y == 0)
12        goto C;
13     else goto A;

14  C: printf("%d\n", x);
15     goto A;

16  D: printf("End of list\n");
17     return 0;
18  }
```

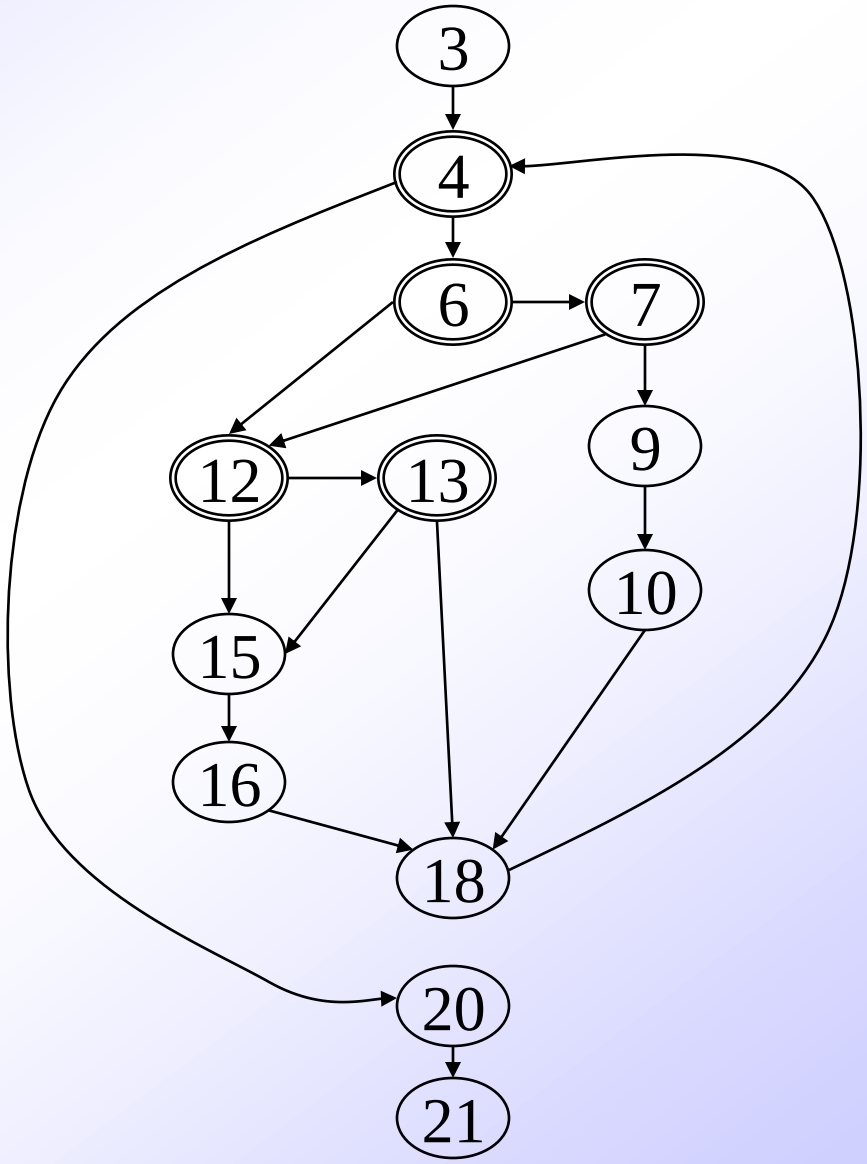


A Sample Function to Diagram and Analyze

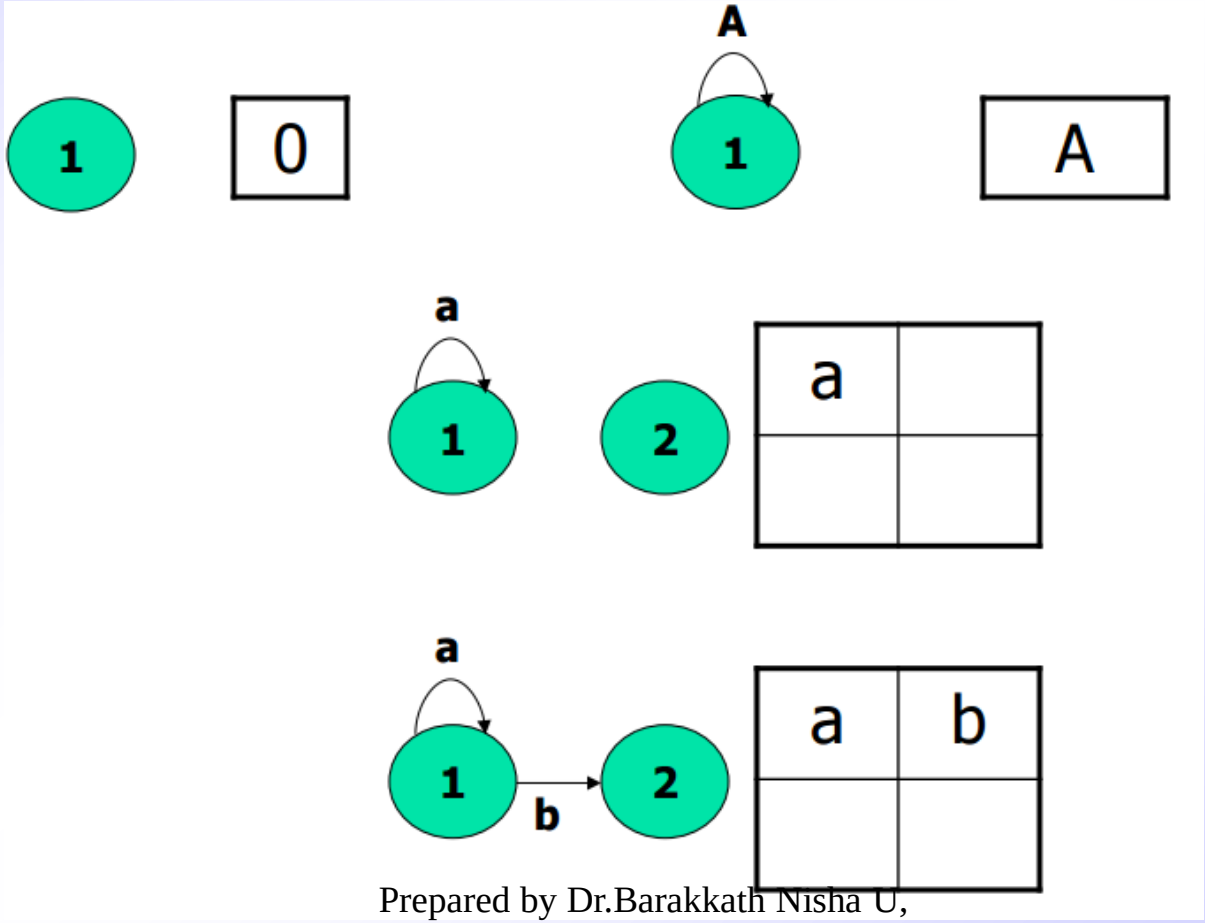
```
int functionZ(int y)
{
int x = 0;

while (x <= (y * y))
    {
    if ((x % 11 == 0) &&
         (x % y == 0))
        {
        printf(“%d”, x);
        x++;
        } // End if
    else if ((x % 7 == 0) ||
                (x % y == 1))
        {
        printf(“%d”, y);
        x = x + 2;
        } // End else
    printf(“\n”);
    } // End while

printf("End of list\n");
return 0;
} // End functionZ
```

A Sample Function to Diagram and Analyze

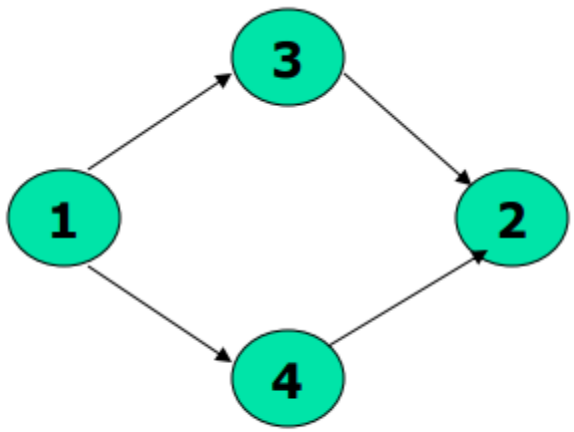
```
 1  int functionZ(int y)
 2  {
 3  int x = 0;

 4  while (x <= (y * y))
 5     {
 6     if ((x % 11 == 0) &&
 7         (x % y == 0))
 8        {
 9        printf("%d", x);
10        x++;
11        } // End if
12     else if ((x % 7 == 0) ||
13              (x % y == 1))
14        {
15        printf("%d", y);
16        x = x + 2;
17        } // End else
18     printf("\n");
19     } // End while

20  printf("End of list\n");
21  return 0;
22  } // End functionZ
```



Graph Matrices



Example



		a	c
	b		
	d		

Cyclomatic Complexity

The cyclomatic complexity obtained by subtracting 1 from the total number of entries in each row and ignoring rows with no entries, we obtain the equivalent number of decisions for each row. Adding these values and then adding 1 to the sum yields the graph's cyclomatic complexity.

		1	1	**2-1=1**
	1			**1-1=0**
	1			**1-1=0**

1+1=2 (cyclomatic complexity)

Control Structure Testing

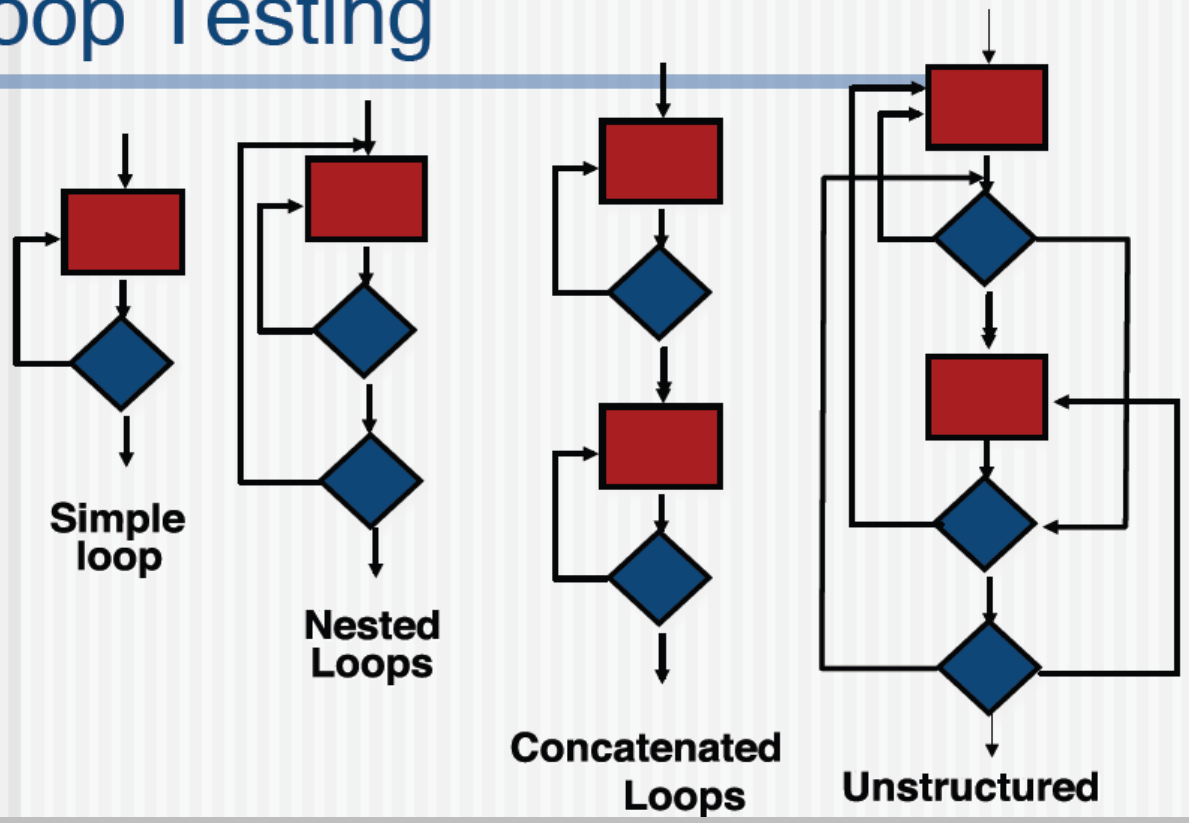
- Condition testing — a test case design method that exercises the logical conditions contained in a program module
- Data flow testing — selects test paths of a program according to the locations of definitions and uses of variables in the program

Control structure testing

Control structure testing is more comprehensive than basis path testing and includes it. This method uses different categories of tests that are listed below.

- *Condition testing*
- *Data flow testing*
- *Loop testing*

Loop Testing



Loop Testing - General

- A white-box testing technique that focuses exclusively on the validity of loop constructs
- Four different classes of loops exist
  - Simple loops
  - Nested loops
  - Concatenated loops
  - Unstructured loops
- Testing occurs by varying the loop boundary values
  - Examples:

```
for (i = 0; i < MAX_INDEX; i++)

while (currentTemp >= MINIMUM_TEMPERATURE)
```

Loop Testing: Simple Loops

Minimum conditions—Simple Loops

1. skip the loop entirely
2. only one pass through the loop
3. two passes through the loop
4. m passes through the loop $m < n$
5. (n-1), n, and (n+1) passes through the loop

where n is the maximum number of allowable passes

Loop Testing: Nested Loops

Nested Loops

Start at the innermost loop. Set all outer loops to their minimum iteration parameter values.

Test the min+1, typical, max-1 and max for the innermost loop, while holding the outer loops at their minimum values.

Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.

Concatenated Loops

If the loops are independent of one another
  then treat each as a simple loop
  else* treat as nested loops
endif*

for example, the final loop counter value of loop 1 is used to initialize loop 2.

Testing of Simple Loops

1) Skip the loop entirely
2) Only one pass through the loop
3) Two passes through the loop
4) m passes through the loop, where m < n
5) n –1, n, n + 1 passes through the loop

‘n’ is the maximum number of allowable passes through the loop

Testing of Nested Loops

1) Start at the <u>innermost</u> loop; set all other loops to <u>minimum</u> values
2) Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter values; add other tests for out-of-range or excluded values
3) <u>Work outward</u>, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to “typical” values
4) Continue until all loops have been tested

Testing of Concatenated Loops

- For independent loops, use the same approach as for simple loops
- Otherwise, use the approach applied for nested loops

Testing of Unstructured Loops

- Redesign the code to reflect the use of structured programming practices
- Depending on the resultant design, apply testing for simple loops, nested loops, or concatenated loops

Black-box Testing

Functional Testing

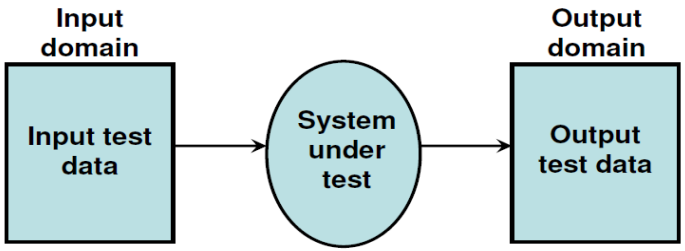
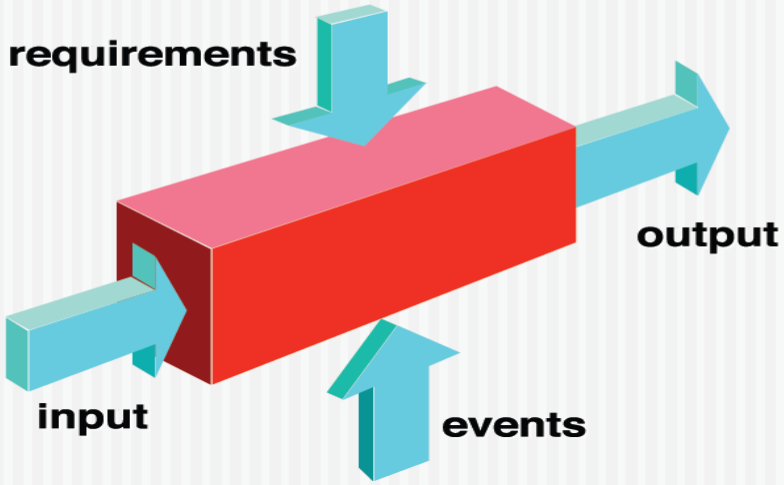


Fig. 3: Black box testing

Black-Box Testing



Black-box Testing

- Complements white-box testing by uncovering different classes of errors
- Focuses on the functional requirements and the information domain of the software
- Used during the later stages of testing after white box testing has been performed
- The tester identifies a set of input conditions that will fully exercise all functional requirements for a program
- The test cases satisfy the following:
  - Reduce, by a count greater than one, the number of additional test cases that must be designed to achieve reasonable testing
  - Tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific task at hand

Black-box Testing Categories

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

Questions answered by Black-box Testing

- How is functional validity tested?
- How are system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundary values of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

Equivalence Partitioning

- A black-box testing method that divides the input domain of a program into classes of data from which test cases are derived
- An ideal test case single-handedly uncovers a complete class of errors, thereby reducing the total number of test cases that must be developed
- Test case design is based on an evaluation of equivalence classes for an input condition
- An equivalence class represents a set of valid or invalid states for input conditions
- From each equivalence class, test cases are selected so that the largest number of attributes of an equivalence class are exercise at once

Guidelines for Defining Equivalence Classes

- If an input condition specifies a range, one valid and two invalid equivalence classes are defined
  - Input range: 1 – 10 Eq classes: {1..10}, {x < 1}, {x > 10}
- If an input condition requires a specific value, one valid and two invalid equivalence classes are defined
  - Input value: 250 Eq classes: {250}, {x < 250}, {x > 250}
- If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined
  - Input set: {-2.5, 7.3, 8.4} Eq classes: {-2.5, 7.3, 8.4}, {any other x}
- If an input condition is a Boolean value, one valid and one invalid class are define
  - Input: {true condition} Eq classes: {true condition}, {false condition}

Boundary Value Analysis

- A greater number of errors occur at the boundaries of the input domain rather than in the "center"
- Boundary value analysis is a test case design method that complements equivalence partitioning
  - It selects test cases at the edges of a class
  - It derives test cases from both the input domain and output domain

Boundary Value Analysis

Consider a program with two input variables x and y. These input variables have specified boundaries as:

$$a \le x \le b$$
$$c \le y \le d$$

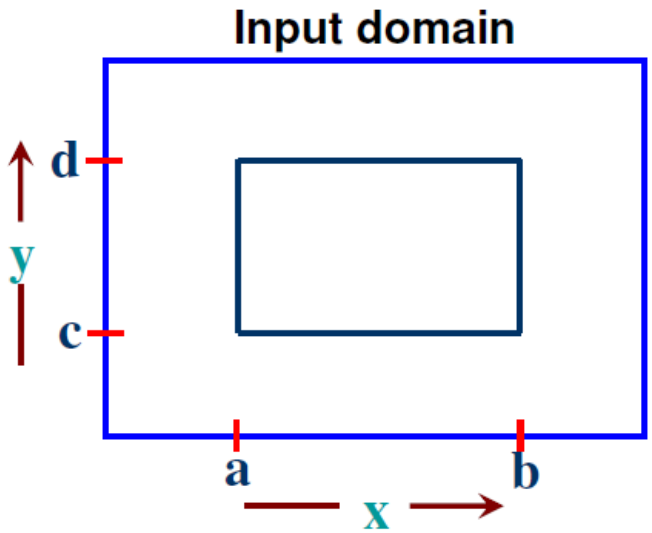


Fig.4: Input domain for program having two input variables

The boundary value analysis test cases for our program with two inputs variables (x and y) that may have any value from 100 to 300 are: (200,100), (200,101), (200,200), (200,299), (200,300), (100,200), (101,200), (299,200) and (300,200). This input domain is shown in Fig. 8.5. Each dot represent a test case and inner rectangle is the domain of legitimate inputs. Thus, for a program of n variables, boundary value analysis yield **4n + 1** test cases.

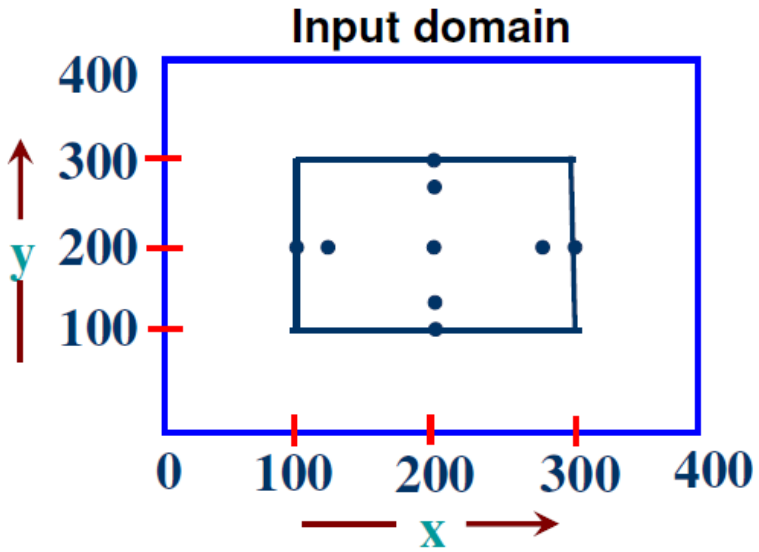


Fig. 5: Input domain of two variables x and y with boundaries [100,300] each

Guidelines for Boundary Value Analysis

- 1. If an input condition specifies a range bounded by values *a* and *b*, test cases should be designed with values *a* and *b* as well as values just above and just below *a* and *b*
- 2. If an input condition specifies a number of values, test case should be developed that exercise the minimum and maximum numbers. Values just above and just below the minimum and maximum are also tested
- Apply guidelines 1 and 2 to output conditions; produce output that reflects the minimum and the maximum values expected; also test the values just below and just above
- If internal program data structures have prescribed boundaries (e.g., an array), design a test case to exercise the data structure at its minimum and maximum boundaries

Object-Oriented Testing Methods

Introduction

- It is necessary to test an object-oriented system at a variety of different levels
- The goal is to uncover errors that may occur as classes collaborate with one another and subsystems communicate across architectural layers
  - Testing begins "in the small" on methods within a class and on collaboration between classes
  - As class integration occurs, use-based testing and fault-based testing are applied
  - Finally, use cases are used to uncover errors during the software validation phase
- Conventional test case design is driven by an input-process-output view of software
- Object-oriented testing focuses on designing appropriate sequences of methods to exercise the states of a class

Testing Implications for Object-Oriented Software

- Because attributes and methods are encapsulated in a class, testing methods from outside of a class is generally unproductive
- Testing requires reporting on the state of an object, yet encapsulation can make this information somewhat difficult to obtain
- Built-in methods should be provided to report the values of class attributes in order to get a snapshot of the state of an object
- Inheritance requires retesting of each new context of usage for a class
  - If a subclass is used in an entirely different context than the super class, the super class test cases will have little applicability and a new set of tests must be designed

Applicability of Conventional Testing Methods

- White-box testing can be applied to the operations defined in a class
  - Basis path testing and loop testing can help ensure that every statement in an method has been tested
- Black-box testing methods are also appropriate
  - Use cases can provide useful input in the design of black-box tests

Fault-based Testing

- The objective in fault-based testing is to design tests that have a high likelihood of uncovering plausible faults
- Fault-based testing begins with the analysis model
  - The tester looks for plausible faults (i.e., aspects of the implementation of the system that may result in defects)
  - To determine whether these faults exist, test cases are designed to exercise the design or code
- If the analysis and design models can provide insight into what is likely to go wrong, then fault-based testing can find a significant number of errors

Fault-based Testing (continued)

- Integration testing looks for plausible faults in method calls or message connections (i.e., client/server exchange)
- Three types of faults are encountered in this context
  - Unexpected result
  - Wrong method or message used
  - Incorrect invocation
- The behavior of a method must be examined to determine the occurrence of plausible faults as methods are invoked
- Testing should exercise the attributes of an object to determine whether proper values occur for distinct types of object behavior
- The focus of integration testing is to determine whether errors exist in the calling code, not the called code

Random Order Testing (at the Class Level)

- Certain methods in a class may constitute a minimum behavioral life history of an object (e.g., open, seek, read, close); consequently, they may have implicit order dependencies or expectations designed into them
- Using the methods for a class, a variety of method sequences are generated randomly and then executed
- The goal is to detect these order dependencies or expectations and make appropriate adjustments to the design of the methods

Partition Testing (at the Class Level)

- Similar to equivalence partitioning for conventional software
- Methods are grouped based on one of three partitioning approaches
- State-based partitioning categorizes class methods based on their ability to change the state of the class
  - Tests are designed in a way that exercise methods that change state and those that do not change state
- Attribute-based partitioning categorizes class methods based on the attributes that they use
  - Methods are partitioned into those that read an attribute, modify an attribute, or do not reference the attribute at all
- Category-based partitioning categorizes class methods based on the generic function that each performs
  - Example categories are initialization methods, computational methods, and termination methods

Multiple Class Testing

- Class collaboration testing can be accomplished by applying random testing, partition testing, scenario-based testing and behavioral testing
- The following sequence of steps can be used to generate multiple class random test cases
  1) For each client class, use the list of class methods to generate a series of random test sequences; use these methods to send messages to server classes
  2) For each message that is generated, determine the collaborator class and the corresponding method in the server object
  3) For each method in the server object (invoked by messages from the client object), determine the messages that it transmits
  4) For each of these messages, determine the next level of methods that are invoked and incorporate these into the test sequence

Software Testing Strategies

- A strategy for software testing integrates the design of software test cases into a well-planned series of steps that result in successful development of the software
- The strategy provides a road map that describes the steps to be taken, when, and how much effort, time, and resources will be required
- The strategy incorporates test planning, test case design, test execution, and test result collection and evaluation
- The strategy provides guidance for the practitioner and a set of milestones for the manager
- Because of time pressures, progress must be measurable and problems must surface as early as possible

A Strategic Approach to Testing

General Characteristics of Strategic Testing

- To perform effective testing, a software team should conduct effective formal technical reviews
- Testing begins at the component level and work outward toward the integration of the entire computer-based system
- Different testing techniques are appropriate at different points in time
- Testing is conducted by the developer of the software and (for large projects) by an independent test group
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy

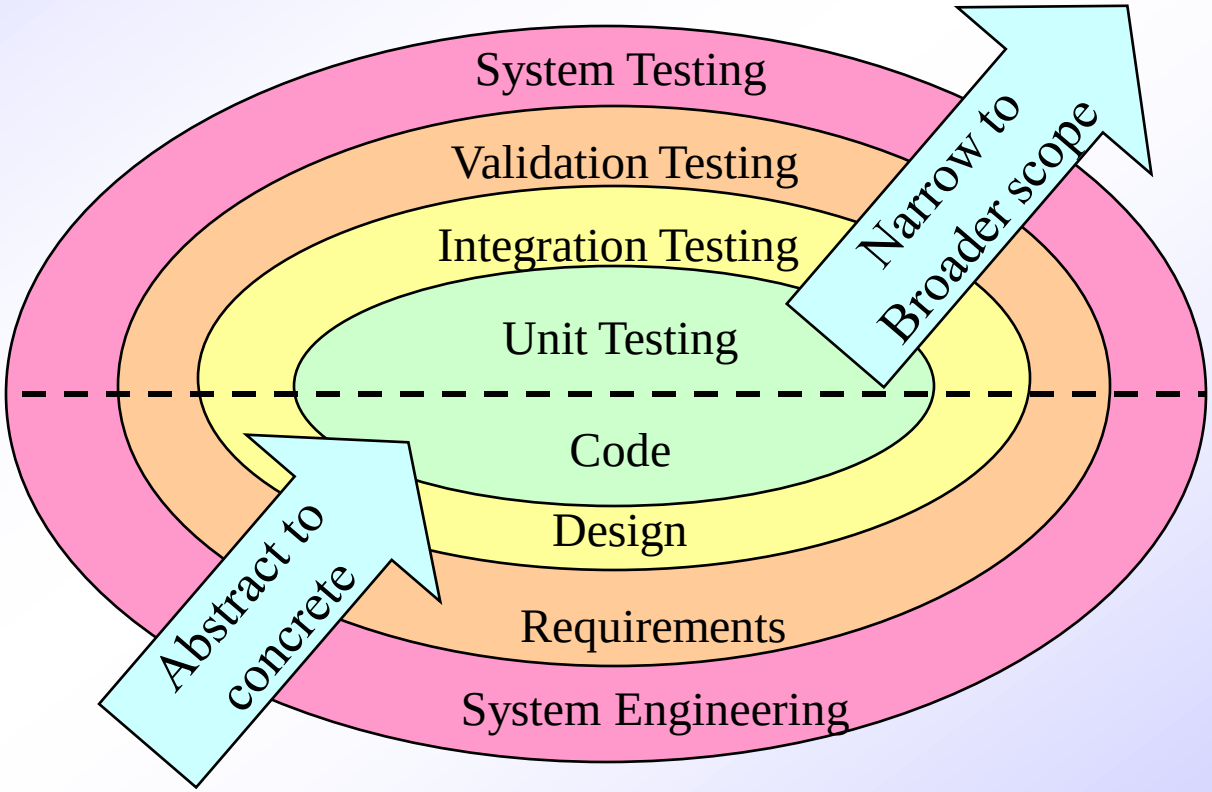
Verification and Validation

- Software testing is part of a broader group of activities called verification and validation that are involved in software quality assurance
- Verification (Are the algorithms coded correctly?)
  - The set of activities that ensure that software correctly implements a specific function or algorithm
- Validation (Does it meet user requirements?)
  - The set of activities that ensure that the software that has been built is traceable to customer requirements

Organizing for Software Testing

- Testing should aim at "breaking" the software
- Common misconceptions
  - The developer of software should do no testing at all
- Reality: Independent test group
  - Removes the inherent problems associated with letting the builder test the software that has been built
  - Removes the conflict of interest that may otherwise be present

A Strategy for Testing Conventional Software



Levels of Testing for Conventional Software

- Unit testing
  - Concentrates on each component/function of the software as implemented in the source code
  - makes heavy use of testing techniques that exercise specific control paths to detect errors in each software component individually
- Integration testing
  - Focuses on the design and construction of the software architecture
  - focuses on issues associated with verification and program construction as components begin interacting with one another
- Validation testing
  - Requirements are validated against the constructed software
  - provides assurance that the software validation criteria (established during requirements analysis) meets all functional, behavioral, and performance requirements
- System testing
  - The software and other system elements are tested as a whole
  - verifies that all system elements mesh properly and that overall system function and performance has been achieved

Testing Strategy applied to Conventional Software

- Unit testing
  - Exercises specific paths in a component's control structure to ensure complete coverage and maximum error detection
  - Components are then assembled and integrated
- Integration testing
  - Focuses on inputs and outputs, and how well the components fit together and work together
- Validation testing
  - Provides final assurance that the software meets all functional, behavioral, and performance requirements
- System testing
  - Verifies that all system elements (software, hardware, people, databases) mesh properly and that overall system function and performance is achieved

When is Testing Complete?

- There is no definitive answer to this question
- Every time a user executes the software, the program is being tested
- Sadly, testing usually stops when a project is running out of time, money, or both
- One approach is to divide the test results into various severity levels
  - Then consider testing to be complete when certain levels of errors no longer occur or have been repaired or eliminated

Software Test Strategy- Issues

- Specify product requirements in a quantifiable manner long before testing commences
- State testing objectives explicitly in measurable terms
- Understand the user of the software (through use cases) and develop a profile for each user category
- Develop a testing plan that emphasizes rapid cycle testing to get quick feedback to control quality levels and adjust the test strategy
- Build robust software that is designed to test itself and can diagnose certain kinds of errors
- Use effective formal technical reviews as a filter prior to testing to reduce the amount of testing required
- Conduct formal technical reviews to assess the test strategy and test cases themselves
- Develop a continuous improvement approach for the testing process through the gathering of metrics

Test Strategies for Conventional Software

Unit Testing

- Focuses testing on the function or software module
- Concentrates on the internal processing logic and data structures
- Is simplified when a module is designed with high cohesion
  - Reduces the number of test cases
  - Allows errors to be more easily predicted and uncovered
- Concentrates on critical modules and those with **high cyclomatic complexity** when testing resources are limited

Targets for Unit Test Cases

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

Common Computational Errors in Execution Paths

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

Other Errors to Uncover

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

Problems to uncover in Error Handling

- Error description is unintelligible or ambiguous
- Error noted does not correspond to error encountered
- Error condition causes operating system intervention prior to error handling
- Exception condition processing is incorrect
- Error description does not provide enough information to assist in the location of the cause of the error

Drivers and Stubs for Unit Testing

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

Integration Testing

- Defined as a systematic technique for constructing the software architecture
  - At the same time integration is occurring, conduct tests to uncover errors associated with interfaces
- Objective is to take unit tested modules and build a program structure based on the prescribed design
- Sandwich testing uses top-down tests for upper levels of program structure coupled with bottom-up tests for subordinate levels
- Testers should strive to indentify critical modules having the following requirements
- Overall plan for integration of software and the specific tests are documented in a test specification
- Two Approaches
  - Non-incremental Integration Testing
  - Incremental Integration Testing

Non-incremental Integration Testing

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

Incremental Integration Testing

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

Top-down Integration

- Modules are integrated by moving downward through the control hierarchy, beginning with the main module
- Subordinate modules are incorporated in either a depth-first or breadth-first fashion
  - DF: All modules on a major control path are integrated
  - BF: All modules directly subordinate at each level are integrated
- Advantages
  - This approach verifies major control or decision points early in the test process
- Disadvantages
  - Stubs need to be created to substitute for modules that have not been built or tested yet; this code is later discarded
  - Because stubs are used to replace lower level modules, no significant data flow can occur until much later in the integration/testing process

Bottom-up Integration

- Integration and testing starts with the most atomic modules in the control hierarchy
- Advantages
  - This approach verifies low-level data processing early in the testing process
  - Need for stubs is eliminated
- Disadvantages
  - Driver modules need to be built to test the lower-level modules; this code is later discarded or expanded into a full-featured version
  - Drivers inherently do not contain the complete algorithms that will eventually use the services of the lower-level modules; consequently, testing may be incomplete or more testing may be needed later when the upper level modules are available

Sandwich Integration

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

Regression Testing

- Regression testing – used to check for defects propagated to other modules by changes made to existing program
  - Each new addition or change to base lined software may cause problems with functions that previously worked flawlessly

- Representative sample of existing test cases is used to exercise all software functions.
  - Regression testing re-executes a small subset of tests that have already been conducted
    - Ensures that changes have not propagated unintended side effects
    - Helps to ensure that changes do not introduce unintended behavior or additional errors
    - May be done manually or through the use of automated capture/playback tools

- Additional test cases focusing software functions likely to be affected by the change.
- Tests cases that focus on the changed software components.

  - Regression test suite contains three different classes of test cases
    - A representative sample of tests that will exercise all software functions
    - Additional tests that focus on software functions that are likely to be affected by the change
    - Tests that focus on the actual software components that have been changed

Smoke Testing

- Taken from the world of hardware
  - Power is applied and a technician checks for sparks, smoke, or other dramatic signs of fundamental failure
- Designed as a pacing mechanism for time-critical projects
  - Allows the software team to assess its project on a frequent basis
- Includes the following activities
  - The software is compiled and linked into a build
  - A series of breadth tests is designed to expose errors that will keep the build from properly performing its function
    - The goal is to uncover “show stopper” errors that have the highest likelihood of throwing the software project behind schedule
  - The build is integrated with other builds and the entire product is smoke tested daily
    - Daily testing gives managers and practitioners a realistic assessment of the progress of the integration testing
  - After a smoke test is completed, detailed test scripts are executed

Benefits of Smoke Testing

- Integration risk is minimized
  - Daily testing uncovers incompatibilities and show-stoppers early in the testing process, thereby reducing schedule impact
- The quality of the end-product is improved
  - Smoke testing is likely to uncover both functional errors and architectural and component-level design errors
- Error diagnosis and correction are simplified
  - Smoke testing will probably uncover errors in the newest components that were integrated
- Progress is easier to assess
  - As integration testing progresses, more software has been integrated and more has been demonstrated to work
  - Managers get a good indication that progress is being made

Test Strategies for Object-Oriented Software

Object-Oriented Unit Testing

- smallest testable unit is the encapsulated class or object
- similar to system testing of conventional software
- do not test operations in isolation from one another
- driven by class operations and state behavior, not algorithmic detail and data flow across module interface

Test Strategies for Object-Oriented Software

- With object-oriented software, you can no longer test a single operation in isolation (conventional thinking)
- Traditional top-down or bottom-up integration testing has little meaning
- Class testing for object-oriented software is the equivalent of unit testing for conventional software
  - Focuses on operations encapsulated by the class and the state behavior of the class
- Drivers can be used
  - To test operations at the lowest level and for testing whole groups of classes
  - To replace the user interface so that tests of system functionality can be conducted prior to implementation of the actual interface
- Stubs can be used
  - In situations in which collaboration between classes is required but one or more of the collaborating classes has not yet been fully implemented

Test Strategies for Object-Oriented Software (continued)

- Two different object-oriented testing strategies
  - Thread-based testing
    - 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 that no side effects occur
  - Use-based testing
    - First tests the independent classes that use very few, if any, server classes
    - Then the next layer of classes, called dependent classes, are integrated
    - This sequence of testing layer of dependent classes continues until the entire system is constructed

Validation Testing

Background

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

Acceptance Testing

- Making sure the software works correctly for intended user in his or her normal work environment.
- Alpha testing
  - Conducted at the developer's site by end users
  - Software is used in a natural setting with developers watching intently
  - Testing is conducted in a controlled environment
- Beta testing
  - Conducted at end-user sites
  - Developer is generally not present
  - It serves as a live application of the software in an environment that cannot be controlled by the developer
  - The end-user records all problems that are encountered and reports these to the developers at regular intervals
- After beta testing is complete, software engineers make software modifications and prepare for release of the software product to the entire customer base

Acceptance Testing

- Making sure the software works correctly for intended user in his or her normal work environment.
- Alpha test – version of the complete software is tested by customer under the supervision of the developer at the developer's site
- Beta test – version of the complete software is tested by customer at his or her own site without the developer being present

System Testing

- ✓Series of tests whose purpose is to exercise a computer-based system
- ✓The focus of these system tests cases identify interfacing errors

Different Types

- Recovery testing
  - Tests for recovery from system faults
  - Forces the software to fail in a variety of ways and verifies that recovery is properly performed
  - Tests reinitialization, checkpointing mechanisms, data recovery, and restart for correctness
- Security testing
  - Verifies that protection mechanisms built into a system will, in fact, protect it from improper access
- Stress testing
  - Executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- Performance testing
  - Tests the run-time performance of software within the context of an integrated system
  - Often coupled with stress testing and usually requires both hardware and software instrumentation
  - Can uncover situations that lead to degradation and possible system failure

END

Software Project Planning

Software Project Planning

After the finalization of SRS, we would like to estimate size, cost and development time of the project. Also, in many cases, customer may like to know the cost and development time even prior to finalization of the SRS.

Software Project Planning

In order to conduct a successful software project, we must understand:

- Scope of work to be done
- The risk to be incurred
- The resources required
- The task to be accomplished
- The cost to be expended
- The schedule to be followed

Software Project Planning

Software planning begins before technical work starts, continues as the software evolves from concept to reality, and culminates only when the software is retired.

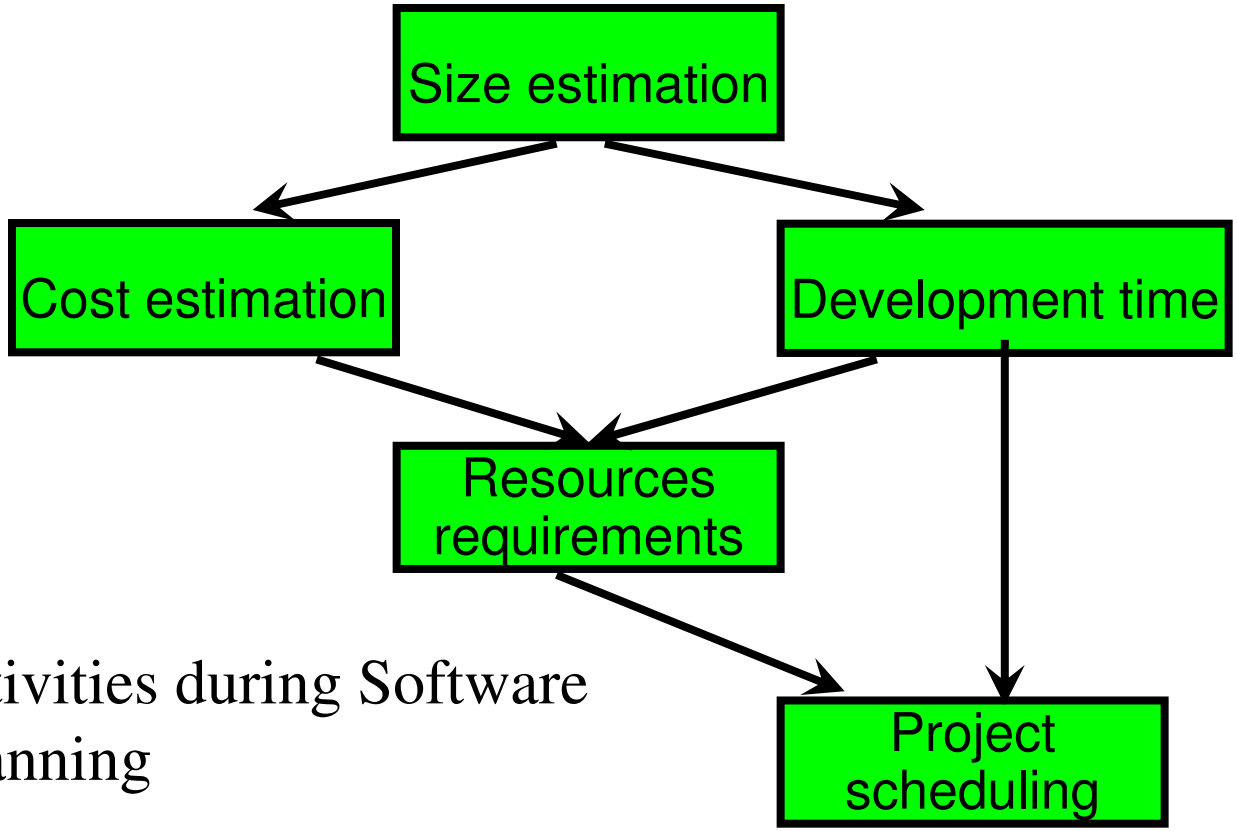


Fig. 1: Activities during Software Project Planning

Size Estimation

Lines of Code (LOC)

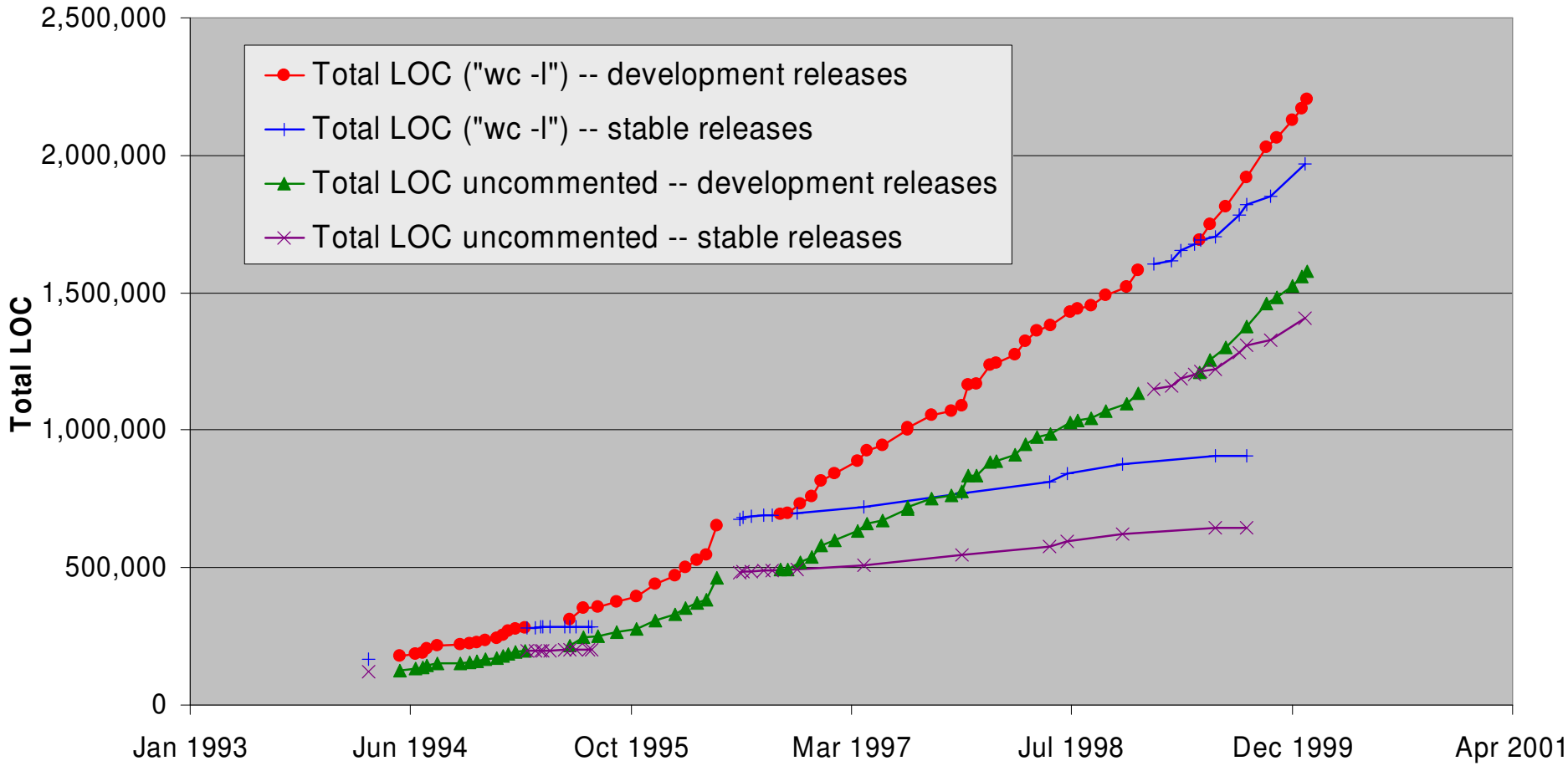
If LOC is simply a count of the number of lines then figure shown below contains 18 LOC .

When comments and blank lines are ignored, the program in figure 2 shown below contains 17 LOC.

Fig. 2: Function for sorting an array

```
1.  int. sort (int x[ ], int n)
2.  {
3.  int i, j, save, im1;
4.  /*This function sorts array x in ascending order */
5.  If (n<2) return 1;
6.  for (i=2; i<=n; i++)
7.  {
8.  im1=i-1;
9.  for (j=1; j<=im; j++)
10. if (x[i] < x[j])
11. {
12. Save = x[i];
13. x[i] = x[j];
14. x[j] = save;
15. }
16. }
17. return 0;
18. }
```

Growth of Lines of Code (LOC)



Furthermore, if the main interest is the size of the program for specific functionality, it may be reasonable to include executable statements. The only executable statements in figure shown above are in lines 5-17 leading to a count of 13. The differences in the counts are 18 to 17 to 13. One can easily see the potential for major discrepancies for large programs with many comments or programs written in language that allow a large number of descriptive but non-executable statement. Conte has defined lines of code as:

Software Project Planning

"A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program header, declaration, and executable and non-executable statements".

This is the predominant definition for lines of code used by researchers. By this definition, figure shown above has 17 LOC.

Function Count

Alan Albrecht while working for IBM, recognized the problem in size measurement in the 1970s, and developed a technique (which he called Function Point Analysis), which appeared to be a solution to the size measurement problem.

Software Project Planning

The principle of Albrecht's function point analysis (FPA) is that a system is decomposed into functional units.

- Inputs : information entering the system
- Outputs : information leaving the system
- Enquiries : requests for instant access to information
- Internal logical files : information held within the system
- External interface files : information held by other system that is used by the system being analyzed.

The FPA functional units are shown in figure given below:

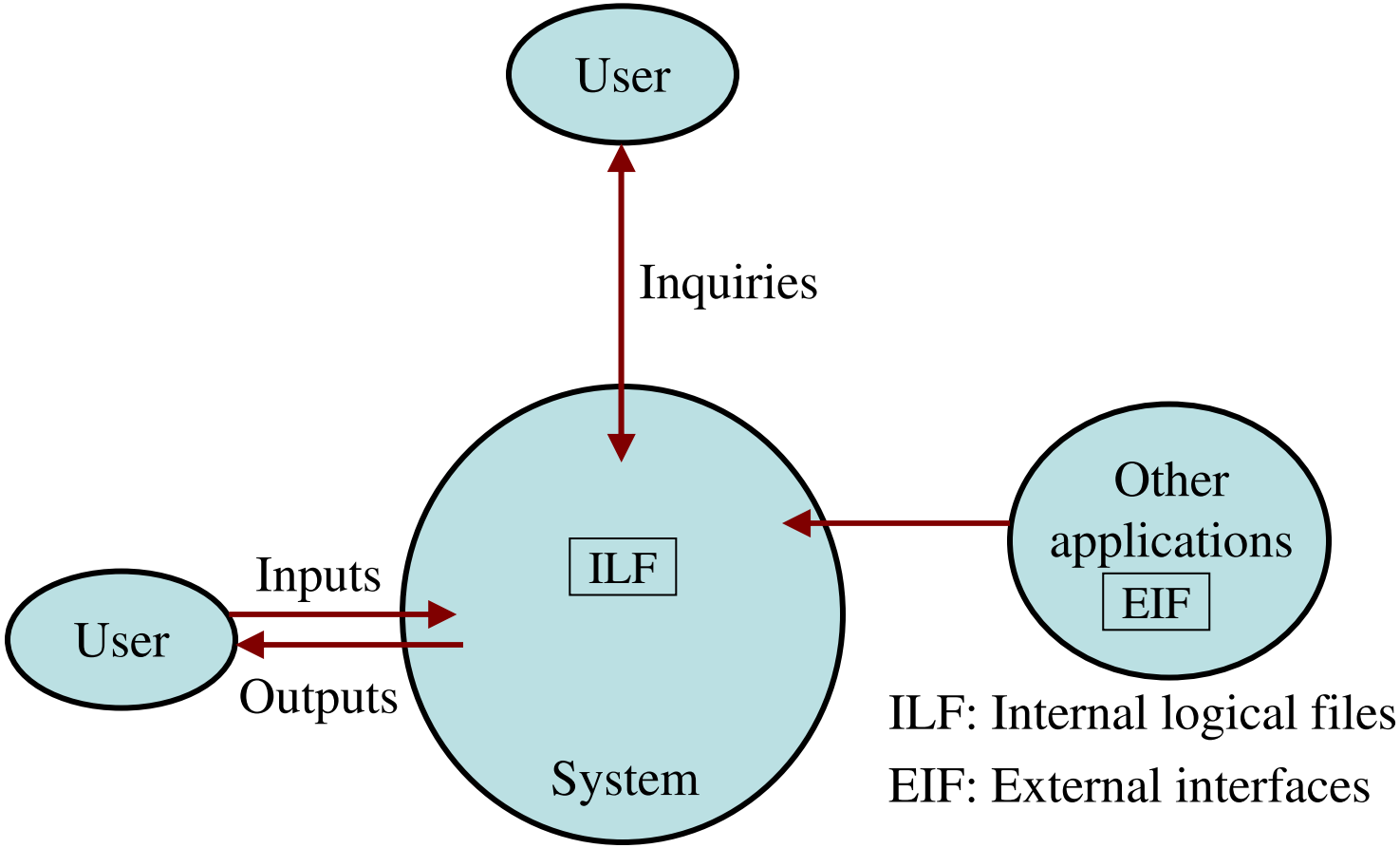


Fig. 3: FPAs functional units System

Software Project Planning

The five functional units are divided in two categories:

(i) Data function types

- Internal Logical Files (ILF): A user identifiable group of logical related data or control information maintained within the system.
- External Interface files (EIF): A user identifiable group of logically related data or control information referenced by the system, but maintained within another system. This means that EIF counted for one system, may be an ILF in another system.

(ii) Transactional function types

- External Input (EI): An EI processes data or control information that comes from outside the system. The EI is an elementary process, which is the smallest unit of activity that is meaningful to the end user in the business.
- External Output (EO): An EO is an elementary process that generate data or control information to be sent outside the system.
- External Inquiry (EQ): An EQ is an elementary process that is made up to an input-output combination that results in data retrieval.

Software Project Planning

Special features

- Function point approach is independent of the language, tools, or methodologies used for implementation; i.e. they do not take into consideration programming languages, data base management systems, processing hardware or any other data base technology.
- Function points can be estimated from requirement specification or design specification, thus making it possible to estimate development efforts in early phases of development.

Software Project Planning

- Function points are directly linked to the statement of requirements; any change of requirements can easily be followed by a re-estimate.
- Function points are based on the system user's external view of the system, non-technical users of the software system have a better understanding of what function points are measuring.

Counting function points

Functional Units	Weighting factors		
	Low	Average	High
External Inputs (EI)	3	4	6
External Output (EO)	4	5	7
External Inquiries (EQ)	3	4	6
External logical files (ILF)	7	10	15
External Interface files (EIF)	5	7	10

Table 1 : Functional units with weighting factors

Software Project Planning

Table 2: UFP calculation table

Functional Units	Count Complexity			Complexity Totals	Functional Unit Totals
External Inputs (EIs)		Low x 3	=		
		Average x 4	=		
		High x 6	=		
External Outputs (EOs)		Low x 4	=		
		Average x 5	=		
		High x 7	=		
External Inquiries (EQs)		Low x 3	=		
		Average x 4	=		
		High x 6	=		
External logical Files (ILFs)		Low x 7	=		
		Average x 10	=		
		High x 15	=		
External Interface Files (EIFs)		Low x 5	=		
		Average x 7	=		
		High x 10	=		
Total Unadjusted Function Point Count					

The weighting factors are identified for all functional units and multiplied with the functional units accordingly. The procedure for the calculation of Unadjusted Function Point (UFP) is given in table shown above.

Software Project Planning

The procedure for the calculation of UFP in mathematical form is given below:

$$UFP = \sum_{i=1}^{5} \sum_{J=1}^{3} Z_{ij} w_{ij}$$

Where i indicate the row and j indicates the column of Table 1

W_{ij} : It is the entry of the ith row and j^{th} column of the table 1

Zij : It is the count of the number of functional units of Type *i* that have been classified as having the complexity corresponding to column *j*.

Software Project Planning

Organizations that use function point methods develop a criterion for determining whether a particular entry is Low, Average or High. Nonetheless, the determination of complexity is somewhat subjective.

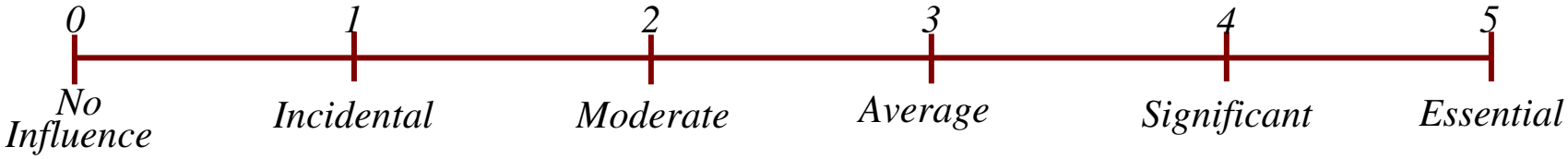
$$FP = UFP * CAF$$

Where CAF is complexity adjustment factor and is equal to $[0.65 + 0.01 \times \Sigma F_i]$. The F_i (i=1 to 14) are the degree of influence and are based on responses to questions noted in table 3.

Software Project Planning

Table 3 : Computing function points.

Rate each factor on a scale of 0 to 5.



Number of factors considered (F_i)

1. Does the system require reliable backup and recovery ?
2. Is data communication required ?
3. Are there distributed processing functions ?
4. Is performance critical ?
5. Will the system run in an existing heavily utilized operational environment ?
6. Does the system require on line data entry ?
7. Does the on line data entry require the input transaction to be built over multiple screens or operations ?
8. Are the master files updated on line ?
9. Is the inputs, outputs, files, or inquiries complex ?
10. Is the internal processing complex ?
11. Is the code designed to be reusable ?
12. Are conversion and installation included in the design ?
13. Is the system designed for multiple installations in different organizations ?
14. Is the application designed to facilitate change and ease of use by the user ?

Software Project Planning

Functions points may compute the following important metrics:

Productivity	=	FP / persons-months
Quality	=	Defects / FP
Cost	=	Rupees / FP
Documentation	=	Pages of documentation per FP

These metrics are controversial and are not universally acceptable. There are standards issued by the International Functions Point User Group (IFPUG, covering the Albrecht method) and the United Kingdom Function Point User Group (UFPGU, covering the MK11 method). An ISO standard for function point method is also being developed.

Example: 4.1

Consider a project with the following functional units:

Number of user inputs	= 50
Number of user outputs	= 40
Number of user enquiries	= 35
Number of user files	= 06
Number of external interfaces	= 04

Assume all complexity adjustment factors and weighting factors are average. Compute the function points for the project.

Solution

We know

$$UFP = \sum_{i=1}^{5} \sum_{J=1}^{3} Z_{ij} w_{ij}$$

UFP = 50 x 4 + 40 x 5 + 35 x 4 + 6 x 10 + 4 x 7

= 200 + 200 + 140 + 60 + 28 = 628

CAF = $(0.65 + 0.01\ \Sigma F_i)$

= (0.65 + 0.01 (14 x 3)) = 0.65 + 0.42 = 1.07

FP = UFP x CAF

= 628 x 1.07 = 672

Example:4.2

An application has the following:

10 low external inputs, 12 high external outputs, 20 low internal logical files, 15 high external interface files, 12 average external inquiries, and a value of complexity adjustment factor of 1.10.

What are the unadjusted and adjusted function point counts ?

Solution

Unadjusted function point counts may be calculated using as:

$$UFP = \sum_{i=1}^{5} \sum_{J=1}^{3} Z_{ij} w_{ij}$$

= 10 x 3 + 12 x 7 + 20 x 7 + 15 + 10 + 12 x 4

= 30 + 84 +140 + 150 + 48

= 452

FP = UFP x CAF

= 452 x 1.10 = 497.2.

Software Project Planning

Example: 4.3

Consider a project with the following parameters.

(*i*) External Inputs:

(a) 10 with low complexity

(b) 15 with average complexity

(c) 17 with high complexity

(*ii*) External Outputs:

(a) 6 with low complexity

(b) 13 with high complexity

(*iii*) External Inquiries:

(a) 3 with low complexity

(b) 4 with average complexity

(c) 2 high complexity

Software Project Planning

(*iv*) Internal logical files:

(a) 2 with average complexity

(b) 1 with high complexity

(v) External Interface files:

(a) 9 with low complexity

In addition to above, system requires

i. Significant data communication

ii. Performance is very critical

iii. Designed code may be moderately reusable

iv. System is not designed for multiple installation in different organizations.

Other complexity adjustment factors are treated as average. Compute the function points for the project.

Software Project Planning

Solution: Unadjusted function points may be counted using table 2

Functional Units	Count	Complexity		Complexity Totals	Functional Unit Totals
External Inputs (EIs)	10	Low x 3	=	30	
	15	Average x 4	=	60	
	17	High x 6	=	102	192
External Outputs (EOs)	6	Low x 4	=	24	
	0	Average x 5	=	0	
	13	High x 7	=	91	115
External Inquiries (EQs)	3	Low x 3	=	9	
	4	Average x 4	=	16	
	2	High x 6	=	12	37
External logical Files (ILFs)	0	Low x 7	=	0	
	2	Average x 10	=	20	
	1	High x 15	=	15	35
External Interface Files (EIFs)	9	Low x 5	=	45	
	0	Average x 7	=	0	
	0	High x 10	=	0	45
Total Unadjusted Function Point Count					424

Software Project Planning

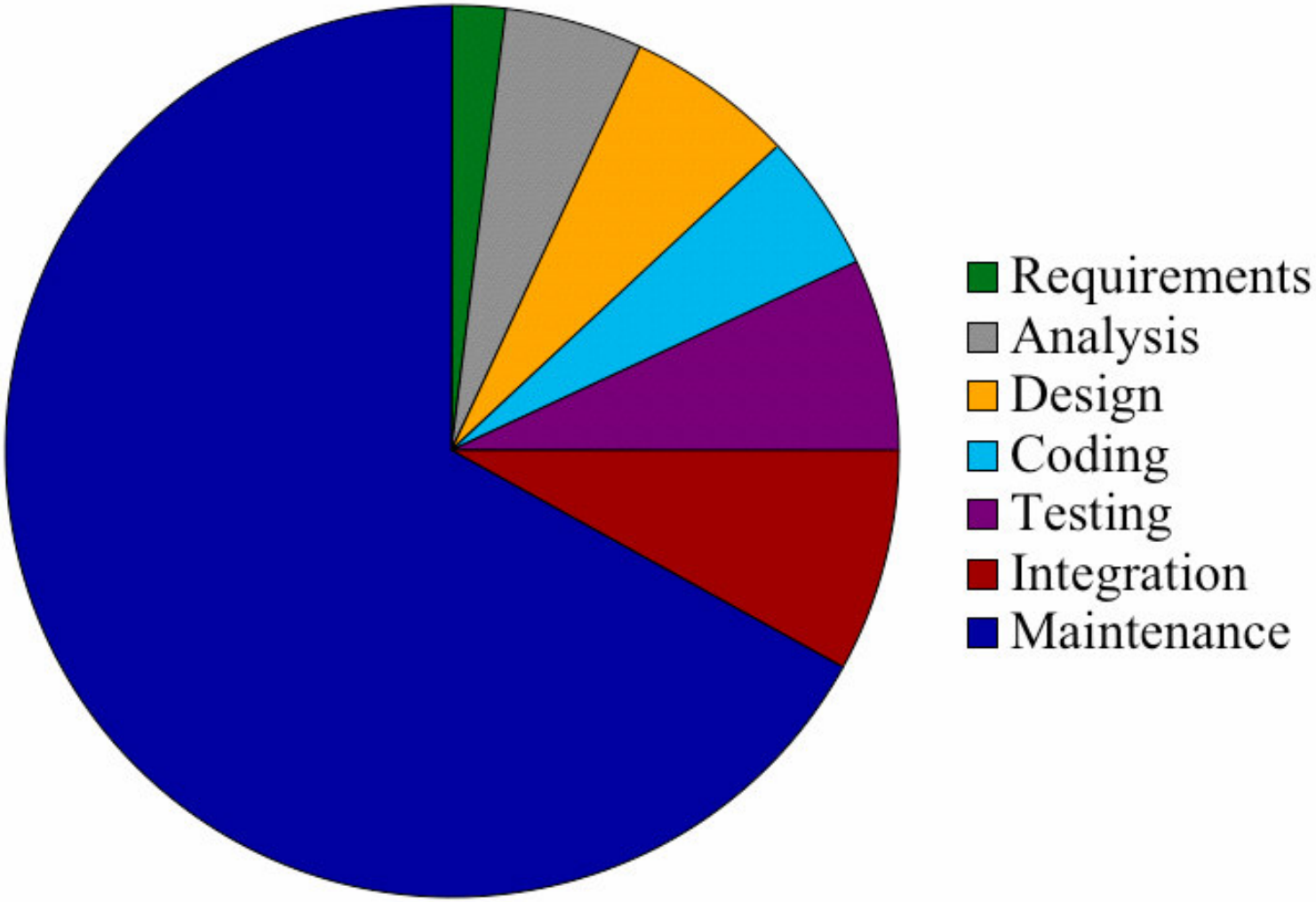
$$\sum_{i=1}^{14} F_i = 3+4+3+5+3+3+3+3+3+3+2+3+0+3=41$$

$$\begin{aligned} CAF &= (0.65 + 0.01 \text{ x } \Sigma F_i) \\ &= (0.65 + 0.01 \text{ x } 41) \\ &= 1.06 \\ FP &= UFP \text{ x } CAF \\ &= 424 \text{ x } 1.06 \\ &= 449.44 \end{aligned}$$

Hence $\boxed{FP = 449}$

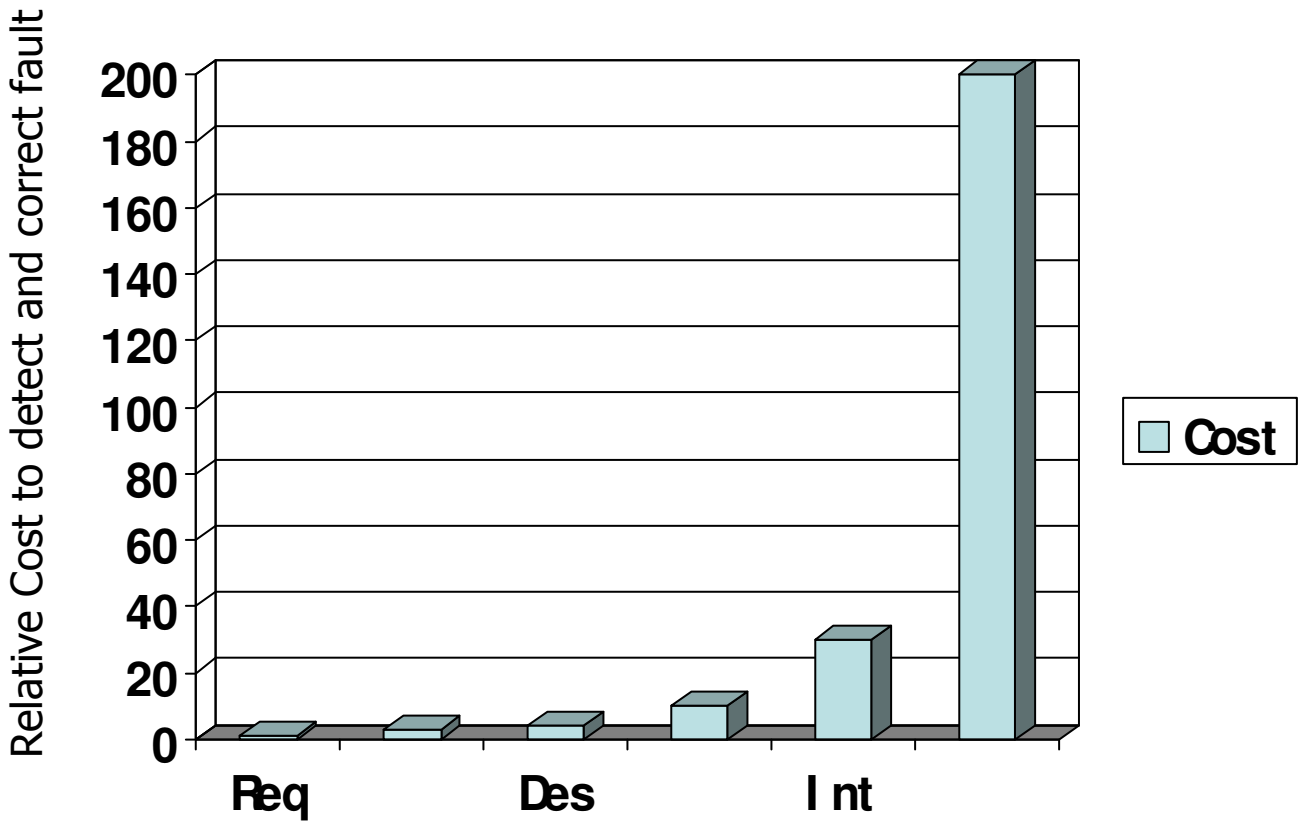
Software Project Planning

Relative Cost of Software Phases



Software Project Planning

Cost to Detect and Fix Faults



Cost Estimation

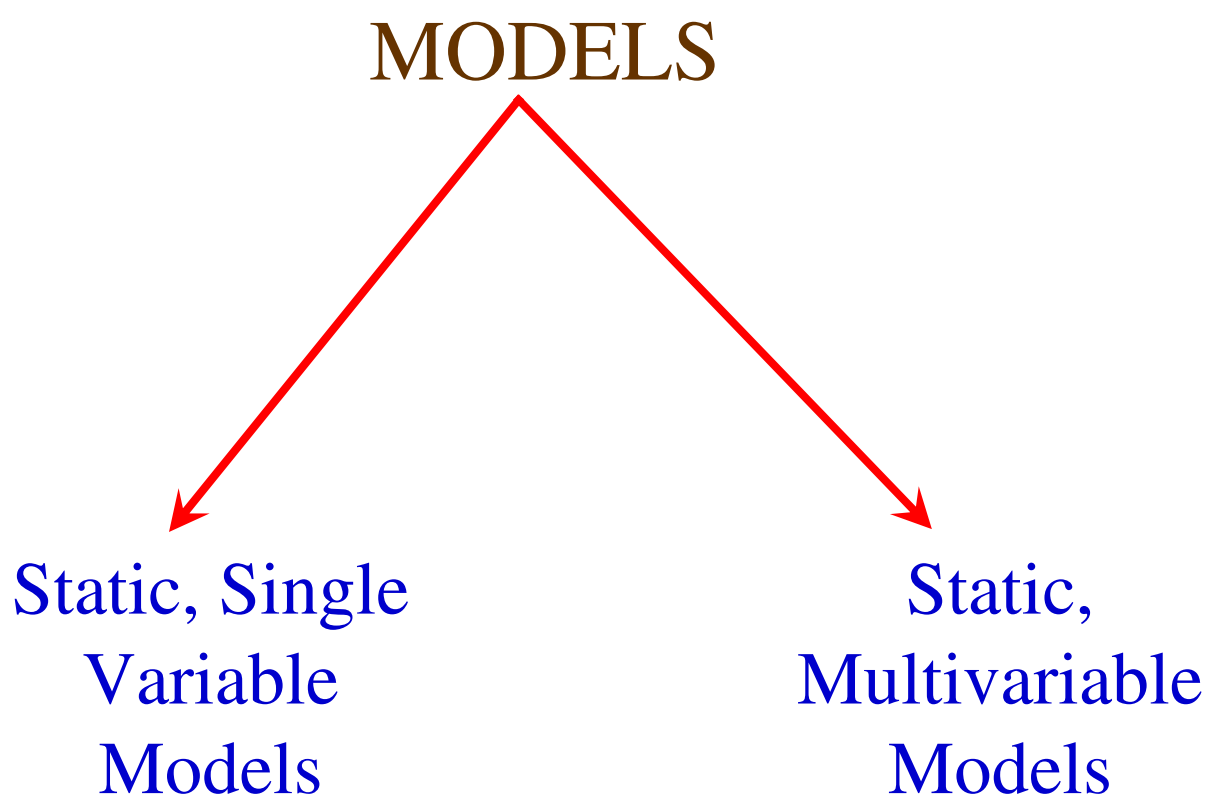
A number of estimation techniques have been developed and are having following attributes in common :

- Project scope must be established in advance
- Software metrics are used as a basis from which estimates are made
- The project is broken into small pieces which are estimated individually

To achieve reliable cost and schedule estimates, a number of options arise:

- Delay estimation until late in project
- Use simple decomposition techniques to generate project cost and schedule estimates
- Develop empirical models for estimation
- Acquire one or more automated estimation tools

Software Project Planning



Static, Single Variable Models

Methods using this model use an equation to estimate the desired values such as cost, time, effort, etc. They all depend on the same variable used as predictor (say, size). An example of the most common equations is :

$$\mathbf{C = a\,L^{b}} \qquad \text{(i)}$$

C is the cost, L is the size and a,b are constants

$$E = 1.4\,L^{0.93}$$

$$DOC = 30.4\,L^{0.90}$$

$$D = 4.6\,L^{0.26}$$

Effort (E in Person-months), documentation (DOC, in number of pages) and duration (D, in months) are calculated from the number of lines of code (L, in thousands of lines) used as a predictor.

Static, Multivariable Models

These models are often based on equation (i), they actually depend on several variables representing various aspects of the software development environment, for example method used, user participation, customer oriented changes, memory constraints, etc.

$$E = 5.2\ L^{0.91}$$

$$D = 4.1\ L^{0.36}$$

The productivity index uses 29 variables which are found to be highly correlated to productivity as follows:

$$I = \sum_{i=1}^{29} W_i X_i$$

Software Project Planning

Example: 4.4

Compare the Walston-Felix model with the SEL model on a software development expected to involve 8 person-years of effort.

(a)Calculate the number of lines of source code that can be produced.

(b)Calculate the duration of the development.

(c)Calculate the productivity in LOC/PY

(d)Calculate the average manning

Solution

The amount of manpower involved = 8 PY = 96 person-months

(a) Number of lines of source code can be obtained by reversing equation to give:

$$L = (E/a)^{1/b}$$

Then

$$L(SEL) = (96/1.4)^{1/0.93} = 94264 \text{ LOC}$$

$$L(SEL) = (96/5.2)^{1/0.91} = 24632 \text{ LOC.}$$

Software Project Planning

(b) Duration in months can be calculated by means of equation

$$D(SEL) = 4.6\ (L)^{0.26}$$

$$= 4.6\ (94.264)^{0.26} = 15 \text{ months}$$

$$D(W\text{-}F) = 4.1\ L^{0.36}$$

$$= 4.1(24.632)^{0.36} = 13 \text{ months}$$

(c) Productivity is the lines of code produced per person/month (year)

$$P(SEL) = \frac{94264}{8} = 11783\ LOC / Person - Years$$

$$P(W - F) = \frac{24632}{8} = 3079\ LOC / Person - Years$$

Software Project Planning

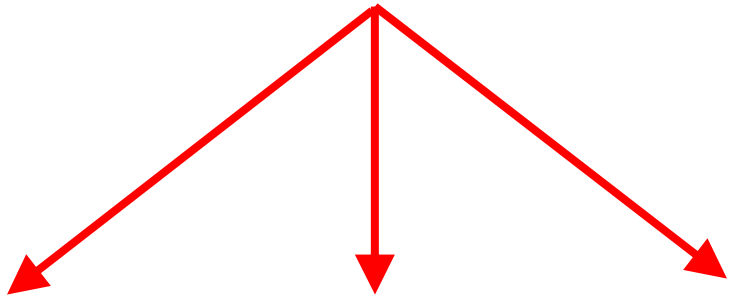
(d) Average manning is the average number of persons required per month in the project.

$$M(SEL) = \frac{96P - M}{15M} = 6.4 Persons$$

$$M(W - F) = \frac{96P - M}{13M} = 7.4 Persons$$

The Constructive Cost Model (COCOMO)

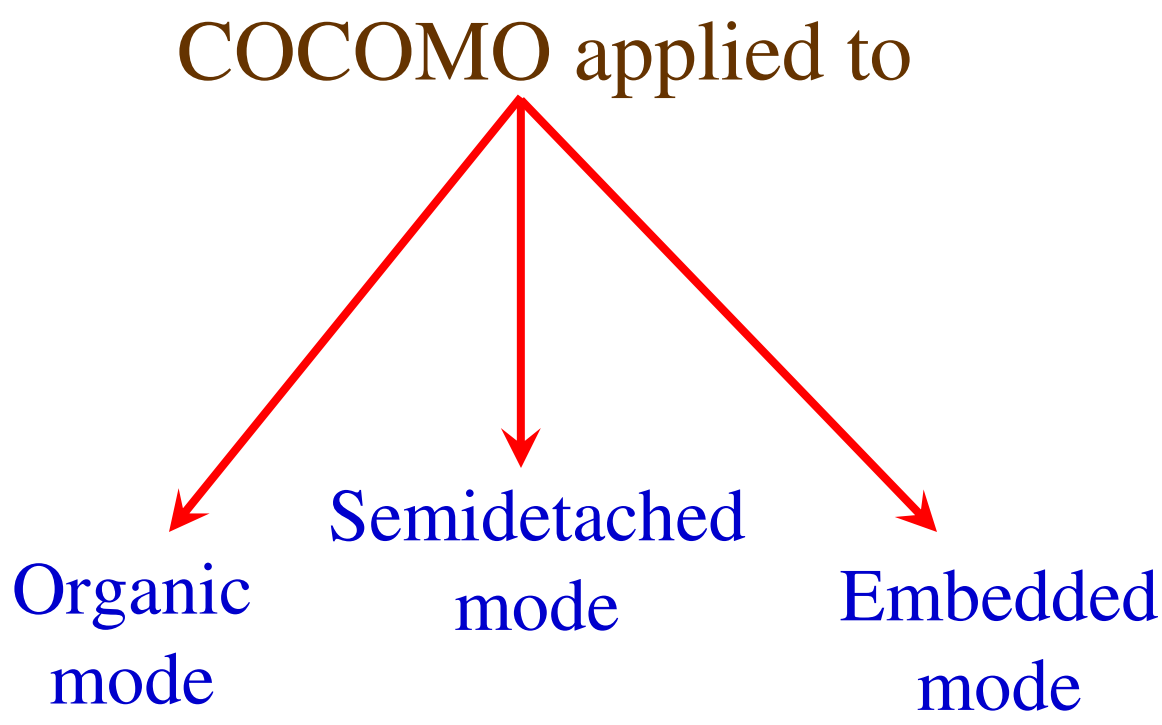
Constructive Cost model
(COCOMO)



Basic Intermediate Detailed

Model proposed by
B. W. Boehm's
through his book
Software Engineering Economics in 1981

Software Project Planning



Software Project Planning

Mode	Project size	Nature of Project	Innovation	Deadline of the project	Development Environment
Organic	Typically 2-50 KLOC	Small size project, experienced developers in the familiar environment. For example, pay roll, inventory projects etc.	Little	Not tight	Familiar & In house
Semi detached	Typically 50-300 KLOC	Medium size project, Medium size team, Average previous experience on similar project. For example: Utility systems like compilers, database systems, editors etc.	Medium	Medium	Medium
Embedded	Typically over 300 KLOC	Large project, Real time systems, Complex interfaces, Very little previous experience. For example: ATMs, Air Traffic Control etc.	Significant	Tight	Complex Hardware/ customer Interfaces required

Table 4: The comparison of three COCOMO modes

Basic Model

Basic COCOMO model takes the form

$$E = a_b (KLOC)^{b_b}$$

$$D = c_b (E)^{d_b}$$

where E is effort applied in Person-Months, and D is the development time in months. The coefficients a_b, b_b, c_b and d_b are given in table 4 (a).

Software Project Planning

Software Project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Table 4(a): Basic COCOMO coefficients

Software Project Planning

When effort and development time are known, the average staff size to complete the project may be calculated as:

$$\text{Average staff size } (SS) = \frac{E}{D} Persons$$

When project size is known, the productivity level may be calculated as:

$$\text{Productivity } (P) = \frac{KLOC}{E} KLOC / PM$$

Example: 4.5

Suppose that a project was estimated to be 400 KLOC. Calculate the effort and development time for each of the three modes i.e., organic, semidetached and embedded.

Solution

The basic COCOMO equation take the form:

$$E = a_b (KLOC)^{b_b}$$

$$D = c_b (KLOC)^{d_b}$$

Estimated size of the project = 400 KLOC

(i) Organic mode

$$E = 2.4(400)^{1.05} = 1295.31 \text{ PM}$$

$$D = 2.5(1295.31)^{0.38} = 38.07 \text{ PM}$$

Software Project Planning

(ii) Semidetached mode

$$E = 3.0(400)^{1.12} = 2462.79 \text{ PM}$$

$$D = 2.5(2462.79)^{0.35} = 38.45 \text{ PM}$$

(iii) Embedded mode

$$E = 3.6(400)^{1.20} = 4772.81 \text{ PM}$$

$$D = 2.5(4772.8)^{0.32} = 38 \text{ PM}$$

Software Project Planning

Example: 4.6

A project size of 200 KLOC is to be developed. Software development team has average experience on similar type of projects. The project schedule is not very tight. Calculate the effort, development time, average staff size and productivity of the project.

Solution

The semi-detached mode is the most appropriate mode; keeping in view the size, schedule and experience of the development team.

Hence $E = 3.0(200)^{1.12} = 1133.12$ PM

$D = 2.5(1133.12)^{0.35} = 29.3$ PM

Average staff size $(SS) = \frac{E}{D} Persons$

$$= \frac{1133.12}{29.3} = 38.67 Persons$$

Software Project Planning

$$\text{Productivity} = \frac{KLOC}{E} = \frac{200}{1133.12} = 0.1765\, KLOC / PM$$

$$P = 176\ LOC / PM$$

Intermediate Model

Cost drivers

(*i*) Product Attributes

- Required s/w reliability
- Size of application database
- Complexity of the product

(*ii*) Hardware Attributes

- Run time performance constraints
- Memory constraints
- Virtual machine volatility
- Turnaround time

Software Project Planning

(*iii*) Personal Attributes

- Analyst capability
- Programmer capability
- Application experience
- Virtual m/c experience
- Programming language experience

(*iv*) Project Attributes

- Modern programming practices
- Use of software tools
- Required development Schedule

Software Project Planning

Multipliers of different cost drivers

Cost Drivers	RATINGS					
	Very low	Low	Nominal	High	Very high	Extra high
Product Attributes						
RELY	0.75	0.88	1.00	1.15	1.40	--
DATA	--	0.94	1.00	1.08	1.16	--
CPLX	0.70	0.85	1.00	1.15	1.30	1.65
Computer Attributes						
TIME	--	--	1.00	1.11	1.30	1.66
STOR	--	--	1.00	1.06	1.21	1.56
VIRT	--	0.87	1.00	1.15	1.30	--
TURN	--	0.87	1.00	1.07	1.15	--

Software Project Planning

Cost Drivers	RATINGS					
	Very low	Low	Nominal	High	Very high	Extra high
Personnel Attributes						
ACAP	1.46	1.19	1.00	0.86	0.71	--
AEXP	1.29	1.13	1.00	0.91	0.82	--
PCAP	1.42	1.17	1.00	0.86	0.70	--
VEXP	1.21	1.10	1.00	0.90	--	--
LEXP	1.14	1.07	1.00	0.95	--	--
Project Attributes						
MODP	1.24	1.10	1.00	0.91	0.82	--
TOOL	1.24	1.10	1.00	0.91	0.83	--
SCED	1.23	1.08	1.00	1.04	1.10	--

Table 5: Multiplier values for effort calculations

Software Project Planning

Intermediate COCOMO equations

$$E = a_i (KLOC)^{b_i} * EAF$$

$$D = c_i (E)^{d_i}$$

Project	a_i	b_i	c_i	d_i
Organic	3.2	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

Table 6: Coefficients for intermediate COCOMO

Detailed COCOMO Model

Detailed COCOMO

Phase-Sensitive effort multipliers

Cost drivers

design & test

Manpower allocation for each phase

Three level product hierarchy

Modules subsystem

System level

Software Project Planning

Development Phase

Plan / Requirements

EFFORT	:	6% to 8%
DEVELOPMENT TIME	:	10% to 40%

% depend on mode & size

Software Project Planning

Design

Effort	:	16% to 18%
Time	:	19% to 38%

Programming

Effort	:	48% to 68%
Time	:	24% to 64%

Integration & Test

Effort	:	16% to 34%
Time	:	18% to 34%

Principle of the effort estimate

Size equivalent

As the software might be partly developed from software already existing (that is, re-usable code), a full development is not always required. In such cases, the parts of design document (DD%), code (C%) and integration (I%) to be modified are estimated. Then, an adjustment factor, A, is calculated by means of the following equation.

$$A = 0.4\ DD + 0.3\ C + 0.3\ I$$

The size equivalent is obtained by

$$S\ (\text{equivalent}) = (S \times A) / 100$$

$$E_p = \mu_p E$$

$$D_p = \tau_p D$$

Lifecycle Phase Values of μ_p

Mode & Code Size	Plan & Requirements	System Design	Detailed Design	Module Code & Test	Integration & Test
Organic Small S≈2	0.06	0.16	0.26	0.42	0.16
Organic medium S≈32	0.06	0.16	0.24	0.38	0.22
Semidetached medium S≈32	0.07	0.17	0.25	0.33	0.25
Semidetached large S≈128	0.07	0.17	0.24	0.31	0.28
Embedded large S≈128	0.08	0.18	0.25	0.26	0.31
Embedded extra large S≈320	0.08	0.18	0.24	0.24	0.34

Table 7 : Effort and schedule fractions occurring in each phase of the lifecycle

Lifecycle Phase Values of τ_p

Mode & Code Size	Plan & Requirements	System Design	Detailed Design	Module Code & Test	Integration & Test
Organic Small S≈2	0.10	0.19	0.24	0.39	0.18
Organic medium S≈32	0.12	0.19	0.21	0.34	0.26
Semidetached medium S≈32	0.20	0.26	0.21	0.27	0.26
Semidetached large S≈128	0.22	0.27	0.19	0.25	0.29
Embedded large S≈128	0.36	0.36	0.18	0.18	0.28
Embedded extra large S≈320	0.40	0.38	0.16	0.16	0.30

Table 7 : Effort and schedule fractions occurring in each phase of the lifecycle

Software Project Planning

Distribution of software life cycle:

1. Requirement and product design
   (a) Plans and requirements
   (b) System design
2. Detailed Design
   (a) Detailed design
3. Code & Unit test
   (a) Module code & test
4. Integrate and Test
   (a) Integrate & Test

Software Project Planning

Example: 4.7

A new project with estimated 400 KLOC embedded system has to be developed. Project manager has a choice of hiring from two pools of developers: Very highly capable with very little experience in the programming language being used

Or

Developers of low quality but a lot of experience with the programming language. What is the impact of hiring all developers from one or the other pool ?

Solution

This is the case of embedded mode and model is intermediate COCOMO.

Hence $$E = a_i (KLOC)^{d_i}$$

$$= 2.8\ (400)^{1.20} = 3712 \text{ PM}$$

Case I: Developers are very highly capable with very little experience in the programming being used.

EAF = 0.82 x 1.14 = 0.9348

E = 3712 x .9348 = 3470 PM

D = $2.5\ (3470)^{0.32}$ = 33.9 M

Software Project Planning

Case II: Developers are of low quality but lot of experience with the programming language being used.

EAF = 1.29 x 0.95 = 1.22

E = 3712 x 1.22 = 4528 PM

D = $2.5\ (4528)^{0.32}$ = 36.9 M

Case II requires more effort and time. Hence, low quality developers with lot of programming language experience could not match with the performance of very highly capable developers with very litter experience.

Software Project Planning

Example: 4.8

Consider a project to develop a full screen editor. The major components identified are:

I. Screen edit

II. Command Language Interpreter

III. File Input & Output

IV. Cursor Movement

V. Screen Movement

The size of these are estimated to be 4k, 2k, 1k, 2k and 3k delivered source code lines. Use COCOMO to determine

1. Overall cost and schedule estimates (assume values for different cost drivers, with at least three of them being different from 1.0)
2. Cost & Schedule estimates for different phases.

Solution

Size of five modules are:

Screen edit	= 4 KLOC
Command language interpreter	= 2 KLOC
File input and output	= 1 KLOC
Cursor movement	= 2 KLOC
Screen movement	= 3 KLOC
Total	**= 12 KLOC**

Software Project Planning

Let us assume that significant cost drivers are

i. Required software reliability is high, i.e.,1.15

ii. Product complexity is high, i.e.,1.15

iii. Analyst capability is high, i.e.,0.86

iv. Programming language experience is low,i.e.,1.07

v. All other drivers are nominal

EAF = 1.15x1.15x0.86x1.07 = 1.2169

Software Project Planning

(a) The initial effort estimate for the project is obtained from the following equation

$$E = a_i\,(KLOC)^{bi} \times EAF$$

$$= 3.2(12)^{1.05} \times 1.2169 = 52.91 \text{ PM}$$

Development time $D = C_i(E)^{di}$

$$= 2.5(52.91)^{0.38} = 11.29 \text{ M}$$

(b) Using the following equations and referring Table 7, phase wise cost and schedule estimates can be calculated.

$$E_p = \mu_p E$$

$$D_p = \tau_p D$$

Software Project Planning

Since size is only 12 KLOC, it is an organic small model. Phase wise effort distribution is given below:

System Design	= 0.16 x 52.91 = 8.465 PM
Detailed Design	= 0.26 x 52.91 = 13.756 PM
Module Code & Test	= 0.42 x 52.91 = 22.222 PM
Integration & Test	= 0.16 x 52.91 = 8.465 Pm

Now Phase wise development time duration is

System Design	= 0.19 x 11.29 = 2.145 M
Detailed Design	= 0.24 x 11.29 = 2.709 M
Module Code & Test	= 0.39 x 11.29 = 4.403 M
Integration & Test	= 0.18 x 11.29 = 2.032 M

COCOMO-II

The following categories of applications / projects are identified by COCOMO-II and are shown in fig. 4 shown below:

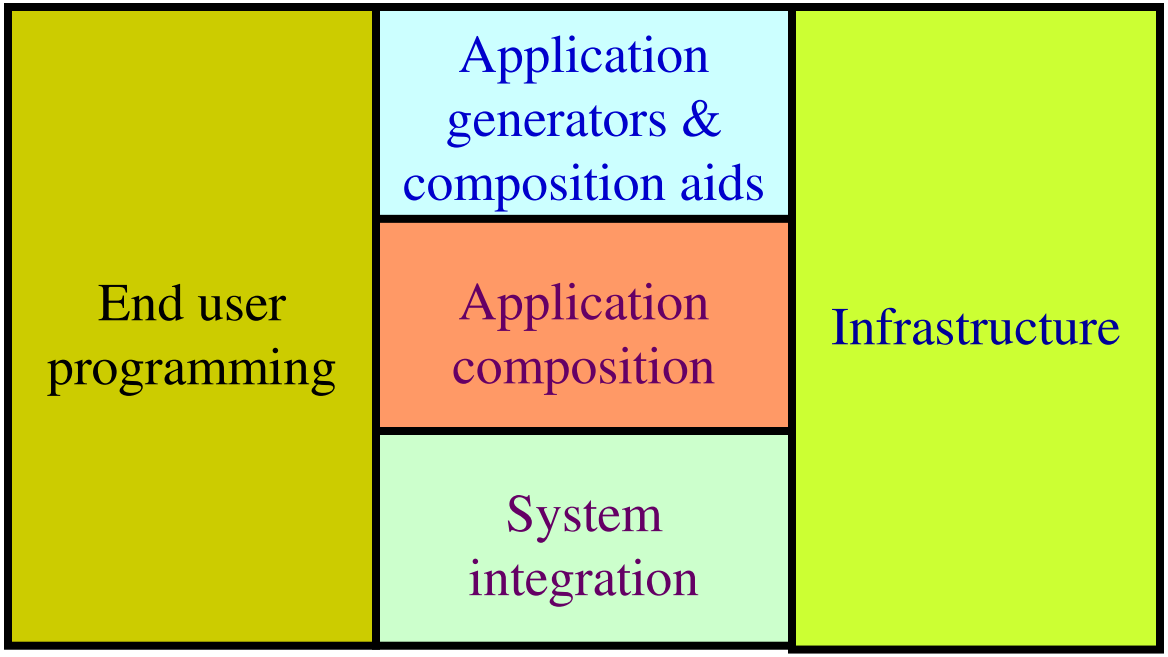


Fig. 4 : Categories of applications / projects

Software Project Planning

Stage No	Model Name	Application for the types of projects	Applications
Stage I	Application composition estimation model	Application composition	In addition to application composition type of projects, this model is also used for prototyping (if any) stage of application generators, infrastructure & system integration.
Stage II	Early design estimation model	Application generators, infrastructure & system integration	Used in early design stage of a project, when less is known about the project.
Stage III	Post architecture estimation model	Application generators, infrastructure & system integration	Used after the completion of the detailed architecture of the project.

Table 8: Stages of COCOMO-II

Application Composition Estimation Model

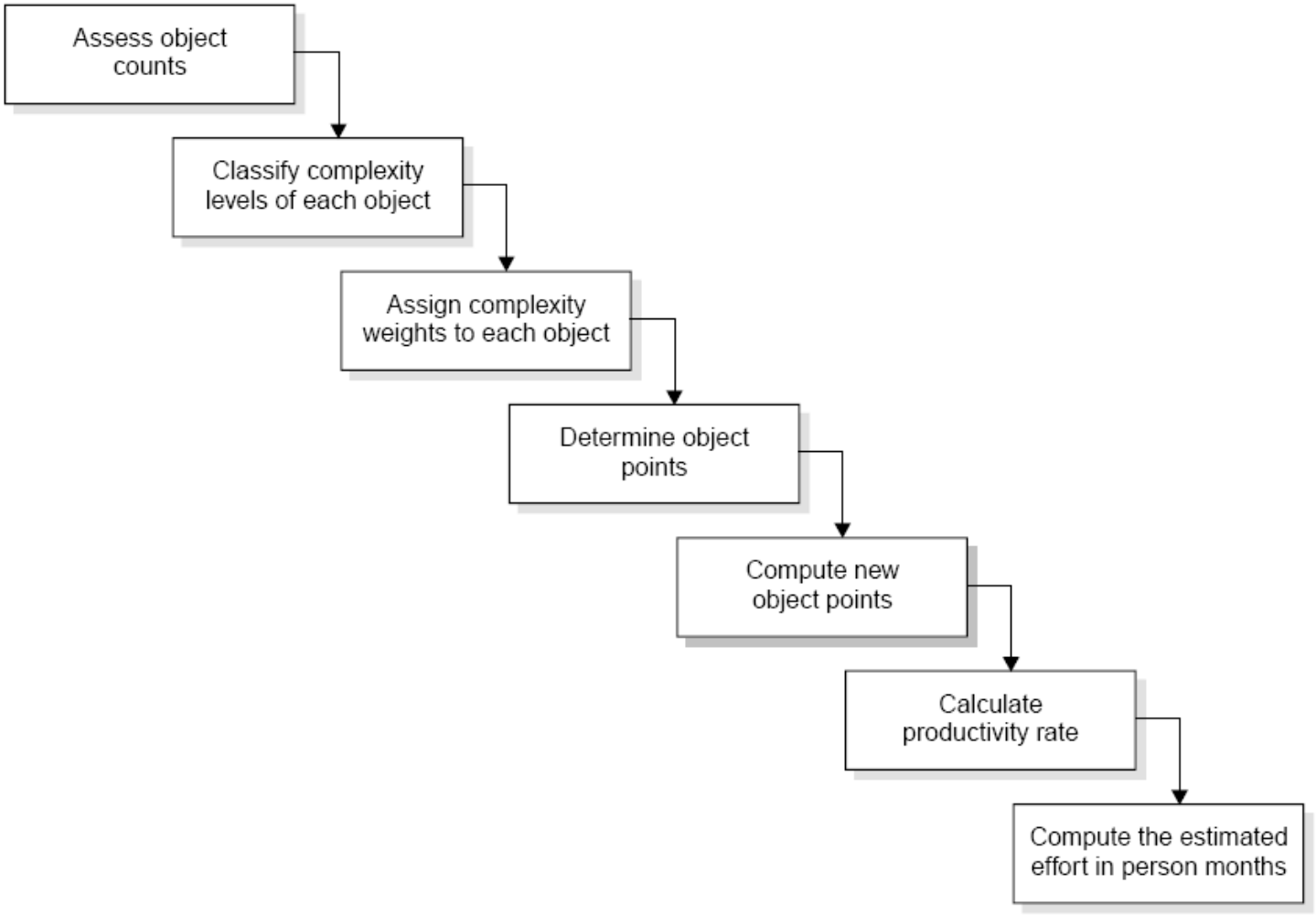


Fig.5: Steps for the estimation of effort in person months

Software Engineering (3rd ed.), By K.K Aggarwal & Yogesh Singh, Copyright © New Age International Publishers, 2007

Software Project Planning

i. Assess object counts: Estimate the number of screens, reports and 3 GL components that will comprise this application.

ii. Classification of complexity levels: We have to classify each object instance into simple, medium and difficult complexity levels depending on values of its characteristics.

Number of views contained	*# and sources of data tables*		
	Total < 4 (< 2 server < 3 client)	*Total < 8 (2 – 3 server 3 – 5 client)*	*Total 8 + (> 3 server, > 5 client)*
< 3	Simple	Simple	Medium
3 – 7	Simple	Medium	Difficult
> 8	Medium	Difficult	Difficult

Table 9 (a): For screens

Software Project Planning

Number of sections contained	# and sources of data tables		
	Total < 4 (< 2 server < 3 client)	Total < 8 (2 – 3 server 3 – 5 client)	Total 8 + (> 3 server, > 5 client)
0 or 1	Simple	Simple	Medium
2 or 3	Simple	Medium	Difficult
4 +	Medium	Difficult	Difficult

Table 9 (b): For reports

Software Project Planning

iii. Assign complexity weight to each object : The weights are used for three object types i.e., screen, report and 3GL components using the Table 10.

Object Type	*Complexity Weight*		
	Simple	*Medium*	*Difficult*
Screen	1	2	3
Report	2	5	8
3GL Component	—	—	10

Table 10: Complexity weights for each level

Software Project Planning

iv. Determine object points: Add all the weighted object instances to get one number and this known as object-point count.

v. Compute new object points: We have to estimate the percentage of reuse to be achieved in a project. Depending on the percentage reuse, the new object points (NOP) are computed.

$$\textbf{NOP} = \frac{(\text{object points}) * (100 - \%\text{reuse})}{100}$$

NOP are the object points that will need to be developed and differ from the object point count because there may be reuse.

Software Project Planning

vi. Calculation of productivity rate: The productivity rate can be calculated as:

Productivity rate (PROD) = NOP/Person month

Developer's experience & capability; ICASE maturity & capability	*PROD (NOP/PM)*
Very low	4
Low	7
Nominal	13
High	25
Very high	50

Table 11: Productivity values

Software Project Planning

vii. Compute the effort in Persons-Months: When PROD is known, we may estimate effort in Person-Months as:

$$\textbf{Effort in PM} = \frac{NOP}{PROD}$$

Software Project Planning

Example: 4.9

Consider a database application project with the following characteristics:

I. The application has 4 screens with 4 views each and 7 data tables for 3 servers and 4 clients.

II. The application may generate two report of 6 sections each from 07 data tables for two server and 3 clients. There is 10% reuse of object points.

The developer's experience and capability in the similar environment is low. The maturity of organization in terms of capability is also low. Calculate the object point count, New object points and effort to develop such a project.

Solution

This project comes under the category of application composition estimation model.

Number of screens = 4 with 4 views each

Number of reports = 2 with 6 sections each

From Table 9 we know that each screen will be of medium complexity and each report will be difficult complexity.

Using Table 10 of complexity weights, we may calculate object point count.

$$= 4 \times 2 + 2 \times 8 = 24$$

$$\textbf{NOP} = \frac{24 * (100 - 10)}{100} = 21.6$$

Software Project Planning

Table 11 gives the low value of productivity (PROD) i.e. 7.

$$\text{Efforts in PM} = \frac{\text{NOP}}{\text{PROD}}$$

$$\text{Efforts} = \frac{21.6}{7} = 3.086 \text{ PM}$$

The Early Design Model

The COCOMO-II models use the base equation of the form

$$PM_{nominal} = A * (size)^B$$

where

$PM_{nominal}$ = Effort of the project in person months

A = Constant representing the nominal productivity, provisionally set to 2.5

B = Scale factor

Size = Software size

Software Project Planning

Scale factor	Explanation	Remarks
Precedentness	Reflects the previous experience on similar projects. This is applicable to individuals & organization both in terms of expertise & experience	Very low means no previous experiences, Extra high means that organization is completely familiar with this application domain.
Development flexibility	Reflect the degree of flexibility in the development process.	Very low means a well defined process is used. Extra high means that the client gives only general goals.
Architecture/ Risk resolution	Reflect the degree of risk analysis carried out.	Very low means very little analysis and Extra high means complete and through risk analysis.

Cont…

Table 12: Scaling factors required for the calculation of the value of B

Scale factor	Explanation	Remarks
Team cohesion	Reflects the team management skills.	Very low means no previous experiences, Extra high means that organization is completely familiar with this application domain.
Process maturity	Reflects the process maturity of the organization. Thus it is dependent on SEI-CMM level of the organization.	Very low means organization has no level at all and extra high means organization is related as highest level of SEI-CMM.

Table 12: Scaling factors required for the calculation of the value of B

Software Project Planning

Scaling factors	Very low	Low	Nominal	High	Very high	Extra high
Precedent ness	6.20	4.96	3.72	2.48	1.24	0.00
Development flexibility	5.07	4.05	3.04	2.03	1.01	0.00
Architecture/ Risk resolution	7.07	5.65	4.24	2.83	1.41	0.00
Team cohesion	5.48	4.38	3.29	2.19	1.10	0.00
Process maturity	7.80	6.24	4.68	3.12	1.56	0.00

Table 13: Data for the Computation of B

The value of B can be calculated as:

B=0.91 + 0.01 * (Sum of rating on scaling factors for the project)

Early design cost drivers

There are seven early design cost drivers and are given below:

i. Product Reliability and Complexity (RCPX)

ii. Required Reuse (RUSE)

iii. Platform Difficulty (PDIF)

iv. Personnel Capability (PERS)

v. Personnel Experience (PREX)

vi. Facilities (FCIL)

vii. Schedule (SCED)

Post architecture cost drivers

There are 17 cost drivers in the Post Architecture model. These are rated on a scale of 1 to 6 as given below :

Very Low	*Low*	*Nominal*	*High*	*Very High*	*Extra High*
1	2	3	4	5	6

The list of seventeen cost drivers is given below :

i. Reliability Required (RELY)

ii. Database Size (DATA)

iii. Product Complexity (CPLX)

iv. Required Reusability (RUSE)

Software Project Planning

v. Documentation (DOCU)

vi. Execution Time Constraint (TIME)

vii. Main Storage Constraint (STOR)

viii. Platform Volatility (PVOL)

ix. Analyst Capability (ACAP)

x. Programmers Capability (PCAP)

xi. Personnel Continuity (PCON)

xii. Analyst Experience (AEXP)

Software Project Planning

xiii. Programmer Experience (PEXP)

xiv. Language & Tool Experience (LTEX)

xv. Use of Software Tools (TOOL)

xvi. Site Locations & Communication Technology between Sites (SITE)

xvii. Schedule (SCED)

Mapping of early design cost drivers and post architecture cost drivers

The 17 Post Architecture Cost Drivers are mapped to 7 Early Design Cost Drivers and are given in Table 14

Early Design Cost Drivers	Counter part Combined Post Architecture Cost drivers
RCPX	RELY, DATA, CPLX, DOCU
RUSE	RUSE
PDIF	TIME, STOR, PVOL
PERS	ACAP, PCAP, PCON
PREX	AEXP, PEXP, LTEX
FCIL	TOOL, SITE
SCED	SCED

Table 14: Mapping table

Software Engineering (3rd ed.), By K.K Aggarwal & Yogesh Singh, Copyright © New Age International Publishers, 2007

Software Project Planning

Product of cost drivers for early design model

i. Product Reliability and Complexity (RCPX): The cost driver combines four Post Architecture cost drivers which are RELY, DATA, CPLX and DOCU.

RCPX	*Extra Low*	*Very Low*	*Low*	*Nominal*	*High*	*Very High*	*Extra High*
Sum of RELY, DATA, CPLX, DOCU ratings	5, 6	7, 8	9-11	12	13-15	16-18	19-21
Emphasis on reliability, documentation	Very Little	Little	Some	Basic	Strong	Very Strong	Extreme
Product complexity	Very Simple	Simple	Some	Moderate	Complex	Very Complex	Extremely Complex
Database size	Small	Small	Small	Moderate	Large	Very Large	Very Large

Software Project Planning

ii. Required Reuse (RUSE) : This early design model cost driver is same as its Post architecture Counterpart. The RUSE rating levels are (as per Table 16):

	Vary Low	*Low*	*Nominal*	*High*	*Very High*	*Extra High*
RUSE	1	2	3	4	5	6
		None	Across project	Across program	Across product line	Across multiple product line

Software Project Planning

iii. Platform Difficulty (PDIF) : This cost driver combines TIME, STOR and PVOL of Post Architecture Cost Drivers.

PDIF	*Low*	*Nominal*	*High*	*Very High*	*Extra High*
Sum of Time, STOR & PVOL ratings	8	9	10-12	13-15	16-17
Time & storage constraint	≤ 50%	≤ 50%	65%	80%	90%
Platform Volatility	Very stable	Stable	Somewhat stable	Volatile	Highly Volatile

Software Project Planning

iv. Personnel Capability (PERS) : This cost driver combines three Post Architecture Cost Drivers. These drivers are ACAP, PCAP and PCON.

PERS	*Extra Low*	*Very Low*	*Low*	*Nominal*	*High*	*Very High*	*Extra High*
Sum of ACAP, PCAP, PCON ratings	3, 4	5, 6	7, 8	9	10, 11	12, 13	14, 15
Combined ACAP & PCAP Percentile	20%	39%	45%	55%	65%	75%	85%
Annual Personnel Turnover	45%	30%	20%	12%	9%	5%	4%

Software Project Planning

v. Personnel Experience (PREX) : This early design driver combines three Post Architecture Cost Drivers, which are AEXP, PEXP and LTEX.

PREX	*Extra Low*	*Very Low*	*Low*	*Nominal*	*High*	*Very High*	*Extra High*
Sum of AEXP, PEXP and LTEX ratings	3, 4	5, 6	7, 8	9	10, 11	12, 13	14, 15
Applications, Platform, Language & Tool Experience	≤ 3 months	5 months	9 months	1 year	2 year	4 year	6 year

Software Project Planning

vi. Facilities (FCIL): This depends on two Post Architecture Cost Drivers, which are TOOL and SITE.

FCIL	*Extra Low*	*Very Low*	*Low*	*Nominal*	*High*	*Very High*	*Extra High*
Sum of TOOL & SITE ratings	2	3	4, 5	6	7, 8	9, 10	11
Tool support	Minimal	Some	Simple CASE tools	Basic life cycle tools	Good support of tools	Very strong use of tools	Very strong & well integrated tools
Multisite conditions development support	Weak support of complex multisite development	Some support	Moderate support	Basic support	Strong support	Very strong support	Very strong support

Software Project Planning

vii. Schedule (SCED) : This early design cost driver is the same as Post Architecture Counterpart and rating level are given below using table 16.

SCED	*Very Low*	*Low*	*Nominal*	*High*	*Very High*
Schedule	75% of Nominal	85%	100%	130%	160%

Software Project Planning

The seven early design cost drivers have been converted into numeric values with a Nominal value 1.0. These values are used for the calculation of a factor called "Effort multiplier" which is the product of all seven early design cost drivers. The numeric values are given in Table 15.

Early design Cost drivers	*Extra Low*	*Very Low*	*Low*	*Nominal*	*High*	*Very High*	*Extra High*
RCPX	.73	.81	.98	1.0	1.30	1.74	2.38
RUSE	—	—	0.95	1.0	1.07	1.15	1.24
PDIF	—	—	0.87	1.0	1.29	1.81	2.61
PERS	2.12	1.62	1.26	1.0	0.83	0.63	0.50
PREX	1.59	1.33	1.12	1.0	0.87	0.71	0.62
FCIL	1.43	1.30	1.10	1.0	0.87	0.73	0.62
SCED	—	1.43	1.14	1.0	1.0	1.0	—

Table 15: Early design parameters

Software Project Planning

The early design model adjusts the nominal effort using 7 effort multipliers (EMs). Each effort multiplier (also called drivers) has 7 possible weights as given in Table 15. These factors are used for the calculation of adjusted effort as given below:

$$PM_{adjusted} = PM_{\text{nominal}} \times \left[\prod_{i=7}^{7} EM_i \right]$$

$PM_{adjusted}$ effort may very even up to 400% from $PM_{nominal}$

Hence $PM_{adjusted}$ is the fine tuned value of effort in the early design phase

Example: 4.10

A software project of application generator category with estimated 50 KLOC has to be developed. The scale factor (B) has low precedentness, high development flexibility and low team cohesion. Other factors are nominal. The early design cost drivers like platform difficult (PDIF) and Personnel Capability (PERS) are high and others are nominal. Calculate the effort in person months for the development of the project.

Solution

Here B = 0.91 + 0.01 * (Sum of rating on scaling factors for the project)

= 0.91 + 0.01 * (4.96 + 2.03 + 4.24 + 4.38 + 4.68)

= 0.91 + 0.01(20.29)=1.1129

$PM_{nominal} = A*(size)^B$

$= 2.5 * (50)^{1.1129} = 194.41$ Person months

The 7 cost drivers are

PDIF = high (1.29)

PERS = high (0.83)

RCPX = nominal (1.0)

RUSE = nominal (1.0)

PREX = nominal (1.0)

FCIL = nominal (1.0)

SCEO = nominal (1.0)

Software Project Planning

$$PM_{adjusted} = PM_{\text{nominal}} \times \left[\prod_{i=7}^{7} EM_i \right]$$

= 194.41 * [1.29 x 0.83)

= 194.41 x 1.07

= 208.155 Person months

Post Architecture Model

The post architecture model is the most detailed estimation model and is intended to be used when a software life cycle architecture has been completed. This model is used in the development and maintenance of software products in the application generators, system integration or infrastructure sectors.

$$PM_{adjusted} = PM_{\text{nominal}} \times \left[\prod_{i=7}^{17} EM_i \right]$$

EM : Effort multiplier which is the product of 17 cost drivers.

The 17 cost drivers of the Post Architecture model are described in the table 16.

Software Project Planning

Cost driver	*Purpose*	*Very low*	*Low*	*Nominal*	*High*	*Very High*	*Extra High*
RELY (Reliability required)	Measure of the extent to which the software must perform its intended function over a period of time	Only slight inconvenience	Low, easily recoverable losses	Moderate, easily recoverable losses	High financial loss	Risk to human life	—
DATA (Data base size)	Measure the affect of large data requirements on product development	—	$\frac{\text{Database size(D)}}{\text{Prog. size (P)}} < 10$	$10 \le \frac{D}{P} < 100$	$100 \le \frac{D}{P} < 1000$	$\frac{D}{P} \ge 1000$	—
CPLX (Product complexity)	Complexity is divided into five areas: Control operations, computational operations, device dependent operations, data management operations & User Interface management operations.	See Table 4.17					
DOCU Documentation	Suitability of the project's documentation to its life cycle needs	Many life cycle needs uncovered	Some needs uncovered	Adequate	Excessive for life cycle needs	Very Excessive	—

Table 16: Post Architecture Cost Driver rating level summary

Cont…

Software Project Planning

TIME (Execution Time constraint)	Measure of execution time constraint on software	—	—	≤ 50% use of a available execution time	70%	85%	95%
STOR (Main storage constraint)	Measure of main storage constraint on software	—	—	≤ 50% use of available storage	70%	85%	95%
PVOL (Platform Volatility)	Measure of changes to the OS, compilers, editors, DBMS etc.	—	Major changes every 12 months & minor changes every 1 month	Major: 6 months Minor: 2 weeks	Major: 2 months Minor: 1 week	Major: 2 week Minor: 2 days	—
ACAP (Analyst capability)	Should include analysis and design ability, efficiency & thoroughness, and communication skills.	15th Percentile	35th Percentile	55th Percentile	75th Percentile	90th Percentile	—

Table 16: Post Architecture Cost Driver rating level summary

Cont…

Software Project Planning

PCAP (Programmers capability)	Capability of Programmers as a team. It includes ability, efficiency, thoroughness & communication skills	15th Percentile	35th Percentile	55th Percentile	75th Percentile	90th Percentile	—
PCON (Personnel Continuity)	Rating is in terms of Project's annual personnel turnover	48%/year	24%/year	12%/year	6%/year	3%/year	—
AEXP (Applications Experience)	Rating is dependent on level of applications experience.	≤ 2 months	6 months	1 year	3 year	6 year	—
PEXP (Platform experience)	Measure of Platform experience	≤ 2 months	6 months	1 year	3 year	6 year	—

Table 16: Post Architecture Cost Driver rating level summary

Cont…

LTEX (Language & Tool experience)	Rating is for Language & tool experience	≤ 2 months	6 months	1 year	3 year	6 year	—
TOOL (Use of software tools)	It is the indicator of usage of software tools	No use	Beginning to use	Some use	Good use	Routine & habitual use	—
SITE (Multisite development)	Site location & Communication technology between sites	International with some phone & mail facility	Multicity & multi company with individual phones, FAX	Multicity & multi company with Narrow band mail	Same city or Metro with wideband electronic communication	Same building or complex with wideband electronic communication & Video conferencing	Fully co-located with interactive multimedia
SCED (Required Development Schedule)	Measure of Schedule constraint. Ratings are defined in terms of percentage of schedule stretch-out or acceleration with respect to nominal schedule	75% of nominal	85%	100%	130%	160%	—

Table 16: Post Architecture Cost Driver rating level summary

Software Project Planning

Product complexity is based on control operations, computational operations, device dependent operations, data management operations and user interface management operations. Module complexity rating are given in table 17.

The numeric values of these 17 cost drivers are given in table 18 for the calculation of the product of efforts i.e., effort multiplier (EM). Hence PM adjusted is calculated which will be a better and fine tuned value of effort in person months.

Software Project Planning

	Control Operations	Computational Operations	Device-dependent Operations	Data management Operations	User Interface Management Operations
Very Low	Straight-line code with a few non-nested structured programming operators: Dos. Simple module composition via procedure calls or simple scripts.	Evaluation of simple expressions: e.g., A=B+C*(D-E)	Simple read, write statements with simple formats.	Simple arrays in main memory. Simple COTSDB queries, updates.	Simple input forms, report generators.
Low	Straight forward nesting of structured programming operators. Mostly simple predicates	Evaluation of moderate-level expressions: e.g., D=SQRT(B**2-4*A*C)	No cognizance needed of particular processor or I/O device characteristics. I/O done at GET/PUT level.	Single file sub setting with no data structure changes, no edits, no intermediate files, Moderately complex COTS-DB queries, updates.	User of simple graphics user interface (GUI) builders.

Table 17: Module complexity ratings

Cont…

Software Project Planning

	Control Operations	Computational Operations	Device-dependent Operations	Data management Operations	User Interface Management Operations
Nominal	Mostly simple nesting. Some inter module control Decision tables. Simple callbacks or message passing, including middleware supported distributed processing.	Use of standard maths and statistical routines. Basic matrix/ vector operations.	I/O processing includes device selection, status checking and error processing.	Multi-file input and single file output. Simple structural changes, simple edits. Complex COTS-DB queries, updates.	Simple use of widget set.
High	Highly nested structured programming operators with many compound predicates. Queue and stack control. Homogeneous, distributed processing. Single processor soft real time control.	Basic numerical analysis: multivariate interpolation, ordinary differential equations. Basic truncation, round off concerns.	Operations at physical I/O level (physical storage address translations; seeks, read etc.) Optimized I/O overlap.	Simple triggers activated by data stream contents. Complex data restructuring.	Widget set development and extension. Simple voice I/O multimedia.

Table 17: Module complexity ratings

Cont…

Software Project Planning

	Control Operations	Computational Operations	Device-dependent Operations	Data management Operations	User Interface Management Operations
Very High	Reentrant and recursive coding. Fixed-priority interrupt handling. Task synchronization, complex callbacks, heterogeneous distributed processing. Single processor hard real time control.	Difficult but structured numerical analysis: near singular matrix equations, partial differential equations. Simple parallelization.	Routines for interrupt diagnosis, servicing, masking. Communication line handling. Performance intensive embedded systems.	Distributed database coordination. Complex triggers. Search optimization.	Moderately complex 2D/3D, dynamic graphics, multimedia.
Extra High	Multiple resource scheduling with dynamically changing priorities. Microcode-level control. Distributed hard real time control.	Difficult and unstructured numerical analysis: highly accurate analysis of noisy, stochastic data. Complex parallelization.	Device timing dependent coding, micro programmed operations. Performance critical embedded systems.	Highly coupled, dynamic relational and object structures. Natural language data management.	Complex multimedia, virtual reality.

Table 17: Module complexity ratings

Cost Driver	Rating					
	Very Low	Low	Nominal	High	Very High	Extra High
RELY	0.75	0.88	1.00	1.15	1.39	
DATA		0.93	1.00	1.09	1.19	
CPLX	0.75	0.88	1.00	1.15	1.30	1.66
RUSE		0.91	1.00	1.14	1.29	1.49
DOCU	0.89	0.95	1.00	1.06	1.13	
TIME			1.00	1.11	1.31	1.67
STOR			1.00	1.06	1.21	1.57
PVOL		0.87	1.00	1.15	1.30	
ACAP	1.50	1.22	1.00	0.83	0.67	
PCAP	1.37	1.16	1.00	0.87	0.74	

Table 18: 17 Cost Drivers

Cont…

Cost Driver	Rating					
	Very Low	Low	Nominal	High	Very High	Extra High
PCON	1.24	1.10	1.00	0.92	0.84	
AEXP	1.22	1.10	1.00	0.89	0.81	
PEXP	1.25	1.12	1.00	0.88	0.81	
LTEX	1.22	1.10	1.00	0.91	0.84	
TOOL	1.24	1.12	1.00	0.86	0.72	
SITE	1.25	1.10	1.00	0.92	0.84	0.78
SCED	1.29	1.10	1.00	1.00	1.00	

Table 18: 17 Cost Drivers

Schedule estimation

Development time can be calculated using $PM_{adjusted}$ as a key factor and the desired equation is:

$$TDEV_{\text{nominal}} = [\phi \times (PM_{adjusted})^{(0.28+0.2(B-0.091))}] * \frac{SCED\%}{100}$$

where Φ = constant, provisionally set to 3.67

$TDEV_{nominal}$ = calendar time in months with a scheduled constraint

B = Scaling factor

$PM_{adjusted}$ = Estimated effort in Person months (after adjustment)

Size measurement

Size can be measured in any unit and the model can be calibrated accordingly. However, COCOMO II details are:

i. Application composition model uses the size in object points.

ii. The other two models use size in KLOC

Early design model uses unadjusted function points. These function points are converted into KLOC using Table 19. Post architecture model may compute KLOC after defining LOC counting rules. If function points are used, then use unadjusted function points and convert it into KLOC using Table 19.

Software Project Planning

Language	SLOC/UFP
Ada	71
AI Shell	49
APL	32
Assembly	320
Assembly (Macro)	213
ANSI/Quick/Turbo Basic	64
Basic-Compiled	91
Basic-Interpreted	128
C	128
C++	29

Table 19: Converting function points to lines of code

Cont…

Software Project Planning

Language	SLOC/UFP
ANSI Cobol 85	91
Fortan 77	105
Forth	64
Jovial	105
Lisp	64
Modula 2	80
Pascal	91
Prolog	64
Report Generator	80
Spreadsheet	6

Table 19: Converting function points to lines of code

Example: 4.11

Consider the software project given in example 4.10. Size and scale factor (B) are the same. The identified 17 Cost drivers are high reliability (RELY), very high database size (DATA), high execution time constraint (TIME), very high analyst capability (ACAP), high programmers capability (PCAP). The other cost drivers are nominal. Calculate the effort in Person-Months for the development of the project.

Solution

Here B = 1.1129

$PM_{nominal}$ = 194.41 Person-months

$$PM_{adjusted} = PM_{\text{nominal}} \times \left[\prod_{i=7}^{17} EM_i \right]$$

= 194.41 x (1.15 x 1.19 x 1.11 x 0.67 x 0.87)

= 194.41 x 0.885

= 172.05 Person-months

Putnam Resource Allocation Model

Norden of IBM

Rayleigh curve

Model for a range of hardware development projects.

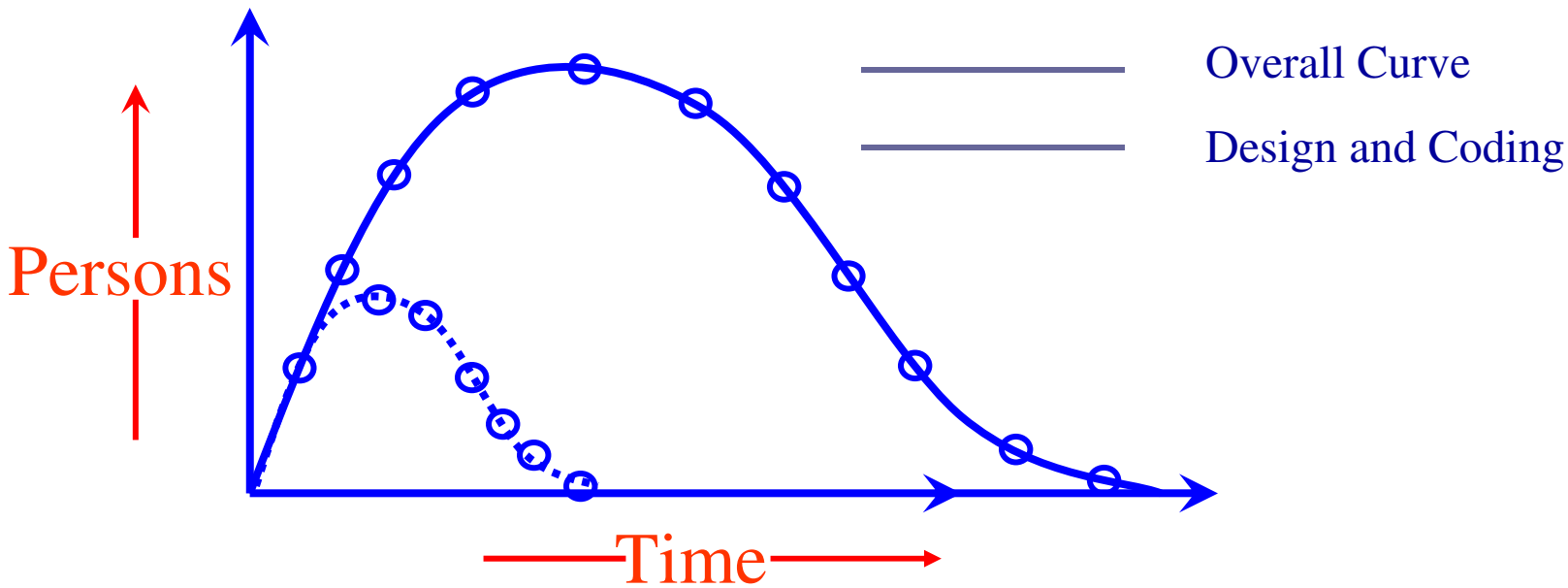


Fig.6: The Rayleigh manpower loading curve

Software Engineering (3rd ed.), By K.K Aggarwal & Yogesh Singh, Copyright © New Age International Publishers, 2007

Software Project Planning

Putnam observed that this curve was a close approximation at project level and software subsystem level.

No. of projects = 150

The Norden / Rayleigh Curve

The curve is modeled by differential equation

$$m(t) = \frac{dy}{dt} = 2kate^{-at^2} \qquad \text{---------- (1)}$$

$\frac{dy}{dt}$ = manpower utilization rate per unit time

a = parameter that affects the shape of the curve

K = area under curve in the interval [0, ∞]

t = elapsed time

Software Project Planning

On Integration on interval [o, t]

$$y(t) = K\,[1-e^{-at^2}] \quad \text{----------(2)}$$

Where y(t): cumulative manpower used upto time t.

$$y(0) = 0$$

$$y(\infty) = k$$

The cumulative manpower is null at the start of the project, and grows monotonically towards the total effort K (area under the curve).

Software Project Planning

$$\frac{d^2y}{dt^2} = 2kae^{-at^2}[1-2at^2] = 0$$

$$t_d^2 = \frac{1}{2a}$$

"t_d": time where maximum effort rate occurs

Replace "t_d" for *t* in equation (2)

$$E = y(t) = k\left(1 - e^{\frac{t_d^2}{2t_d^2}}\right) = K(1 - e^{-0.5})$$

$$E = y(t) = 0.3935\ k$$

$$a = \frac{1}{2t_d^2}$$

Software Project Planning

Replace "a" with $\frac{1}{2t_d^2}$ in the Norden/Rayleigh model. By making this substitution in equation we have

$$m(t) = \frac{2K}{2t_d^2} te^{-\frac{t^2}{2t_d^2}}$$

$$= \frac{K}{t_d^2} te^{-\frac{t^2}{2t_d^2}}$$

Software Project Planning

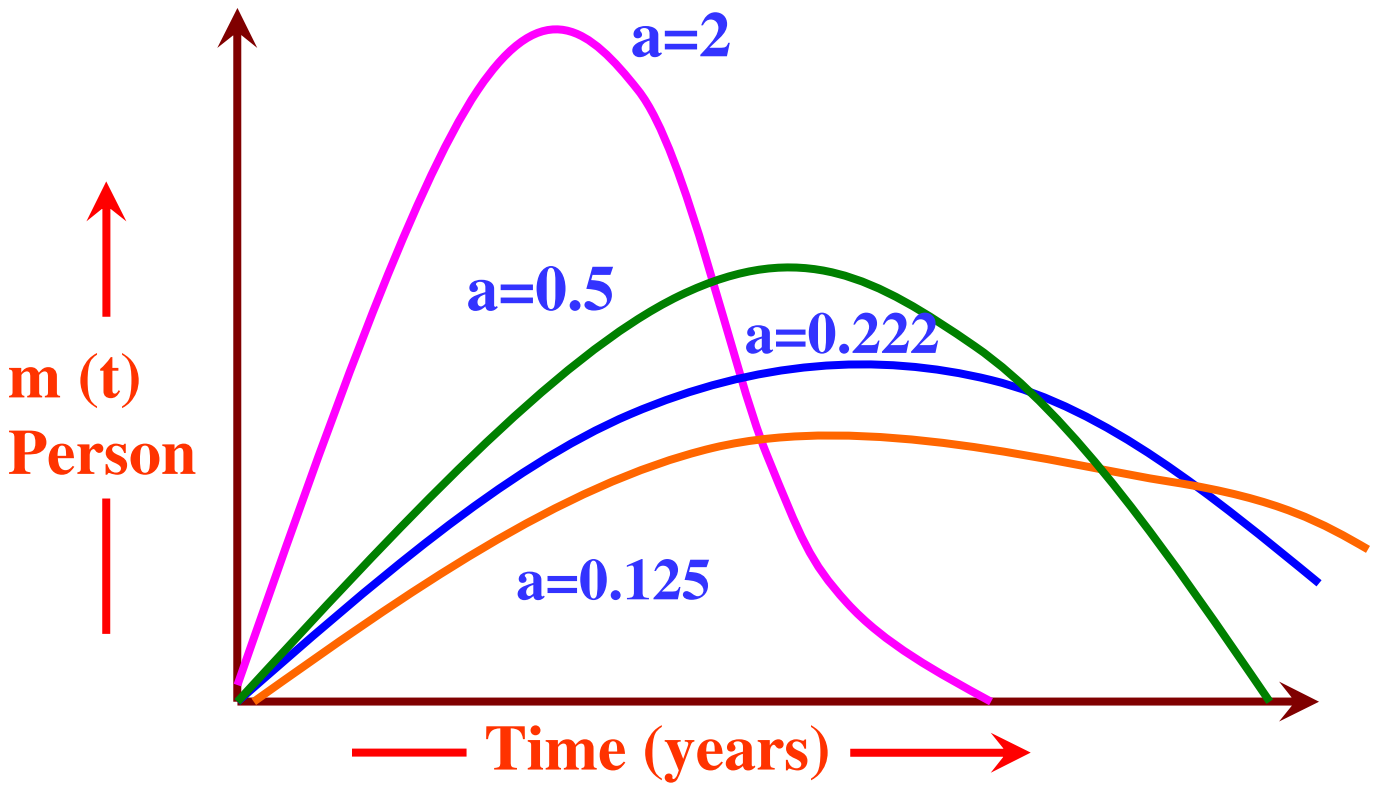


Fig.7: Influence of parameter 'a' on the manpower distribution

Software Project Planning

At time $t=t_d$, peak manning $m(t_d)$ is obtained and denoted by m_o.

$$m_o = \frac{k}{t_d \sqrt{e}}$$

k = Total project cost/effort in person-years.

t_d = Delivery time in years

m_0 = No. of persons employed at the peak

e = 2.71828

Example: 4.12

A software development project is planned to cost 95 MY in a period of 1 year and 9 months. Calculate the peak manning and average rate of software team build up.

Solution

Software development cost k=95 MY

Peak development time t_d = 1.75 years

Peak manning $m_o = \frac{k}{t_d \sqrt{e}}$

$$\frac{95}{1.75 \times 1.648} = 32.94 = 33 \ persons$$

Average rate of software team build up

$$= \frac{m_0}{t_d} = \frac{33}{1.75} = 18.8 \, persons / \ year \ or \ 1.56 \ person / month$$

Example: 4.13

Consider a large-scale project for which the manpower requirement is K=600 PY and the development time is 3 years 6 months.

(a)Calculate the peak manning and peak time.

(b)What is the manpower cost after 1 year and 2 months?

Solution

(a) We know t_d=3 years and 6 months = 3.5 years

NOW $$m_0 = \frac{K}{t_d \sqrt{e}}$$

$$\therefore \quad m_0 = 600/(3.5\text{x}1.648) \cong 104 \text{ persons}$$

(b) We know

$$y(t) = K\left[1 - e^{-at^2}\right]$$

t = 1 year and 2 months

= 1.17 years

$$a = \frac{1}{2t_d^2} = \frac{1}{2\times(3.5)^2} = 0.041$$

$$y(1.17) = 600\left[1 - e^{-0.041(1.17)^2}\right]$$

= 32.6 PY

Difficulty Metric

Slope of manpower distribution curve at start time t=0 has some useful properties.

$$m'(t) = \frac{d^2 y}{dt^2} = 2kae^{-at^2}(1 - 2at^2)$$

Then, for t=0

$$m'(0) = 2Ka = \frac{2K}{2t_d^2} = \frac{K}{t_d^2}$$

Software Project Planning

The ratio $\frac{K}{t_d^2}$ is called difficulty and denoted by D, which is measured in person/year :

$$D = \frac{k}{t_d^2} \text{ persons/year}$$

Software Project Planning

Project is difficult to develop if

Manpower demand is high

When time schedule is short

Software Project Planning

Peak manning is defined as:

$$m_0 = \frac{k}{t_d \sqrt{e}}$$

$$D = \frac{k}{t_d^2} = \frac{m_0 \sqrt{e}}{t_d}$$

Thus difficult projects tend to have a higher peak manning for a given development time, which is in line with Norden's observations relative to the parameter "a".

Manpower buildup

D is dependent upon "K". The derivative of D relative to "K" and "t_d" are

$$D'(t_d) = \frac{-2k}{t_d^3} \; persons / year^2$$

$$D'(k) = \frac{1}{t_d^2} \; year^{-2}$$

Software Project Planning

$D^1(K)$ will always be very much smaller than the absolute value of $D^1(t_d)$. This difference in sensitivity is shown by considering two projects

Project A : Cost = 20 PY & t_d = 1 year
Project B : Cost = 120 PY & t_d = 2.5 years

The derivative values are

Project A : $D`(t_d) = -40$ & $D`(K) = 1$
Project B : $D`(t_d) = -15.36$ & $D`(K) = 0.16$

This shows that a given software development is time sensitive.

Software Project Planning

Putnam observed that

Difficulty derivative relative to time

Behavior of s/w development

If project scale is increased, the development time also increase to such an extent that $\frac{k}{t_d^3}$ remains constant around a value which could be 8,15,27.

Software Project Planning

It is represented by D_0 and can be expressed as:

$$D_0 = \frac{k}{t_d^3} \, person / year^2$$

D_0 =8, new s/w with many interfaces & interactions with other systems.

D_0 =15, New standalone system.

D_0 =27, The software is rebuild form existing software.

Example: 4.14

Consider the example 4.13 and calculate the difficulty and manpower build up.

Solution

We know

Difficulty $$D = \frac{K}{t_d^2}$$

$$= \frac{600}{(3.5)^2} = 49 \ person/year$$

Manpower build up can be calculated by following equation

$$D_0 = \frac{K}{t_d^3}$$

$$= \frac{600}{(3.5)^3} = 14 \ person/year^2$$

Productivity Versus Difficulty

Productivity = No. of LOC developed per person-month

$$P \propto D^{\beta}$$

Avg. productivity

$$P = \frac{LOC\ produced}{cumulative\ manpower\ used\ to\ produce\ code}$$

Software Project Planning

P = S/E

$$P = \phi D^{-2/3}$$

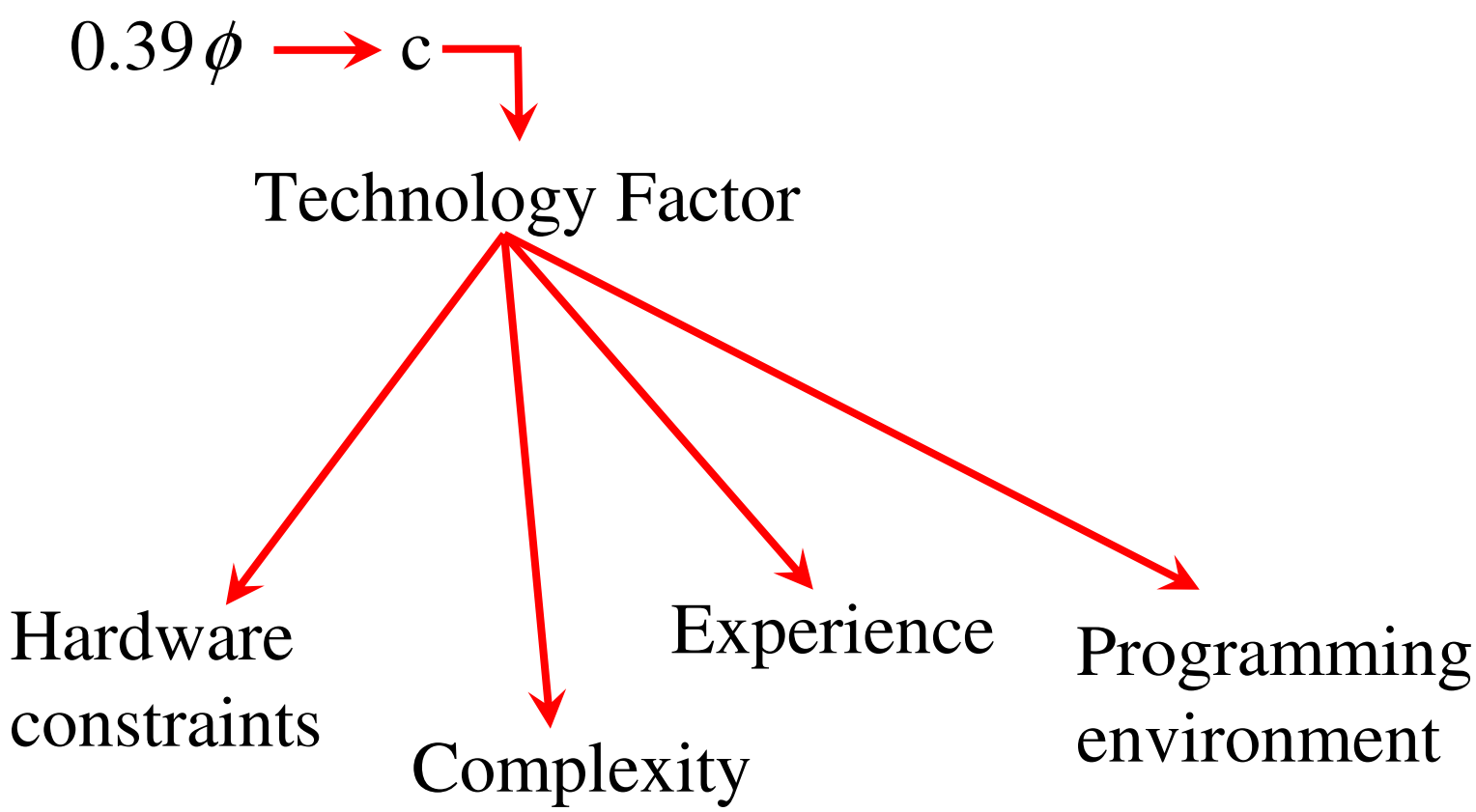
$$S = \phi D^{-2/3} E$$

$$= \phi D^{-2/3} (0.3935\ K)$$

$$S = \phi \left[\frac{k}{t_d^2} \right]^{-\frac{2}{3}} k(0.3935)$$

$$S = 0.3935 \phi K^{1/3} t_d^{\ 4/3}$$

Software Project Planning



Software Project Planning

C → 610 – 57314

K : P-Y

T : Years

$$S = CK^{1/3} t_d^{4/3}$$

$$C = S.K^{-1/3} t_d^{-4/3}$$

The trade off of time versus cost

$$K^{1/3} t_d^{4/3} = S / C$$

$$K = \frac{1}{t_d^4} \left(\frac{S}{C} \right)^3$$

Software Project Planning

C = 5000

S = 5,00,000 LOC

$$K = \frac{1}{t_d^4}(100)^3$$

t_d (years)	K (P-Y)
5.0	1600
4.0	3906
3.5	6664
3.0	12346

Table 20: (Manpower versus development time)

Development Subcycle

All that has been discussed so far is related to project life cycle as represented by project curve

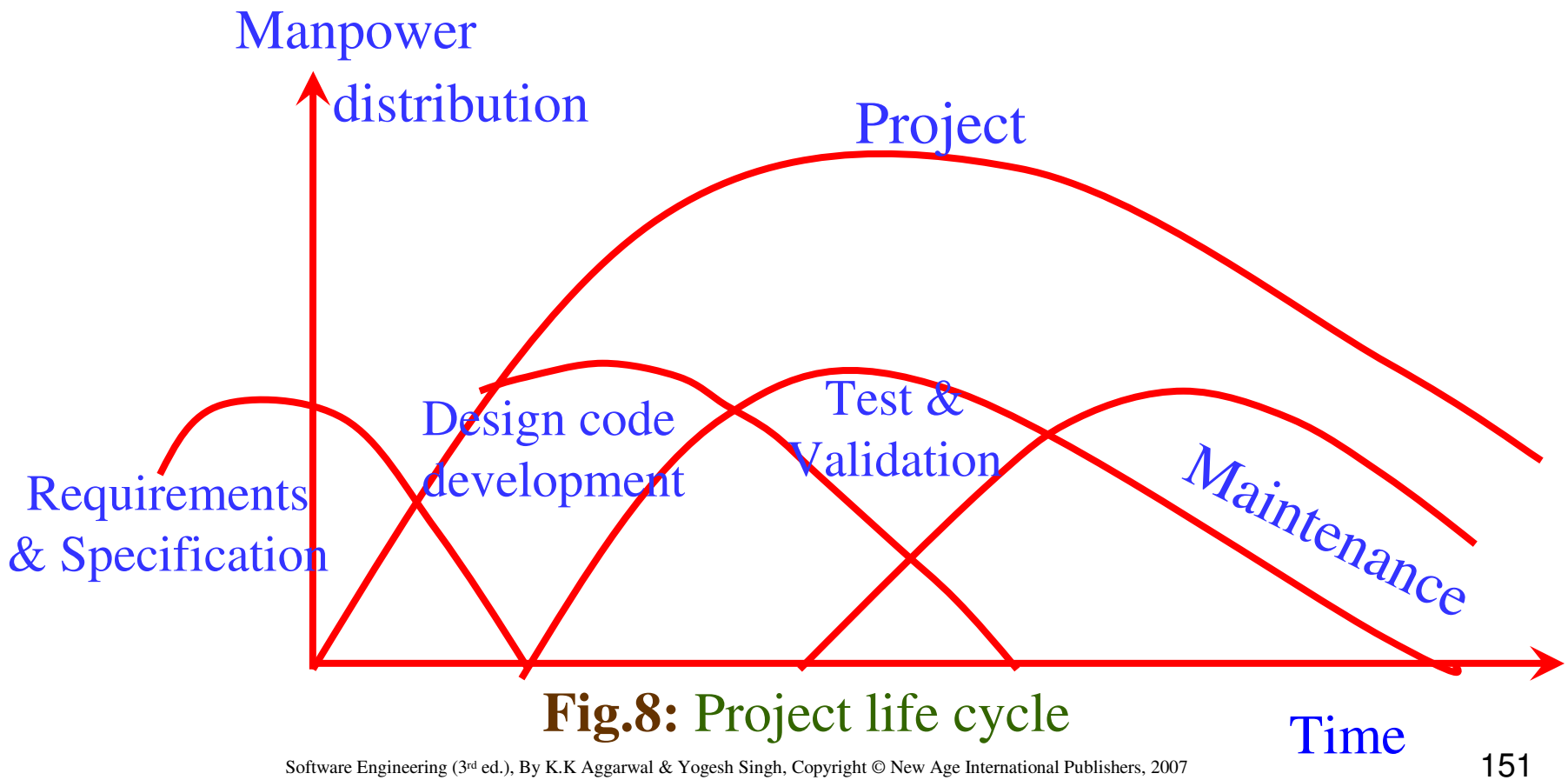
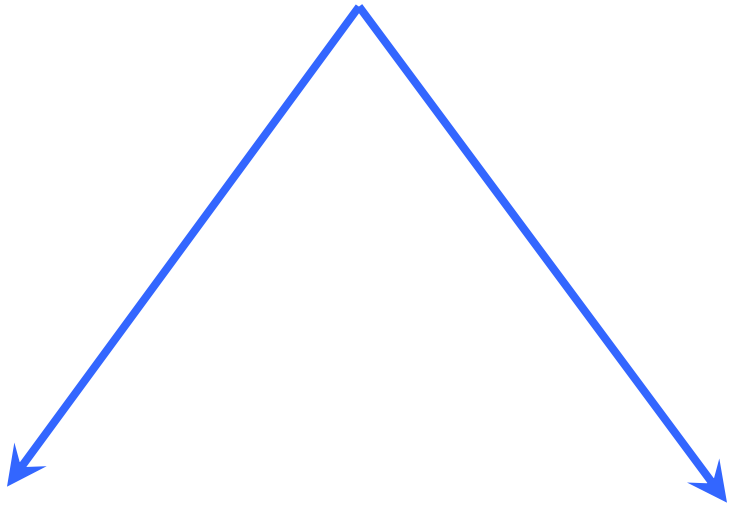


Fig.8: Project life cycle

Software Engineering (3rd ed.), By K.K Aggarwal & Yogesh Singh, Copyright © New Age International Publishers, 2007

Project life cycle

Project curve is the addition of two curves



Development Curve

Test & Validation Curve

Software Project Planning

$$\therefore\ m_d(t) = 2k_d bt\, e^{-bt^2}$$

$$y_d(t) = K_d\,[1-e^{-bt^2}]$$

An examination of $m_d(t)$ function shows a non-zero value of m_d at time t_d.

This is because the manpower involved in design & coding is still completing this activity after t^d in form of rework due to the validation of the product.

Nevertheless, for the model, a level of completion has to be assumed for development.

It is assumed that 95% of the development will be completed by the time t_d.

Software Project Planning

$$\frac{y_d(t)}{K_d} = 1 - e^{-bt^2} = 0.95$$

$\therefore$ We may say that $b = \dfrac{1}{2t_{od}^2}$

T_{od}: time at which development curve exhibits a peak manning.

$$t_{od} = \frac{t_d}{\sqrt{6}}$$

Software Project Planning

Relationship between K_d & K must be established.

At the time of origin, both cycles have the same slope.

$$\left(\frac{dm}{dt}\right)_o = \frac{K}{t_d^2} = \frac{K_d}{t_{od}^2} = \left(\frac{dm_d}{dt}\right)_o$$

$K_d = K/6$

$$D = \frac{K}{t_d^2} = \frac{K_d}{t_{od}^2}$$

Software Project Planning

This does not apply to the manpower build up D_0.

$$D_o = \frac{K}{t_d^3} = \frac{K_d}{\sqrt{6}t_{od}^3}$$

Conte investigated that

Larger projects → reasonable

Medium & small projects → overestimate

Example: 4.15

A software development requires 90 PY during the total development sub-cycle. The development time is planned for a duration of 3 years and 5 months

(a) Calculate the manpower cost expended until development time

(b) Determine the development peak time

(c) Calculate the difficulty and manpower build up.

Solution

(a) Duration t_d = 3.41 years

We know from equation $$\frac{y_d(t)}{K_d} = 1 - e^{-bt_d} = 0.95$$

$$\frac{y_d(t_d)}{K_d} = 0.95$$

$$Y_d(t_d) = 0.95 \times 90$$

= 85.5 PY

(b) We know from equation $t_{od} = \frac{t_d}{\sqrt{6}}$

$$t_{od} = \frac{t_d}{\sqrt{6}} = 3.41 / 2.449 = 1.39 \ years$$

$$\cong 17 \ months$$

Software Project Planning

(c) Total Manpower development

$$K_d = y_d(t_d)/0.95$$

= 85.5 / 0.95 = 90

$$K = 6K_d = 90 \times 6 = 540PY$$

$$D = K/t_d^2 = 540/(3.41)^2 = 46 \text{ persons/years}$$

$$D_o = \frac{K}{t_d^3} = 540/(3.41)^3 = 13.6 \text{ persons/years}^2$$

Example:4.16

A software development for avionics has consumed 32 PY up to development cycle and produced a size of 48000 LOC. The development of project was completed in 25 months. Calculate the development time, total manpower requirement, development peak time, difficulty, manpower build up and technology factor.

Solution:

Development time t_d = 25 months = 2.08 years

Total manpower development $k_d = \frac{Y_d(t_d)}{0.95} = \frac{32}{0.95} = 33.7\ PY$

Development peak time $t_{od} = \frac{(t_d)}{\sqrt{6}} = 0.85\ years = 10\ months$

$$K = 6K_d = 6 \text{ x } 33.7 = 202 \text{ PY}$$

$$D = \frac{k}{t_d^2} = \frac{202}{(2.08)^2} = 46.7\ pesons / years$$

Software Project Planning

$$D_0 = \frac{k}{t_d^3} = \frac{202}{(2.08)^3} = 22.5 \; Persons / year^2$$

Technology factor

$$C = SK^{-1/3} t_d^{-4/3}$$

$$= 3077$$

Example 4.17

What amount of software can be delivered in 1 year 10 months in an organization whose technology factor is 2400 if a total of 25 PY is permitted for development effort.

Solution:

t_d = 1.8 years

K_d = 25 PY

K = 25 x 6 = 150 PY

C = 2400

We know $S = CK^{1/3} t_d^{4/3}$

= 2400 x 5.313 x 2.18 = 27920 LOC

Example 4.18

The software development organization developing real time software has been assessed at technology factor of 2200. The maximum value of manpower build up for this type of software is D_o=7.5. The estimated size to be developed is S=55000 LOC.

(a) Determine the total development time, the total development manpower cost, the difficulty and the development peak manning.

(b) The development time determined in (a) is considered too long. It is recommended that it be reduced by two months. What would happen?

Solution

We have $$S = CK^{1/3} t_d^{\ 4/3}$$

$$\left(\frac{s}{c}\right)^3 = kt_d^4$$

which is also equivalent to $$\left(\frac{S}{C}\right)^3 = D_o t_d^7$$

then $$t_d = \left[\frac{1}{D_0}\left(\frac{S}{C}\right)^3\right]^{1/7}$$

Software Project Planning

$$Since \quad \frac{S}{C} = 25$$

$t_d = 3$ years

$$K = D_0 t_d^3 = 7.5 \times 27 = 202\ PY$$

Total development manpower cost $K_d = \frac{202}{06} = 33.75 PY$

$D = D_0 t_d = 22.5$ persons / year

$$t_{od} = \frac{t_d}{\sqrt{6}} = \frac{3}{\sqrt{6}} = 1.2\ years$$

Software Project Planning

$$M_d(t) = 2k_d\, bte^{-bt^2}$$

$$Y_d(t) = k_d\,(1-e^{-bt^2})$$

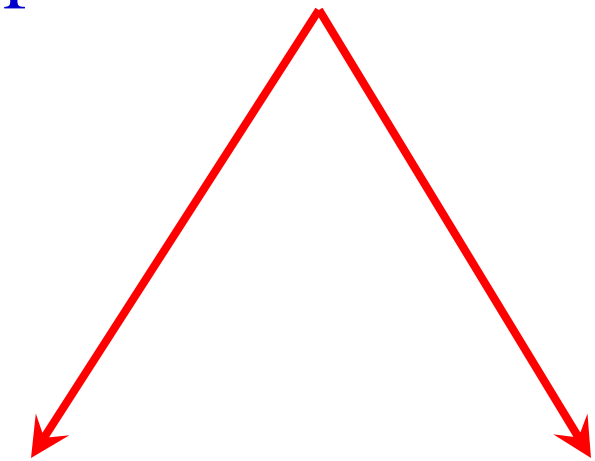
Here $t = t_{od}$

Peak manning $= m_{od} = Dt_{od}e^{-1/2}$

$= 22.5 \times 1.2 \times .606 = 16$ persons

Software Project Planning

III. If development time is reduced by 2 months



Developing s/w at higher manpower build-up

Producing less software

(i) Increase Manpower Build-up

$$D_o = \frac{1}{t_d^7}\left(\frac{S}{C}\right)^3$$

Now t_d = 3 years – 2 months = 2.8 years

$$D_o = (25)^3 / (2.8)^7 = 11.6 \; persons / years$$

$$k = D_0 t_d^3 = 254 \, PY$$

$$K_d = \frac{254}{6} = 42.4 \, PY$$

Software Project Planning

$D = D_0 t_d = 32.5$ persons / year

The peak time is $t_{od} = 1.14$ years

Peak manning $m_{od} = Dt_{od}\, e^{-0.5}$

$= 32.5 \times 1.14 \times 0.6$

$= 22$ persons

Note the huge increase in peak manning & manpower cost.

(ii) Produce Less Software

$$\left(\frac{S}{C}\right)^3 = D_0 t_d^7 = 7.5 \times (2.8)^7 = 10119.696$$

$$\left(\frac{S}{C}\right)^3 = 21.62989$$

Then for C=2200

S=47586 LOC

Productivity versus difficult

Example 4.19

A stand alone project for which the size is estimated at 12500 LOC is to be developed in an environment such that the technology factor is 1200. Choosing a manpower build up D_o=15, Calculate the minimum development time, total development man power cost, the difficulty, the peak manning, the development peak time, and the development productivity.

Solution

Size (S) = 12500 LOC

Technology factor (C) = 1200

Manpower buildup (D_o) = 15

Now

$$S = CK^{1/3} t_d^{4/3}$$

$$\frac{S}{C} = K^{1/3} t_d^{4/3}$$

$$\left(\frac{S}{C}\right)^3 = K t_d^4$$

Software Project Planning

Also we know $D_o = \frac{K}{t_d^3}$

$$K = D_o t_d^3 = D_o t_d^3$$

Hence $\left(\frac{S}{C}\right)^3 = D_o t_d^7$

Substituting the values, we get $\left(\frac{12500}{1200}\right)^3 = 15 t_d^7$

$$t_d = \left[\frac{(10.416)^3}{15}\right]^{1/7}$$

$$t_d = 1.85 \text{ years}$$

Software Project Planning

(i) Hence Minimum development time (t_d)=1.85 years

(ii) Total development manpower cost $K_d = \frac{K}{6}$

Hence $K = 15 t_d^3$

$$= 15(1.85)^3 = 94.97 \text{ PY}$$

$$K_d = \frac{K}{6} = \frac{94.97}{6} = 15.83\ PY$$

(iii) Difficulty $D = \frac{K}{t_d^2} = \frac{94.97}{(1.85)^2} = 27.75\ Persons/year$

Software Project Planning

(iv) Peak Manning $$m_0 = \frac{K}{t_d \sqrt{e}}$$

$$= \frac{94.97}{1.85 \times 1.648} = 31.15 Person$$

(v) Development Peak time $$t_{od} = \frac{t_d}{\sqrt{6}}$$

$$= \frac{1.85}{2.449} = 0.755 \ years$$

(vi) Development Productivity

$$= \frac{No.of\ lines\ of\ code\ (S)}{effort\ (K_d)}$$

$$= \frac{12500}{15.83} = 789.6\ LOC/PY$$

Software Risk Management

- We Software developers are extremely optimists.
- We assume, everything will go exactly as planned.
- Other view
  - not possible to predict what is going to happen ?

  Software surprises
  - Never good news

Software Project Planning

Risk management is required to reduce this surprise factor

Dealing with concern before it becomes a crisis.

Quantify probability of failure & consequences of failure.

What is risk ?

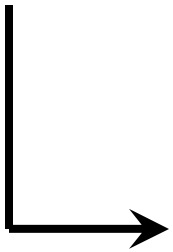
Tomorrow's problems are today's risks.

"Risk is a problem that may cause some loss or threaten the success of the project, but which has not happened yet".

Software Project Planning

Risk management is the process of identifying addressing and eliminating these problems before they can damage the project.

Current problems &



Potential Problems

Typical Software Risk

Capers Jones has identified the top five risk factors that threaten projects in different applications.

1. Dependencies on outside agencies or factors.
   - Availability of trained, experienced persons
   - Inter group dependencies
   - Customer-Furnished items or information
   - Internal & external subcontractor relationships

Software Project Planning

2. Requirement issues

Uncertain requirements

Wrong product

or

Right product badly

Either situation results in unpleasant surprises and unhappy customers.

Software Project Planning

- Lack of clear product vision
- Lack of agreement on product requirements
- Unprioritized requirements
- New market with uncertain needs
- Rapidly changing requirements
- Inadequate Impact analysis of requirements changes

Software Project Planning

3. Management Issues

Project managers usually write the risk management plans, and most people do not wish to air their weaknesses in public.

- Inadequate planning
- Inadequate visibility into actual project status
- Unclear project ownership and decision making
- Staff personality conflicts
- Unrealistic expectation
- Poor communication

Software Project Planning

4. Lack of knowledge

- Inadequate training
- Poor understanding of methods, tools, and techniques
- Inadequate application domain experience
- New Technologies
- Ineffective, poorly documented or neglected processes

Software Project Planning

5. Other risk categories

- Unavailability of adequate testing facilities
- Turnover of essential personnel
- Unachievable performance requirements
- Technical approaches that may not work

Risk Management Activities

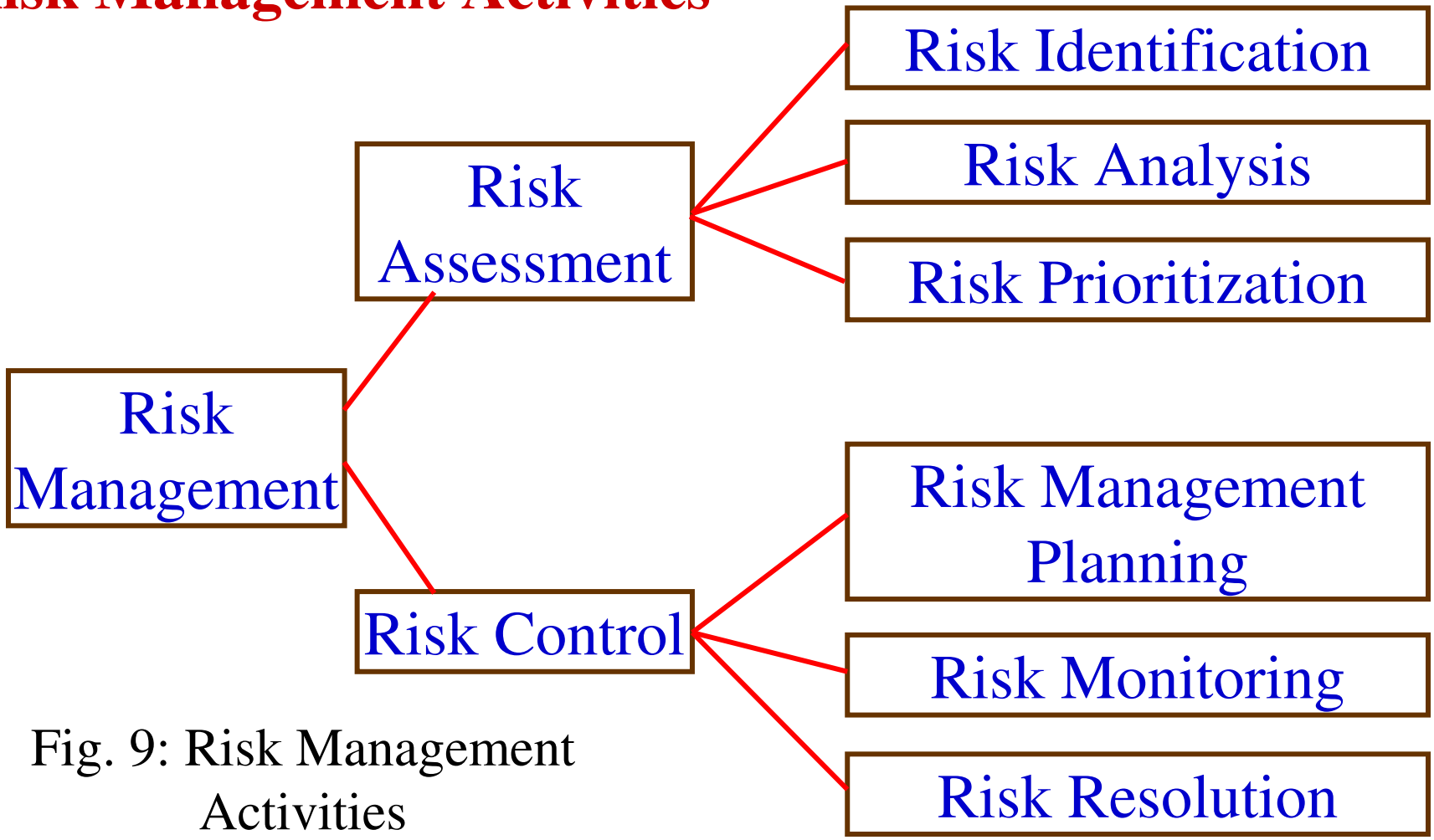


Fig. 9: Risk Management Activities

Risk Assessment

Identification of risks

Risk analysis involves examining how project outcomes might change with modification of risk input variables.

Risk prioritization focus for severe risks.

Risk exposure: It is the product of the probability of incurring a loss due to the risk and the potential magnitude of that loss.

Software Project Planning

Another way of handling risk is the risk avoidance. Do not do the risky things! We may avoid risks by not undertaking certain projects, or by relying on proven rather than cutting edge technologies.

Risk Control

Risk Management Planning produces a plan for dealing with each significant risks.

- Record decision in the plan.

Risk resolution is the execution of the plans of dealing with each risk.

Multiple Choice Questions

Note: Choose most appropriate answer of the following questions:

4.1 After the finalization of SRS, we may like to estimate

(a) Size (b) Cost

(c) Development time (d) All of the above.

4.2 Which one is not a size measure for software

(a) LOC (b) Function Count

(c) Cyclomatic Complexity (d) Halstead's program length

4.3 Function count method was developed by

(a) B.Beizer (b) B.Boehm

(c) M.halstead (d) Alan Albrecht

4.4 Function point analysis (FPA) method decomposes the system into functional units. The total number of functional units are

(a) 2 (b) 5

(c) 4 (d) 1

Multiple Choice Questions

4.5 IFPUG stand for
(a) Initial function point uniform group
(b) International function point uniform group
(c) International function point user group
(d) Initial function point user group

4.6 Function point can be calculated by
(a) UFP * CAF
(b) UFP * FAC
(c) UFP * Cost
(d) UFP * Productivity

4.7 Putnam resource allocation model is based on
(a) Function points
(b) Norden/ Rayleigh curve
(c) Putnam theory of software management
(d) Boehm's observation on manpower utilisation rate

4.8 Manpower buildup for Putnam resource allocation model is

$(a)\, K / t_d^2 \ persons / year^2$
$(b)\, K / t_d^3 \ persons / year^2$
$(c)\, K / t_d^2 \ persons / year$
$(d)\, K / t_d^3 \ persons / year$

Multiple Choice Questions

4.9 COCOMO was developed initially by

(a) B.W.Bohem (b) Gregg Rothermal

(c) B.Beizer (d) Rajiv Gupta

4.10 A COCOMO model is

(a) Common Cost estimation model

(b) Constructive cost Estimation model

(c) Complete cost estimation model

(d) Comprehensive Cost estimation model

4.11 Estimation of software development effort for organic software is COCOMO is

(a) $E=2.4(KLOC)^{1.05}PM$ (b) $E=3.4(KLOC)^{1.06}PM$

(c) $E=2.0(KLOC)^{1.05}PM$ (d) $E-2.4(KLOC)^{1.07}PM$

4.12 Estimation of size for a project is dependent on

(a) Cost (b) Schedule

(c) Time (d) None of the above

4.13 In function point analysis, number of Complexity adjustment factor are

(a) 10 (b) 20

(c) 14 (d) 12

Multiple Choice Questions

4.14 COCOMO-II estimation model is based on

(a) Complex approach
(b) Algorithm approach
(c) Bottom up approach
(d) Top down approach

4.15 Cost estimation for a project may include

(a) Software Cost
(b) Hardware Cost
(c) Personnel Costs
(d) All of the above

4.16 In COCOMO model, if project size is typically 2-50 KLOC, then which mode is to be selected?

(a) Organic
(b) Semidetached
(c) Embedded
(d) None of the above

4.17 COCOMO-II was developed at

(a) University of Maryland
(b) University of Southern California
(c) IBM
(d) AT & T Bell labs

4.18 Which one is not a Category of COCOMO-II

(a) End User Programming
(b) Infrastructure Sector
(c) Requirement Sector
(d) System Integration

Multiple Choice Questions

4.19 Which one is not an infrastructure software?

(a) Operating system
(b) Database management system
(c) Compilers
(d) Result management system

4.20 How many stages are in COCOMO-II?

(a) 2
(b) 3
(c) 4
(d) 5

4.21 Which one is not a stage of COCOMO-II?

(a) Application Composition estimation model
(b) Early design estimation model
(c) Post architecture estimation model
(d) Comprehensive cost estimation model

4.22 In Putnam resource allocation model, Rayleigh curve is modeled by the equation

$(a)\ \ m(t) = 2at\, e^{-at^2}$

$(b)\ \ m(t) = 2Kt\, e^{-at^2}$

$(c)\ \ m(t) = 2Kat\, e^{-at^2}$

$(d)\ \ m(t) = 2Kbt\, e^{-at^2}$

Multiple Choice Questions

4.23 In Putnam resource allocation model, technology factor 'C' is defined as

$(a)\ C = SK^{-1/3}t_d^{-4/3}$ $(b)\ C = SK^{1/3}t_d^{4/3}$

$(c)\ C = SK^{1/3}t_d^{-4/3}$ $(d)\ C = SK^{-1/3}t_d^{4/3}$

4.24 Risk management activities are divided in

(a) 3 Categories (b) 2 Categories

(c) 5 Categories (d) 10 Categories

4.25 Which one is not a risk management activity?

(a) Risk assessment (b) Risk control

(c) Risk generation (d) None of the above

Exercises

4.1 What are various activities during software project planning?

4.2 Describe any two software size estimation techniques.

4.3 A proposal is made to count the size of 'C' programs by number of semicolons, except those occurring with literal strings. Discuss the strengths and weaknesses to this size measure when compared with the lines of code count.

4.4 Design a LOC counter for counting LOC automatically. Is it language dependent? What are the limitations of such a counter?

4.5 Compute the function point value for a project with the following information domain characteristics.

Number of user inputs = 30

Number of user outputs = 42

Number of user enquiries = 08

Number of files = 07

Number of external interfaces = 6

Assume that all complexity adjustment values are moderate.

Exercises

4.6 Explain the concept of function points. Why FPs are becoming acceptable in industry?

4.7 What are the size metrics? How is function point metric advantageous over LOC metric? Explain.

4.8 Is it possible to estimate software size before coding? Justify your answer with suitable example.

4.9 Describe the Albrecht's function count method with a suitable example.

4.10 Compute the function point FP for a payroll program that reads a file of employee and a file of information for the current month and prints cheque for all the employees. The program is capable of handling an interactive command to print an individually requested cheque immediately.

Exercises

4.11 Assume that the previous payroll program is expected to read a file containing information about all the cheques that have been printed. The file is supposed to be printed and also used by the program next time it is run, to produce a report that compares payroll expenses of the current month with those of the previous month. Compute functions points for this program. Justify the difference between the function points of this program and previous one by considering how the complexity of the program is affected by adding the requirement of interfacing with another application (in this case, itself).

4.12 Explain the Walson & Felix model and compare with the SEL model.

4.13 The size of a software product to be developed has been estimated to be 22000 LOC. Predict the manpower cost (effort) by Walston-Felix Model and SEL model.

4.14 A database system is to be developed. The effort has been estimated to be 100 Persons-Months. Calculate the number of lines of code and productivity in LOC/Person-Month.

Exercises

4.15 Discuss various types of COCOMO mode. Explain the phase wise distribution of effort.

4.16 Explain all the levels of COCOMO model. Assume that the size of an organic software product has been estimated to be 32,000 lines of code. Determine the effort required to developed the software product and the nominal development time.

4.17 Using the basic COCOMO model, under all three operating modes, determine the performance relation for the ratio of delivered source code lines per person-month of effort. Determine the reasonableness of this relation for several types of software projects.

4.18 The effort distribution for a 240 KLOC organic mode software development project is: product design 12%, detailed design 24%, code and unit test 36%, integrate and test 28%. How would the following changes, from low to high, affect the phase distribution of effort and the total effort: analyst capability, use of modern programming languages, required reliability, requirements volatility?

Exercises

4.19 Specify, design, and develop a program that implements COCOMO. Using reference as a guide, extend the program so that it can be used as a planning tool.

4.20 Suppose a system for office automation is to be designed. It is clear from requirements that there will be five modules of size 0.5 KLOC, 1.5 KLOC, 2.0 KLOC, 1.0 KLOC and 2.0 KLOC respectively. Complexity, and reliability requirements are high. Programmer's capability and experience is low. All other factors are of nominal rating. Use COCOMO model to determine overall cost and schedule estimates. Also calculate the cost and schedule estimates for different phases.

4.21 Suppose that a project was estimated to be 600 KLOC. Calculate the effort and development time for each of the three modes i.e., organic, semidetached and embedded.

4.22 Explain the COCOMO-II in detail. What types of categories of projects are identified?

Exercises

4.23 Discuss the Infrastructure Sector of COCOMO-II.

4.24 Describe various stages of COCOMO-II. Which stage is more popular and why?

4.25 A software project of application generator category with estimated size of 100 KLOC has to be developed. The scale factor (B) has high percedentness, high development flexibility. Other factors are nominal. The cost drivers are high reliability, medium database size, high Personnel capability, high analyst capability. The other cost drivers are nominal. Calculate the effort in Person-Months for the development of the project.

4.26 Explain the Putnam resource allocation model. What are the limitations of this model?

4.27 Describe the trade-off between time versus cost in Putnam resource allocation model.

4.28 Discuss the Putnam resources allocation model. Derive the time and effort equations.

Exercises

4.29 Assuming the Putnam model, with S=100,000 , C=5000, D_o=15, Compute development time t_d and manpower development K_d.

4.30 Obtain software productivity data for two or three software development programs. Use several cost estimating models discussed in this chapter. How to the results compare with actual project results?

4.31 It seems odd that cost and size estimates are developed during software project planning-before detailed software requirements analysis or design has been conducted. Why do we think this is done? Are there circumstances when it should not be done?

4.32 Discuss typical software risks. How staff turnover problem affects software projects?

4.33 What are risk management activities? Is it possible to prioritize the risk?

Exercises

4.34 What is risk exposure? What techniques can be used to control each risk?

4.35 What is risk? Is it economical to do risk management? What is the effect of this activity on the overall cost of the project?

4.36 There are significant risks even in student projects. Analyze a student project and list all the risk.

Module 3

Software Design

prepared by, U.B.Nisha AP/CSE

Overview of Design

- What is it? A meaningful engineering representation of something that is to be built.
- Who does it? Software engineers --computer architecture
- Why is it important? A house would never be built without a blueprint. Why should software? Without design the system may fail with small changes, is difficult to test and cannot be assessed for quality
- What is the work product? A design specification

Design

Good software design should exhibit:

- *Firmness:* A program should not have any bugs that inhibit its function.
- *Commodity:* A program should be suitable for the purposes for which it was intended.
- *Delight:* The experience of using the program should be satisfying one.

Purpose of Design

- Design is where customer requirements, business needs, and technical considerations all come together in the formulation of a product or system
- The design model provides detail about the software data structures, architecture, interfaces, and components
- The design model can be assessed for quality and be improved before code is generated and tests are conducted
  - Does the design contain errors, inconsistencies, or omissions?
  - Are there better design alternatives?
  - Can the design be implemented within the constraints, schedule, and cost that have been established?

Purpose of Design

The design process involves:

- Diversification (acquisition of alternatives)

  Followed By
- Convergence (elimination of all but one particular configuration)

- A designer must practice diversification and convergence
  - The designer selects from design components, component solutions, and knowledge available through catalogs, textbooks, and experience
  - The designer then chooses the elements from this collection that meet the requirements defined by requirements engineering and analysis modeling
  - Convergence occurs as alternatives are considered and rejected until one particular configuration of components is chosen
- Software design is an iterative process through which requirements are translated into a blueprint for constructing the software
  - Design begins at a high level of abstraction that can be directly traced back to the data, functional, and behavioral requirements
  - As design iteration occurs, subsequent refinement leads to design representations at much lower levels of abstraction

Design Principles

- The design process should not suffer from ‘tunnel vision.’
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should “minimize the intellectual distance” [DAV95] between the software and the problem as it exists in the real world.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.
- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.
- Design is not coding, coding is not design.
- The design should be assessed for quality as it is being created, not after the fact.
- The design should be reviewed to minimize conceptual (semantic) errors.

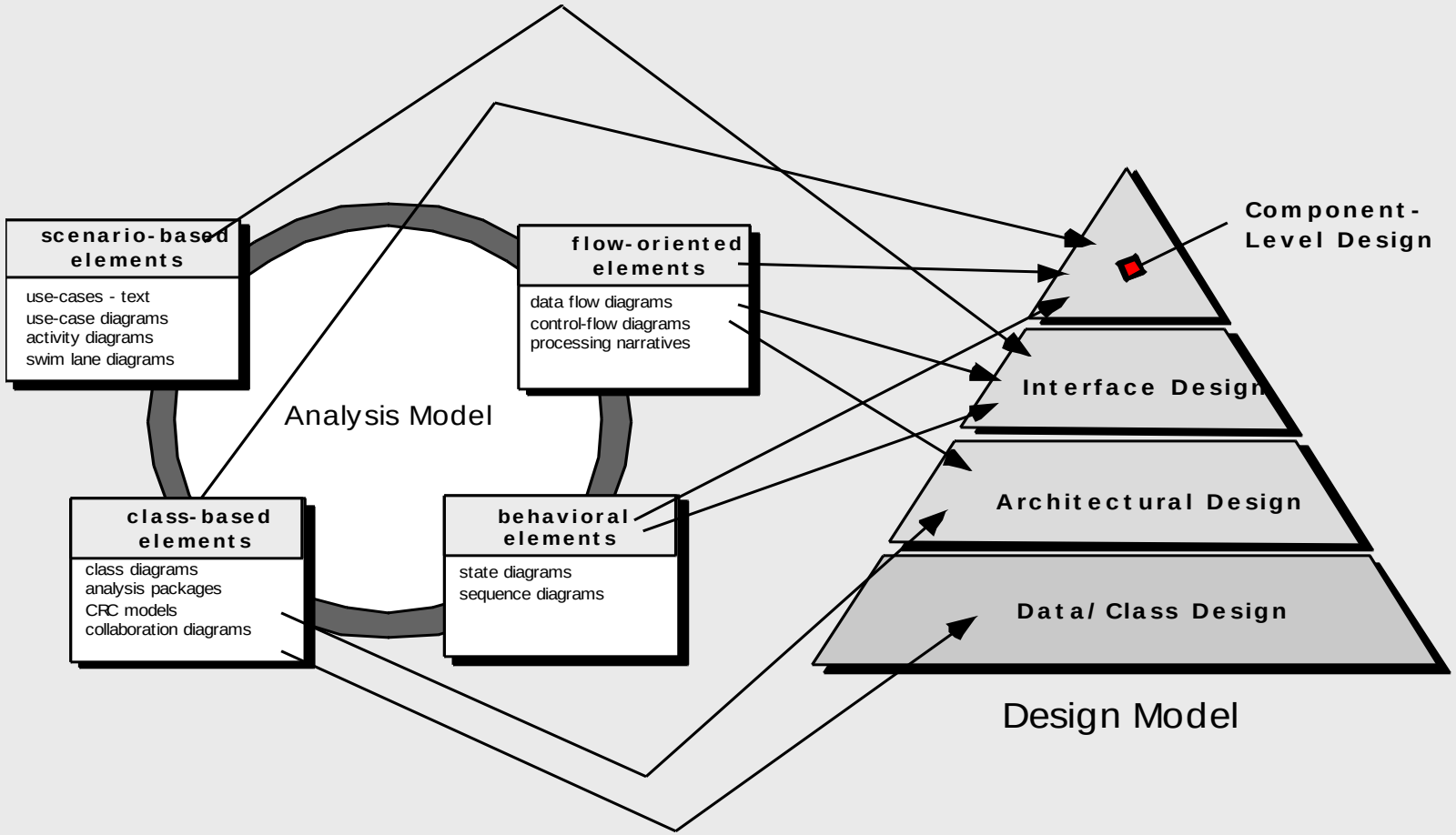
From Davis [DAV95]

From Analysis Model to Design Model

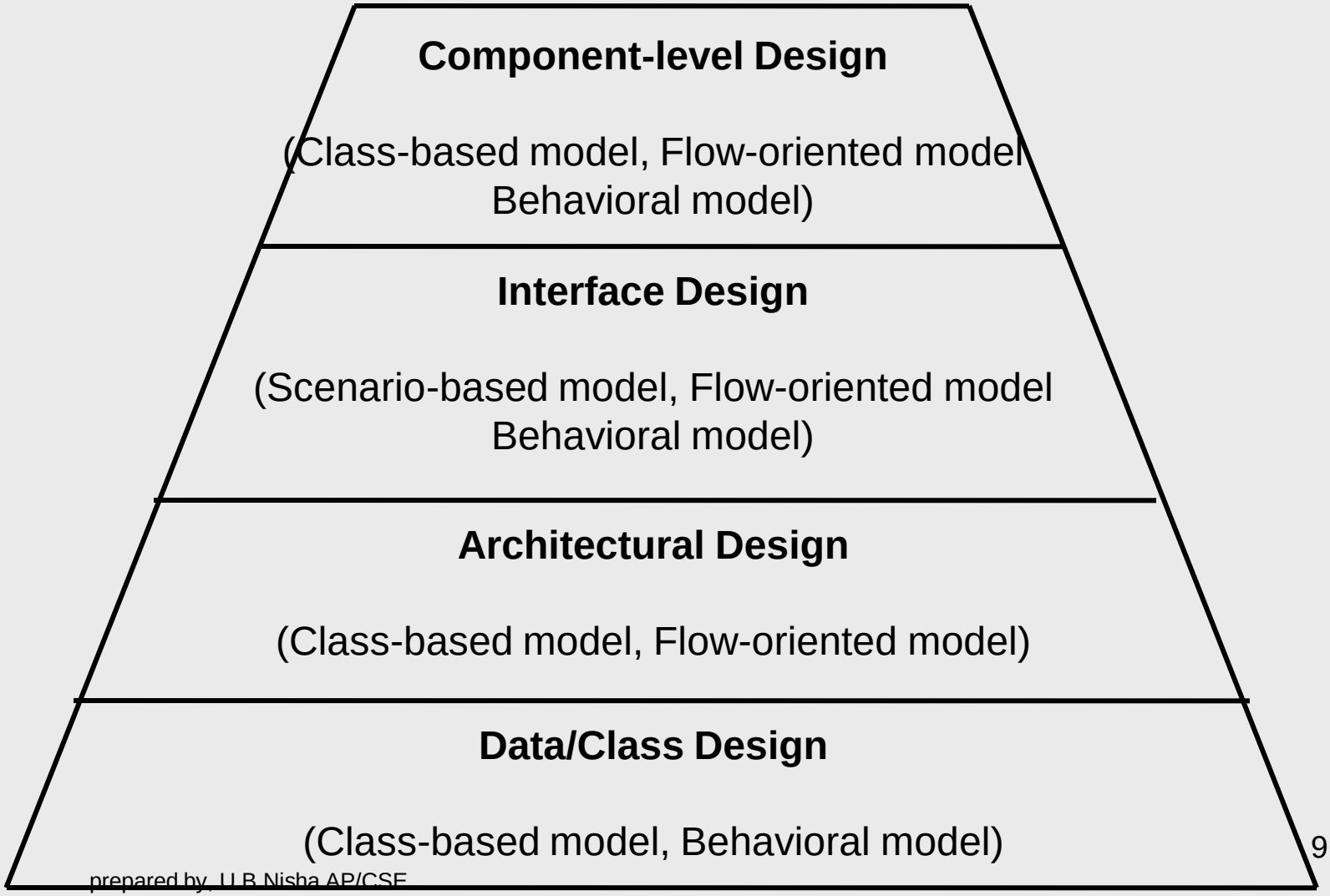
- Each element of the analysis model provides information that is necessary to create the four design models
  - The data/class design transforms analysis classes into design classes along with the data structures required to implement the software
  - The architectural design defines the relationship between major structural elements of the software; architectural styles and design patterns help achieve the requirements defined for the system
  - The interface design describes how the software communicates with systems that interoperate with it and with humans that use it
  - The component-level design transforms structural elements of the software architecture into a procedural description of software components

(More on next slide)

Analysis Model -> Design Model



From Analysis Model to Design Model (continued)



Task Set for Software Design

1) Examine the information domain model and design appropriate data structures for data objects and their attributes
2) Using the analysis model, select an architectural style (and design patterns) that are appropriate for the software
3) Partition the analysis model into design subsystems and allocate these subsystems within the architecture
   a) Design the subsystem interfaces
   b) Allocate analysis classes or functions to each subsystem
4) Create a set of design classes or components
   a) Translate each analysis class description into a design class
   b) Check each design class against design criteria; consider inheritance issues
   c) Define methods associated with each design class
   d) Evaluate and select design patterns for a design class or subsystem

Task Set for Software Design (continued)

5) Design any interface required with external systems or devices
6) Design the user interface
7) Conduct component-level design
   a) Specify all algorithms at a relatively low level of abstraction
   b) Refine the interface of each component
   c) Define component-level data structures
   d) Review each component and correct all errors uncovered
8) Develop a deployment model

Show a physical layout of the system, revealing which components will be located where in the physical computing environment

Design Quality: Quality's Role

- The importance of design is quality
- Design is the place where quality is fostered
  - Provides representations of software that can be assessed for quality
  - Accurately translates a customer's requirements into a finished software product or system
  - Serves as the foundation for all software engineering activities that follow
- Without design, we risk building an unstable system that
  - Will fail when small changes are made
  - May be difficult to test
  - Cannot be assessed for quality later in the software process when time is short and most of the budget has been spent
- The quality of the design is assessed through a series of formal technical reviews or design walkthroughs

Goals of a Good Design

- The design must implement all of the explicit requirements contained in the analysis model
  - It must also accommodate all of the implicit requirements desired by the customer
- The design must be a readable and understandable guide for those who generate code, and for those who test and 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 Quality Guidelines

1) A design should exhibit an architecture that
   a) Has been created using recognizable architectural styles or patterns
   b) Is composed of components that exhibit good design characteristics
   c) Can be implemented in an evolutionary fashion, thereby facilitating implementation and testing
2) A design should be modular; that is, the software should be logically partitioned into elements or subsystems
3) A design should contain distinct representations of data, architecture, interfaces, and components
4) A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns

(more on next slide)

Quality Guidelines (continued)

5) A design should lead to components that exhibit independent functional characteristics
6) A design should lead to interfaces that reduce the complexity of connections between components and with the external environment
7) A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis
8) A design should be represented using a notation that effectively communicates its meaning

Design Concepts

- **Abstraction**—data, procedure, control
- **Architecture**—the overall structure of the software
- **Patterns—**”conveys the essence” of a proven design solution
- **Separation of concerns**—any complex problem can be more easily handled if it is subdivided into pieces
- **Modularity**—compartmentalization of data and function
- **Hiding**—controlled interfaces
- **Functional independence**—single-minded function and low coupling
- **Refinement**—elaboration of detail for all abstractions
- **Aspects**—a mechanism for understanding how global requirements affect design
- **Refactoring**—a reorganization technique that simplifies the design
- **Design Classes**—provide design detail that will enable analysis classes to be implemented

Design Concepts

- Abstraction
  - Procedural abstraction – a sequence of instructions that have a specific and limited function
  - Data abstraction – a named collection of data that describes a data object
- Architecture
  - The overall structure of the software and the ways in which the structure provides conceptual integrity for a system
  - Consists of components, connectors, and the relationship between them
- Patterns
  - A design structure that solves a particular design problem within a specific context
  - It provides a description that enables a designer to determine whether the pattern is applicable, whether the pattern can be reused, and whether the pattern can serve as a guide for developing similar patterns

(more on next slide)

Design Concepts (continued)

- Modularity
  - Separately named and addressable components (i.e., modules) that are integrated to satisfy requirements (divide and conquer principle)
  - Makes software intellectually manageable so as to grasp the control paths, span of reference, number of variables, and overall complexity
- Information hiding
  - The designing of modules so that the algorithms and local data contained within them are inaccessible to other modules
  - This enforces access constraints to both procedural (i.e., implementation) detail and local data structures
- Functional independence
  - Modules that have a "single-minded" function and an aversion to excessive interaction with other modules
  - High cohesion – a module performs only a single task
  - Low coupling – a module has the lowest amount of connection needed with other modules

Design Concepts (continued)

- Stepwise refinement
  - Development of a program by successively refining levels of procedure detail
  - Complements abstraction, which enables a designer to specify procedure and data and yet suppress low-level details
- Refactoring
  - A reorganization technique that simplifies the design (or internal code structure) of a component without changing its function or external behavior
  - Removes redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failures
- Design classes
  - Refines the analysis classes by providing design detail that will enable the classes to be implemented
  - Creates a new set of design classes that implement a software infrastructure to support the business solution

Abstraction

- Wasserman: “Abstraction permits one to concentrate on a problem at some level of abstraction without regard to low level detail.
- At the highest level of abstraction a solution is stated in broad terms using the language of the problem environment.
- At lower level, a procedural orientation is taken.
- At the lowest level of abstraction the solution is stated in a manner that can be directly implemented.

Abstraction(1)

Types of abstraction :

1. Procedural Abstraction :

A named sequence of instructions that has a specific & limited function

Eg: Word OPEN for a door

2. Data Abstraction :

A named collection of data that describes a data object. Data abstraction for door would be a set of attributes that describes the door

(e.g. door type, swing direction, weight, dimension)

Architecture (1)

- Is the structure/organization of program components (modules), the manner in which these components interact, and the structure of data that is used by the components.
- Components can be generalized to represent major system elements and their interactions.
- Goal – derive architectural rendering of system, that serves as a framework from which more detailed design activities are conducted.

Architecture (2)

Architectural design models:

- structural models;
- framework models;
- dynamic models;
- process models;
- functional models.

Patterns

- Describes a design structure that solves a particular design problem within a specific context.
- Should provide description that enables designer to determine:
  - whether pattern is applicable to current work;
  - whether the pattern can be reused;
  - whether the pattern can serve as a guide for developing similar, but functionally or structurally different pattern.

Separation of Concerns

- Any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently
- A concern is a feature or behavior that is specified as part of the requirements model for the software
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

Modularity

- In this concept, software is divided into separately named and addressable components called modules
- Follows “divide and conquer” concept, a complex problem is broken down into several manageable pieces
- Let p1 and p2 be two problems.
- Let E1 and E2 be the effort required to solve them –

If $C(p1)>C(p2)$

Hence $E(p1)>E(p2)$

Modularity

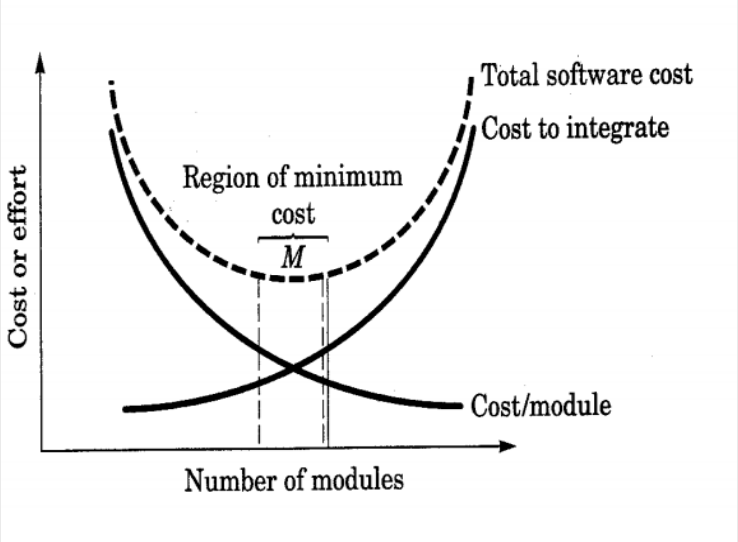
Now—

- Complexity of a problem that combines p1 and p2 is greater than complexity when each problem is consider

  C(p1+p2) > C(p1)+C(p2),

Hence

E(p1+p2) > E(p1)+E(p2) It is easier to solve a complex problem when you break it into manageable pieces



Modularity

5 criteria to evaluate a design method with respect to its modularity-----

- **Modular understandability:** module should be understandable as a standalone unit (no need to refer to other modules)
- **Modular continuity:** If small changes to the system requirements result in changes to individual modules, rather than system wide changes, the impact of side effects will be minimized
- **Modular protection**: If an error occurs within a module then those errors are localized and not spread to other modules
- **Modular Composability:** Design method should enable reuse of existing components.
- **Modular Decomposability:** Complexity of the overall problem can be reduced if the design method provides a systematic mechanism to decompose a problem into sub problems

Information hiding

- Builds up on modularity concept.
- Modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information;
- Hiding defines and enforces access constraints to procedural details and local data structures used within a module.
- Hiding prevents error propagation outside of a module.

Why Information Hiding?

- reduces the likelihood of “side effects”
- limits the global impact of local design decisions
- emphasizes communication through controlled interfaces
- discourages the use of global data
- leads to encapsulation—an attribute of high quality design
- results in higher quality software

Functional Independence

- Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.
- *Cohesion* is an indication of the relative functional strength of a module.
  - A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.
- *Coupling* is an indication of the relative interdependence among modules.
  - 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.

Functional independence

- Each module should address a specific sub function of requirements and have a simple interface.
- Functional independent modules are easier to develop, maintain, and test;
- Error propagation is reduced and reusable modules are possible;
- Assessed using two qualitative criteria:
  - cohesion;
  - coupling.

Types of Cohesion

Def: the degree of interaction within a module

- Worst to Best

6. Coincidental Cohesion
   - Performs multiple unrelated actions
   - Can happen if an organization enforces rigid rules on module size - modules are hacked apart and glued together
   - Worse than no modularity at all
5. Logical Cohesion
   - Module tasks related logically
   - Example: an object that performs all input and output
   - Interface can be difficult to understand (e.g. printf) and code for several actions may be intertwined
4. Temporal Cohesion
   - Tasks executed within the same span of time
   - Example: initialization of data structures

Low

Moderate

More Types of Cohesion

3. Procedural
   - Actions are related and must be executed in a certain order
2. Communication
   - Actions are performed in series and on the same data
   - Example: CalculateTrajectoryAndPrint
   - Damages Reusability
1. Functional or informational cohesion
   - Performs exactly one action OR
   - Performs a number of actions, with separate entry points, all performed on the same data structure
   - Equivalent to a well-designed abstract data type or object

Moderate

High

"Single-minded"

Coupling

- Def: the degree of interaction between modules
- A measure of relative interdependence; strive for low coupling since this reduces the “ripple effect”
- Types of Coupling (Worst to Best):

5. Content Coupling
   - One module directly references the internals of another
   - Example: module *p* branches to a local label of module *q*
   - Almost any change in one requires a change in the other
4. Common Coupling
   - Both modules have access to the same global data area
   - Example: module *p* and *q* have read and write access to the same database element
   - Suffers from all the disadvantages of global variables

High Coupling

↓

Moderate Coupling

Types of Coupling

3. Control Coupling
   - Element of control is transferred between modules
   - Example: Module *q* not only passes information but also informs module *p* as to what action to take
   - These kinds of modules often have logical cohesion
2. Stamp Coupling
   - Whole data structures (records, arrays, object) transferred
   - BUT the called module only operates on part of the data structure
   - Security Risk: allows uncontrolled data access
1. Data Coupling
   - Every argument is either a simple type or a data structure
   - AND all elements are used by the called module
   - Maintenance is easier because regression faults less likely

Moderate Coupling

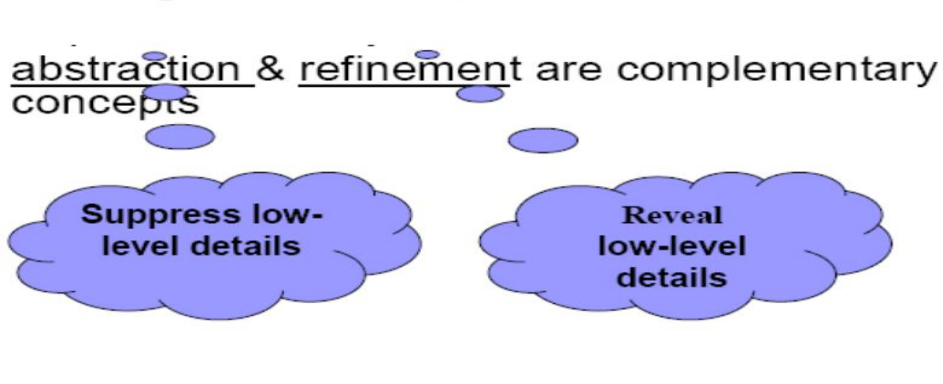
High Couplin g

Refinement

- A top-down design strategy by which program is designed by successively refining levels of procedural detail;
- Abstraction and refinement are complementary features:
  - one specifies procedure and data without details;
  - other allows to elaborate by providing low-level details.

Refinement

- Process of elaboration.
- Start with the statement of function defined at the abstract level, decompose the statement of function in a stepwise fashion until programming language statements are reached.



Refactoring

It is the process of changing a software system in such way that it does not alter the external behavior of the code yet improves its internal structure.

OO Design Concepts

- Design classes
  - Entity classes
  - Boundary classes
  - Controller classes
- Inheritance—all responsibilities of a superclass is immediately inherited by all subclasses
- Messages—stimulate some behavior to occur in the receiving object
- Polymorphism—a characteristic that greatly reduces the effort required to extend the design