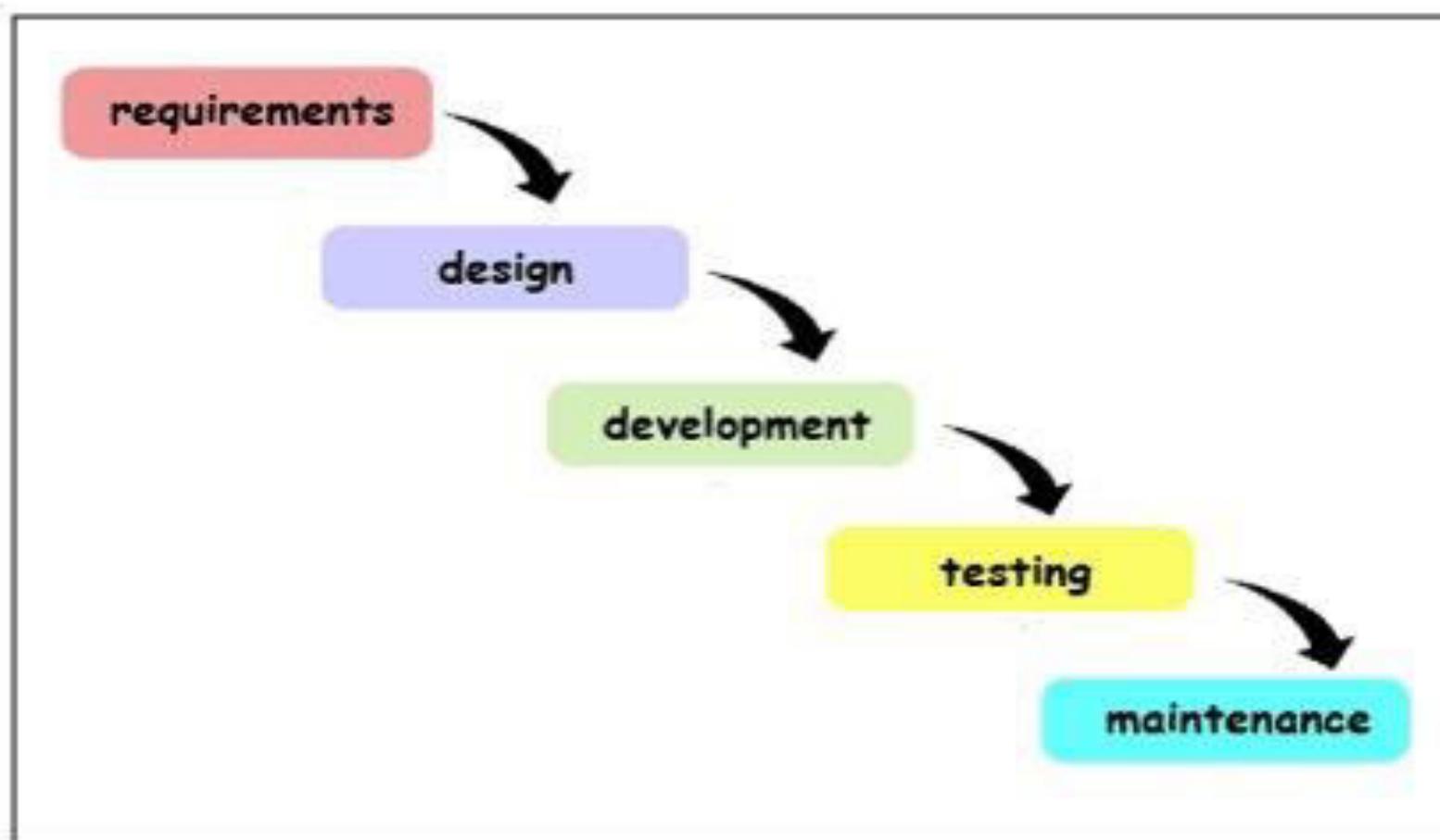


PROCESS MODELS

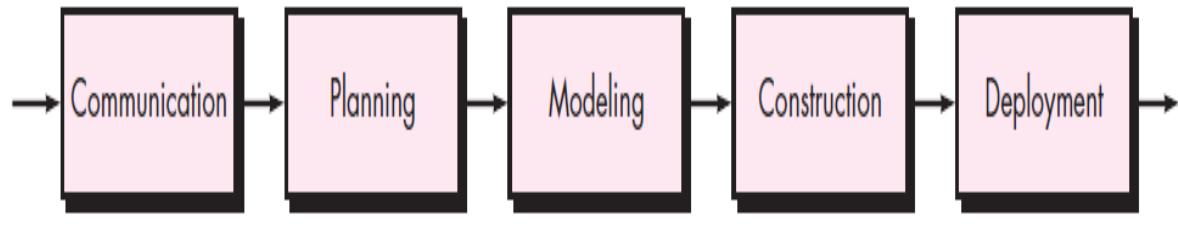
- Perspective Models: The Waterfall
- Incremental Models: Increment and RAD
- Evolutionary Model: Prototype and Spiral
- Specialized Process Models: Component, Formal Methods, AOSD
- Unified Process Model

Software Development Life Cycle (SDLC)

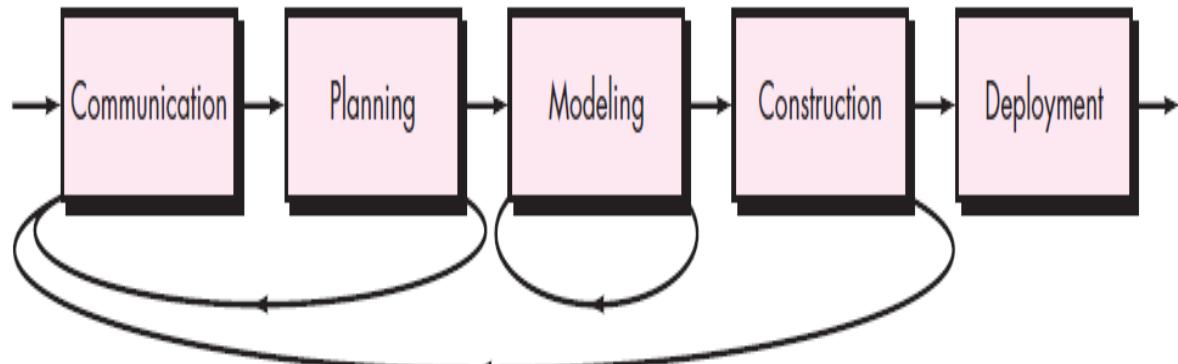
Phases



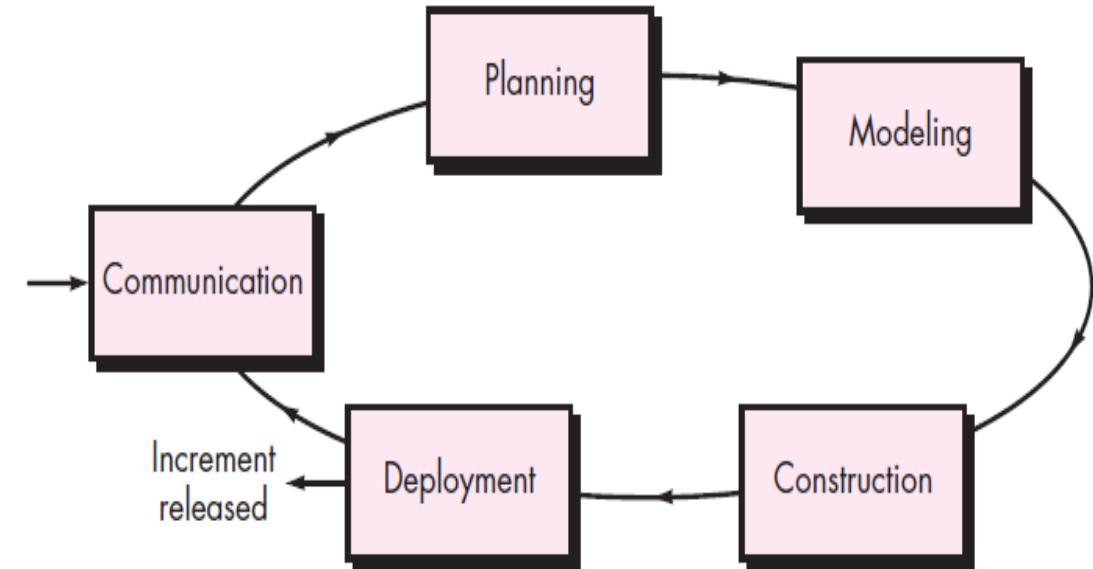
Process Flow



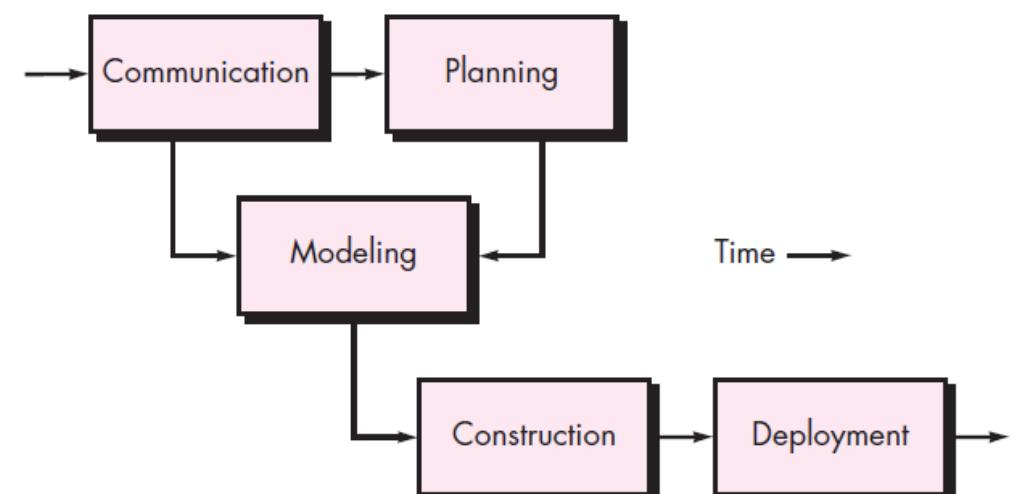
(a) Linear process flow



(b) Iterative process flow



(c) Evolutionary process flow



(d) Parallel process flow

Prescriptive Process Models

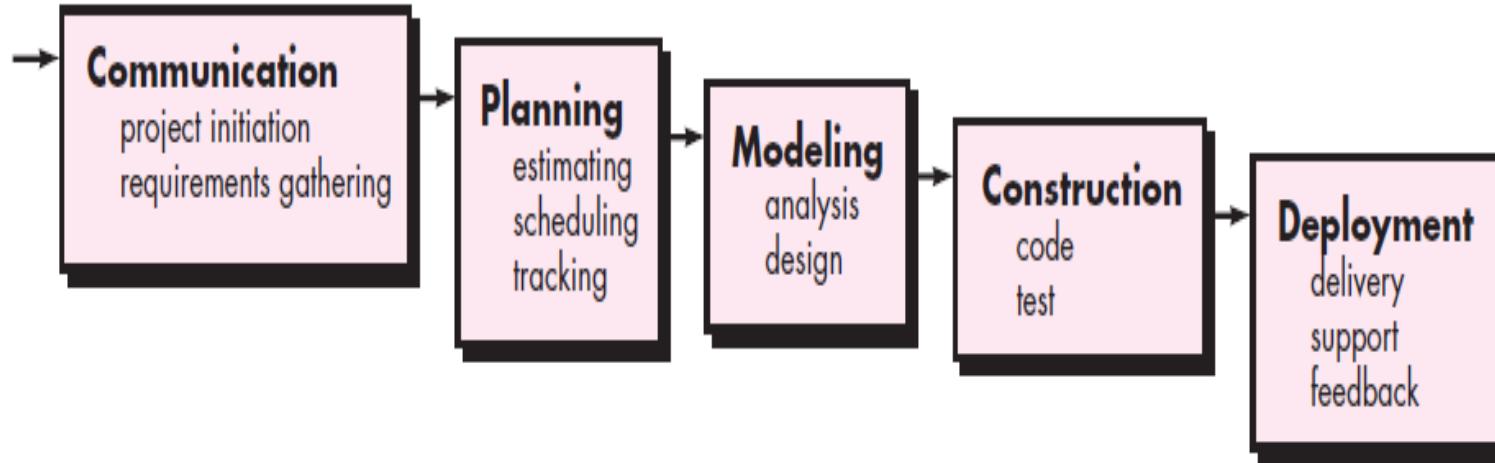
- Often referred to as “conventional” process models
- Prescribe a set of process elements
 - Framework activities
 - Software engineering actions
 - Tasks
 - Work products
 - Quality assurance and
 - Change control mechanisms for each project
- There are a number of prescriptive process models in operation
- Each process model also prescribes a *workflow*
- Various process models differ in their emphasis on different activities and workflow

The Waterfall Model

- Sometimes called the *classic life cycle*
- Suggests a systematic, **sequential (or linear)** approach to s/w development
- The oldest paradigm for s/w engineering

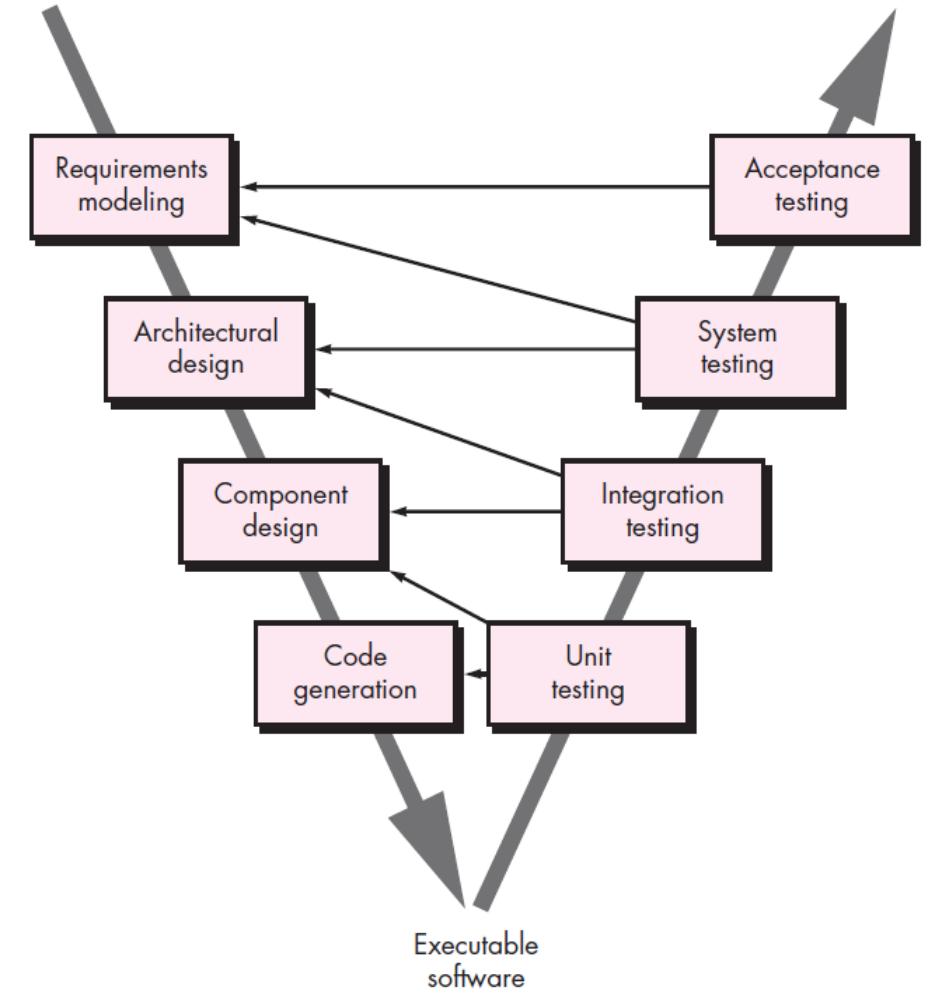
- Works best when –
 - Requirements of a problem are reasonably well understood
 - Well-defined adaptations or enhancements to an existing system must be made
 - Requirements are well-defined and reasonably stable
 - Technology is understood
 - There are no ambiguous requirements
 - Ample resources with required expertise are available freely
 - The project is short

The Waterfall Model



Basic Model

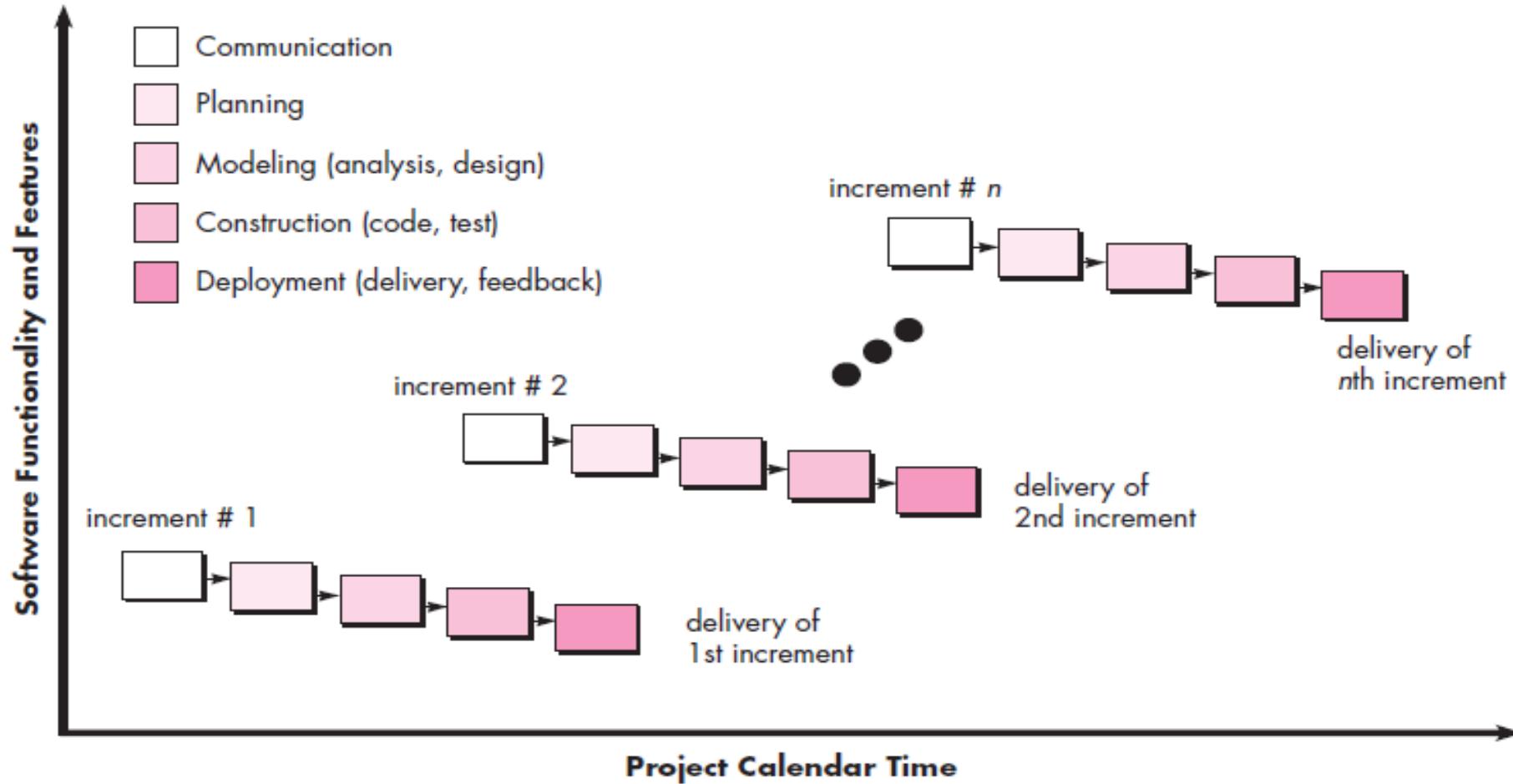
Variations in the Waterfall Model



The Waterfall Model - Problems

- Real projects rarely follow the sequential flow
 - Accommodates iteration indirectly
 - Changes can cause confusion
- It is often difficult for the customer to state all requirements explicitly
 - Has difficulty accommodating the natural uncertainty that exists at the beginning of many projects
- The customer must have patience
 - A working version of the program(s) will not be available until late in the project time-span
 - A major blunder, if undetected until the working program is reviewed, can be disastrous
- Leads to “blocking states” for team members

Incremental Process Models



Incremental Process Models

- Combines elements of the waterfall model applied in an iterative fashion
- Each linear sequence produces deliverable “increments” of the software
- The first increment is often a *core product*
- The core product is used by the customer (or undergoes detailed evaluation)
- Based on evaluation results, a plan is developed for the next increment

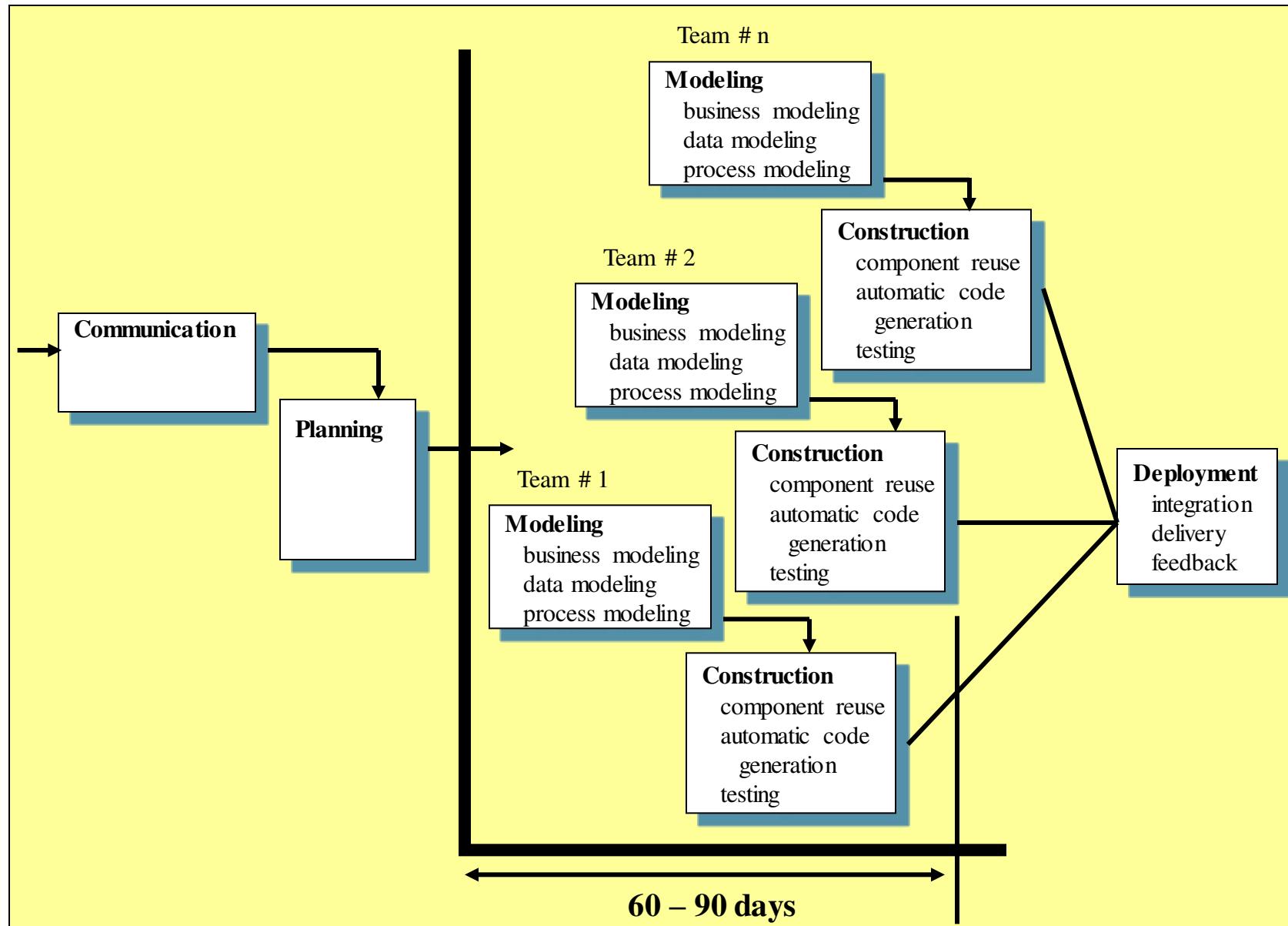
Incremental Process Models

- The incremental process model, like prototyping and other evolutionary approaches, is iterative in nature
- But unlike prototyping, the incremental model focuses on the delivery of an operational product with each increment
- Particularly useful when
 - Staffing is unavailable
 - This model can be used when the requirements of the complete system are clearly defined and understood.
 - Major requirements must be defined; however, some details can evolve with time.
 - There is a need to get a product to the market early.
 - A new technology is being used
 - Resources with needed skill set are not available
 - There are some high-risk features and goals.
- Increments can be planned to manage **technical risks**

The RAD Model

- Rapid Application Development is Incremental Process Model
- Emphasizes on short development cycle
- A “high speed” adaptation of the waterfall model
- Uses a component-based construction approach
- May deliver software within a very short time period (e.g. , 60 to 90 days) if requirements are well understood and project scope is constrained

The RAD Model



The RAD Model

- The time constraints imposed on a RAD project demand “**scalable scope**”
- The application should be modularized and addressed by separate RAD teams
- Integration is required
- Particularly useful when:
 - RAD should be used when there is a need to create a system that can be modularized in 2-3 months of time.
 - It should be used if there's high availability of designers for modeling and the budget is high enough to afford their cost along with the cost of automated code generating tools.
 - RAD SDLC model should be chosen only if resources with high business knowledge are available and there is a need to produce the system in a short span of time (2-3 months).

The RAD Model - Drawbacks

- For large, but scalable projects, RAD requires sufficient human resources
- RAD projects will fail if developers and customers are not committed to the rapid-fire activities
- If a system cannot be properly modularized, building the components necessary for RAD will be problematic
- If high performance is an issue, and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work
- RAD may not be appropriate when **technical risks are high**

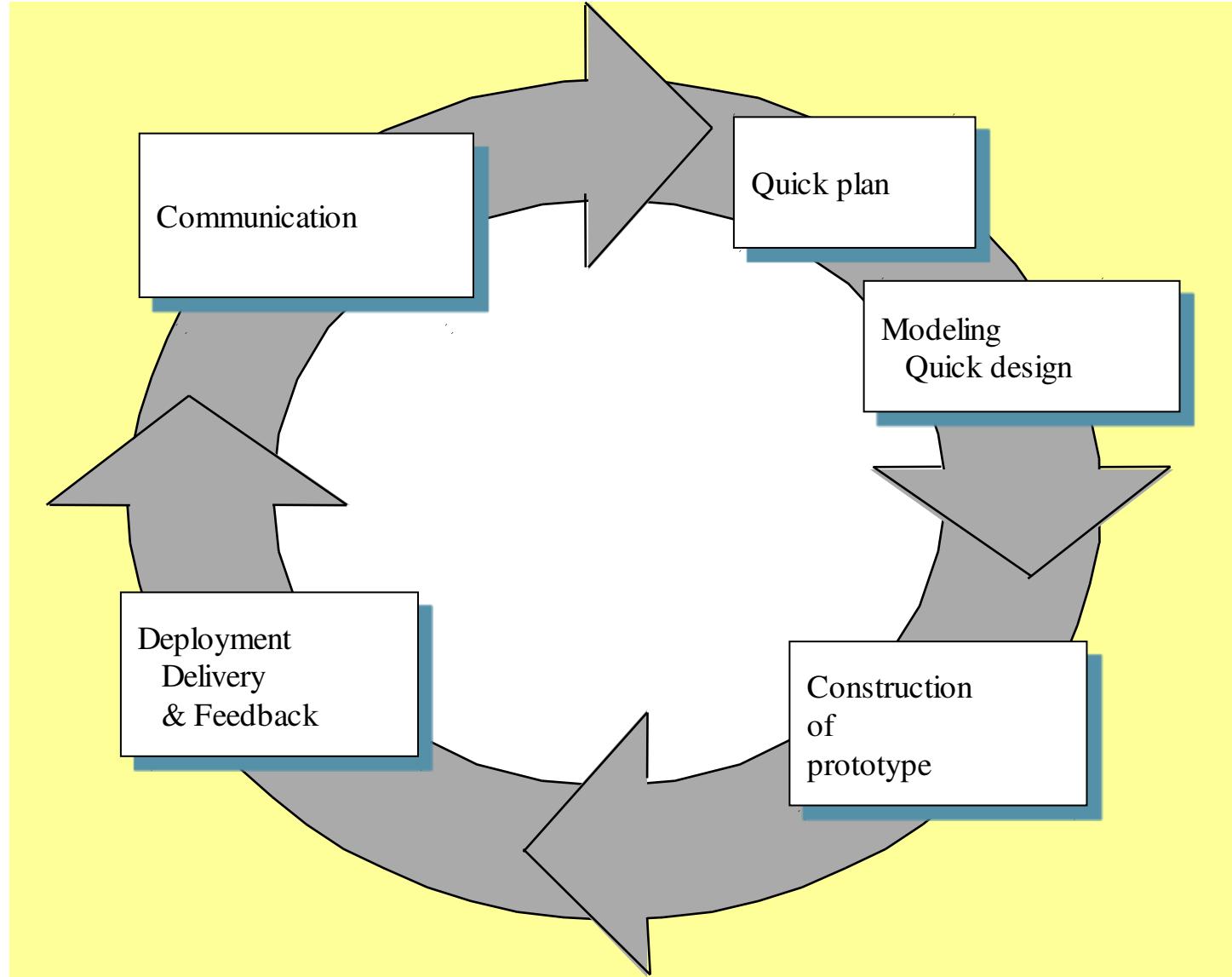
Evolutionary Process Models

- Software, like all complex systems, evolves over a period of time
- Business and product requirements often change as development proceeds, making a straight-line path to an end product is unrealistic
- Evolutionary models are iterative

Prototyping

- Customer defines general objectives but not sure with detailed input, processing and output.
- This model assists the software engineer and the customer to better understand what to be built when requirements are fuzzy.

Prototyping



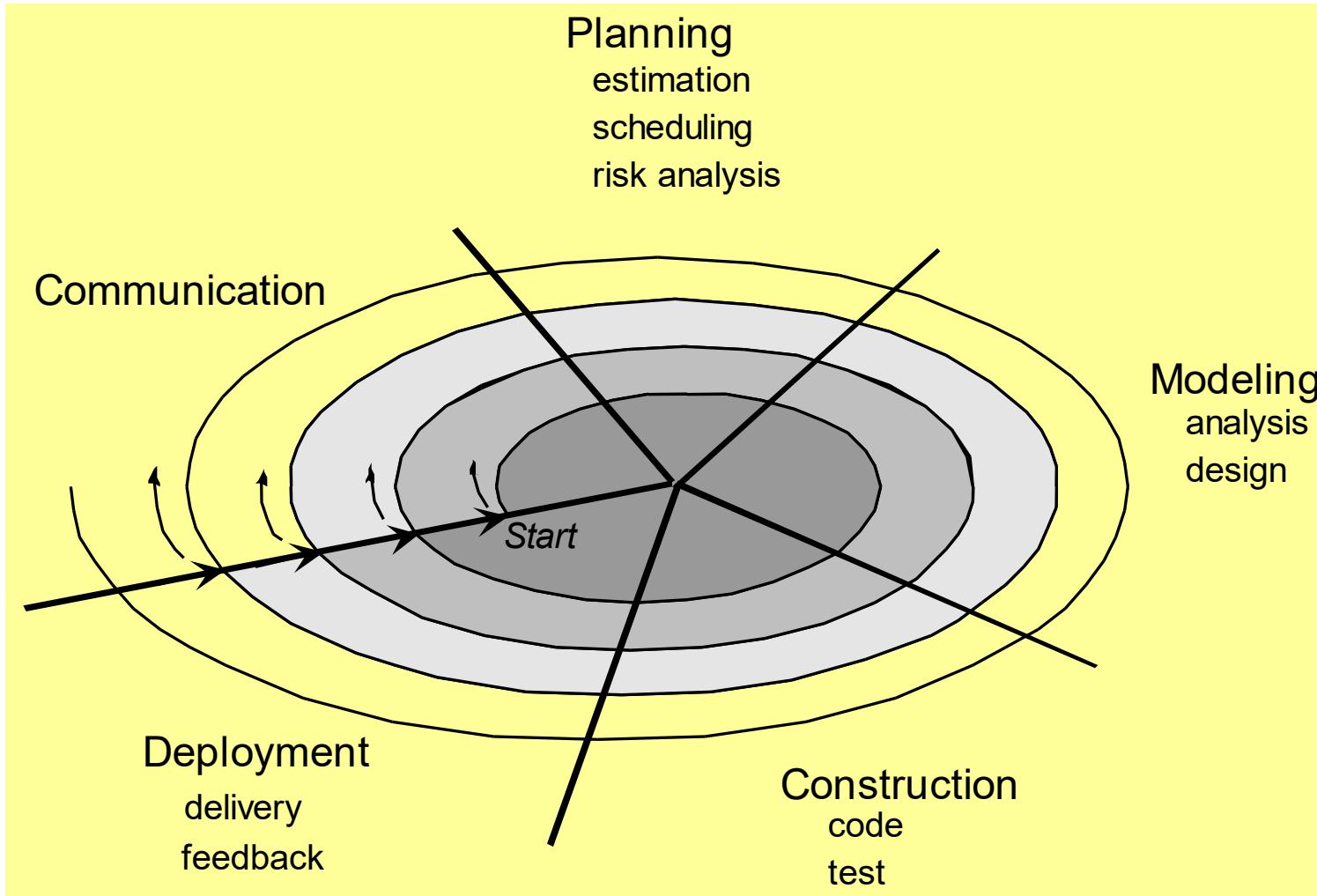
Prototyping - Problems

- Customers may press for immediate delivery of working but inefficient products
- The developer often makes implementation compromises in order to get a prototype working quickly

The Spiral Model

- Couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model
- It provides the potential for rapid development of increasingly more complete versions of the software
- It is a *risk-driven process model* generator
- It has two main distinguishing features
 - Cyclic approach
 - Incrementally growing a system's degree of definition and implementation while decreasing its degree of risk
 - A set of *anchor point milestones*
 - For ensuring stakeholder commitment to feasible and mutually satisfactory system solution

The Spiral Model



The Spiral Model

- Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer s/w
- The circuits around the spiral might represent
 - Concept development project
 - New Product development project
 - Product enhancement project
- The spiral model demands a direct consideration of technical risks at all stages of the project

Particularly useful when

- When costs and risk evaluation is important
- For medium to high-risk projects
- Long-term project commitment unwise because of potential changes to economic priorities
- Users are unsure of their needs
- Requirements are complex
- New product line
- Significant changes are expected (research and exploration)

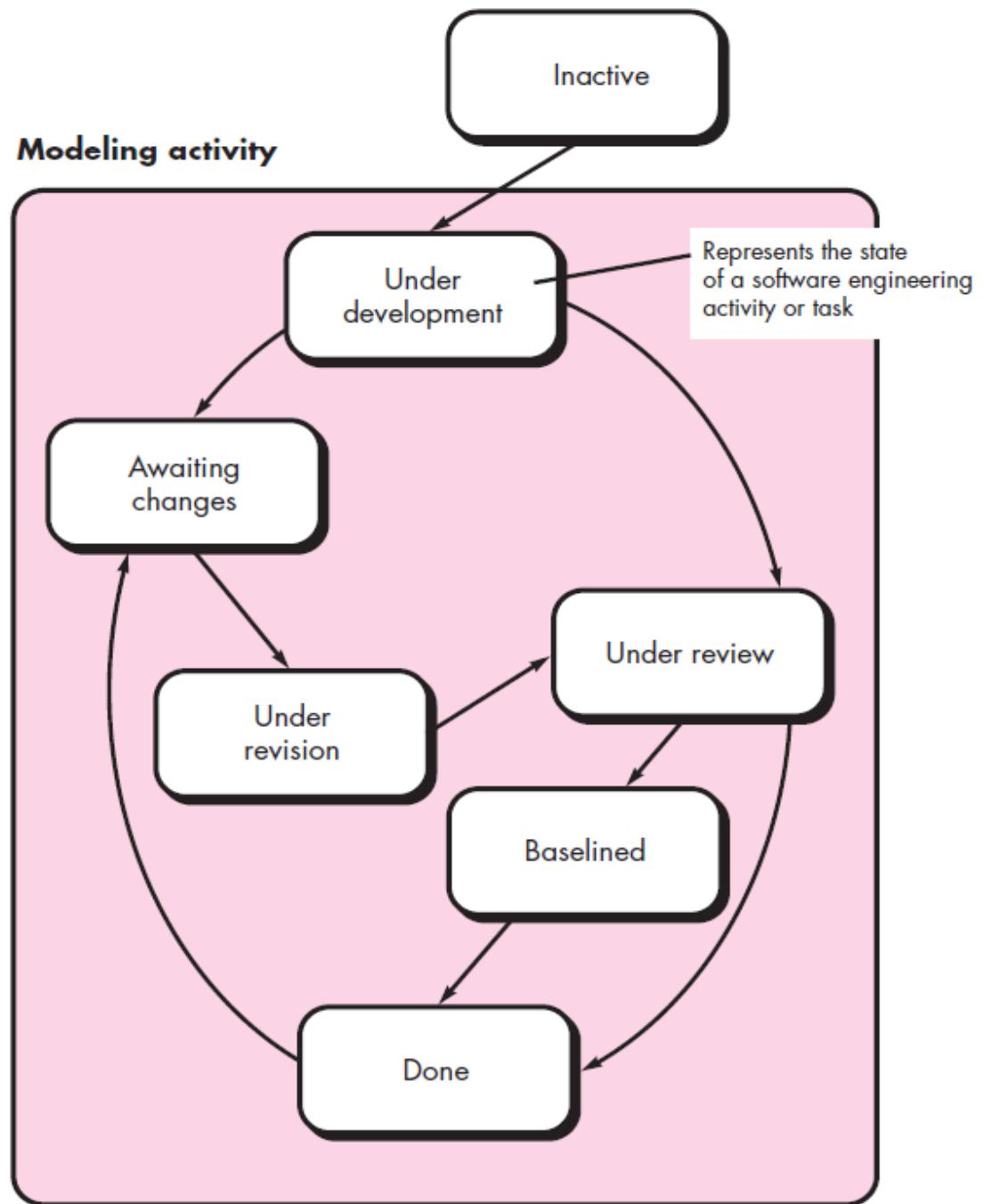
The Spiral Model - Drawbacks

- It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable.
- It demands considerable risk assessment expertise and relies on this expertise for success.
- If a major risk is not uncovered and managed, problems will undoubtedly occur.

The Concurrent Development Model

- Sometimes called *concurrent engineering*
- Can be represented schematically as a series of framework activities, s/w engineering actions and tasks, and their associated states
- Defines a series of events that will trigger transitions from state to state for each of the s/w engineering activities, actions, or tasks
- Applicable to all types of s/w development
- Defines a network of activities
- Events generated at one point in the process network trigger transitions among the states

The Concurrent Development Model



Weaknesses of Evolutionary Process Models

- Uncertainty in the number of total cycles required
 - Most project management and estimation techniques are based on linear layouts of activities
- Do not establish the maximum speed of the evolution
- Software processes should be focused on flexibility and extensibility rather than on high quality, which sounds scary
 - However, we should prioritize the speed of the development over zero defects.

Why?

Specialized Process Models

- Take on many of the characteristics of one or more of the conventional models
- Tend to be applied when a narrowly defined software engineering approach is chosen
- Examples:
 - Component-Based Development
 - The Formal Methods Model
 - Aspect-Oriented Software Development

Component-Based Development

- Commercial off-the-shelf (COTS) software components can be used
- Components should have well-defined interfaces
- Incorporates many of the characteristics of the spiral model
- Evolutionary in nature

Component-Based Development

- Candidate components should be identified first
- Components can be designed as either conventional software modules or object-oriented classes or packages of classes

The Formal Methods Model

- Encompasses a set of activities that leads to formal mathematical specification of computer software
- Have provision to apply a **rigorous, mathematical** notation
- Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily – not through *ad hoc review*, but through the application of mathematical analysis
- Offers the promise of **defect-free** software

The Formal Methods Model – Critical Issues

- The development of formal models is currently quite time-consuming and expensive
- Extensive training is required
- It is difficult to use the models as a communication mechanism for technically unsophisticated customers

Aspect-Oriented Software Development (AOSD)

- Certain “concerns” – customer required properties or areas of technical interest – span the entire s/w architecture
- Example “concerns”
 - Security
 - Fault Tolerance
 - Task synchronization
 - Memory Management
- When concerns cut across multiple system functions, features, and information, they are often referred to as *crosscutting concerns*

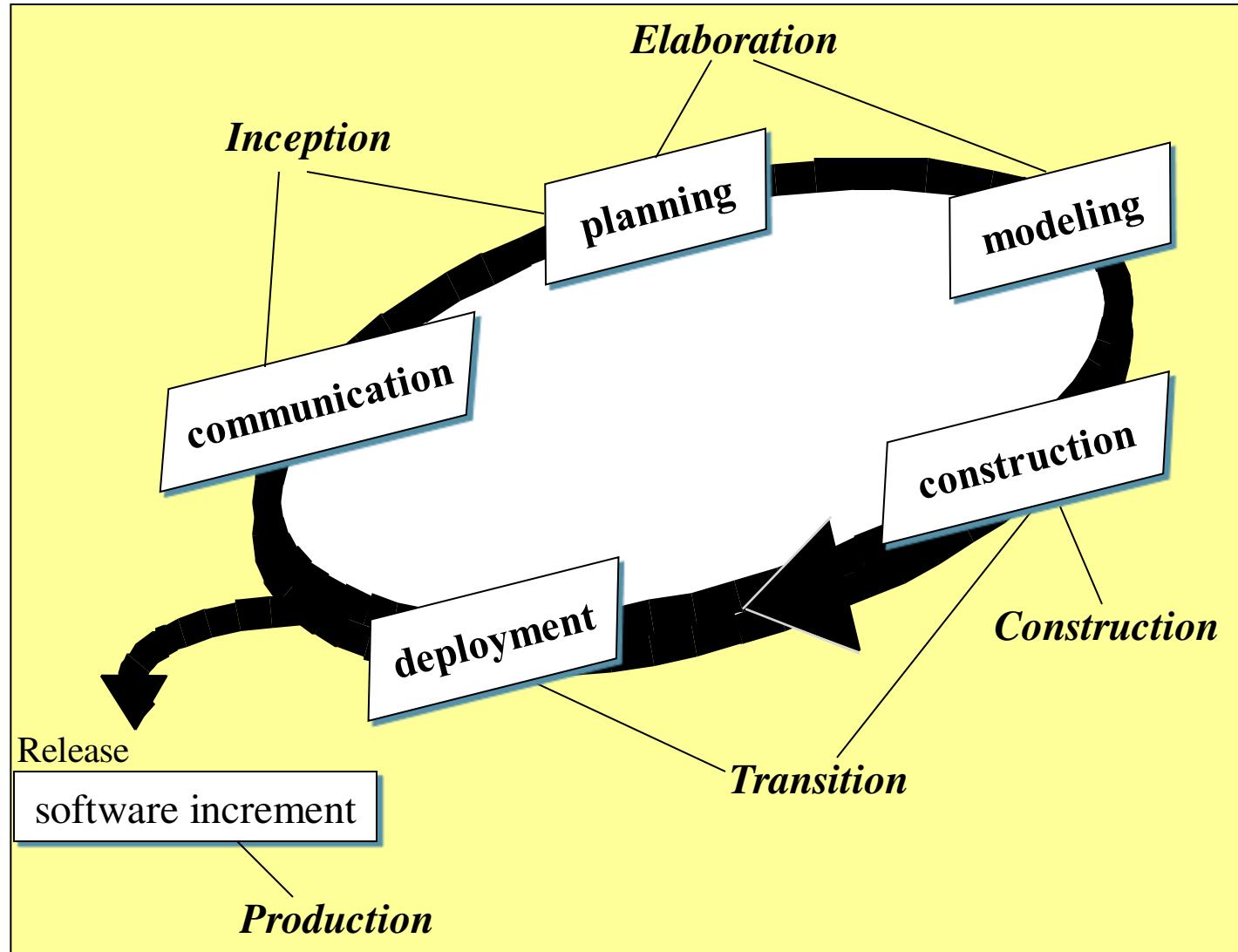
Aspect-Oriented Software Development (AOSD)

- *Aspectual requirements* define those crosscutting concerns that have impact across the s/w architecture
- AOSD or AOP (Aspect-Oriented Programming) provides a process and methodological approach for defining, specifying, designing, and constructing aspects – “mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern”
- A distinct aspect-oriented process has not yet matured
- It is likely that AOSD will adopt characteristics of both the spiral and concurrent process models

The Unified Process (UP)

- It is a use-case driven, architecture-centric, iterative and incremental software process
- UP is an attempt to draw on the best features and characteristics of conventional s/w process models
- Also implements many of the best principles of **agile software development**
- UP is a framework for object-oriented software engineering using **UML (Unified Modeling Language)**

Phases of the Unified Process



Phases of UP - Inception

- Encompasses both customer communication and planning activities
- Fundamental business requirements are described through a set of preliminary use-cases
 - A **use-case** describes a sequence of actions that are performed by an actor (e.g., a person, a machine, another system) as the actor interacts with the software
- A rough architecture for the system is also proposed

Phases of UP - Elaboration

- Encompasses customer communication and modeling activities
- Refines and expands the preliminary use-cases
- Expands the architectural representation to include **five different views of the software**
 - The use-case model
 - The analysis model
 - The design model
 - The implementation model
 - The deployment model
- In some cases, elaboration creates an “**executable architectural baseline**” that represents a “**first cut**” executable system

Phases of UP - Construction

- Makes each use-case operational for end-users
- As components are being implemented, **unit tests** are designed and executed for each
- Integration activities (component assembly and **integration testing**) are conducted
- Use-cases are used to derive a suite of **acceptance tests**

Phases of UP - Transition

- Software is given to end-users for **beta testing**
- The software team creates the necessary support information –
 - User manuals
 - Trouble-shooting guides
 - Installation procedures
- At the conclusion of the transition phase, the software increment becomes a usable software release

Phases of UP - Production

- Coincides with the deployment activity of the generic process
- The on-going use of the software is monitored
- Support for the operating environment (infrastructure) is provided
- Defect reports and requests for changes are submitted and evaluated

Unified Process Work Products

- Inception
 - Vision document
 - Initial use-case model
- Elaboration
 - Analysis model, design model
- Construction
 - Implementation model, deployment model, test model
- Transition
 - Delivered software, beta test reports, general user feedback

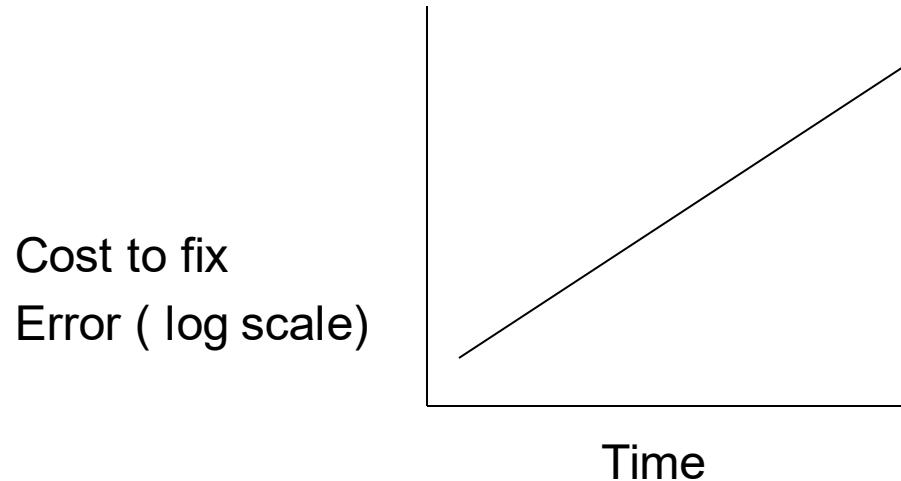
Distribution of effort...

- How programmers spend their time
 - Typing programs 13%
 - Searching and Reading programs 16%
 - Job communication 32%
 - Others 39%
- Programmers spend more time in reading programs than in writing them.
- Writing programs is a small part of their lives.

Defects

- Distribution of error occurrences by phase is
 - Req. - 20%
 - Design - 30%
 - Coding - 50%
- Defects can be injected at any of the major phases.
- Cost of latency: Cost of defect removal increases exponentially with latency time.

Defects...



- Cheapest way to detect and remove defects close to where it is injected.
- Hence must check for defects after every phase.

Factors to be considered for Software Development

- User Satisfaction
- Time
- Quality factors
- Adaptable for changes
- Risk Management
- Evolution

AGILE DEVELOPMENT

Common Fears for Developers

- The project will produce the wrong product.
- The project will produce a product of inferior quality.
- The project will be late.
- We'll have to work 80-hour weeks.
- We'll have to break commitments.
- We won't be having fun.

The Manifesto for Agile Software Development

“We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- *Individuals and interactions* over processes and tools
- *Working software* over comprehensive documentation
- *Customer collaboration* over contract negotiation
- *Responding to change* over following a plan

That is, while there is value in the items on the right, we value the items on the left more.”

-- Kent Beck et al.

What is “Agility”?

- Effective (rapid and adaptive) response to change
- Effective communication among all stakeholders
- Drawing the customer onto the team
- Organizing a team so that it is in control of the work performed

Yielding ...

- Rapid, incremental delivery of software

An Agile Process

- Driven by customer descriptions of what is required (scenarios)
- Recognizes that plans are short-lived
- Develops software iteratively with a heavy emphasis on construction activities
- Delivers multiple ‘software increments’
- Adapts as changes occur

Principles of Agility

- Our highest priority is to **satisfy the customer** through early and continuous delivery of valuable software.
- Welcome **changing requirements**, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the **shorter time scale**.
- Business people and developers must **work together** daily throughout the project.

Principles of Agility

- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

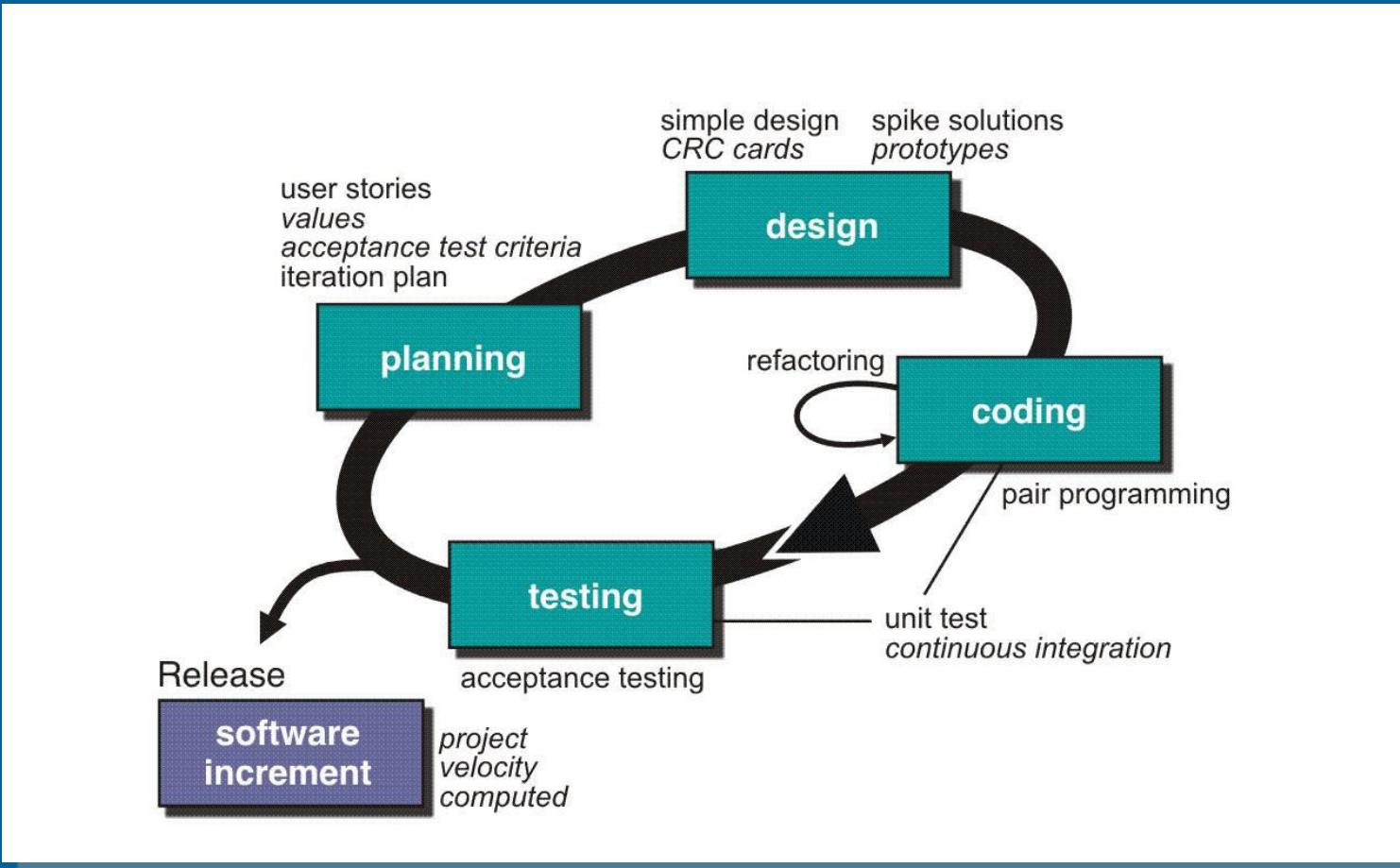
Principles of Agility

- Continuous attention to **technical excellence** and good design enhances agility.
- **Simplicity** - the art of maximizing the amount of work not done - is essential.
- The best architectures, requirements, and designs emerge from **self-organizing teams**.
- At regular intervals, the team reflects on how to become more **effective**, then tunes and adjusts its behavior accordingly.

Extreme Programming (XP)

- The most widely used agile process, originally proposed by Kent Beck [BEC99]
- XP uses an object-oriented approach as its preferred development paradigm
- Defines four (4) framework activities
 - Planning
 - Design
 - Coding
 - Testing

Extreme Programming (XP)



XP - Planning

- Begins with the creation of a set of stories (also called *user stories*)
- Each story is written by the customer and is placed on an index card
- The customer assigns a value (i.e. a priority) to the story
- Agile team assesses each story and assigns a cost
- Stories are grouped to form a deliverable increment
- A commitment is made on delivery date
- After the first increment “project velocity” is used to help define subsequent delivery dates for other increments

XP - Design

- Follows the KIS (keep it simple) principle
- Encourage the use of CRC (class-responsibility-collaborator) cards
- For difficult design problems, suggests the creation of “*spike solutions*”—a design prototype
- Encourages “*refactoring*”—an iterative refinement of the internal program design
- Design occurs both before and after coding commences

XP - Coding

- Recommends the construction of a series of **unit tests** for each of the stories before coding commences
- Encourages “*pair programming*”
 - Mechanism for real-time problem solving and real-time quality assurance
 - Keeps the developers focused on the problem at hand
- Needs continuous integration with other portions (stories) of the s/w, which provides a “**smoke testing**” environment

XP - Testing

- Unit tests should be implemented using a framework to make testing automated. This encourages a regression testing strategy.
- Integration and validation testing can occur on a daily basis
- *Acceptance tests*, also called *customer tests*, are specified by the customer and executed to assess customer visible functionality
- Acceptance tests are derived from user stories

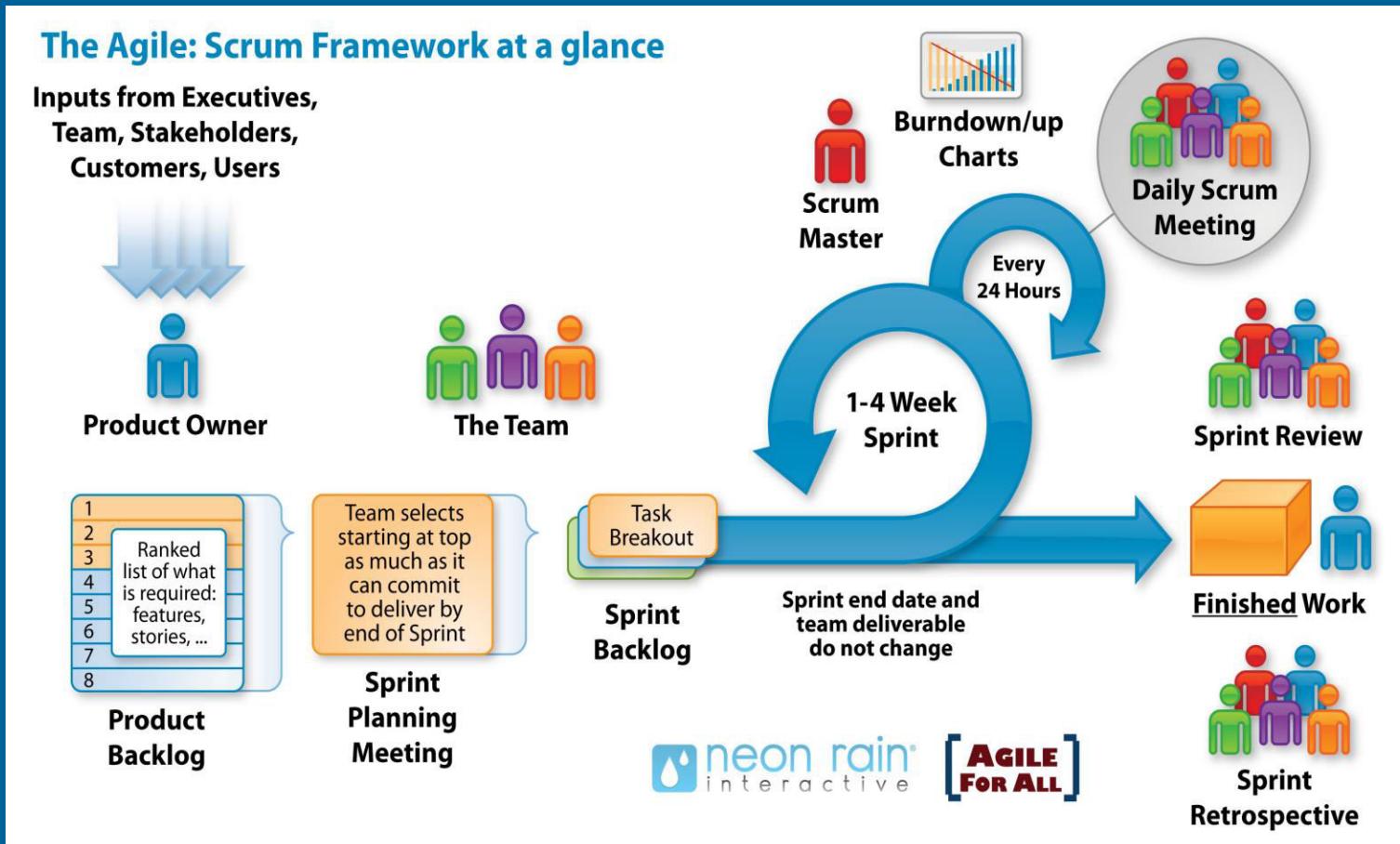
Scrum



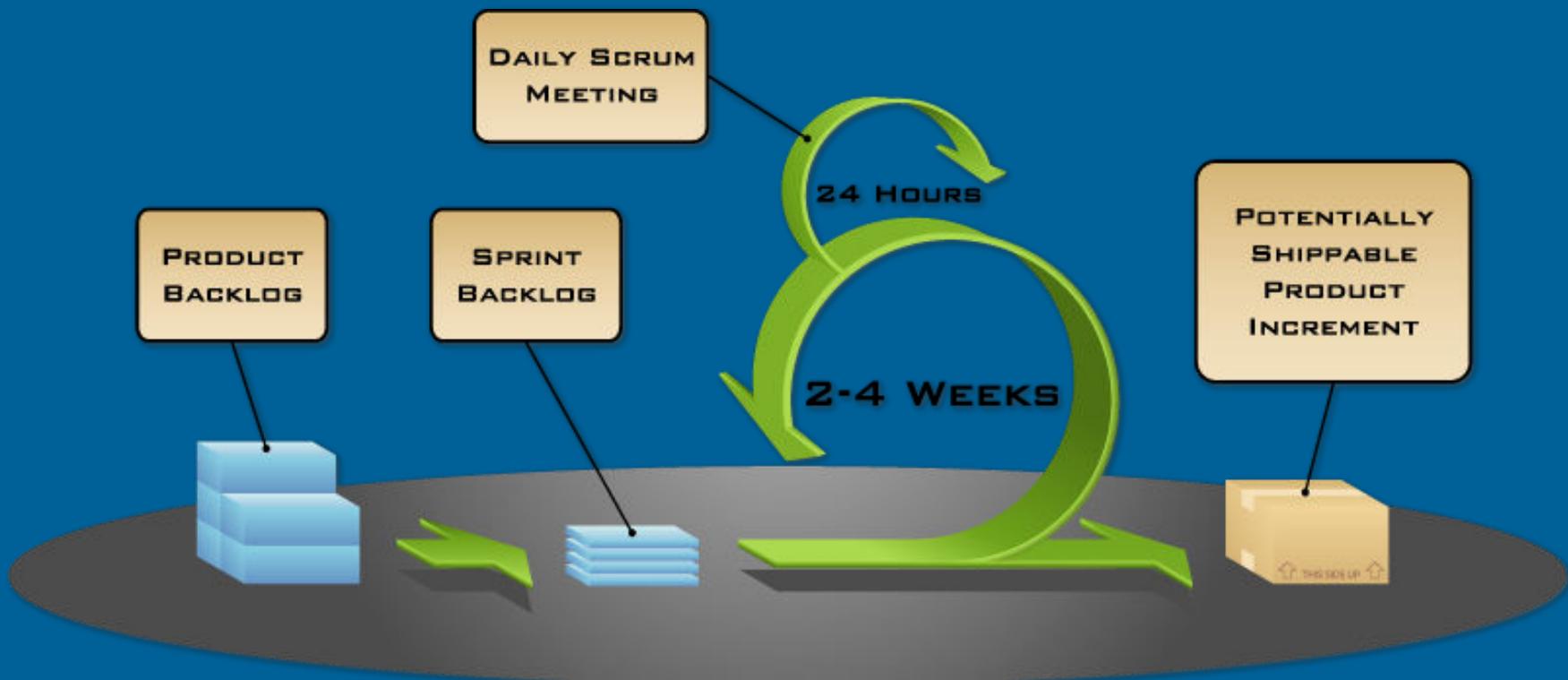
Scrum

- A software development method Originally proposed by Schwaber and Beedle (an activity occurs during a rugby match) in early 1990.
- Scrum—distinguishing features
 - Development work is partitioned into “**packets**”
 - **Testing and documentation are on-going** as the product is constructed
 - Work units occurs in “**sprints**” and is derived from a “**backlog**” of existing changing prioritized requirements
 - Changes are not introduced in sprints (short term but stable) but in backlog.
 - **Meetings are very short** (15 minutes daily) and sometimes conducted without chairs (what did you do since last meeting? What obstacles are you encountering? What do you plan to accomplish by next meeting?)
 - “**demos**” are delivered to the customer with the time-box allocated. May not contain all functionalities. So customers can evaluate and give feedbacks.

Scrum



How Scrum Works?



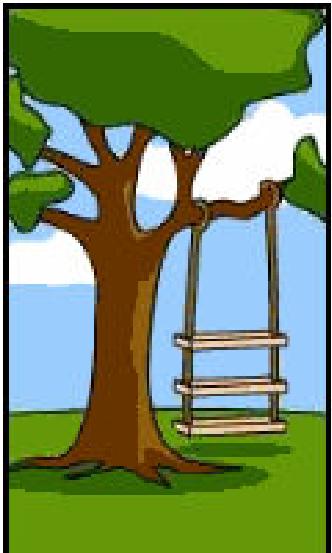
Sprints

- Scrum projects make progress in a series of “sprints”
 - Analogous to XP iterations
- Target duration is one month
 - +/- a week or two
 - But, a constant duration leads to a better rhythm
- Product is designed, coded, and tested during the sprint

Other Agile Processes

- Adaptive Software Development (ASD)
- Dynamic Systems Development Method (DSDM)
- Crystal
- Feature Driven Development
- Agile Modeling (AM)

Requirement Engineering



How the customer explained it



How the Project Leader understood it



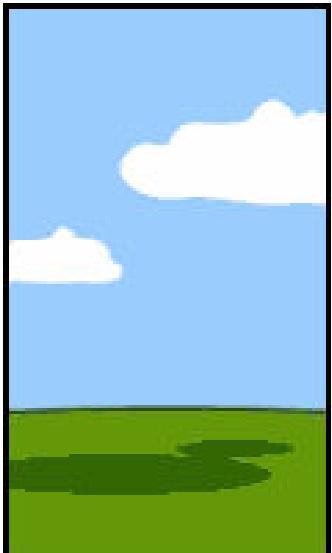
How the Analyst designed it



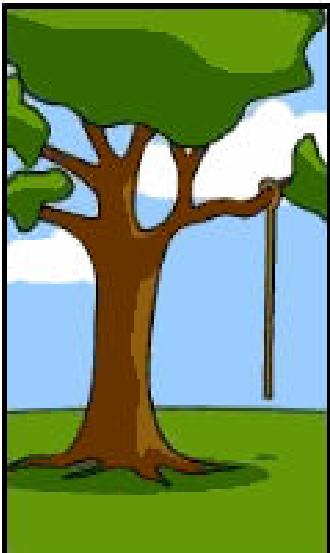
How the Programmer wrote it



How the Business Consultant described it



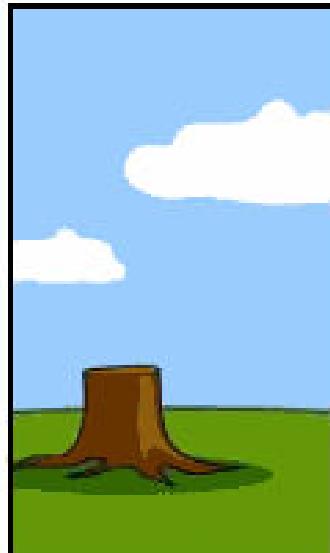
How the project was documented



What operations installed



How the customer was billed



How it was supported



What the customer really needed

1. A Bridge to Design and Construction

- Understanding the requirements of a problem is among the most difficult tasks that face a software engineer.
- Requirements engineering establishes a solid base for design and construction. Without it, the resulting software has a high probability of not meeting customer's needs.
- Requirement engineering builds a bridge to design and construction. But where does the bridge originate?
 - Begins at the feet of the project stakeholders, where business need is defined, user scenarios are described, functions and features are delineated and project constraints are identified.

2. Requirements Engineering Tasks

- Requirements engineering provides the appropriate mechanism for
 - Understanding what the customer wants,
 - Analyzing need,
 - Assessing feasibility,
 - Negotiating a reasonable solution,
 - Specifying the solution unambiguously,
 - Validating the specification and
 - Managing the requirements as they are transformed into an operational system.
- The requirements engineering process is accomplished through the execution of 7 distinct functions- *inception, elicitation, elaboration, negotiation, specification, validation and management*

2.1 Inception

- Expect to do bit of design during requirements work and a bit of requirements work during design
- Software engineers ask a set of context-free questions like
 - Who is behind the request for this work?
 - Who will use the solution?
 - What will be the economic benefit of a successful solution?
 - Is there another source for the solution that you need?
- The intent is to establish
 - A basic understanding of the problem
 - The nature of the solution that is desired
 - Effectiveness of preliminary communication and collaboration between the customer and the developer

2.2 Elicitation

- Questions asked in a manner that no further doubt remains
- Difficulty in requirement elicitation
 - Problems of scope- The boundary of the system is ill-defined or the customer/user specify unnecessary technical detail that may confuse rather than clarify
 - Problems of understanding- The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment etc..
 - Problems of volatility- The requirement change over time

2.3 Elaboration

- The information obtained from the customer during inception and elicitation is expanded and refined during elaboration.
- Elaboration is driven by the creation and refinement of user scenarios that describe how the end-user will interact with the system
- Each user scenario is parsed
- The attributes of each analysis class are defined and the services that are required by each class are identified.
- The relationships and collaboration between classes are identified and a variety of supplementary UML diagrams are produced
- The end-result of elaboration is an analysis model that defines the informational, functional and behavioral domain of the problem

2.4 Negotiation

- It is relatively common for different customers or users to propose conflicting requirements, arguing that their version is “essential for our special needs”.
- The requirement engineer must reconcile these conflicts through a process of *negotiation*.
- Customers, users and other stakeholders are asked to rank requirements and then discuss conflicts in priority.
 - Risks
- Requirements are eliminated, combined and/or modified so that each party achieves some measure of satisfaction.

2.5 Specification

- The formality and format of a specification varies with the size and the complexity of the software to be built.
- A specification can be
 - Written document,
 - Set of graphical models,
 - Formal mathematical model,
 - Collections of usage scenarios,
 - Prototype or any combination of these
- The specification is the final work product produced by the requirements engineer
 - It servers as the foundation for subsequent software engineering activities.

2.6 Validation

- A key concern during requirements validation is consistency.
 - Use the analysis model to ensure that requirement have been consistently stated
- Requirement validation examines the specification to ensure that all software requirements have been stated
 - Unambiguously;
 - That inconsistencies and errors have been detected and corrected;
 - The work products conform to the standards established for the process, the project and the product

2.7 Requirements Management

- Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds.
- When a system is large and complex, determining the connections between requirements can be a daunting task
 - Use traceability tables to make the job a bit easier
- Requirement management begins with the identification
 - Each requirement is assigned a unique identifier
- Once the requirements have been identified, traceability tables are developed
 - Each *traceability* table relates requirements to one or more aspects of the system or its environment

2.7 Requirements Management (contd..)

Traceability tables

- **Features traceability table-** shows how requirements relate to important customer observable system/product features
- **Source traceability table-** identifies the source of each requirement.
- **Dependency traceability table-** indicates how requirements are related to one another
- **Subsystem traceability table-** categories requirements by the subsystem that they govern
- **Interface traceability table-** shows how requirements relate to both internal and external system interfaces

3. Initiating The Requirements Engineering Process

3.1 Identifying the stakeholders

- The stakeholders who benefits in a direct or indirect way from the system which is being developed
 - Business operations managers,
 - Product managers,
 - Marketing people,
 - Internal and external customers,
 - End-users, Consultants, Product engineers, Software engineers,
 - Support and maintenance engineers; and others

3.2 Recognizing multiple viewpoints

- Because many different stakeholders exist, the requirements of the system will be explored from many different points of view.
- For example
 - Marketing group is interested in functions and features that will excite the potential market, making the new system easy to sell.
 - Business managers are interested in a feature set that can be built within budget and that will be ready to meet defined market requirements
 - End-user
 - Software engineers
 - Support engineers

3.3 Working toward collaboration

- The job of the requirements engineer is to identify areas of commonality and areas of conflict or inconsistency

3.4 Asking the First questions

- What problem(s) will this solution address?
- Can you show me the business environment in which the solution will be used? etc..

4. Eliciting Requirements

4.1 Collaborative Requirements Gathering

- Meetings are conducted and attended by both software engineers and customers
- An agenda is suggested that is formal enough to cover all important point. etc..

4.2 Quality Function Deployment

- It is a technique that translates the needs of the customer into technical requirements for software
- QFD identifies three types of requirements
 - **Normal requirements**- These requirements reflect objectives and goals stated for a system.
 - **Expected requirements**- These requirements are implicit to the system (customer does not explicitly state them).
 - **Exciting requirements**- These requirements reflect features that go beyond the customer's expectations.

4.3 User Scenarios

- The scenarios (often called use-cases) provide a description of how the system will be used

Use-case Diagram

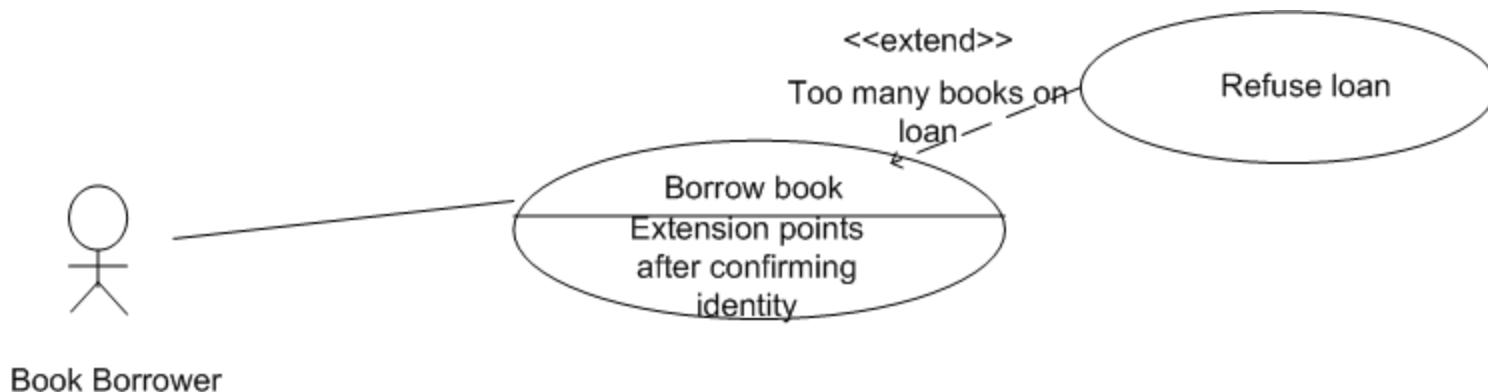
- Actor should be noun
 - E.g.: Student, Person
- Function should be verb/ verb noun
 - E.g.: Withdraw, Borrow books
- Instance of use-case is scenario
- Actor becomes class
- Use-case becomes functions

Validating Requirements

- Once the analysis model is created, it is examined for consistency, omissions and ambiguity.
- A review analysis model addresses the following questions
 - Is each requirement consistent with the overall objectives for the system/product?
 - Is each requirement bounded and unambiguous?
 - Do any requirements conflict with other requirements?
 - Is each requirement achievable in the technical environment that will house the system or product? etc etc..

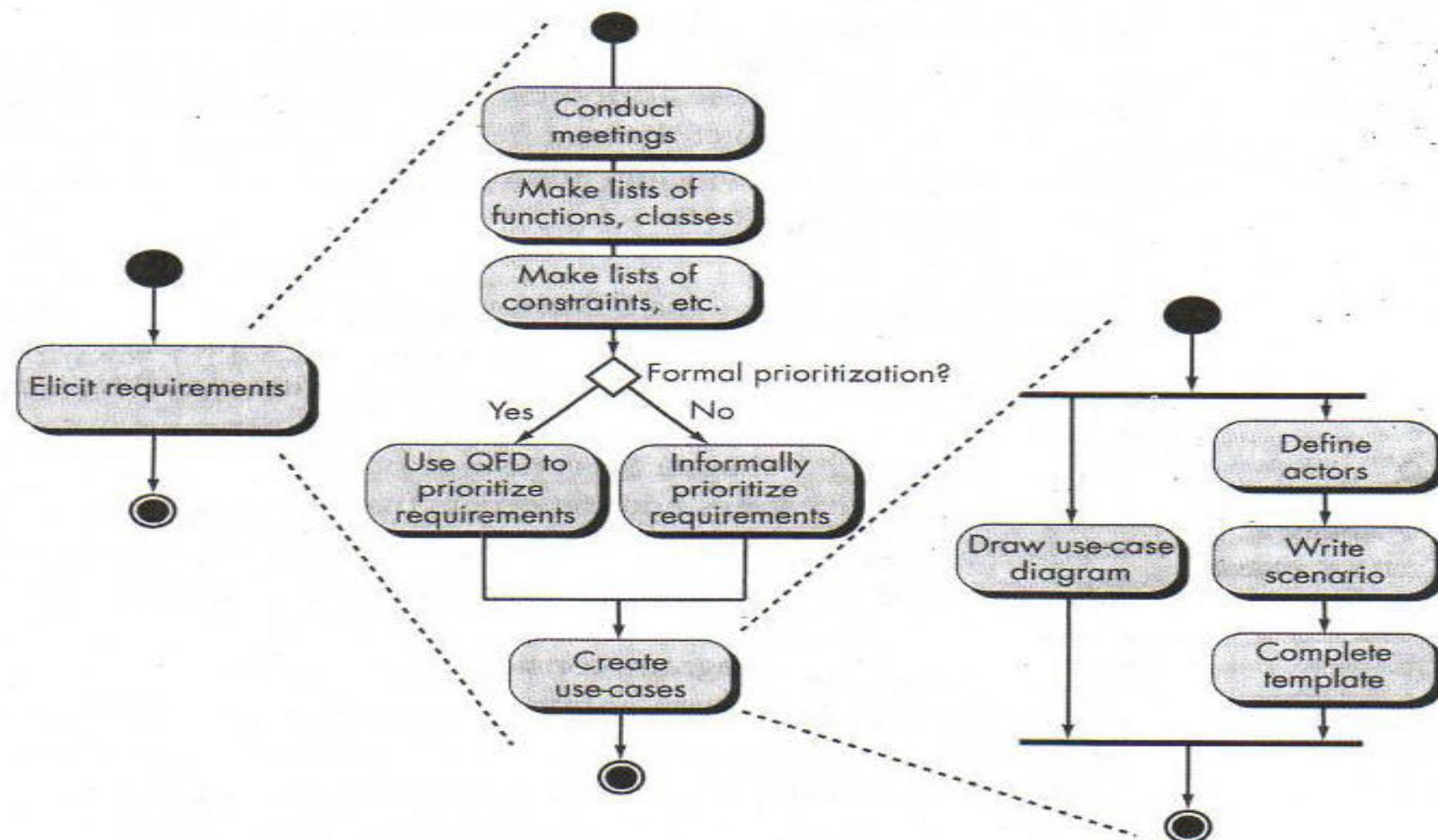
<<extend>> with *extension point*

- The extension point is specified for a base use case and is referenced by an extend relationship between the base use case and the extension use case.



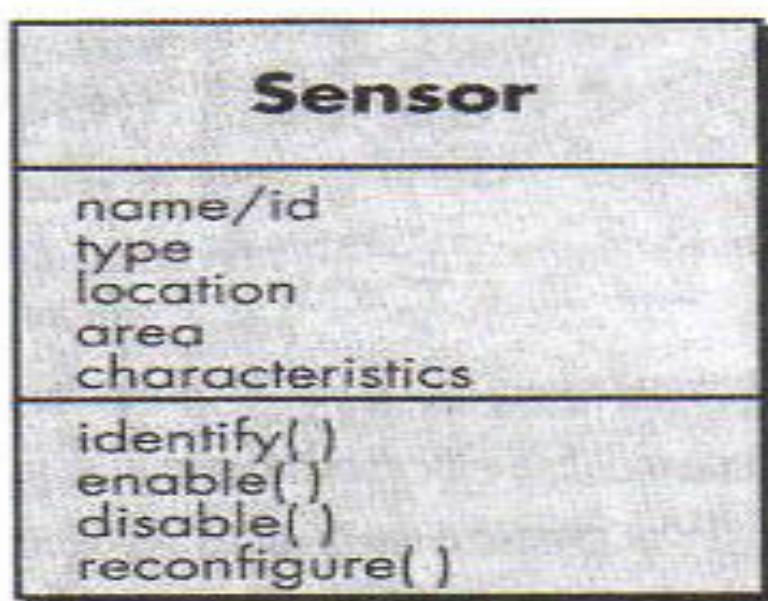
Building the Analysis Model

- Scenario based elements



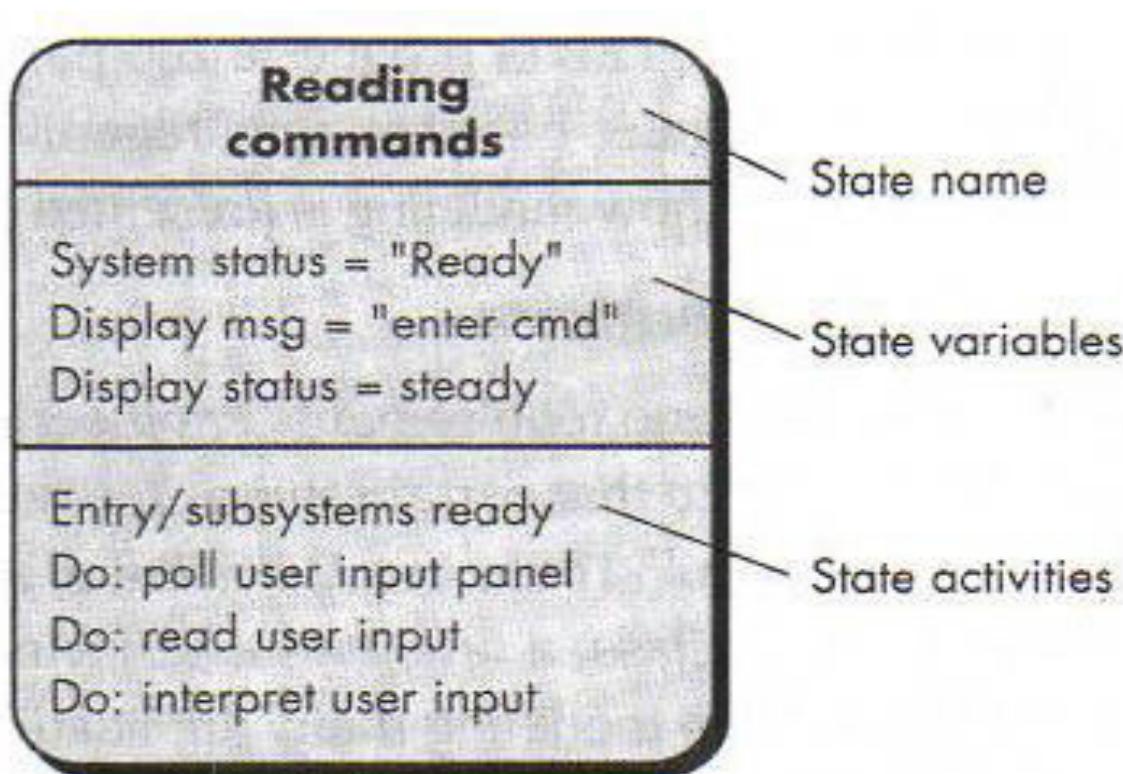
Building the Analysis Model

- Class-based elements



Building the Analysis Model

- State Machine Diagram



Class Based Modeling

Approaches for Identifying Classes

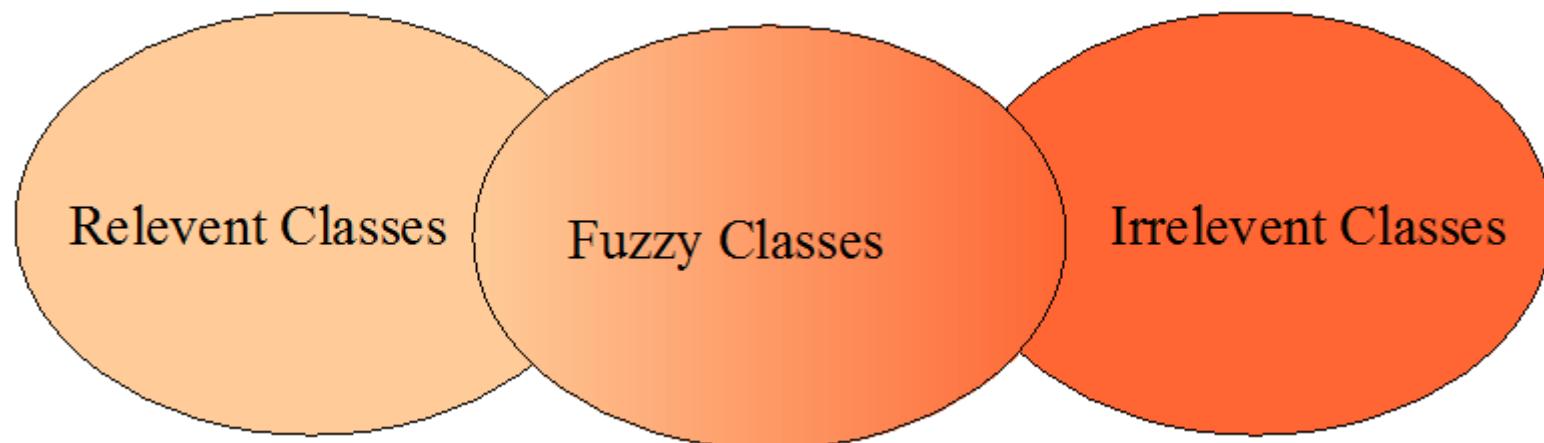
- The noun phrase approach.
- The common class patterns approach.
- The class responsibilities collaboration (CRC) approach.
- The use-case driven approach.

1. Noun-phrase approach

- Initial list of noun phrases
- Eliminate irrelevant classes
- Reviewing redundant classes
- Review adjective classes
- Review possible attributes
- Review the class purpose

Noun Phrase Strategy (Contd...)

- Change all plurals to singular and make a list, which can then be divided into three categories.



1. **Relevant classes**
2. **Fuzzy classes(class that are not sure about)**
3. **Irrelevant classes**

1.1 Initial list of Noun phrases

- Guidelines for identifying *tentative* classes
 - Look for nouns in the System Requirements Specification (SRS)
 - Some classes are implicit or taken from general knowledge
 - All classes are to be taken from problem domain
 - Carefully choose and define class names

1.2 Eliminate Irrelevant classes

- Noun-phrase are put into 3 categories
 - Relevant classes
 - Fuzzy classes
 - Irrelevant classes - unnecessary, so omit

1.3 Reviewing Redundant classes

- Redundant classes from Relevant and Fuzzy classes
 - Do not keep two classes that express the same information
 - In case of more than one to describe the same idea, choose the word used by user of the system

1.4 Review Adjective classes

- Adjectives are used in many ways
 - Adjectives can suggest a different kind of object
 - Different use of the same object
- Find whether the object represented by noun behave differently when the adjective is applied to it.
- Make a new class only if the behavior is different

1.5 Review possible Attributes

- Attribute classes
 - Classes defined only as values should be restated as attributes and not classes

1.6 Review the class purpose

- Each class must have a purpose and every class should be clearly defined.
- Formulate a statement of purpose for each candidate class.
- Otherwise, eliminate the candidate class

Guidelines For Identifying Classes

- The followings are guidelines for selecting classes in your application:
 - Look for nouns and noun phrases in the problem statement.
 - Some classes are implicit or taken from general knowledge.
 - All classes must make sense in the application domain.
 - Avoid computer implementation classes, defer it to the design stage.
 - Carefully choose and define class names.

Guidelines For Identifying Classes (Contd...)

- External Entities (e.g. other systems, devices, people)
- Things (e.g. reports, displays, letters, signals, results)
- Occurrences or Events (e.g. property transfer, fund transfer, robot movements)
- Roles (e.g. manager, engineer, salesperson)
- Organizational Units (e.g. division, group, team)
- Places (e.g. manufacturing floor, loading dock, back door)
- Structures (e.g. sensors, vehicles, computers)
- Not Objects/Attributes (e.g. number, type)

1. Identify potential classes and define above attributes based on above mentioned points.
2. Then apply selection characteristics to retain with potential classes.

Selection characteristics of Potential Classes

- **Retained Information:** information about class must be remembered so that the system should function.
- **Needed Services:** must have set of identifiable operations that can change the value of attributes in some way.
- **Multiple Attributes:** multiple attributes should be represented in the class
- **Common Attributes:** set of common attributes can be defined or applied to all instances of that class.
- **Common Operations:** set of common operations can be defined or applied to all instances of that class.
- **Essential Requirements:** information producing or consuming by external entities should be considered as classes

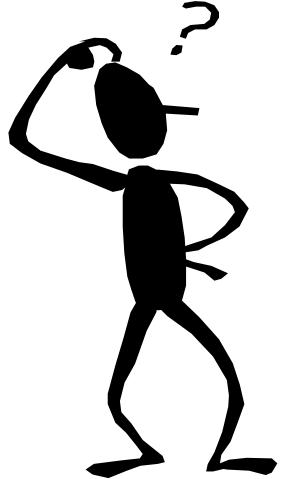
Guidelines For Refining Classes

Redundant Classes:

- Do not keep two classes that express the same information.
- If more than one word is being used to describe the same idea, select the one that is the most meaningful in the context of the system.



Guidelines For Refining Classes (Contd..)



Adjective Classes:

- Does the object represented by the noun behave differently when the adjective is applied to it?
- If the use of the adjective signals that the behavior of the object is different, then make a new class.
- For example, If *Adult Membership* and *Youth Membership* behave differently, than they should be classified as different classes.

Guidelines For Refining Classes (Con't)

Attribute Classes:

- Tentative objects which are used only as values should be defined or restated as attributes and not as a class.
- For example the demographics of Membership are not classes but attributes of the Membership class.

Guidelines For Refining Classes (Con't)

Irrelevant Classes:

- Each class must have a purpose and every class should be clearly defined and necessary.
- If you cannot come up with a statement of purpose, simply eliminate the candidate class.



Identifying a list of candidate classes: Example 1

Books and journals The library contains books and journals. It may have several copies of a given book. Some of the books are for short term loans only. All other books may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of staff may borrow up to 12 items at one time. Only members of staff may borrow journals.

Borrowing The system must keep track of when books and journals are borrowed and returned, enforcing the rules described above.

- Take a coherent, concise statement of the requirement of the system
- Underline its noun and noun phrases, that is, identify the words and phrases that denote things
- This gives a list of candidate classes, which we can then whittle down and modify to get an initial class list for the system

In this particular case we discard

- **Library**, because it is outside the scope of our system
- **Short term loan**, because a loan is really an event, which so far as we know is not a useful object in this system
- **Member of the library**, which is redundant
- **Week**, because it is a measure, not a thing
- **Item**, because it is vague (we need to clarify it)
- **Time**, because it is outside the scope of the system
- **System**, because it is part of the meta-language of requirements description, not a part of domain
- **Rule**, for the same reason

This leaves:

- Book
- Journal
- Copy (of book)
- Library member
- Member of staff

Example 2:

System domain

In the hospital, there are a number of wards, each of which may be empty or have on it one or more patients. There are two types of ward, male wards and female wards. A ward can only have patients of the specified sex on it. Every ward has a fixed capacity, which is the maximum number of patients that can be on it at one time (i.e. the capacity is the number of beds in the ward). Different wards may have different capacities. Each ward has a unique name. The hospital has an Administration department that is responsible for recording information about the hospital's wards and the patients that are on each ward.

The doctors in the hospital are organised into teams, each of which has a unique team code (such as Orthopaedics A, or Paediatrics). Each team is headed by a consultant doctor who is the only consultant doctor in the team; the rest of the team are all junior doctors, at least one of whom must be at grade 1. Each doctor is in exactly one team. The Administration department keeps a record of these teams and the doctors allocated to each team.

Each patient is on a single ward and is under the care of a single team of doctors; the consultant who heads that team is responsible for the patient. A patient may be treated by any number of doctors but they must all be in the team that cares for the patient. A doctor can treat any number of patients.

| | |
|---|--------------------------------|
| hospital | |
| number of wards | doctor |
| patient | team |
| type of ward | team code |
| male ward | Orthopaedics A |
| female ward | Paediatrics |
| ward | grade 1 |
| sex | consultant doctor (consultant) |
| capacity (maximum number of patients, number of beds) | junior doctor |
| name of ward | record of teams and doctors |
| Administration department | care |
| information | number of doctors |
| | number of patients |

Eliminated Classes

- hospital, Administration department (outside the scope of the system)
- number of wards (language idiom – the requirements document could equally well have referred to ‘wards’ or ‘some wards’)
- type, capacity, name (attributes of ward)
- sex (attribute of patient)
- information (generic term referring to the behaviour of the system)
- team code (attribute of team)
- record of teams and doctors (part of the behaviour required of the system)
- number of doctors, number of patients (language idiom)

You may have noticed that there are three nouns,
‘Orthopaedics A,
‘Paediatrics’ and
‘grade 1’,
which have not been eliminated using the guidelines for rejection, yet
which common sense suggests should not be modelled by classes.
‘Orthopaedics A and ‘Paediatrics’ are in fact given in the requirements
document as example values of ‘team code’, which was identified
above as an attribute of team, and ‘grade 1’ is a value of an attribute,
‘grade’, of junior doctor.

male ward

female ward

ward

patient

doctor team

consultant doctor

junior doctor

care

Example 3:

A store wants to automate its inventory. It has point-of-sale terminals that can record all of the items and quantities that a customer purchases. Another terminal is also available for the customer service desk to handle returns. It has a similar terminal in the loading dock to handle arriving shipments from suppliers. The meat department and produce department have terminals to enter losses/discounts due to spoilage.

Identify nouns

Store

Inventory

Point-of-sale terminal

Terminals

Items

Quantity

Purchase

Customer

Customer service desk

Handle returns

Returns

Loading dock

Shipment

Handle shipment

Suppliers

Meat department

Produce department

Department

Enter losses

Enter discount

Spoilage

Eliminate irrelevant nouns

Store

Point-of-sale terminal

inventory

Item

Customer

Customer service desk

Handle returns

Returns

Handle shipment

Shipment

Meat department

Produce department

Department

Enter losses

Enter discount

Eliminate redundancies

Store

Point-of-sale terminal

Item

Customer service desk

Handle returns

Handle shipment

Meat department

Produce department

Enter losses

Enter discount

The final set of classes and objects after the elimination process.

Point-of-sale terminal

Item

Handling return

Handling shipment

Enter losses

Enter discount

Meat department

Produce department

Store

2. Common Class patterns approach

- Based on a knowledge base of the common classes proposed by various researchers
 - Concept class
 - Event class
 - Organization class
 - People class
 - Place class
 - Tangible things and device class
 - ***Review the class purpose***

2.1 Concept Class

- Idea or understanding of the problem
- Principles that are not tangible, and used to organize or keeping track of business activities

E.g.: Performance, reservation

2.2 Event class

- Things happen at a given date and time
- Points in time that must be recorded
 - E.g.: Request, order

2.3 Organization class

- A collection of people, resources, facilities, or group to which the users belong

E.g.: Accounting department

2.4 People class (roles class)

- Different roles users play in interacting with application
- Guideline
 - Classes that represent users of the system
 - E.g.: Operator, clerk
 - Classes representing people who do not use the system but about whom information is kept by the system
 - E.g.: Employee, client, teacher, manager

2.5 Places class

- Areas set aside for people or things
 - Physical locations that the system must keep information about
- E.g.: Building, store, site, travel office

2.6 Tangible things

- Physical objects or groups of objects that are tangible and devices with which application interacts

E.g.: Car, ATM

ATM Case Study –System Requirements

- The bank client must be able to deposit an amount to and withdraw an amount from his or her accounts using the touch screen at the ATM kiosk. Each transaction must be recorded, and the client must be able to review all transactions performed against a given account. Recorded transactions must include the date, time, transaction type, amount and account balance after the transaction
- A bank client can have two types of accounts: a checking account and savings account. For each checking account, one related savings account can exist.
- Access to the bank accounts is provided by a PIN code consisting of four integer digits between 0 and 9

ATM Case Study –System Requirements

- One PIN code allows access to all accounts held by a bank client
- No receipts will be provided for any account transactions
- The bank application operates for a single banking institution only
- Neither a checking nor a savings account can have a negative balance. The system should automatically withdraw money from a related savings account if the requested withdrawal amount on the checking account is more than its current balance. If the balance on a savings account is less than the withdrawal amount requested, the transaction will stop and the bank client will be notified.

Apply Common Class patterns to ATM Case study

- Concept class - Availability
- Events class – Account, Checking account, Savings account, Transaction class
- Organization class – Bank class
- People class – BankClient class
- Places class – Building
- Tangible things and devices class – ATM Kiosk

Guidelines for Naming Classes

- The class should describe a single object, so it should be the singular form of noun.
- Use names that the users are comfortable with.
- The name of a class should reflect its intrinsic nature.

Guidelines for Naming Classes (Con't)

- By the convention, the class name must begin with an upper case letter.
- For compound words, capitalize the first letter of each word - for example, LoanWindow.

Summary

- Finding classes is not easy.
- The more practice you have, the better you get at identifying classes.
- There is no such thing as the “right set of classes.”
- Finding classes is an incremental and iterative process.

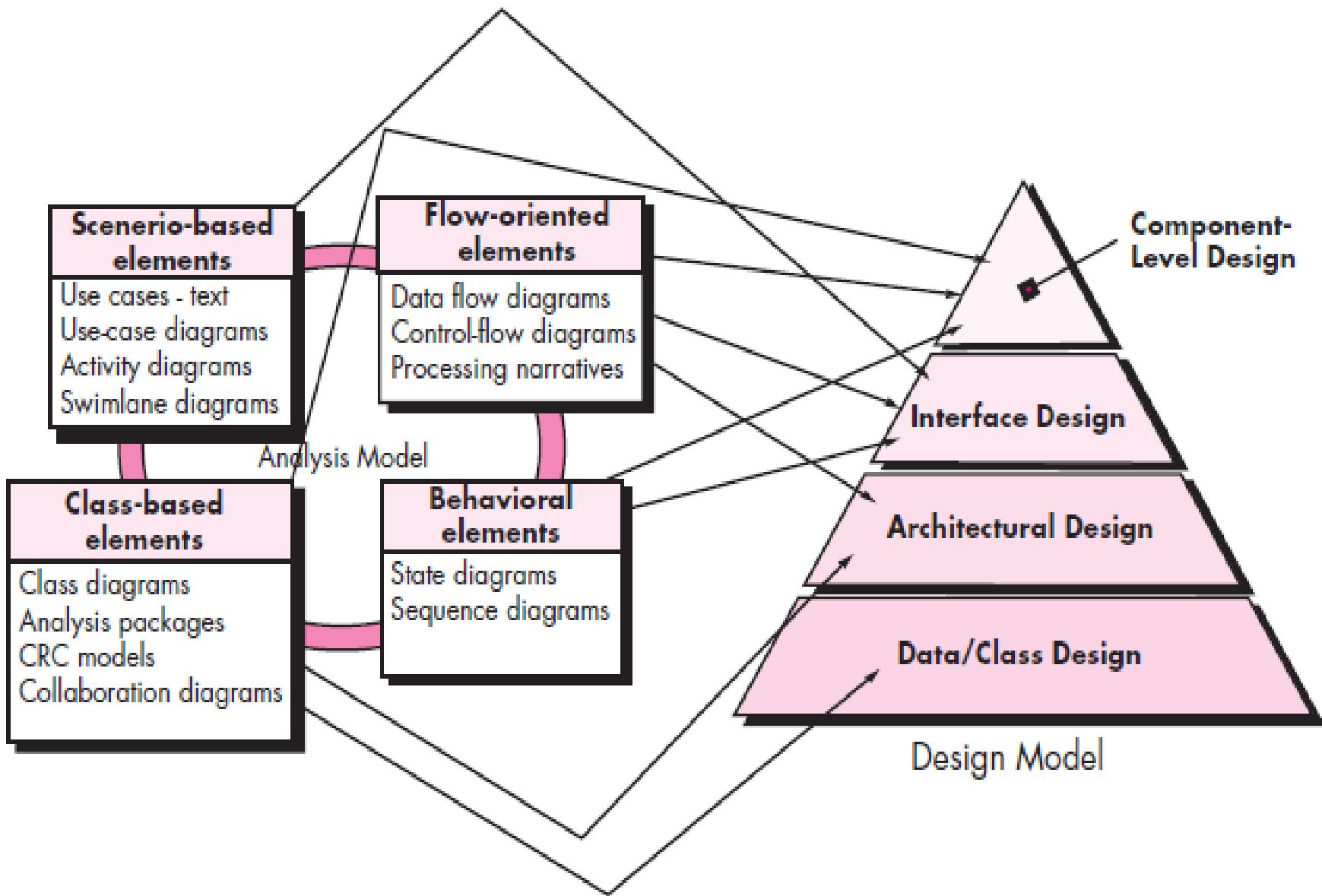


DESIGN ENGINEERING

Design model

Components of a design model :

- **Data Design**
 - Transforms information domain model into data structures required to implement software
- **Architectural Design**
 - Defines relationship among the major structural elements of a software
- **Interface Design**
 - Describes how the software communicates with systems that interact with it and with humans.
- **Component level Design**
 - Transforms structural elements of the architecture into a procedural description of software components



► Software Design -- An iterative process transforming requirements into a “blueprint” for constructing the software.

Goals of the design process:

1. The design must implement all of the explicit requirements contained in the analysis model and it must accommodate all of the implicit requirements desired by the customer.
2. The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
3. The design should address the data, functional and behavioral domains from an implementation perspective

Quality Guidelines

1. A design should exhibit an **architecture** that
 - (1) has been created using recognizable architectural styles or patterns,
 - (2) is composed of components that exhibit good design characteristics
 - (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
2. A design should be **modular**; that is, the software should be logically partitioned into elements or subsystems

3. A design should contain distinct representations of data, architecture, interfaces, and components.
4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning

Quality attributes

- **Functionality** – ability of the system to do the work for which it was intended
- **Usability** – ease of use (user friendliness)
- **Reliability** – failure free software operation
- **Performance** - responsiveness
- **Supportability** – identifying and resolving issue when software fails

Design Attribute for a Good Software

- Cohesion
- Coupling
- Layering
- Abstraction
- Fan-in - Is the number of modules that call a given module.
- Fan-out - Is the numbers of modules that called by a given module.
- Scope of re-use
- Error of isolation
- Clear Decomposition
- Modularity - refers to the extent to which a software may be divided into smaller modules
- Correctness, maintainability, understandability and efficiency

Design Concepts

Abstraction

- ▶ At the highest level of abstraction a solution is stated in broad terms
- ▶ At lower levels of abstraction, a more detailed description of the solution is provided.

Design Concepts(Abstraction)

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)

Design Concepts

- *Software architecture* refers to “the overall structure of the software and the ways in which that structure provides conceptual integrity for a system”
- A set of properties should be specified as part of an architectural design:
- **Structural properties.** This aspect of the architectural design representation defines the components of a system (e.g., modules, objects) and the manner in which those components are packaged and interact with one another.
- **Extra-functional properties.** The architectural design description should address how the design architecture achieves requirements for performance, reliability, security, adaptability, and other system characteristics.
- **Families of related systems.** The design should have the ability to reuse architectural building blocks.

Design Concepts



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

Design Concepts(Modularity)

Five 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.

Design Concepts(Modularity)

Modular Composability

- Design method should enable reuse of existing components.

Modular Decomposability

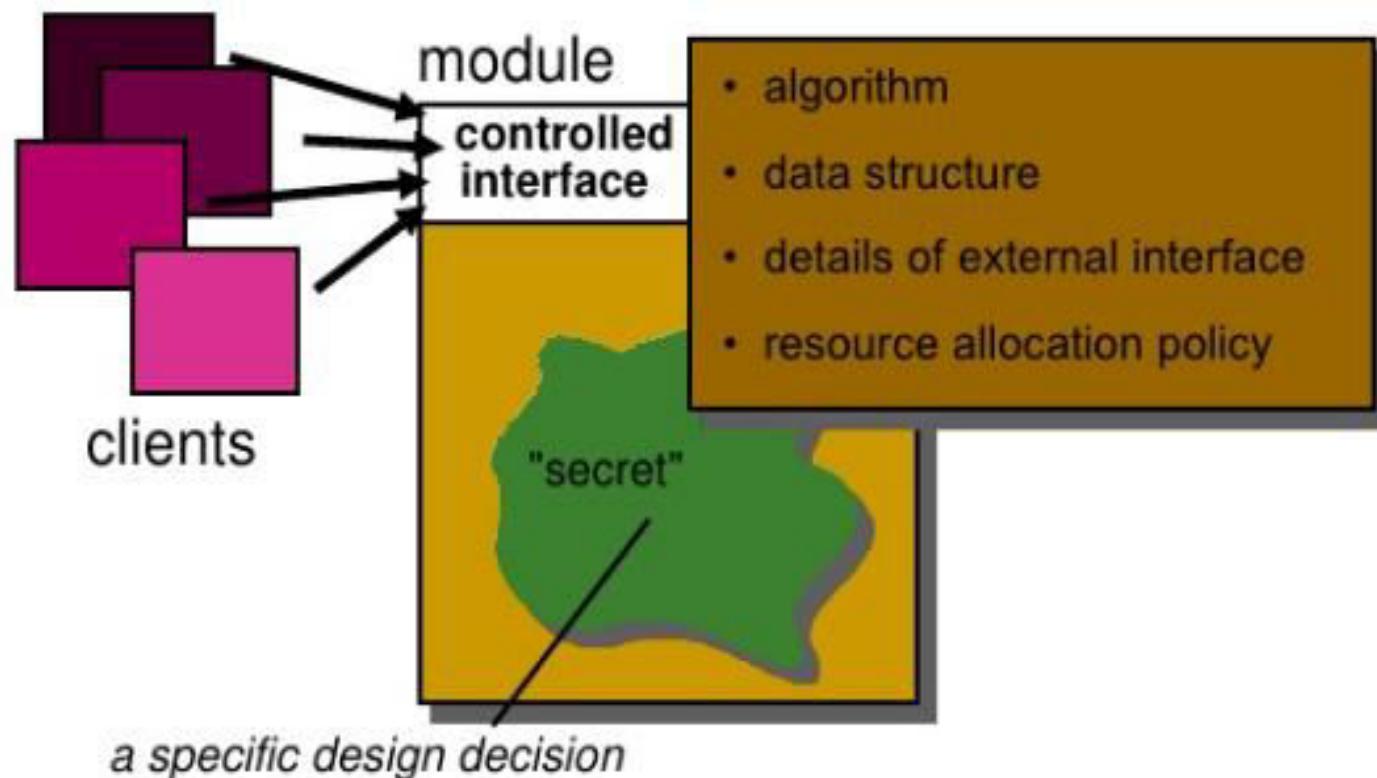
- Complexity of the overall problem can be reduced if the design method provides a systematic mechanism to decompose a problem into sub problems

Design Concepts

Information Hiding

- Information (data and procedure) contained within a module should be inaccessible to other modules that have no need for such information.
- Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module.

Information hiding



Design Concepts

Functional Independence

- Functional independence is achieved by developing modules with “single minded” function and an aversion to excessive interaction with other modules.
- Measured using 2 qualitative criteria:
 1. Cohesion : Measure of the relative strength of a module.
 2. Coupling : Measure of the relative interdependence among modules.

Cohesion

- Definition
 - The degree to which all elements of a component are directed towards a single task.
 - The degree to which all elements directed towards a task are contained in a single component.
 - The degree to which all responsibilities of a single class are related.
- All elements of component are directed toward and essential for performing the same task.

Cohesion

Different types of cohesion :

1. Functional Cohesion (Highest):

A functionally cohesive module contains elements that all contribute to the execution of one and only one problem-related task.

Examples of functionally cohesive modules are

Compute cosine of an angle

Calculate net employee salary

Cohesion

2. Sequential Cohesion :

A *sequentially cohesive* module is one whose elements are involved in activities such that output data from one activity serves as input data to the next.

Eg: Module read and validate customer record

- Read record
- Validate customer record

Here output of one activity is input to the second

Cohesion

3. Communicational cohesion

A *communicationally cohesive* module is one whose elements contribute to activities that use the same input or output data.

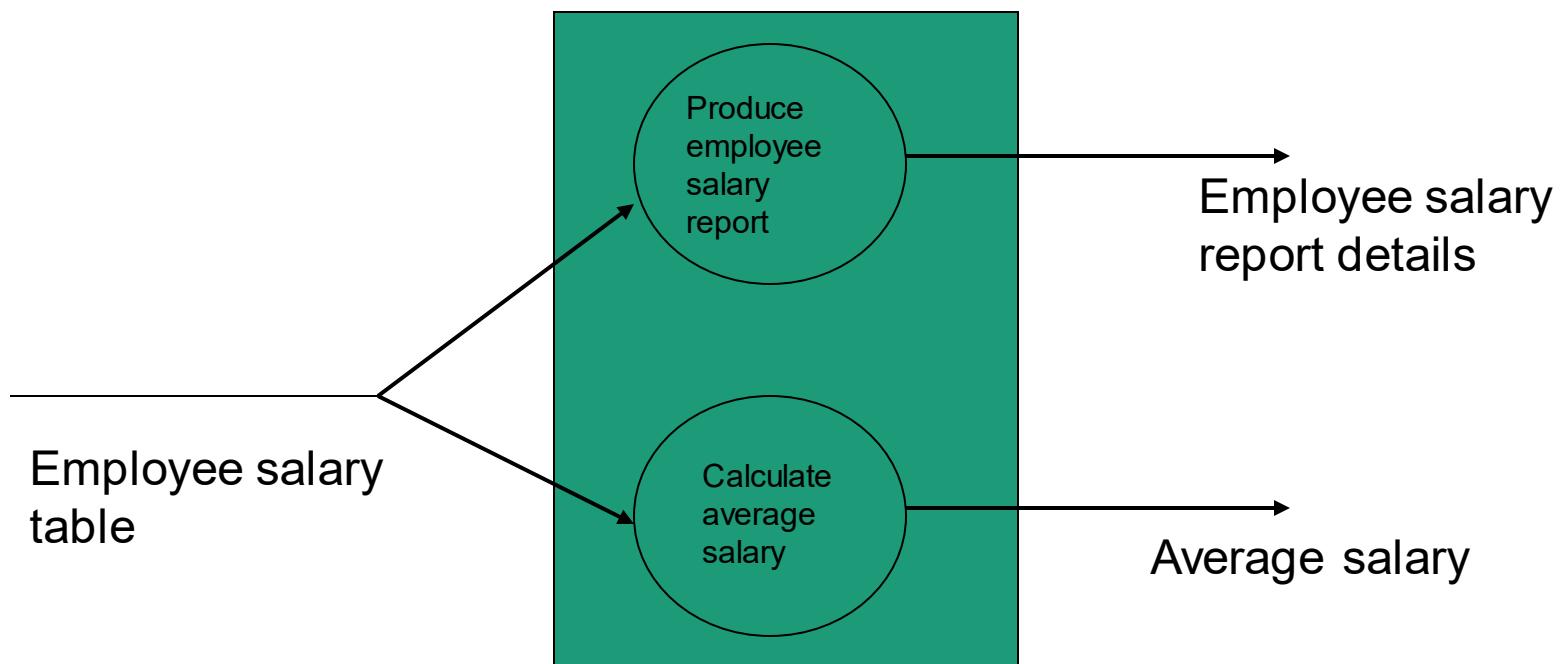
Suppose we wish to find out some facts about a book
For instance, we may wish to

- FIND TITLE OF BOOK
- FIND PRICE OF BOOK
- FIND PUBLISHER OF BOOK
- FIND AUTHOR OF BOOK

These four activities are related because they all work on the same input data, the book, which makes the “module” communicationaly cohesive.

Cohesion

Eg: module which produces employee salary report and calculates average salary



Cohesion

4. Procedural Cohesion

A procedurally cohesive module is one whose elements are involved in different and possibly unrelated activities in which control flows from each activity to the next.

Eg:

addRecord

changeRecord

deleteRecord

Cohesion

5. Temporal Cohesion

A temporally cohesive module is one whose elements are involved in activities that are related in time.

Eg:

An exception handler that

- Closes all open files
- Creates an error log
- Notifies user

Cohesion

6. Logical Cohesion

A logically cohesive module is one whose elements contribute to activities of the same general category in which the activity or activities to be executed are selected from outside the module.

A logically cohesive module contains a number of activities of the same general kind.

To use the module, we pick out just the piece(s) we need.

Eg:

Read from file

Read from database

Cohesion

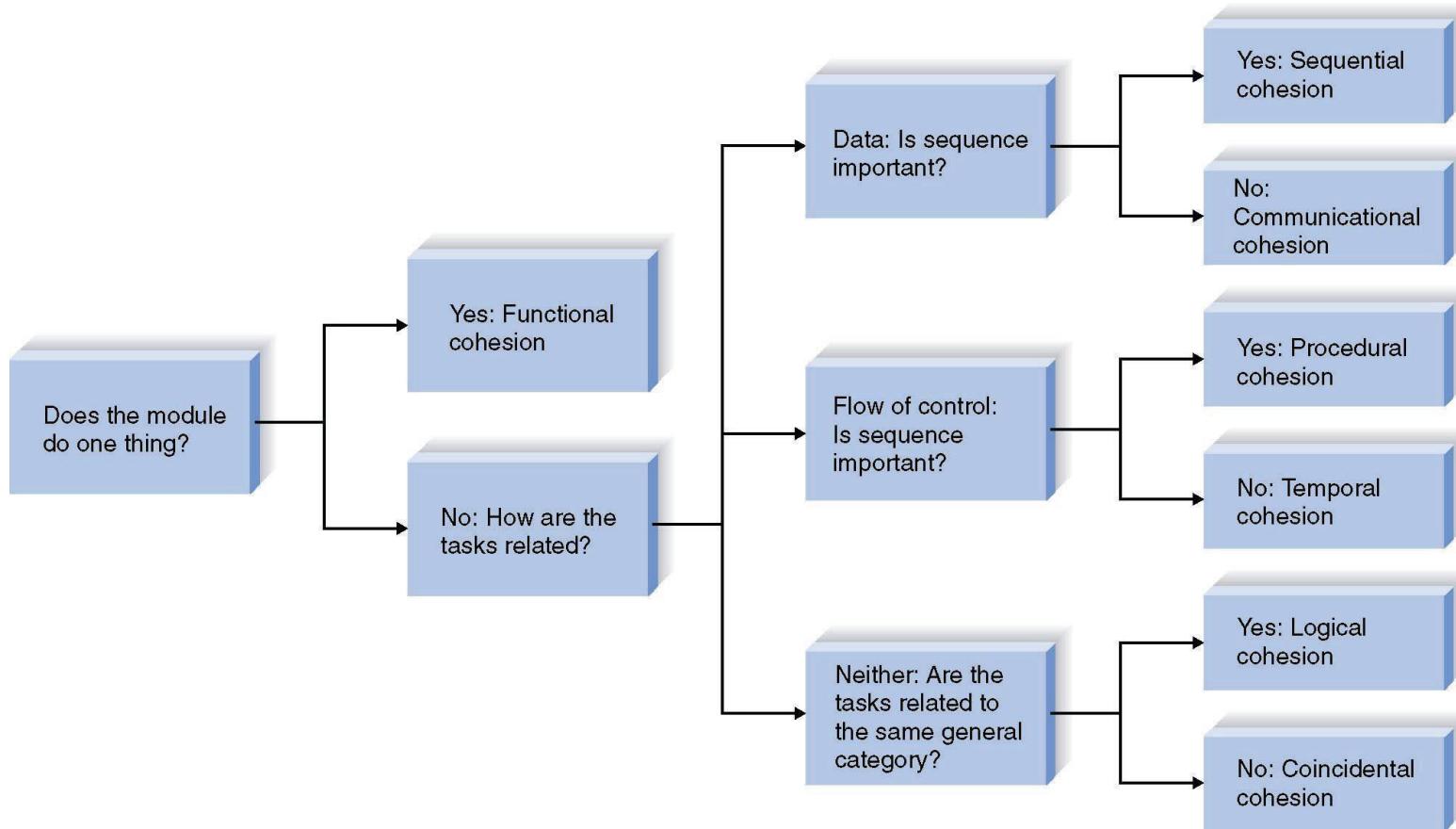
7. Coincidental Cohesion (Lowest)

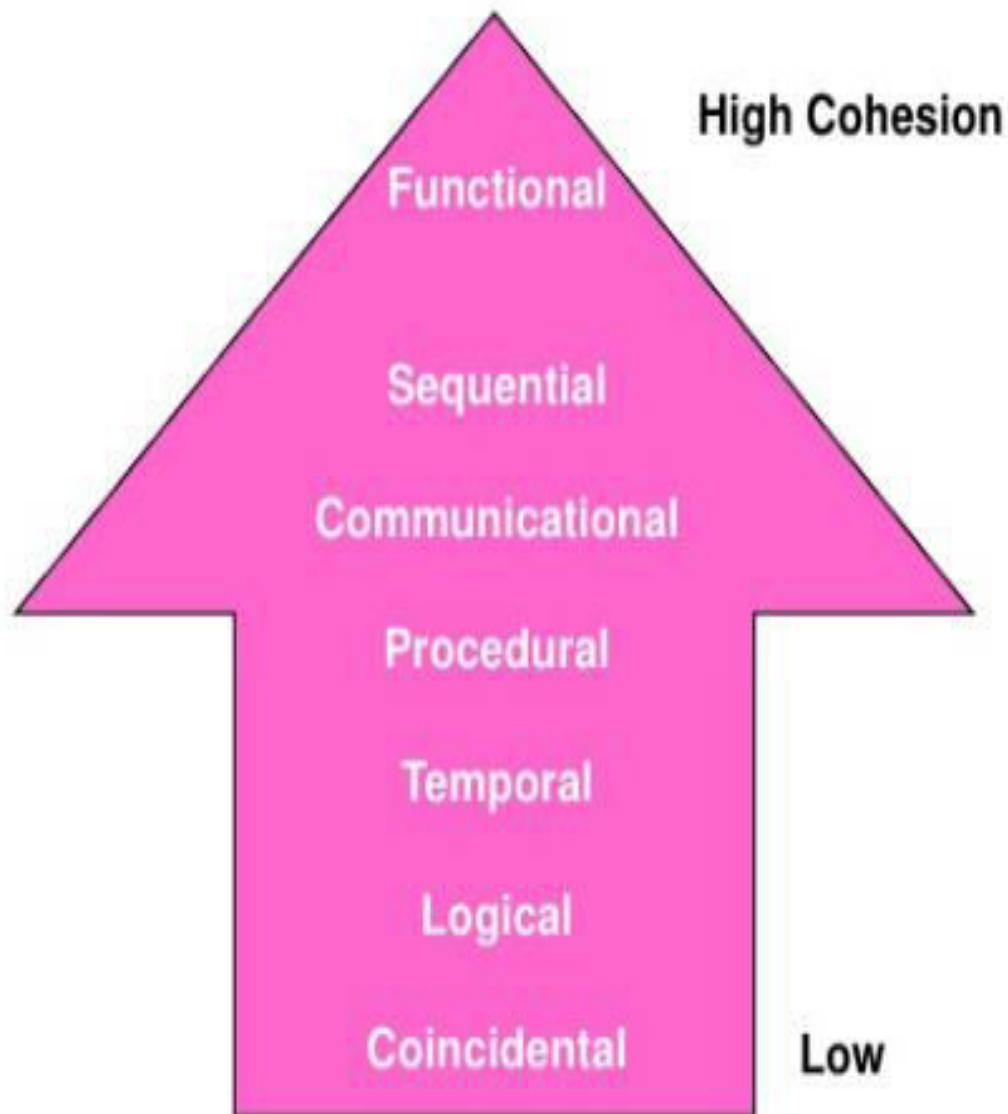
A coincidentally cohesive module is one whose elements contribute to activities with no meaningful relationship to one another.

Eg. When a large program is modularized by arbitrarily segmenting the program into several small modules.

Cohesion

A decision tree to show module cohesion





Coupling

- Measure of interconnection among modules in a software structure
- Strive for lowest coupling possible.

Coupling

Types of coupling

1. Data coupling (Most desirable)

Two modules are data coupled, if they communicate through parameters where each parameter is an elementary piece of data.

e.g. an integer, a float, a character, etc. This data item should be problem related and not be used for control purpose.

2. Stamp Coupling

Two modules are said to be stamp coupled if a data structure is passed as parameter but the called module operates on some but not all of the individual components of the data structure.

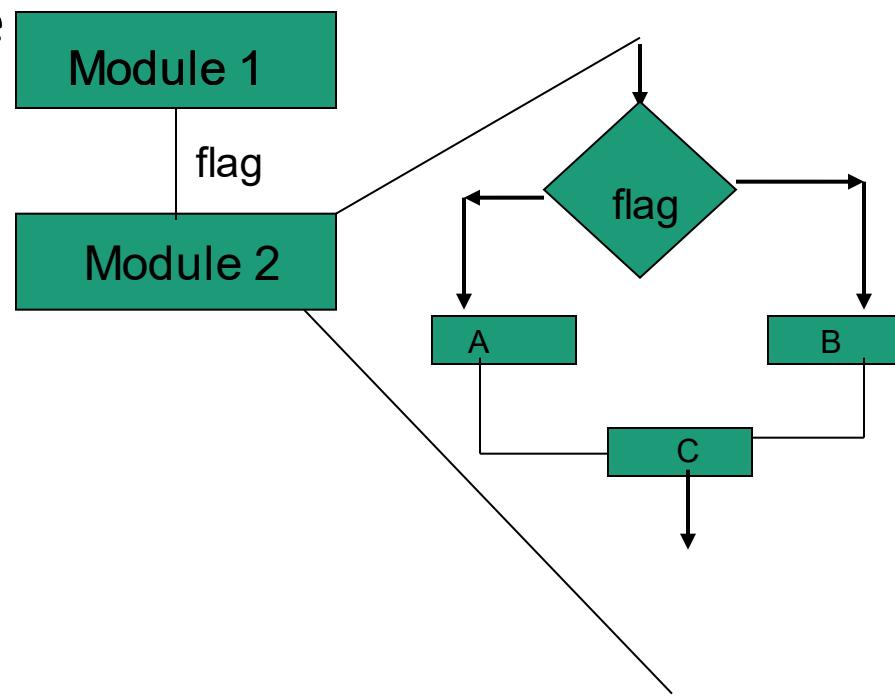
Coupling

3. Control Coupling

Two modules are said to be control coupled if one module passes a control element to the other module.

This control element affects /controls the internal logic of the called module

Eg: flags



Coupling

4. Common Coupling

Takes place when a number of modules access a data item in a global data area.

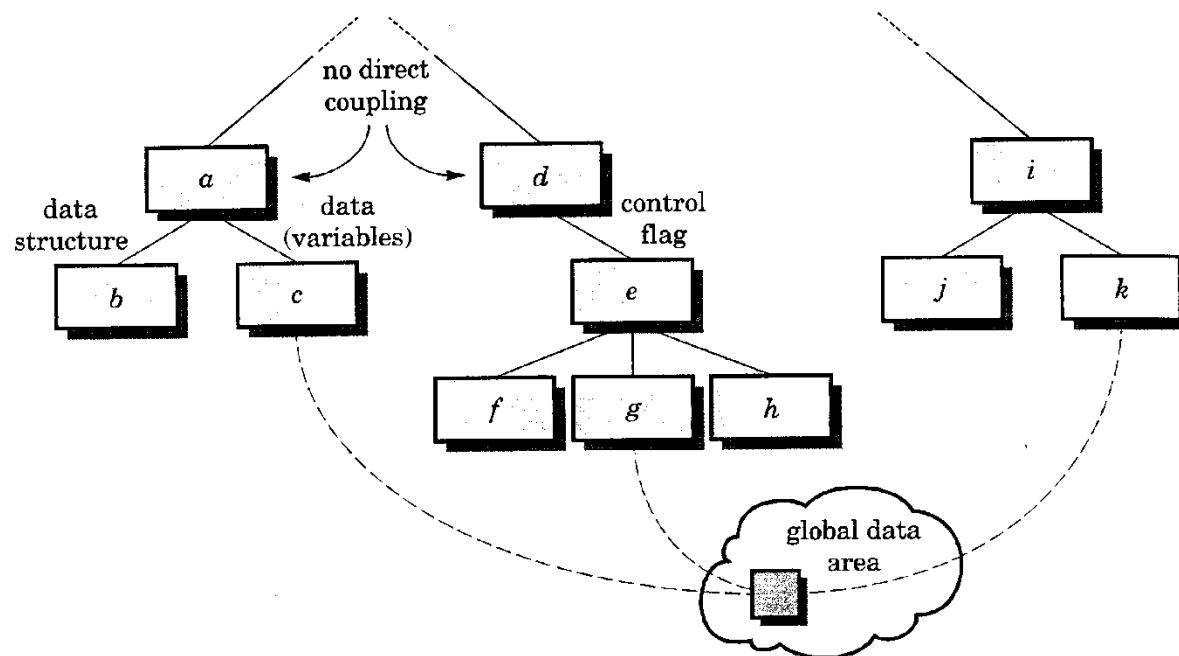


FIGURE 13.8. Types of coupling

Modules c, g and k exhibit common coupling

Coupling

5. Content coupling (Least desirable)

Two modules are said to be content coupled if one module branches into another module or modifies data within another.

Eg:

```
int func1(int a)
{ printf("func1");
  a+=2;
  goto F2A;
  return a;
}
```

```
void func2(void)
{ printf("func2");
  F2A : printf("At F2A")
}
```

- Eg: Part of a program that handles lookup for a customer.

When customer not found, component adds customer by directly modifying the contents of the data structure containing customer data.

Design Concepts

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.

Stepwise refinement



Design Concepts

Refactoring

- Reorganization technique that simplifies the design (or code) of a component without changing its function or behavior
- “Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.”

- When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design.
- For example, a first design iteration might yield a component that exhibits low cohesion (i.e., it performs three functions that have only limited relationship to one another). After careful consideration, you may decide that the component should be refactored into three separate components, each exhibiting high cohesion

Design Concepts

Design Classes

- As the design model evolves, you will define a set of *design classes* that refine the analysis classes by providing design detail that will enable the classes to be implemented and implement a software infrastructure that supports the business solution.

- *User interface classes*
- define all abstractions that are necessary for human-computer interaction (HCI). In many cases, HCI occurs within the context of a *metaphor* (e.g., a checkbook, an order form) and the design classes for the interface may be visual representations of the elements of the metaphor.

- *Business domain classes*
- are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.
- *Process classes*
- implement lower-level business abstractions required to fully manage the business domain classes.

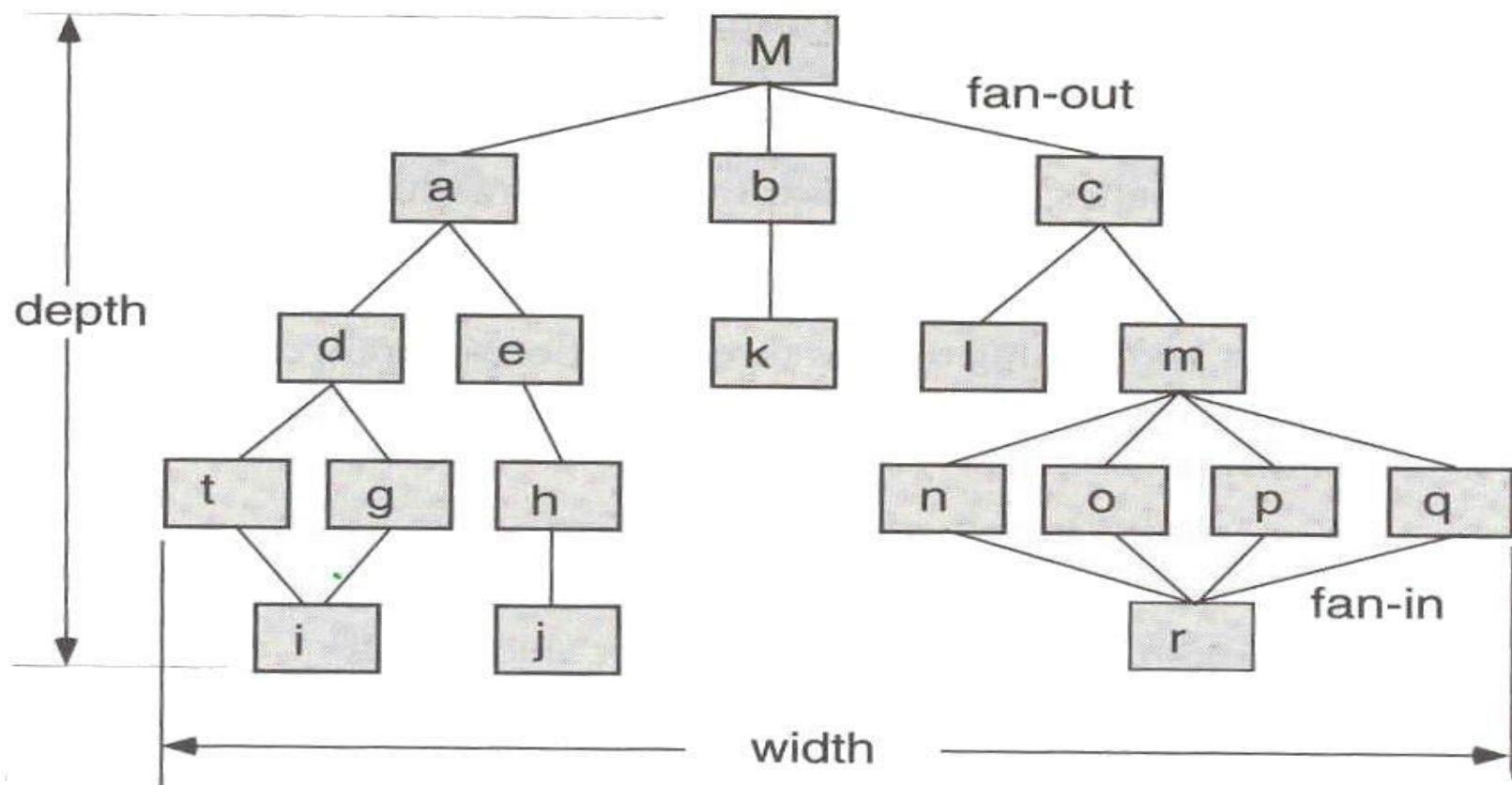
- *Persistent classes*
- represent data stores (e.g., a database) that will persist beyond the execution of the software.
- *System classes*
- implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world

Four characteristics of well-formed design class:

- Complete and sufficient
- Primitiveness (easier to understand, reusable)
- High cohesion
- Low coupling

Design Concepts

Control Hierarchy



Design Concepts(Control Hierarchy)

- Also called program structure
- Represent the organization of program components.
- Does not represent procedural aspects of software such as decisions, repetitions etc.
 - **Depth** –No. of levels of control (distance between the top and bottom modules in program control structure)
- **Width**- Span of control.
- **Fan-out** -no. of modules that are directly controlled by another module
- **Fan-in** - how many modules directly control a given module

Design Concepts(Control Hierarchy)

Super ordinate -module that control another module

Subordinate - module controlled by another

Software Testing Techniques

(Source: Pressman, R. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2005)

Overview

■ Test Case Design

□ White Box

- Control Flow Graph
- Cyclomatic Complexity
- Basic Path Testing

□ Black Box

- Equivalence Classes
- Boundary Value Analysis

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

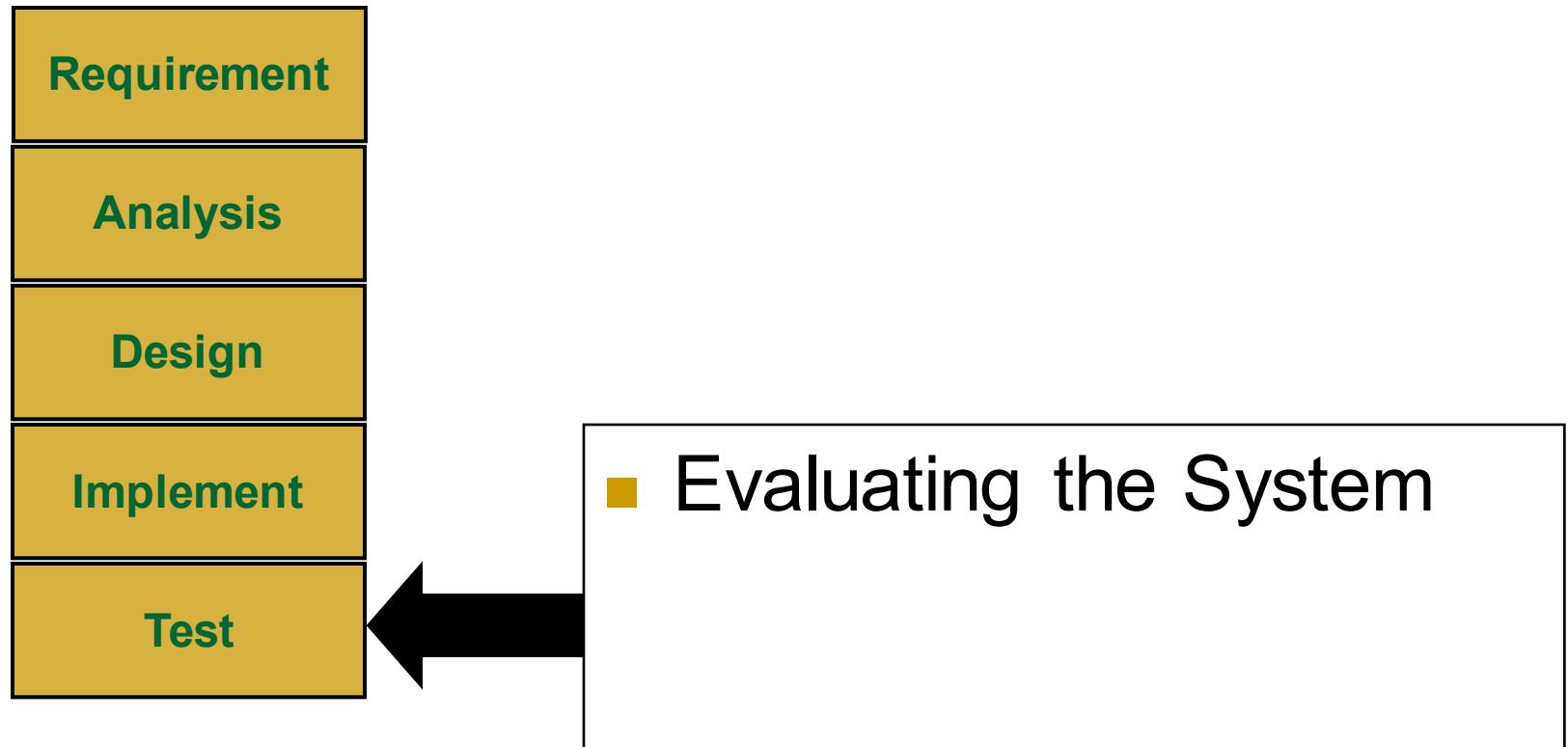
White-box Testing

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”

Where are we now?



White Box Testing: Introduction

- Test Engineers have access to the source code.
- Typical at the Unit Test level as the programmers have knowledge of the internal logic of code.
- Tests are based on coverage of:
 - Code statements;
 - Branches;
 - Paths;
 - Conditions.
- Most of the testing techniques are based on *Control Flow Graph* (denoted as CFG) of a code fragment.

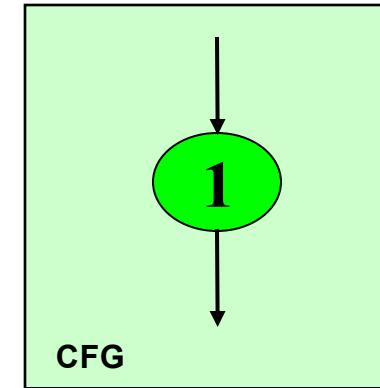
Control Flow Graph: Introduction

- An abstract representation of a structured program/function/method.
- Consists of two major components:
 - *Node*:
 - Represents a stretch of sequential code statements with no branches.
 - *Directed Edge* (also called *arc*):
 - Represents a branch, alternative path in execution.
- Path:
 - A collection of *Nodes* linked with *Directed Edges*.

Simple Examples

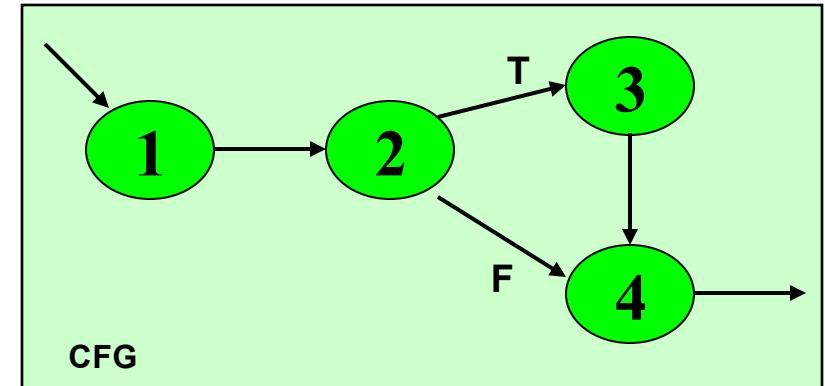
```
Statement1;  
Statement2;  
Statement3;  
Statement4;
```

Can be represented as **one** node as there is no branch.



```
Statement1;  
Statement2;  
  
if x < 10 then  
    Statement3;  
  
Statement4;
```

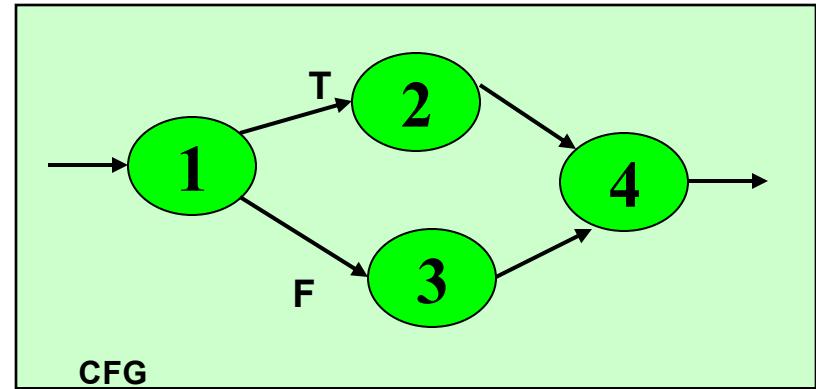
1
2
3
4



More Examples

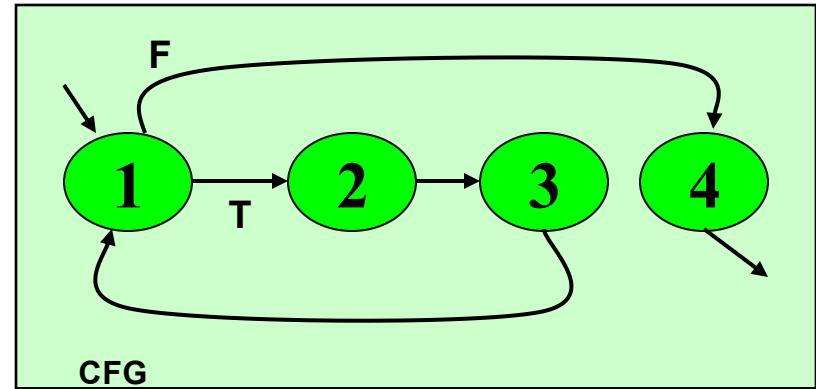
```
if x > 0 then  
    Statement1;  
else  
    Statement2;
```

1
2
3



```
while x < 10 {  
    Statement1;  
    X++; }
```

1
2
3



Question: Why is there a node 4 in both CFGs?

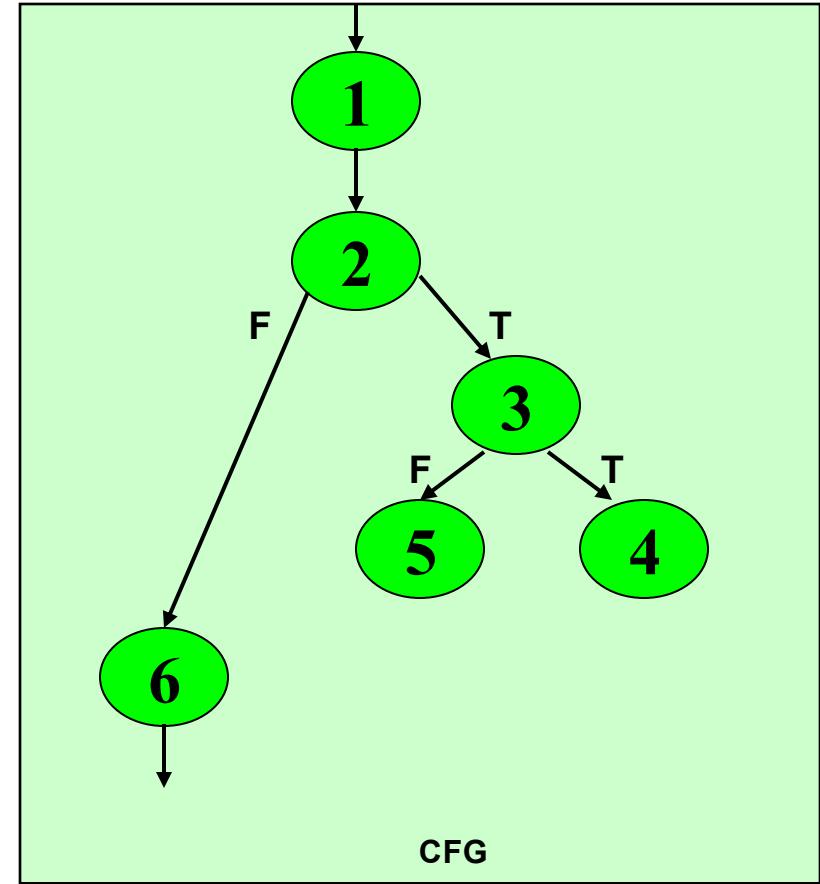
Notation Guide for CFG

- A CFG should have:
 - 1 entry arc (known as a directed edge, too).
 - 1 exit arc.
- All nodes should have:
 - At least 1 entry arc.
 - At least 1 exit arc.
- A **Logical Node** that does not represent any actual statements can be added as a joining point for several incoming edges.
 - Represents a logical closure.
 - Example:
 - Node 4 in the if-then-else example from previous slide.

Example: Minimum Element

```
min = A[0];  
I = 1;  
  
while (I < N) {  
    if (A[I] < min)  
        min = A[I];  
    I = I + 1;  
}  
print min
```

1
2
3
4
5
6

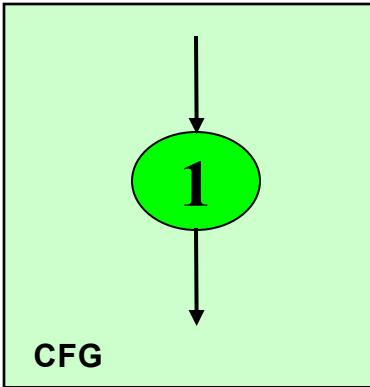


Note: The CFG is **INCOMPLETE**. Try to complete it

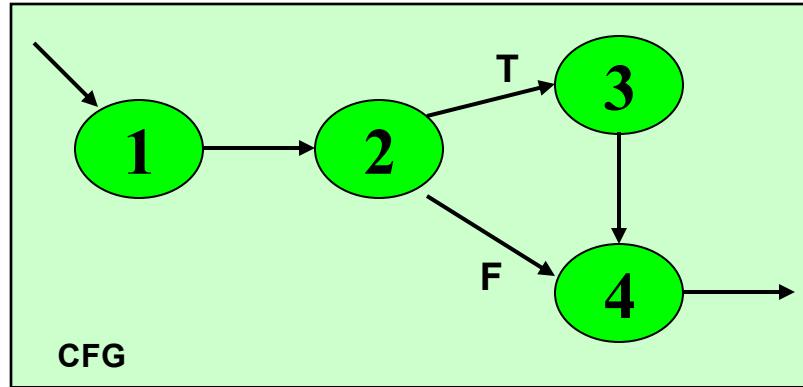
Number of Paths through CFG

- Given a program, how do we exercise all statements and branches at least once?
- Translating the program into a CFG, an equivalent question is:
 - Given a CFG, how do we cover all arcs and nodes at least once?
- Since a path is a trail of nodes linked by arcs, this is similar to ask:
 - Given a CFG, what is the set of paths that can cover all arcs and nodes?

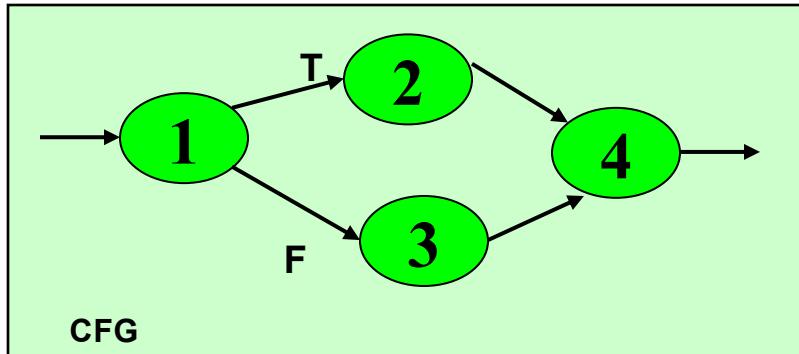
Example



- Only **one** path is needed:
 - [1]



- **Two** paths are needed:
 - [1 - 2 - 4]
 - [1 - 2 - 3 - 4]



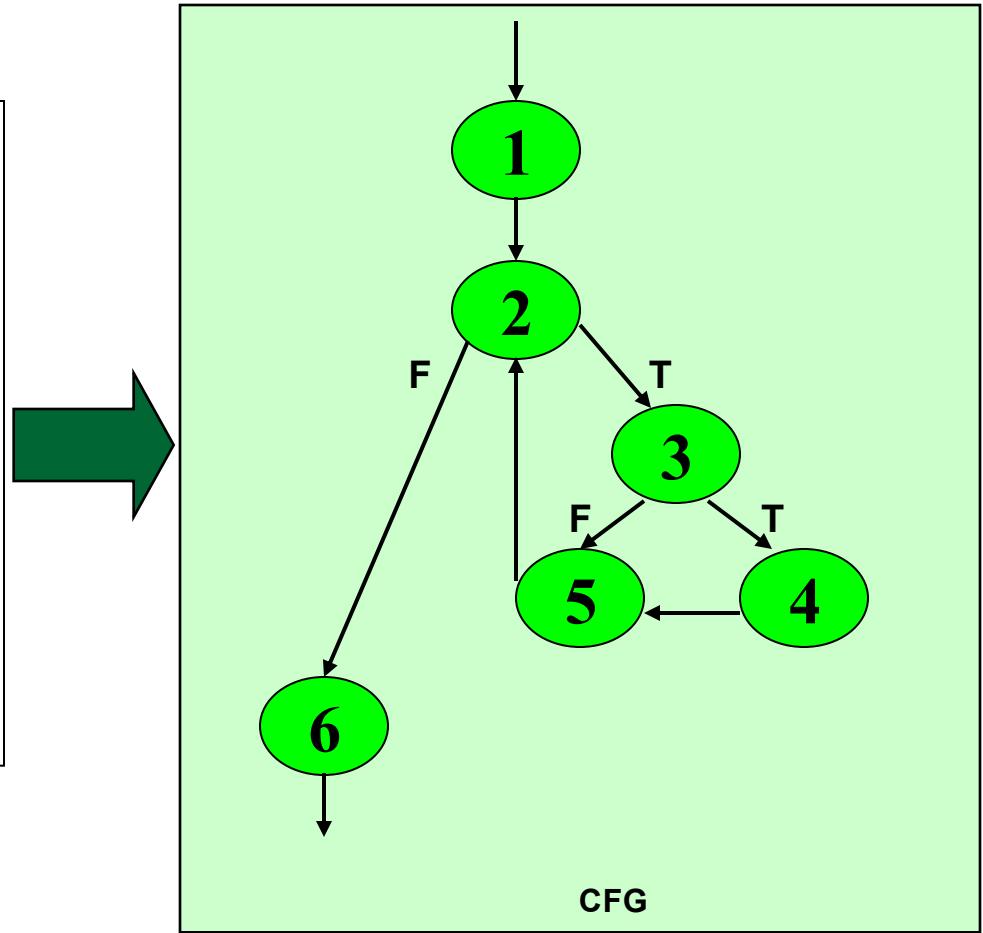
- **Two** paths are needed:
 - [1 - 2 - 4]
 - [1 - 3 - 4]

White Box Testing: Path Based

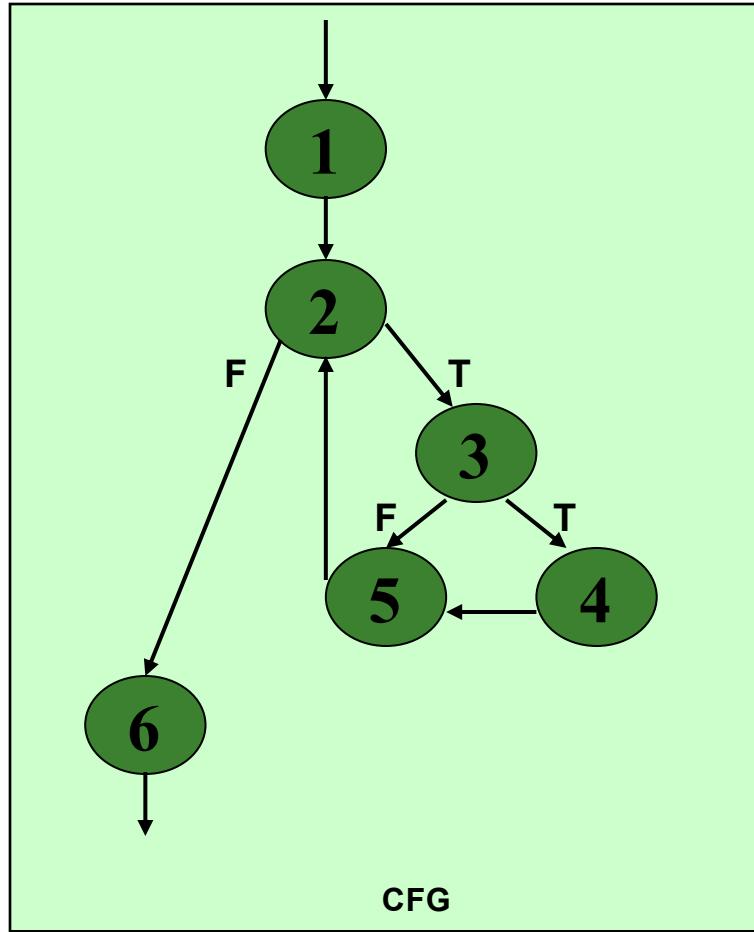
- A generalized technique to find out the number of paths needed (known as *cyclomatic complexity*) to cover all arcs and nodes in CFG.
- Steps:
 1. Draw the CFG for the code fragment.
 2. Compute the *cyclomatic complexity number C*, for the CFG.
 3. Find at most **C** paths that cover the nodes and arcs in a CFG, also known as **Basic Paths Set**;
 4. Design test cases to force execution along paths in the **Basic Paths Set**.

Path Based Testing: Step 1

```
min = A[0];  
I = 1;  
  
while (I < N) {  
    if (A[I] < min)  
        min = A[I];  
    I = I + 1;  
}  
print min
```

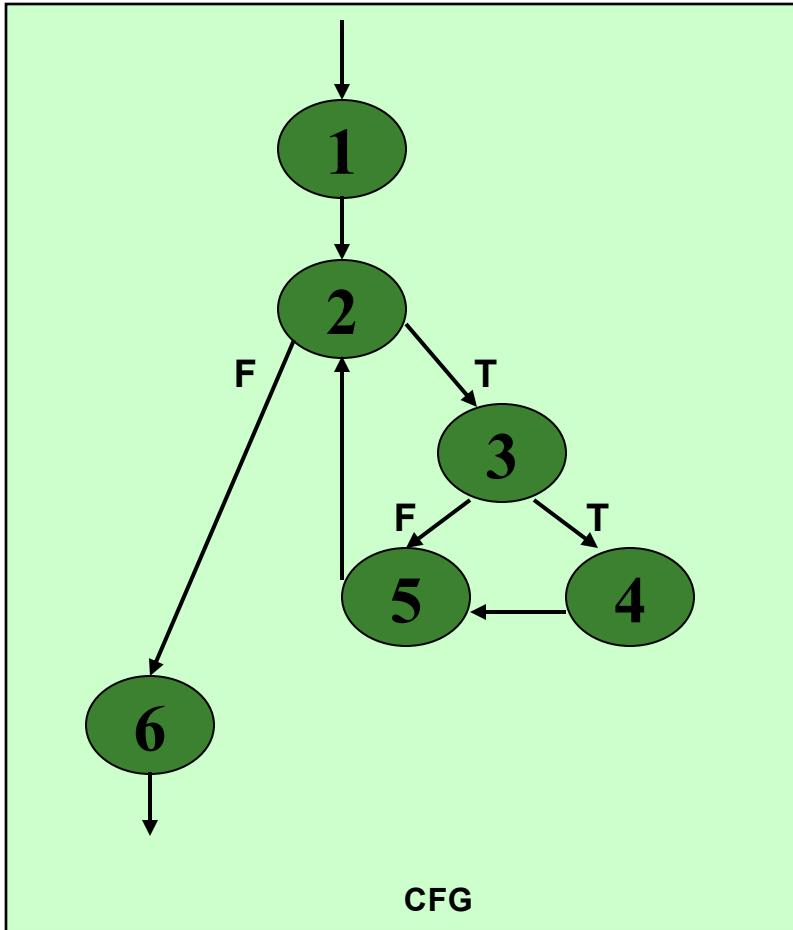


Path Base Testing: Step 2



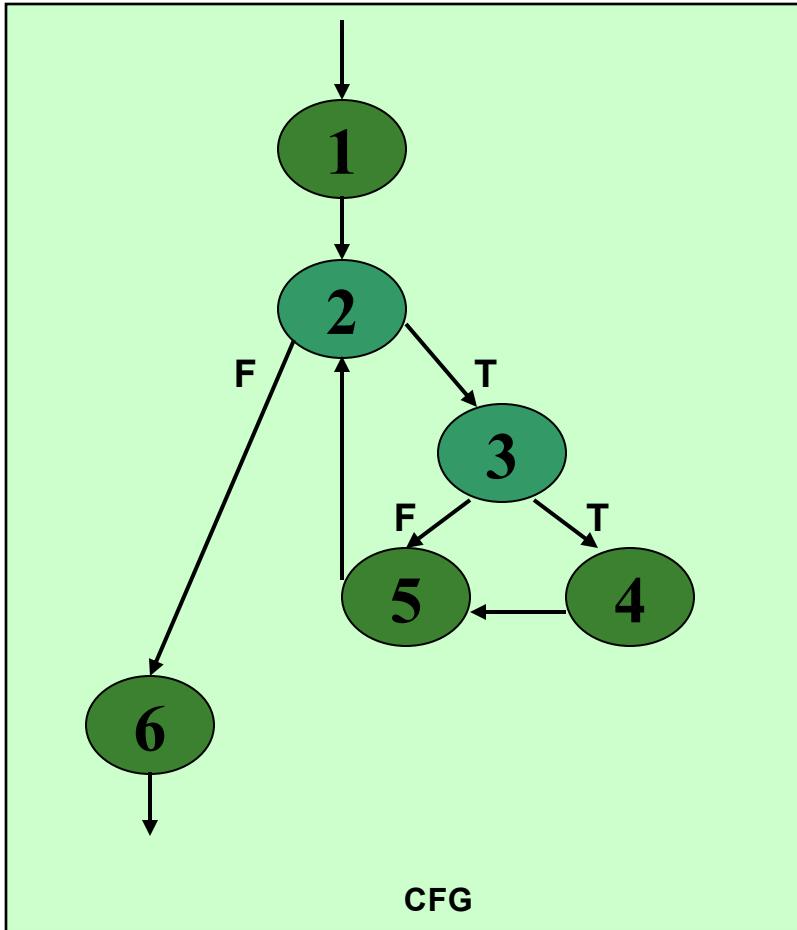
- Cyclomatic complexity =
 - The number of ‘regions’ in the graph; OR
 - The number of predicates + 1.
 - Number of edges-Number of nodes+2 (E-N+2)

Path Base Testing: Step 2



- **Region:** Enclosed area in the CFG.
 - Do not forget the outermost region.
- In this example:
 - 3 Regions (see the circles with different colors).
 - Cyclomatic Complexity = 3
- Alternative way in next slide.

Path Base Testing: Step 2

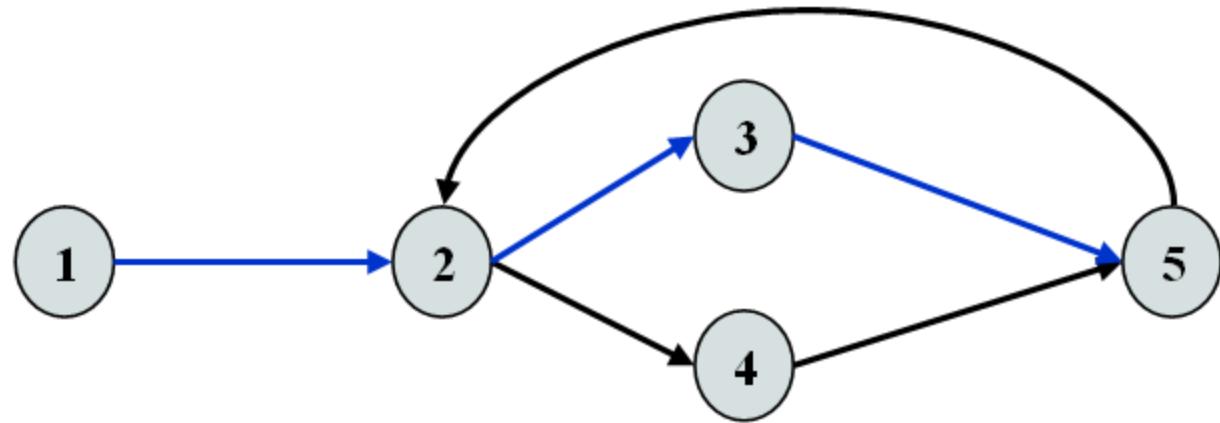


- **Predicates:**
 - Nodes with multiple exit arcs.
 - Corresponds to branch/conditional statement in program.
- In this example:
 - Predicates = 2
 - (Node 2 and 3)
 - Cyclomatic Complexity
 $= 2 + 1$
 $= 3$

Path Base Testing: Step 3

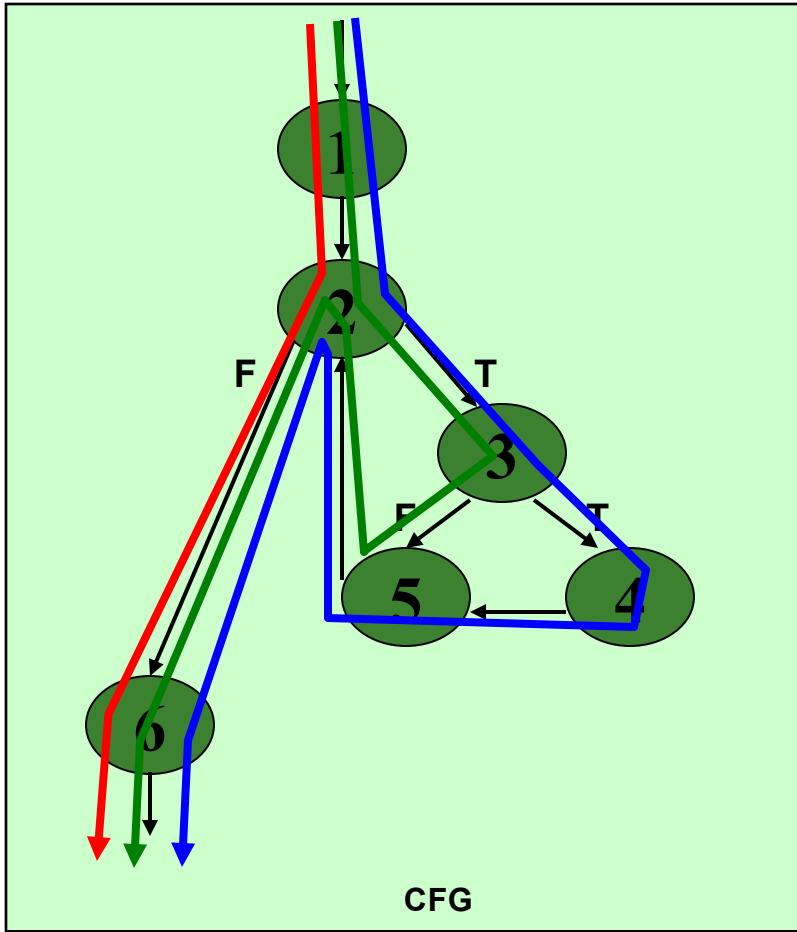
- Independent path:
 - An **executable** or **realizable path** through the graph from the start node to the end node that has not been traversed before.
 - **Must** move along **at least one arc** that has not been yet traversed (an unvisited arc).
 - The objective is to cover all statements in a program by independent paths.
- The number of independent paths to discover \leq cyclomatic complexity number.
- Decide the Basis Path Set:
 - It is the maximal set of *independent paths* in the flow graph.
 - **NOT** a unique set.

Example



- 1-2-3-5 can be the first independent path; 1-2-4-5 is another; 1-2-3-5-2-4-5 is one more.
- There are only these 3 independent paths. The basis path set is then having 3 paths.
- Alternatively, if we had identified 1-2-3-5-2-4-5 as the first independent path, there would be no more independent paths.
- The number of independent paths therefore can vary according to the order we identify them.

Path Base Testing: Step 3



- Cyclomatic complexity = 3.
- Need at most 3 independent paths to cover the CFG.
- In this example:
 - [1 – 2 – 6]
 - [1 – 2 – 3 – 5 – 2 – 6]
 - [1 – 2 – 3 – 4 – 5 – 2 – 6]

Path Base Testing: Step 4

- Prepare a test case for each independent path.
- In this example:
 - Path: [1 – 2 – 6]
 - Test Case: $A = \{ 5, \dots \}$, $N = 1$
 - Expected Output: 5
 - Path: [1 – 2 – 3 – 5 – 2 – 6]
 - Test Case: $A = \{ 5, 9, \dots \}$, $N = 2$
 - Expected Output: 5
 - Path: [1 – 2 – 3 – 4 – 5 – 2 – 6]
 - Test Case: $A = \{ 8, 6, \dots \}$, $N = 2$
 - Expected Output: 6
- These tests will result a complete decision and statement coverage of the code.

Try to verify that the test cases actually force execution along a desired path.

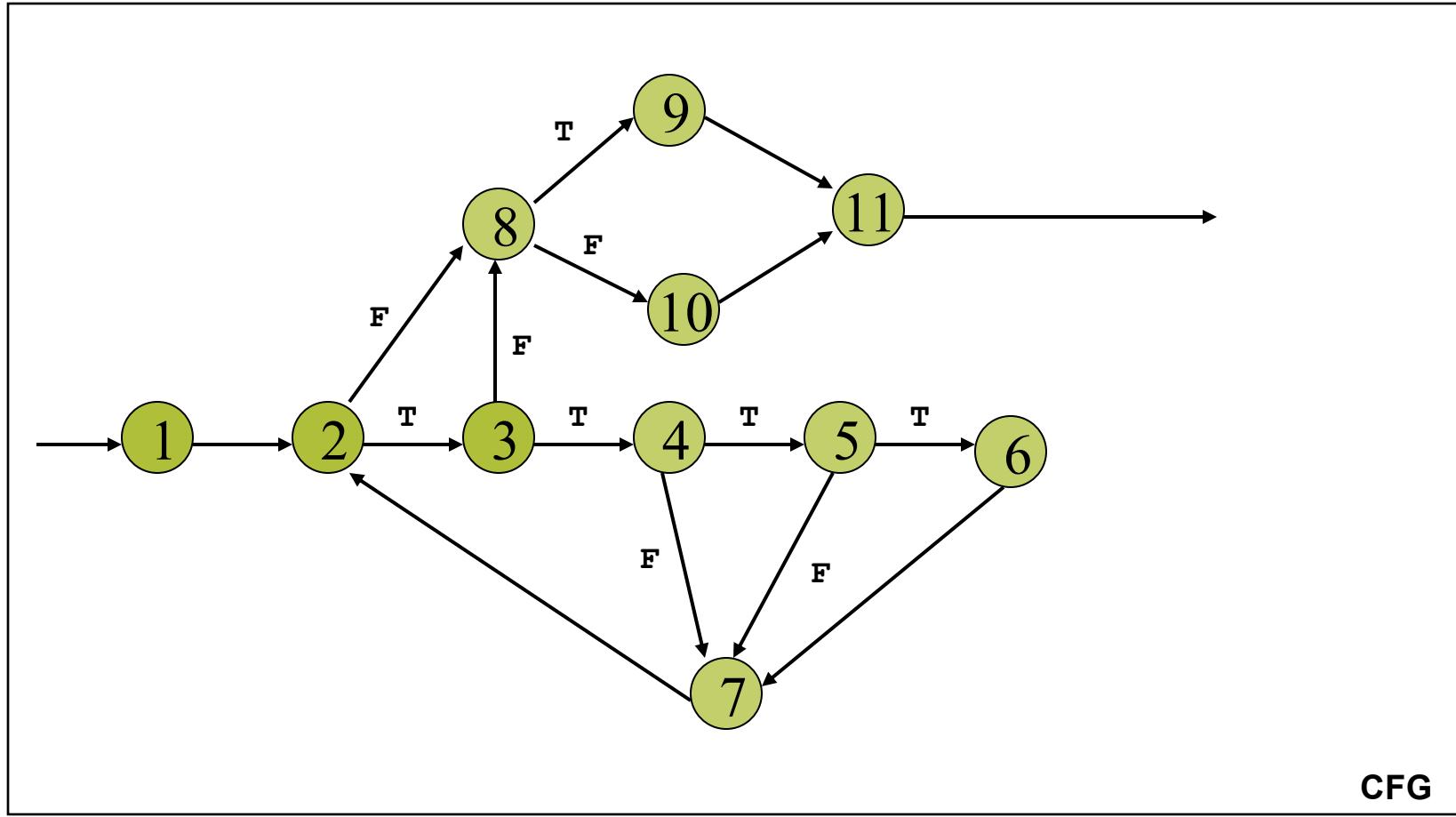
Another Example

```
int average (int[ ] value, int min, int max, int N) {  
    int i, totalValid, sum, mean;  
    i = totalValid = sum = 0;  
    while ( i < N && value[i] != -999 ) {  
        if (value[i] >= min && value[i] <= max) {  
            totalValid += 1; sum += value[i];  
        }  
        i += 1;  
    }  
    if (totalValid > 0)  
        mean = sum / totalValid;  
    else  
        mean = -999;  
    return mean;  
}
```

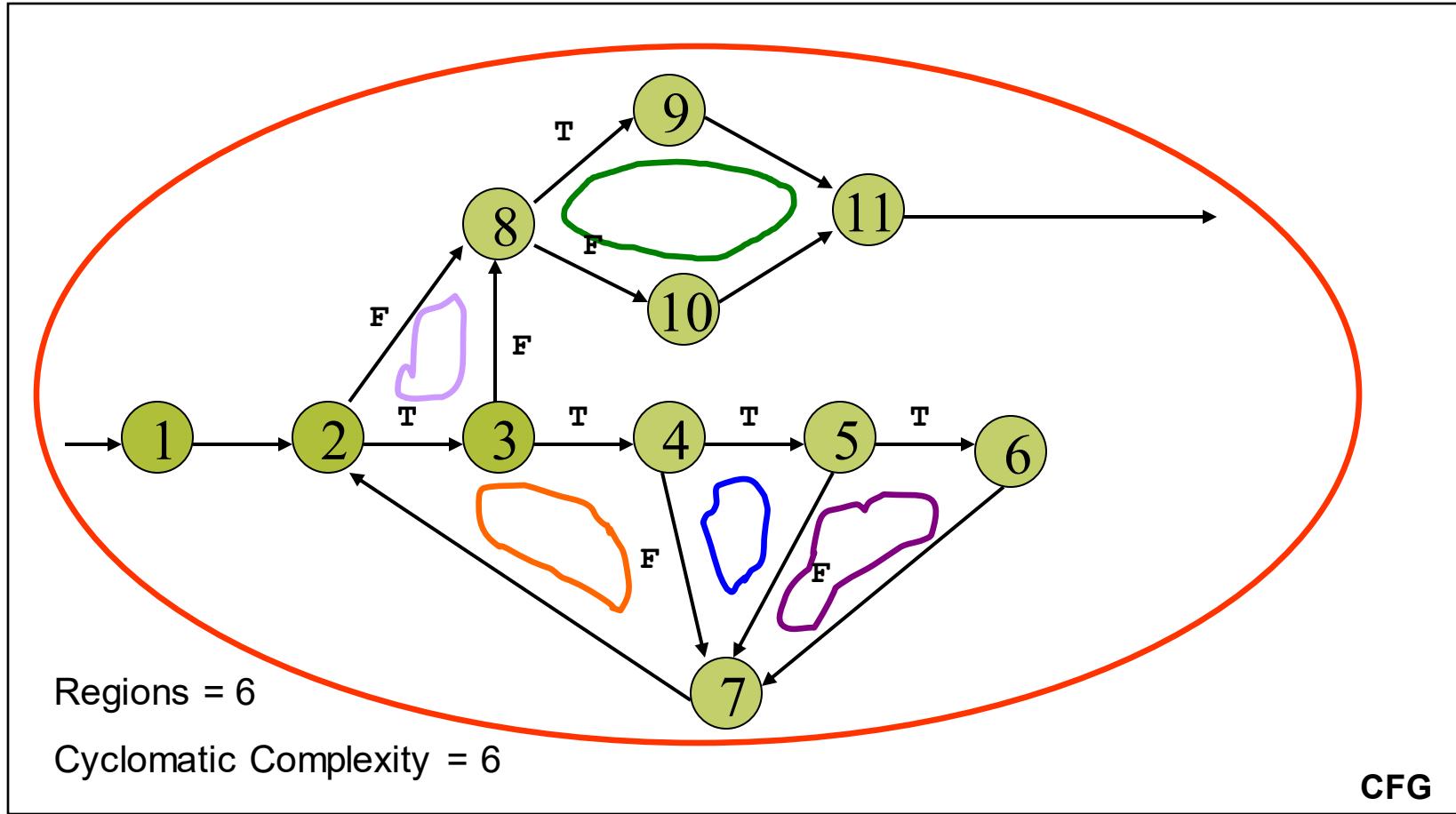
Step 1: Draw CFG

```
int average (int[ ] value, int min, int max, int N) {  
    int i, totalValid, sum, mean;  
    i = totalValid = sum = 0; } 1  
    while ( i < N && value[i] != -999 ) {  
        if (value[i] >= min && value[i] <= max) {  
            totalValid += 1; sum += value[i]; } 6  
        }  
        i += 1; 7  
    }  
    if (totalValid > 0) 8  
        mean = sum / totalValid; 9  
    else  
        mean = -999; 10  
    return mean; 11  
}
```

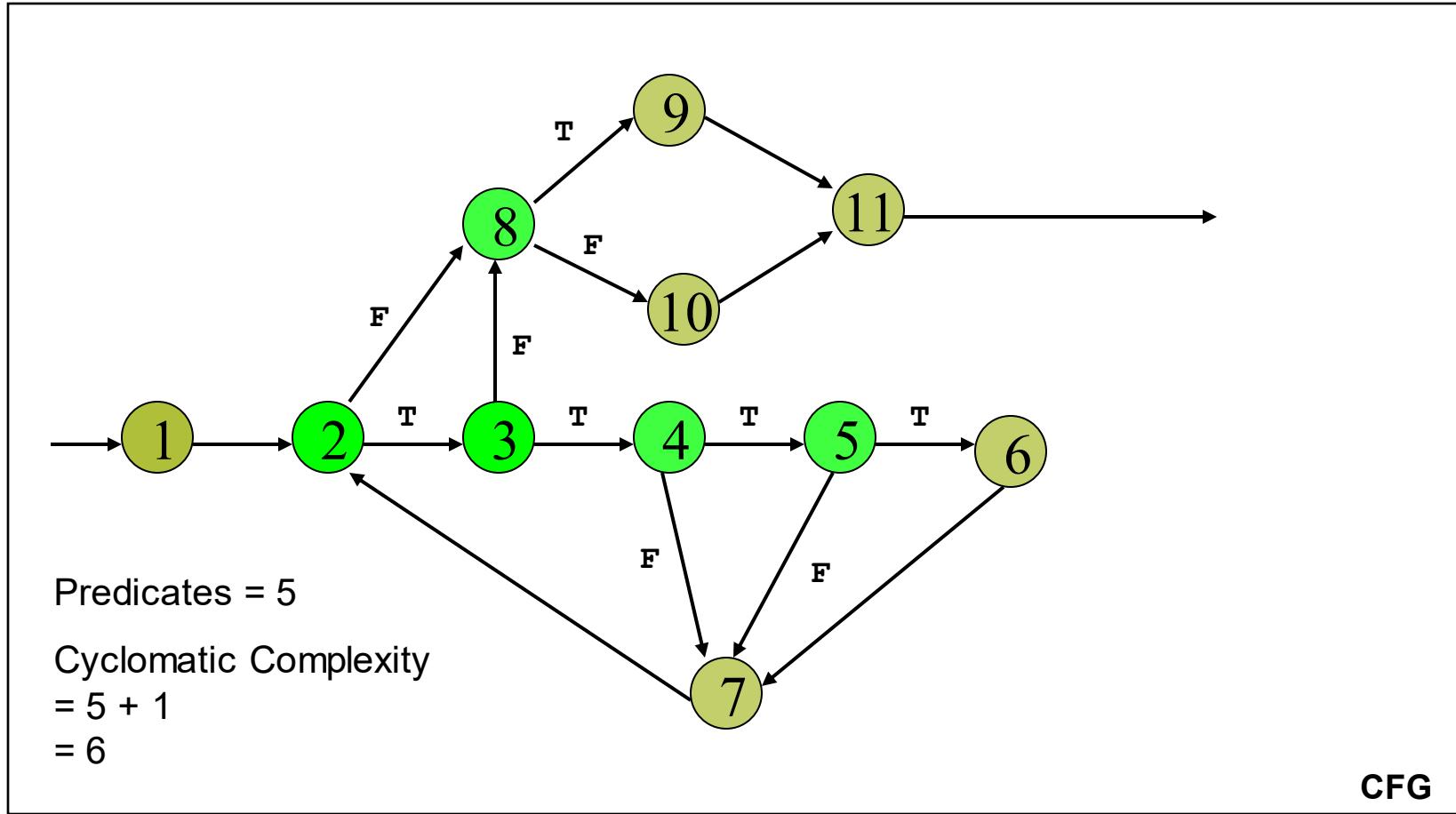
Step 1: Draw CFG



Step 2: Find Cyclomatic Complexity



Step 2: Find Cyclomatic Complexity



Step 3: Find Basic Path Set

- Find at most 6 independent paths.
- Usually, simpler path == easier to find a test case.
- However, some of the simpler paths are not possible (not realizable):
 - Example: [1 – 2 – 8 – 9 – 11].
 - Not Realizable (i.e., impossible in execution).
 - Verify this by tracing the code.
- Basic Path Set:
 - [1 – 2 – 8 – 10 – 11].
 - [1 – 2 – 3 – 8 – 10 – 11].
 - [1 – 2 – 3 – 4 – 7 – 2 – 8 – 10 – 11].
 - [1 – 2 – 3 – 4 – 5 – 7 – 2 – 8 – 10 – 11].
 - [1 – (2 – 3 – 4 – 5 – 6 – 7) – 2 – 8 – 9 – 11].
- In the last case, (...) represents possible repetition.

Step 4: Derive Test Cases

- Path:
 - [1 – 2 – 8 – 10 – 11]
- Test Case:
 - value = { ... } **irrelevant.**
 - N = 0
 - min, max **irrelevant.**
- Expected Output:
 - average = -999

```
... i = 0; 1
while (i < N &&
       value[i] != -999) {
    .....
}
if (totalValid > 0) 8
    .....
else
    mean = -999; 10
return mean; 11
```

Step 4: Derive Test Cases

Path:

- [1 – 2 – 3 – 8 – 10 – 11]

Test Case:

- value = { -999 }
- N = 1
- min, max irrelevant

Expected Output:

- average = -999

```
... i = 0; 1
while (i < N && 2
      value[i] != -999) 3
...
}
if (totalValid > 0) 8
...
else
    mean = -999; 10
return mean; 11
```

Step 4: Derive Test Cases

- Path:
 - [1 – 2 – 3 – 4 – 7 – 2 – 8 – 10 – 11]
 - Test Case:
 - A single value in the `value[]` array which is smaller than *min*.
 - `value = { 25 }, N = 1, min = 30, max irrelevant.`
 - Expected Output:
 - `average = -999`
-
- Path:
 - [1 – 2 – 3 – 4 – 5 – 7 – 2 – 8 – 10 – 11]
 - Test Case:
 - A single value in the `value[]` array which is larger than *max*.
 - `value = { 99 }, N = 1, max = 90, min irrelevant.`
 - Expected Output:
 - `average = -999`

Step 4: Derive Test Cases

- Path:
 - [1 – 2 – 3 – 4 – 5 – 6 – 7 – 2 – 8 – 9 – 11]
- Test Case:
 - A single valid value in the `value[]` array.
 - `value = { 25 }, N = 1, min = 0, max = 100`
- Expected Output:
 - `average = 25`

OR _____

- Path:
 - [1 – 2 – 3 – 4 – 5 – 6 – 7 – 2 – 3 – 4 – 5 – 6 – 7 – 2 – 8 – 9 – 11]
- Test Case:
 - Multiple valid values in the `value[]` array.
 - `value = { 25, 75 }, N = 2, min = 0, max = 100`
- Expected Output:
 - `average = 50`

Summary: Path Base White Box Testing

- A simple test that:
 - Cover all statements.
 - Exercise all decisions (conditions).
- The cyclomatic complexity is an **upperbound** of the independent paths needed to cover the CFG.
 - If more paths are needed, then either cyclomatic complexity is wrong, or the paths chosen are incorrect.
- Although picking a complicated path that covers more than one unvisited edge is possible all times, it is not encouraged:
 - May be hard to design the test case.

Graph Matrices

- To develop a software tool assists in basis path testing, a data structure, called a graph matrix.
- Square Matrix: tabular representation of flow graph
- Each node represents by number and each edge represents by letter
- Add link weight to each entry which provides additional information about control flow
 - Link weight is 1 (a connection exists)
 - Link weight is 0 (a connection does not exists)

Graph Matrices

- Link weights can be assigned other
 - Probability of edge will be executed
 - Processing time during traversal a link
 - Memory required during traversal a link
 - Resources required during traversal a link

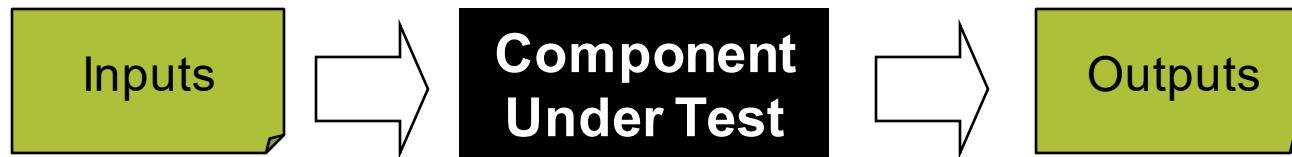
Control Structure Testing : to improve quality of white box testing

- Condition Testing
 - Simple condition
 - Compound condition
- Data Flow Testing
 - Selects test paths of a program according to variables
- Loop Testing
 - Simple Loops, Nested Loops, Concatenated Loops and Unstructured Loops

Black Box Testing

Black Box Testing: Introduction

- Test Engineers have no access to the source code or documentation of internal working.
- The “Black Box” can be:
 - A single unit.
 - A subsystem.
 - The whole system.
- Tests are based on:
 - Specification of the “Black Box”.
 - Providing **inputs** to the “Black Box” and inspect the **outputs**.



Test Case Design

- Two techniques will be covered for the black box testing in this course:
 - Equivalence Partition;
 - Boundary Value Analysis.

Equivalence Partition: Introduction

- To ensure the correct behavior of a “black box”, both valid and invalid cases need to be tested.
- Example:
 - Given the method below:

```
boolean isValidMonth(int m)
```

Functionality: check `m` is [1..12]

Output:

- `true` if `m` is 1 to 12
- `false` otherwise

- Is there a better way to test other than testing **all** integer values $[-2^{31}, \dots, 2^{31}-1]$?

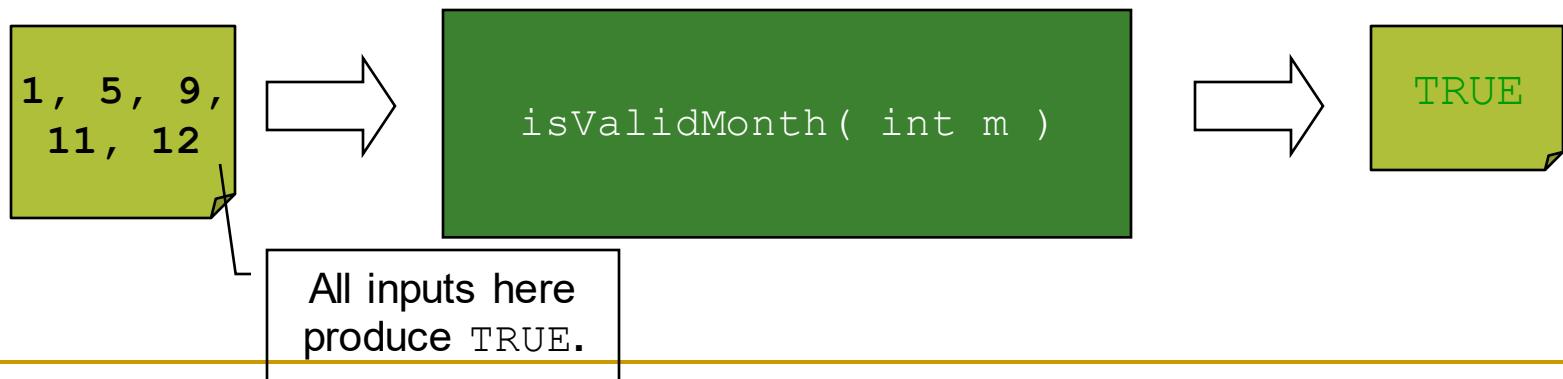
Equivalence Partition

- Experience shows that exhaustive testing is not feasible or necessary:
 - Impractical for most methods.
 - An error in the code would have caused the same failure for many input values:
 - There is no reason why a value will be treated differently from others.
 - E.g., if value 240 fails the testing, it is *likely* that 241 is likely to fail the test too.
- A better way of choosing test cases is needed.

Equivalence Partition

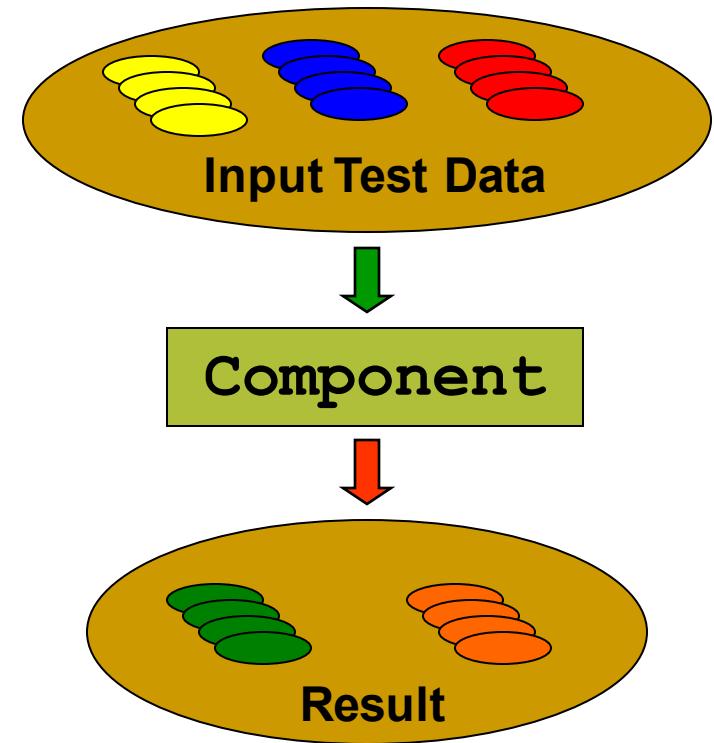
Observations:

- For a method, it is common to have a number of inputs that produce similar outcomes.
- Testing one of the inputs *should be* as good as exhaustively testing all of them.
- So, pick only a few test cases from each “category” of input that produce the same output.



Equivalence Partition: Definition

- Partition input data into *equivalence classes*.
- Data in each equivalence class:
 - Likely to be treated equally by a reasonable algorithm.
 - Produce same output state, i.e., valid/invalid.
- Derive test data for each class.

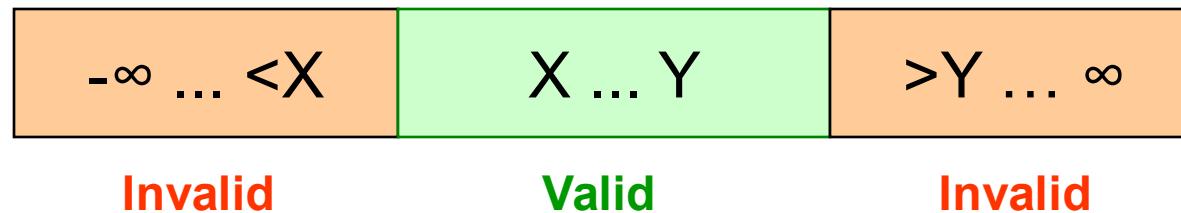


Example (*isValidMonth*)

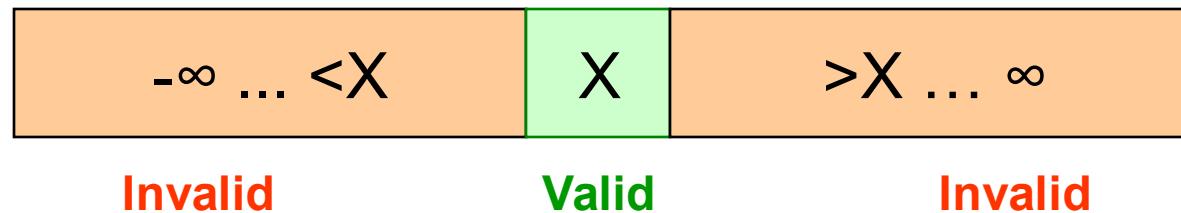
- For the *isValidMonth* example:
 - Input value [1 ... 12] should get a similar treatment.
 - Input values lesser than 1, larger than 12 are two other groups.
- Three partitions:
 - [$-\infty$... 0] should produce an **invalid** result
 - [1 ... 12] should produce a **valid** result
 - [13 ... ∞] should produce an **invalid** result
- Pick one value from each partition as test case:
 - E.g., { -12, 5, 15 }
 - Reduce the number of test cases significantly.

Common Partitions

- If the component specifies an input range, [X ... Y].
 - Three partitions:

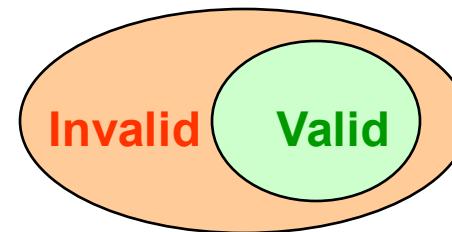


- If the component specifies a single value, [X].
 - Three partitions:

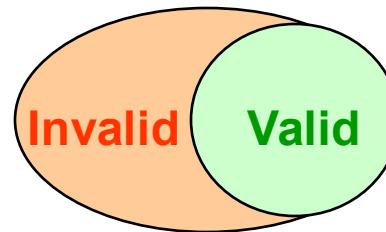


Common Partitions

- If the component specifies member(s) of a set:
 - E.g., The traffic light color = {Green, Yellow, Red}; Vehicles have to stop moving when the traffic light is Red.
 - Two partitions:



- If the component specifies a boolean value.
 - Two partitions:



Combination of Equivalence Classes

- Number of test cases can grow very large when there are multiple parameters.
- Example:
 - Check a phone number with the following format:
 - (XXX) XXX – XXXX
 - In addition:
 - Area Core Present: Boolean value [True or False]
 - Area Code: 3-digit number [200 ... 999] except 911
 - Prefix: 3-digit number not beginning with 0 or 1
 - Suffix: 4-digit number

Example (cont)

- Area Core Present: Boolean value [True or False]
 - Two Classes:
 - [True], [False]
- Area Code: 3-digit number [200 ... 999] except 911
 - Five Classes:
 - $[-\infty \dots 199]$, $[200 \dots 910]$, $[911]$, $[912 \dots 999]$, $[1000 \dots \infty]$
- Prefix: 3-digit number not beginning with 0 or 1
 - Three Classes:
 - $[-\infty \dots 199]$, $[200 \dots 999]$, $[1000 \dots \infty]$
- Suffix: 4-digit number
 - Three Classes:
 - $[-\infty \dots -1]$, $[0000 \dots 9999]$, $[10000 \dots \infty]$

Example (cont)

- A thorough testing would require us to test all combinations of the equivalence classes for each parameter.
- Hence, the total equivalence classes for the example should be the *multiplication* of equivalence classes for each input:
 - $2 \times 5 \times 3 \times 3 = 90$ classes = 90 test cases (!)
- For critical systems, all combinations should be tested to ensure a correct behavior.
- A less stringent testing strategy can be used for normal systems.

Reduction of Test Cases

- A reasonable approach to reduce the test cases is as follows:
 - At least one test for each equivalence class for each parameter:
 - Different equivalence class should be chosen for each parameter in each test to minimize the number of cases.
 - Test all combinations (if possible) where a parameter may affect each other.
 - A few other random combinations.

Example

- As the Area Code has the most classes (i.e., 5), five test cases can be defined which simultaneously try out different equivalence classes for each parameter:

| Test Case | Area Code Present | Area Code | Prefix | Suffix |
|-----------|-------------------|-----------------------|-----------------------|------------------------|
| 1 | False | $[-\infty \dots 199]$ | $[-\infty \dots 199]$ | $[-\infty \dots -1]$ |
| 2 | True | $[200 \dots 910]$ | $[200 \dots 999]$ | $[0000 \dots 9999]$ |
| 3 | True | $[911]$ | $[1000 \dots \infty]$ | $[10000 \dots \infty]$ |
| 4 | True | $[912 \dots 999]$ | $[200 \dots 999]$ | $[0000 \dots 9999]$ |
| 5 | True | $[1000 \dots \infty]$ | $[1000 \dots \infty]$ | $[10000 \dots \infty]$ |

E.g., Actual Test Case Data:
(True, 934, 222, 4321)

Example (cont)

- In this example, Area Code Present affects the interpretation of Area Code, so all combinations between these two parameters should be tested.
 - $2 \times 5 = 10$ test cases.
 - Five combinations have already been tested in the previous test cases, five more would be needed.
- Not counting extra random combinations, this strategy reduces the number of test cases to only 10.

Boundary Value Analysis: Introduction

- It has been found that most errors are caught at the **boundary** of the equivalence classes.
- Not surprising, as the end points of the boundary are usually used in the code for checking:
 - E.g., checking K is in range $[X \dots Y]$:

```
if (K >= X && K <= Y)
```

```
...
```

Easy to make
mistake on the
comparison.

- Hence, when choosing test data using equivalence classes, boundary values should be used.
- Nicely complement the Equivalence Class Testing.

Using Boundary Value Analysis

- If the component specifies a range, [$X \dots Y$]
 - Four values should be tested:
 - **Valid:** X and Y
 - **Invalid:** Just below X (e.g., $X - 1$)
 - **Invalid:** Just above Y (e.g., $Y + 1$)
 - E.g., [1 ... 12]
 - Test Data: { 0, 1, 12, 13 }
- Similar for open interval ($X \dots Y$), i.e., X and Y not inclusive.
 - Four values should be tested:
 - **Invalid:** X and Y
 - **Valid:** Just **above** X (e.g., $X + 1$)
 - **Valid:** Just **below** Y (e.g., $Y - 1$)
 - E.g., (100 ... 200)
 - Test Data: { 100, 101, 199, 200 }

Using Boundary Value Analysis

- If the component specifies a number of values:
 - Define test data using [min value ... max value]:
 - Valid: min, max
 - Invalid: Just below min (e.g., min - 1)
 - Invalid: Just above max (e.g., max + 1)
 - E.g., values = {2, 4, 6, 8} → [2 ... 8]
 - Test Data: {1, 2, 8, 9}
- If a data structure has prescribed boundaries:
 - define test data to exercise the data structure at those boundaries.
 - E.g.,
 - String: Empty String, String with 1 character
 - Array : Empty Array, Array with 1 element, Full Array
- Boundary value analysis is not applicable for data with no meaningful boundary, e.g., the set *color* {Red, Green, Yellow}.

Functionality Testing

- Previous examples have mostly numerical parameters and simplistic functionality, where it is easy to see how Equivalence Class Testing and Boundary Value Analysis can be applied.
- The following example is to illustrate how functionality testing of a method can be accomplished by the black box testing techniques discussed.
- This requires the method to be well specified:
 - The Precondition, Postcondition and Invariant should be available.
 - The Invariant: A property that is preserved by the method, i.e., true before and after the execution of method.

Functionality Testing

- For a well specified method (or component).
 - Use the Precondition:
 - Define Equivalence Classes.
 - Apply Boundary Value Analysis if possible to choose test data from the equivalence classes.
 - Use the Postcondition and the Invariant:
 - Derive expected results.

Example (Searching)

```
boolean Search(  
    List aList, int key)
```

Precondition:

-aList has at least one element

Postcondition:

- true if key is in the aList
- false if key is not in aList

Equivalence Classes

- Sequence with a single value:
 - key found.
 - key not found.
- Sequence of multi values:
 - key found:
 - First element in sequence.
 - Last element in sequence.
 - “Middle” element in sequence.
 - key not found.

Test Data

| Test Case | aList | Key | Expected Result |
|-----------|--------------------|-----|-----------------|
| 1 | [123] | 123 | True |
| 2 | [123] | 456 | False |
| 3 | [1, 6, 3, -4, 5] | 1 | True |
| 4 | [1, 6, 3, -4, 5] | 5 | True |
| 5 | [1, 6, 3, -4, 5] | 3 | True |
| 6 | [1, 6, 3, -4, 5] | 123 | False |

Example (Stack – Push Method)

```
void push (Object obj) throws FullStackException
```

Precondition:

- ! full()

Postconditions:

- if !full() on entry then
 - top() == obj && size() == old size() + 1
 - else throw FullStackException

Invariant:

- size() >= 0 && size() <= capacity()

- Common methods in the Stack class:
 - full(), top(), size(), capacity()

Test Data

- Precondition: stack is not full (i.e., boolean).

- Two equivalence classes can be defined.

- **Valid Case:** Stack is not full.

- **Input:** a non-full stack, an object `obj`
 - **Expected result:**

```
top() == obj  
size() == old size() + 1  
0 <= size() <= capacity()
```

- **Invalid Case:** Stack is full.

- **Input:** a full stack, an object `obj`
 - **Expected result:**

`FullStackException` is thrown

```
0 <= size() <= capacity()
```

Example

- A HR department of company is following employment process where applications are sorted out based on a person's age. A person under 15 as well as senior citizen older than 55 is not eligible to hire. Person less than 18 can be hired on a part-time basis only. Adult persons above 18 can hire as full time employees. Design Test Cases using Equivalence Partitioning and Boundary Value Analysis for above scenario.

Answer

■ Equivalence Partitioning

- Age Below 15
- Age 16 to 18
- Age 18 to 55
- Age above 55

■ Boundary Value Analysis:

- Test case values for Attribute Age: 14, 15, 16, 17, 18, 19, 54, 55, 56



Metrics for Process and Projects



What is Software metrics?

Software metrics (process and project) are quantitative measures. Measurement can be applied:

- To the software process with the intent of improvement
- To assist in estimation, quality control, productivity assessment, and project control
- To help assess the quality of technical work products
- To assist in tactical decision making as a project proceeds



Why we need to measure?

- ▶ To **characterize** in an effort to gain an understanding of process, products, resources and environments and to establish baselines for comparisons with future assessments.
- ▶ To **evaluate** to determine status with respect to plans
- ▶ To **predict** by gaining understanding of relationships among processes and products and building models of these relationships
- ▶ To **improve** by identifying roadblocks, root causes, inefficiencies, and other opportunities for improving product and process performance.



Metric and Indicator

- ▶ A **metric** is a quantitative measure of the degree to which a system, component, or process possesses a given attribute.
- ▶ An **indicator** is a metric or combination of metrics that provide insight into the software process, a software project, or the product itself.
- ▶ A software engineer collects measures and develops metrics so that indicators will be obtained.



Process Metrics

- ▶ Process metrics are collected across all projects and over long periods of time.
- ▶ Process metrics provide a set of process indicators that lead to long-term software process improvement
- ▶ The only way to know how/where to improve any process is to
 - ▶ Measure specific attributes of the process
 - ▶ Develop a set of meaningful metrics based on these attributes
 - ▶ Use the metrics to provide indicators that will lead to a strategy for improvement

Project metrics

Enable a software project manager to

- ▶ Assess the status of an ongoing project
- ▶ Track potential risks
- ▶ Uncover problem areas before they "go critical"
- ▶ Adjust work flow or tasks



Project metrics

- ▶ Evaluate the project team's ability to control quality of software engineering work products
- ▶ Estimate effort and time duration
- ▶ Every project should measure input, output and result

Use of Project Metrics

- ▶ The first application of project metrics occurs during estimation
 - ▶ Metrics from past projects are used as a basis for estimating time and effort
- ▶ As a project proceeds, the amount of time and effort expended are compared to original estimates
- ▶ As technical work commences, other project metrics become important
 - ▶ Production rates are measured (represented in terms of models created, review hours, function points, and delivered source lines of code)
 - ▶ Error uncovered during each generic framework activity (i.e., communication, planning, modeling, construction, deployment) are measured

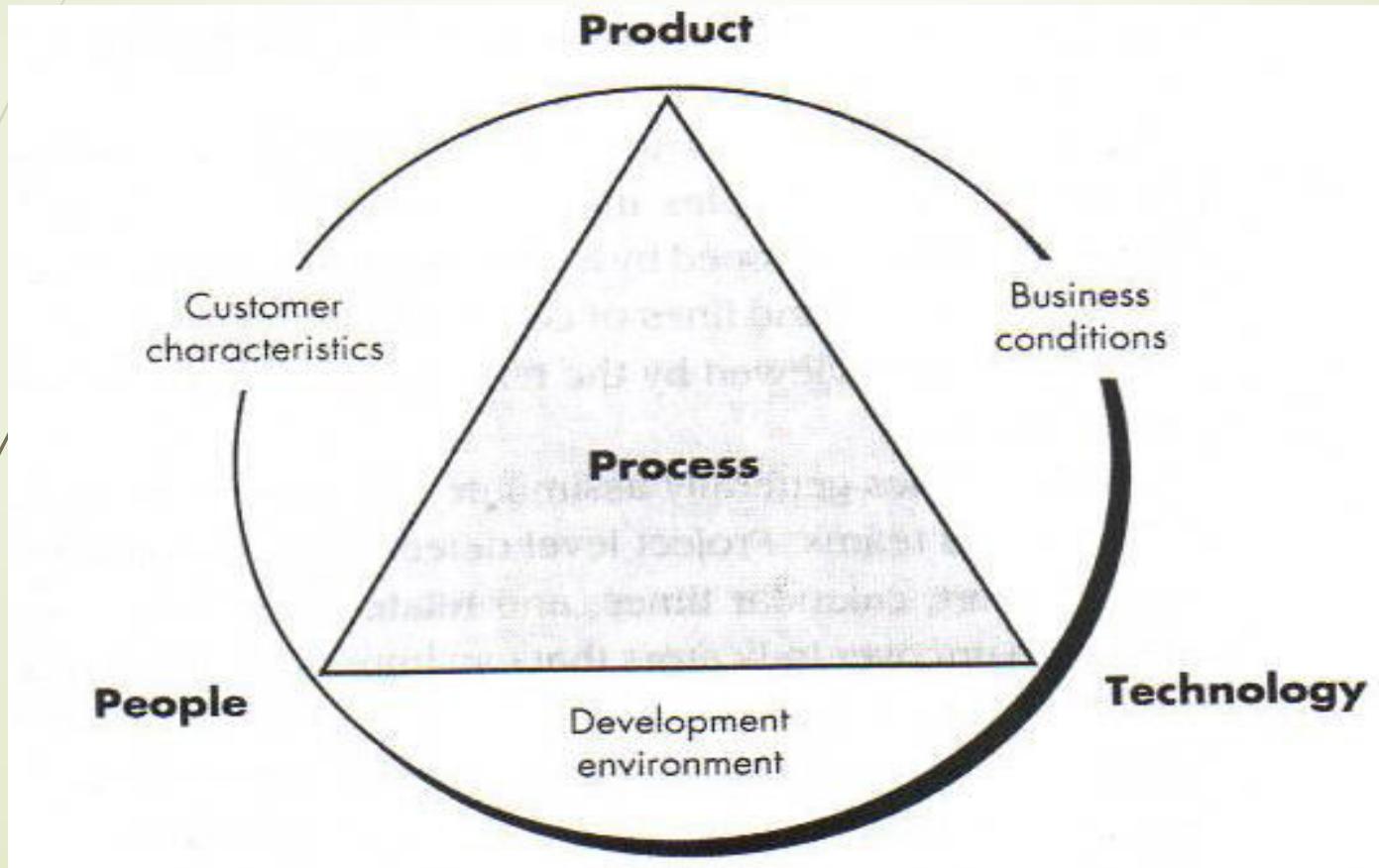
Use of Project Metrics

- ▶ Project metrics are used to
 - ▶ Minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks
 - ▶ Assess product quality on an ongoing basis and, when necessary, to modify the technical approach to improve quality
- ▶ In summary
 - ▶ As quality improves, defects are minimized
 - ▶ As defects go down, the amount of rework required during the project is also reduced
 - ▶ As rework goes down, the overall project cost is reduced

Summary: Project Metrics

- Used by project manager and a software team to adapt project workflow and technical activities.
- Metrics collected from past projects used to estimate effort and time.
- Project metrics measures: review hours, function points, delivered source lines, errors uncovered during each software
- Intent of project metrics is twofold: 1. these metrics are used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks.

Determinants for Software quality & Organizational Effectiveness



Private and Public metric

There are "private and public" uses for different types of process data

- Data *private* to the individual
 - Serve as an indicator for the individual only
- Eg : Defect rates, Errors found during development

Public metric

- Defects reported for major software functions
- Errors found during formal technical reviews
- Lines of code or function points per module/function

Software Measurement

- ▶ Direct measures
 - Software process (cost)
 - Software product (lines of code (LOC), execution speed, defects reported over some set period of time)
- ▶ Indirect measures
 - Software product (Functionality, quality, complexity, efficiency, reliability, maintainability)
- ▶ Many factors affect software work, it is difficult (don't use metrics) to compare individuals/team

Size-Oriented Metrics

- ▶ Derived by normalizing quality and/or productivity measures by considering the "size" of the software
 - ▶ Metrics include
 - ▶ Errors per KLOC
 - Errors per person-month
 - ▶ Defects per KLOC
 - KLOC per person-month
 - ▶ Dollars per KLOC
 - Dollars per page of documentation
 - ▶ Pages of documentation per KLOC
 - ▶ Opponents argue that KLOC measurements
 - ▶ Are dependent on the programming language
 - ▶ Penalize well-designed but short programs
 - ▶ Cannot easily accommodate nonprocedural languages
 - ▶ Require a level of detail that may be difficult to achieve



| Project | LOC | Effort | \$ (000) | Pp. doc. | Errors | Defects | People |
|---------|--------|--------|----------|----------|--------|---------|--------|
| alpha | 12,100 | 24 | 168 | 365 | 134 | 29 | 3 |
| beta | 27,200 | 62 | 440 | 1224 | 321 | 86 | 5 |
| gamma | 20,200 | 43 | 314 | 1050 | 256 | 64 | 6 |
| • | • | • | • | • | • | | |
| • | • | • | • | • | • | | |
| • | • | • | • | • | • | | |

Function-Oriented Metrics

- ▶ It is a measure of the functionality delivered by the application as a normalization value
- ▶ Eg. Number of input, number of output, number of files, number of interfaces

Function-oriented Metrics

- ▶ Function-oriented metrics use a measure of the functionality delivered by the application as a normalization value
- ▶ Most widely used metric of this type is the function point:

$$FP = \text{count total} * [0.65 + 0.01 * \text{sum (value adj. factors)}]$$

- ▶ Function point values on past projects can be used to compute, for example, the average number of lines of code per function point (e.g., 60)

Function Point Controversy

- ▶ Like the KLOC measure, function point use also has proponents and opponents
- ▶ Proponents claim that
 - ▶ FP is programming language independent
 - ▶ FP is based on data that are more likely to be known in the early stages of a project, making it more attractive as an estimation approach
- ▶ Opponents claim that
 - ▶ FP requires some “sleight of hand” because the computation is based on subjective data
 - ▶ Counts of the information domain can be difficult to collect after the fact
 - ▶ FP has no direct physical meaning...it's just a number

Reconciling LOC and FP Metrics

- ▶ Relationship between LOC and FP depends upon
 - ▶ The programming language that is used to implement the software
 - ▶ The quality of the design
- ▶ FP and LOC have been found to be relatively accurate predictors of software development effort and cost
 - ▶ However, a historical baseline of information must first be established
- ▶ LOC and FP can be used to estimate object-oriented software projects
 - ▶ However, they do not provide enough granularity for the schedule and effort adjustments required in the iterations of an evolutionary or incremental process
- ▶ The table on the next slide provides a rough estimate of the average LOC to one FP in various programming languages

LOC Per Function Point

| Language | Average | Median | Low | High |
|--------------|---------|--------|-----|------|
| Ada | 154 | -- | 104 | 205 |
| Assembler | 337 | 315 | 91 | 694 |
| C | 162 | 109 | 33 | 704 |
| C++ | 66 | 53 | 29 | 178 |
| COBOL | 77 | 77 | 14 | 400 |
| Java | 55 | 53 | 9 | 214 |
| PL/1 | 78 | 67 | 22 | 263 |
| Visual Basic | 47 | 42 | 16 | 158 |

Object-oriented Metrics

- ▶ Number of scenario scripts (i.e., use cases)
 - ▶ This number is directly related to the size of an application and to the number of test cases required to test the system
- ▶ Number of key classes (the highly independent components)
 - ▶ Key classes are defined early in object-oriented analysis and are central to the problem domain
 - ▶ This number indicates the amount of effort required to develop the software
 - ▶ It also indicates the potential amount of reuse to be applied during development
- ▶ Number of support classes
 - ▶ Support classes are required to implement the system but are not immediately related to the problem domain (e.g., user interface, database, computation)
 - ▶ This number indicates the amount of effort and potential reuse

Object-oriented Metrics

- ▶ Average number of support classes per key class
 - ▶ Key classes are identified early in a project (e.g., at requirements analysis)
 - ▶ Estimation of the number of support classes can be made from the number of key classes
 - ▶ GUI applications have between two and three times more support classes as key classes
 - ▶ Non-GUI applications have between one and two times more support classes as key classes
- ▶ Number of subsystems
 - ▶ A subsystem is an aggregation of classes that support a function that is visible to the end user of a system

Metrics for Software Quality

► Correctness

- This is the number of defects per KLOC, where a defect is a verified lack of conformance to requirements
- Defects are those problems reported by a program user after the program is released for general use

► Maintainability

- This describes the ease with which a program can be corrected if an error is found, adapted if the environment changes, or enhanced if the customer has changed requirements
- Mean time to change (MTTC) : the time to analyze, design, implement, test, and distribute a change to all users
 - Maintainable programs on average have a lower MTTC

Defect Removal Efficiency

- ▶ Defect removal efficiency provides benefits at both the project and process level
- ▶ It is a measure of the filtering ability of QA activities as they are applied throughout all process framework activities
 - ▶ It indicates the percentage of software errors found before software release
- ▶ It is defined as $DRE = E / (E + D)$
 - ▶ E is the number of errors found before delivery of the software to the end user
 - ▶ D is the number of defects found after delivery
- ▶ As D increases, DRE decreases (i.e., becomes a smaller and smaller fraction)
- ▶ The ideal value of DRE is 1, which means no defects are found after delivery
- ▶ DRE encourages a software team to institute techniques for finding as many errors as possible before delivery

Process and Project Metrics

Software metrics

Software metrics (process and project) are quantitative measures

Measurement can be applied

- To the software process with the intent of improvement
- To assist in estimation, quality control, productivity assessment, and project control
- To help assess the quality of technical work products
- To assist in tactical decision making as a project proceeds

Metric and Indicator

- A **metric** is a quantitative measure of the degree to which a system, component, or process possesses a given attribute.
- An **indicator** is a metric or combination of metrics that provide insight into the software process, a software project, or the product itself.
- A software engineer collects measures and develops metrics so that indicators will be obtained .

Process Metrics

- Process metrics are collected across all projects and over long periods of time.
- Process metrics provide a set of process indicators that lead to long-term software process improvement

Project metrics

Enable a software project manager to

- Assess the status of an ongoing project
- Track potential risks
- Uncover problem areas before they "go critical"
- Adjust work flow or tasks

Project metrics

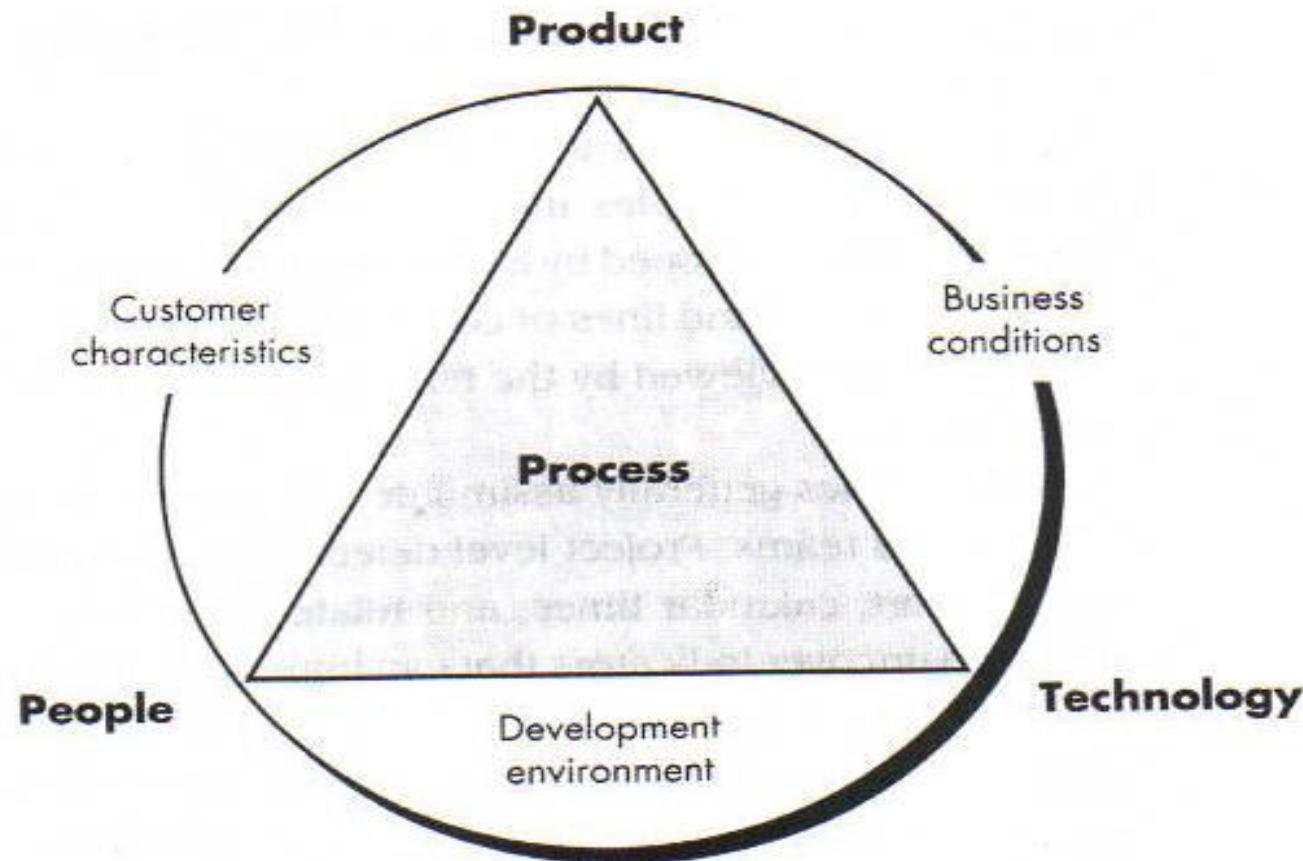
- Evaluate the project team's ability to control quality of software engineering work products.
- Estimate effort and time duration
- Every project should measure input, output and result

Process Metrics and Software Process Improvement

The only rational way to improve any process is

- To measure specific attributes of the process
- Develop a set of meaningful metrics based on these attributes
- Use the metrics to provide indicators that will lead to a strategy for improvement

Determinants for Software quality & Organizational Effectiveness



Private and Public metric

There are "private and public" uses for different types of process data

- Data *private* to the individual
 - Serve as an indicator for the individual only
- Eg : Defect rates, Errors found during development

Public metric

- Defects reported for major software functions
- Errors found during formal technical reviews
- Lines of code or function points per module/function

Software Measurement

- Direct measures
 - Software process (cost)
 - Software product (lines of code (LOC), execution speed, defects reported over some set period of time)
- Indirect measures
 - Software product (Functionality, quality, complexity, efficiency, reliability, maintainability)
- Many factors affect software work, it is difficult (don't use metrics) to compare individuals/team

Size-Oriented Metrics

- Derived by normalizing quality and/or productivity measures by considering the "size" of the software
- Metrics include
 - Errors per KLOC, Defects per KLOC, Dollars per KLOC, Pages of documentation per KLOC

| Project | LOC | Effort | \$ (000) | Pp. doc. | Errors | Defects | People |
|---------|--------|--------|----------|----------|--------|---------|--------|
| alpha | 12,100 | 24 | 168 | 365 | 134 | 29 | 3 |
| beta | 27,200 | 62 | 440 | 1224 | 321 | 86 | 5 |
| gamma | 20,200 | 43 | 314 | 1050 | 256 | 64 | 6 |
| : | : | : | : | : | : | : | |
| : | : | : | : | : | : | : | |
| : | : | : | : | : | : | : | |

Function-Oriented Metrics

- It is a measure of the functionality delivered by the application as a normalization value
- Eg. Number of input, number of output, number of files, number of interfaces

Reconciling LOC & FC Metrics

- The relationship between lines of code and function points depends programming language
- Rough estimates of the average number of lines of code required to build one function point in various programming languages

| Programming Language | LOC per Function point | | | |
|----------------------|------------------------|--------|-----|------|
| | Avg. | Median | Low | High |
| Access | 35 | 38 | 15 | 47 |
| Ada | 154 | — | 104 | 205 |
| APS | 86 | 83 | 20 | 184 |
| ASP 69 | 62 | — | 32 | 127 |
| Assembler | 337 | 315 | 91 | 694 |
| C | 162 | 109 | 33 | 704 |
| C++ | 66 | 53 | 29 | 178 |
| Clipper | 38 | 39 | 27 | 70 |
| COBOL | 77 | 77 | 14 | 400 |

Metrics in the Process Domain

Metrics in the Process Domain

- Process metrics are collected across all projects and over long periods of time
- They are used for making strategic decisions
- The intent is to provide a set of process indicators that lead to long-term software process improvement
- The only way to know how/where to improve any process is to
 - Measure specific attributes of the process
 - Develop a set of meaningful metrics based on these attributes
 - Use the metrics to provide indicators that will lead to a strategy for improvement

Metrics in the Process Domain (continued)

- We measure the effectiveness of a process by deriving a set of metrics based on outcomes of the process such as
 - Errors uncovered before release of the software
 - Defects delivered to and reported by the end users
 - Work products delivered
 - Human effort expended
 - Calendar time expended
 - Conformance to the schedule
 - Time and effort to complete each generic activity

Etiquette of Process Metrics

- Use common sense and organizational sensitivity when interpreting metrics data
- Provide regular feedback to the individuals and teams who collect measures and metrics
- Don't use metrics to evaluate individuals
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them
- Never use metrics to threaten individuals or teams
- Metrics data that indicate a problem should not be considered “negative”
 - Such data are merely an indicator for process improvement
- Don't obsess on a single metric to the exclusion of other important metrics

Metrics in the Project Domain

Metrics in the Project Domain

- Project metrics enable a software project manager to
 - Assess the status of an ongoing project
 - Track potential risks
 - Uncover problem areas before their status becomes critical
 - Adjust work flow or tasks
 - Evaluate the project team's ability to control quality of software work products
- Many of the same metrics are used in both the process and project domain
- Project metrics are used for making tactical decisions
 - They are used to adapt project workflow and technical activities

Use of Project Metrics

- The first application of project metrics occurs during estimation
 - Metrics from past projects are used as a basis for estimating time and effort
- As a project proceeds, the amount of time and effort expended are compared to original estimates
- As technical work commences, other project metrics become important
 - Production rates are measured (represented in terms of models created, review hours, function points, and delivered source lines of code)
 - Error uncovered during each generic framework activity (i.e, communication, planning, modeling, construction, deployment) are measured

Use of Project Metrics (continued)

- Project metrics are used to
 - Minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks
 - Assess product quality on an ongoing basis and, when necessary, to modify the technical approach to improve quality
- In summary
 - As quality improves, defects are minimized
 - As defects go down, the amount of rework required during the project is also reduced
 - As rework goes down, the overall project cost is reduced

Software Project Planning

The overall goal of project planning is to establish a pragmatic strategy for controlling, tracking, and monitoring a complex technical project.

Why?

So the end result gets done on time, with quality!

Estimation

- Estimation of resources, cost, and schedule for a software engineering effort requires
 - experience
 - access to good historical information (metrics)
 - the courage to commit to quantitative predictions when qualitative information is all that exists
- Estimation carries inherent risk and this risk leads to uncertainty

Estimation Techniques

- Past (similar) project experience
- Conventional estimation techniques
 - task breakdown and effort estimates
 - size (e.g., FP) estimates
- Empirical models
- Automated tools

Conventional Two types of Metrics

- Size oriented
- Function Point oriented

Size-oriented Metrics

- Derived by normalizing quality and/or productivity measures by considering the size of the software produced
- Thousand lines of code (KLOC) are often chosen as the normalization value
- Metrics include
 - Errors per KLOC
 - Defects per KLOC
 - Dollars per KLOC
 - Pages of documentation per KLOC
 - Errors per person-month
 - KLOC per person-month
 - Dollars per page of documentation

Size-oriented Metrics (continued)

- Size-oriented metrics are not universally accepted as the best way to measure the software process
- Opponents argue that KLOC measurements
 - Are dependent on the programming language
 - Penalize well-designed but short programs
 - Cannot easily accommodate nonprocedural languages
 - Require a level of detail that may be difficult to achieve

Example: LOC Approach

Average productivity for systems of this type = 620 LOC/pm.

Burdened labor rate = \$8000 per month, the cost per line of code is approximately \$13.

Based on the LOC estimate and the historical productivity data, the total estimated project cost is **\$431,000** and the estimated effort is 54 person-months.

An Example of LOC-Based Estimation

| Function | Estimated LOC |
|--|---------------|
| User interface and control facilities (UCIF) | 2300 |
| Two-dimensional geometric analysis | 5300 |
| Three-dimensional geometric analysis | 6800 |
| Database management | 3350 |
| Computer graphics display facilities | 4950 |
| Peripheral control function | 2100 |
| Design Analysis Modules | 8400 |
| <i>Estimated lines of code</i> | 33200 |

Function-Oriented Metrics

- Function-oriented metrics use a measure of the functionality delivered by the application as a normalization value
- Most widely used metric of this type is the function point:

$$FP = \text{count total} * [0.65 + 0.01 * \text{sum (value adj. factors)}]$$

- Function point values on past projects can be used to compute, for example, the average number of lines of code per function point (e.g., 60)

Function Points (5 characteristics)

Based on a combination of program 5 characteristics

The number of :

- External (user) inputs: input transactions that update internal files
- External (user) outputs: reports, error messages
- User interactions: inquiries
- Logical internal files used by the system:

Example a purchase order logical file composed of 2 physical files/tables Purchase_Order and Purchase_Order_Item

- External interfaces: files shared with other systems

Function Points (FP)

- A weight is associated with each of the above 5 characteristics
- Weight range:
 - from 3 for simple feature to
 - 15 for complex feature
- The function point count is computed by multiplying each raw count by the weight and summing all values

FP Calculation

| <u>measurement parameter</u> | <u>count</u> | <u>weighting factor</u> | | | = | <input type="text"/> |
|------------------------------|----------------------|-------------------------|------|---------|---|----------------------|
| | | simple | avg. | complex | | |
| number of user inputs | <input type="text"/> | X 3 | 4 | 6 | = | <input type="text"/> |
| number of user outputs | <input type="text"/> | X 4 | 5 | 7 | = | <input type="text"/> |
| number of user inquiries | <input type="text"/> | X 3 | 4 | 6 | = | <input type="text"/> |
| number of files | <input type="text"/> | X 7 | 10 | 15 | = | <input type="text"/> |
| number of ext.interfaces | <input type="text"/> | X 5 | 7 | 10 | = | <input type="text"/> |
| count-total | <hr/> | | | | | <input type="text"/> |
| complexity multiplier | <hr/> | | | | | <input type="text"/> |
| function points | <hr/> | | | | | <input type="text"/> |

Adjusted Function Points Count Complexity: 14 Factors Fi

14 factors: Each factor is rated on a scale of:

Zero: not important or not applicable

Five: absolutely essential

1. Backup and recovery
2. Data communication
3. Distributed processing functions
4. Is performance critical?
5. Existing operating environment
6. On-line data entry
7. Input transaction built over multiple screens

Adjusted Function Points Count Complexity: 14 Factors Fi

8.Master files updated on-line

9.Complexity of inputs, outputs, files, inquiries

10.Complexity of processing

11.Code design for re-use

12.Are conversion/installation included in design?

13.Multiple installations

14.Application designed to facilitate change by the user

Final computation

- Value adjustment factor (F_i) = SUM (Backup and recovery + Data communication + Performance Criteria etc..)
- $F_{\text{Estimated}} = \text{Count_total} * [0.65 + 0.01 * \sum (F_i)]$

Function Points

- Each of the F_i criteria are given a rating of 0 to 5 as:
 - No Influence = 0; Incidental = 1;
 - Moderate = 2 Average = 3;
 - Significant = 4 Essential = 5

Function-Oriented Metrics

- Once function points are calculated, they are used in a manner analogous to LOC as a measure of software productivity, quality and other attributes, e.g.:
 - productivity FP/person-month
 - quality faults/FP
 - cost \$\$/FP
 - documentation doc_pages/FP

Example: Function Points

| | |
|---|------|
| # of user inputs: {on, off, ext. number} (3) x simple (3) | = 9 |
| # of user outputs: {tone} (1) x simple (4) | = 4 |
| # of user inquiries: 0 x simple | = 0 |
| # of files: {mapping table} (1)x simple (7) | = 7 |
| # of external interfaces: {memory map} (1) x simple (5) | = 5 |
| Total count | = 25 |

Example 1

- Consider the *average* as the degree of complexity with following functional units and compute the function point for the project
 - Number of input = 40
 - Number of output = 28
 - Number of user enquiries = 32
 - Number of user files = 9
 - Number of external interface = 5
 - Adjustment factor = 1.07

Example 2

- i] Number of user inputs=5, with degree of complexity equal to simple
- ii] Number of user output=10, with degree of complexity equal to average
- iii] Number of user enquiries=5, with degree of complexity equal to complex
- iv] Number of user files=8, with degree of complexity equal to simple
- v] Number of external interfaces=3, with degree of complexity equal to complex

- vi] Backup and recovery= 0.5
- vi] Adjustment factor excluding Backup and recovery is 1.07, compute the function point for the project.

Empirical Estimation Method COCOMO II

The COCOMO II Model

- The Constructive Cost Model (COCOMO) is an algorithmic software cost estimation model
- Based on the complexity the project can be divided into 3 different modes
 - Organic mode (simple)
 - Semi-detached mode (medium)
 - Embedded mode (complex)
- Effort = $a * KLOC^b$, (in person/months)
- Duration = $c * effort^d$, (in months)
- Staffing =Effort/Duration

The COCOMO II Model

- Organic mode (simple)
 - $a = 2.4, b = 1.05, c = 2.5, d = 0.38$
- Semi-detached mode (medium)
 - $a = 3, b = 1.12, c = 2.5, d = 0.35$
- Embedded mode (complex)
 - $a = 3.6, b = 1.2, c = 2.5, d = 0.32$

Assume that the size of an organic type software product has been estimated to be 32,000 lines of source code. Assume that the average salary of software engineers be Rs. 15,000/- per month. Determine the effort required to develop the software product and the nominal development time. From the basic COCOMO estimation formula for organic software:

From the basic COCOMO estimation formula for organic software:

$$\text{Effort} = 2.4 \times (32)^{1.05} = 91 \text{ PM}$$

$$\text{Nominal development time} = 2.5 \times (91)^{0.38} = 14 \text{ months}$$

| | |
|--------------------------------------|---------------------------|
| Cost required to develop the product | $= 14 \times 15,000$ |
| | $= \text{Rs. } 210,000/-$ |

Reconciling LOC and FP Metrics

- Relationship between LOC and FP depends upon
 - The programming language that is used to implement the software
 - The quality of the design
- FP and LOC have been found to be relatively accurate predictors of software development effort and cost
 - However, a historical baseline of information must first be established
- LOC and FP can be used to estimate object-oriented software projects
 - However, they do not provide enough granularity for the schedule and effort adjustments required in the iterations of an evolutionary or incremental process
- The table on the next slide provides a rough estimate of the average LOC to one FP in various programming languages

Metrics for Software Quality

- Correctness
 - This is the number of defects per KLOC, where a defect is a verified lack of conformance to requirements
 - Defects are those problems reported by a program user after the program is released for general use
- Maintainability
 - This describes the ease with which a program can be corrected if an error is found, adapted if the environment changes, or enhanced if the customer has changed requirements
 - Mean time to change (MTTC) : the time to analyze, design, implement, test, and distribute a change to all users
 - Maintainable programs on average have a lower MTTC

Defect Removal Efficiency

- Defect removal efficiency provides benefits at both the project and process level
- It is a measure of the filtering ability of QA activities as they are applied throughout all process framework activities
 - It indicates the percentage of software errors found before software release
- It is defined as $DRE = E / (E + D)$
 - E is the number of errors found before delivery of the software to the end user
 - D is the number of defects found after delivery
- As D increases, DRE decreases (i.e., becomes a smaller and smaller fraction)
- The ideal value of DRE is 1, which means no defects are found after delivery
- DRE encourages a software team to institute techniques for finding as many errors as possible before delivery

Project Scheduling

Introduction

- Involves deciding which tasks would be taken up when
- Project Manager need to do the following
 - Identify all the tasks
 - Break down large tasks into small activities
 - Determine the dependency among activities
 - Estimate the time duration to complete the activities
 - Allocate resources to activities
 - Plan start and end dates
 - Determine the *Critical path*
- Task dependencies
- Milestones

“Failing to plan is planning to fail”

by J. Hinze, *Construction Planning and Scheduling*

- Planning:
 - “what” is going to be done, “how”, “where”, by “whom”, and “when”
 - for effective monitoring and control of complex projects

“It’s about time”

by J. Hinze, *Construction Planning and Scheduling*

- Scheduling:
 - “what” will be done, and “who” will be working
 - relative timing of tasks & time frames
 - a concise description of the plan

“Once you plan your work, you must work your plan”

by J. Hinze, *Construction Planning and Scheduling*

- Planning and Scheduling occurs:
 - **AFTER** you have decided how to do the work
 - “The first idea is not always the best idea.”
- Requires discipline to “work the plan”
 - The act of development useful,
 - But need to monitor and track
 - only then, is a schedule an effective management tool
 - as-built schedules

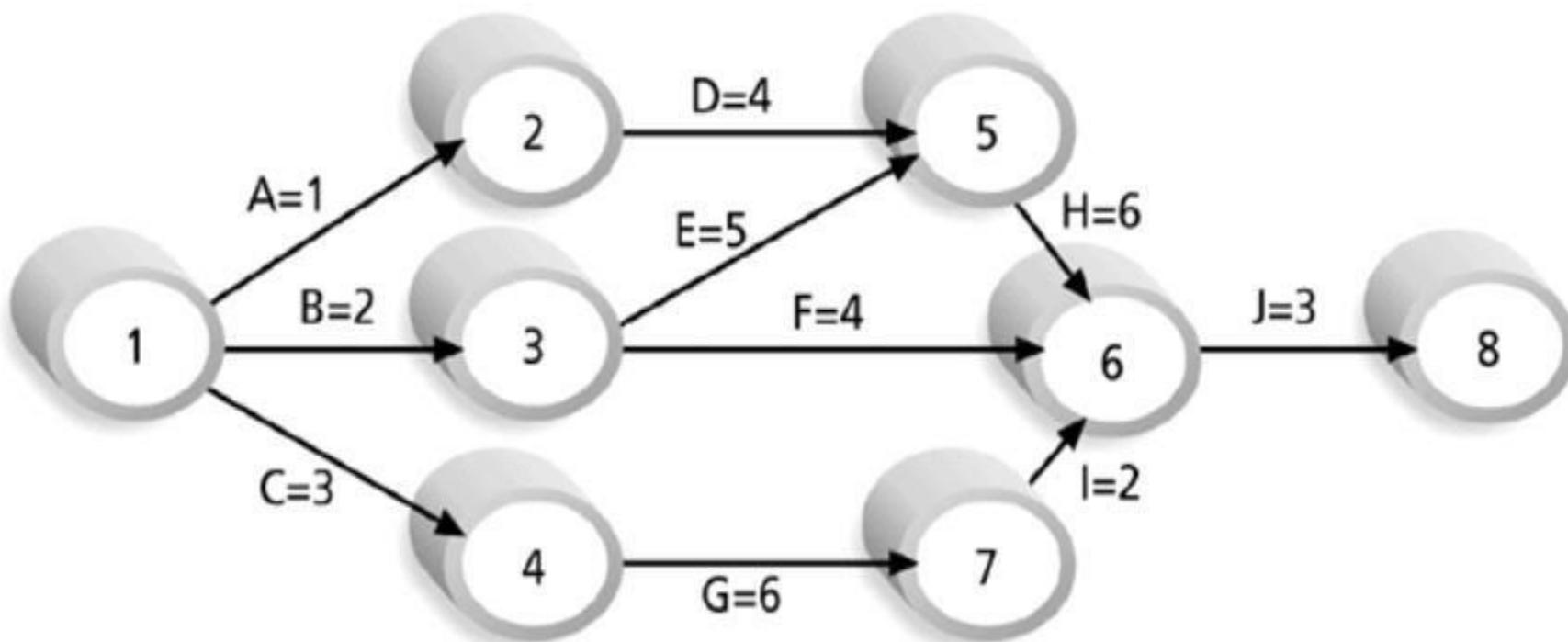
Activity Network

- Activity network shows
 - Different activities
 - Estimated durations
 - Interdependencies

Network Diagrams

- Network diagrams are the preferred technique for showing activity sequencing.
- A **network diagram** is a schematic display of the logical relationships among, or sequencing of, project activities.
- Two main formats are the arrow and precedence diagramming methods.

Activity Network Diagram for Project X



Note: Assume all durations are in days; A=1 means Activity A has a duration of 1 day.

Techniques of Scheduling

- CPM
- PERT
- GANTT

The PERT/CPM Approach for Project Scheduling

- The PERT/CPM approach to project scheduling uses network presentation of the project to
 - Reflect activity precedence relations
 - Activity completion time
- PERT/CPM is used for scheduling activities such that the project's completion time is minimized.

Critical Path Method

- It is the chain of activities that determines the duration of the project.
- Different Parameters:
 - Earliest Start (ES) Time
 - Latest Start (LS) Time
 - Earliest Finish (EF) Time
 - Latest Finish (LF) Time
 - Slack Time (ST) = LS-ES, equivalently can be written as LF-EF
 - Critical task is one with a *zero* slack time
 - Path from start to finish containing only critical task is critical path

Identifying the Activities of a Project

- To determine optimal schedules we need to
 - Identify all the project's activities.
 - Determine the precedence relations among activities.
- Based on this information we can develop managerial tools for project control.

Identifying Activities, Example

KLONE COMPUTERS, INC.

- **KLONE Computers manufactures personal computers.**
- **It is about to design, manufacture, and market the Klonepalm 2000 palmbook computer.**

KLONE COMPUTERS, INC

- There are three major tasks to perform:
 - Manufacture the new computer.
 - Train staff and vendor representatives.
 - Advertise the new computer.
- KLONE needs to develop a precedence relations chart.
- The chart gives a concise set of tasks and their immediate predecessors.

KLONE COMPUTERS, INC

| | <u>Activity</u> | <u>Description</u> |
|--------------------------|-----------------|-------------------------------------|
| Manufacturing activities | A | Prototype model design |
| | B | Purchase of materials |
| | C | Manufacture of prototype model |
| | D | Revision of design |
| | E | Initial production run |
| Training activities | F | Staff training |
| | G | Staff input on prototype models |
| | H | Sales training |
| Advertising activities | I | Pre-production advertising campaign |
| | J | Post-redesign advertising campaign |

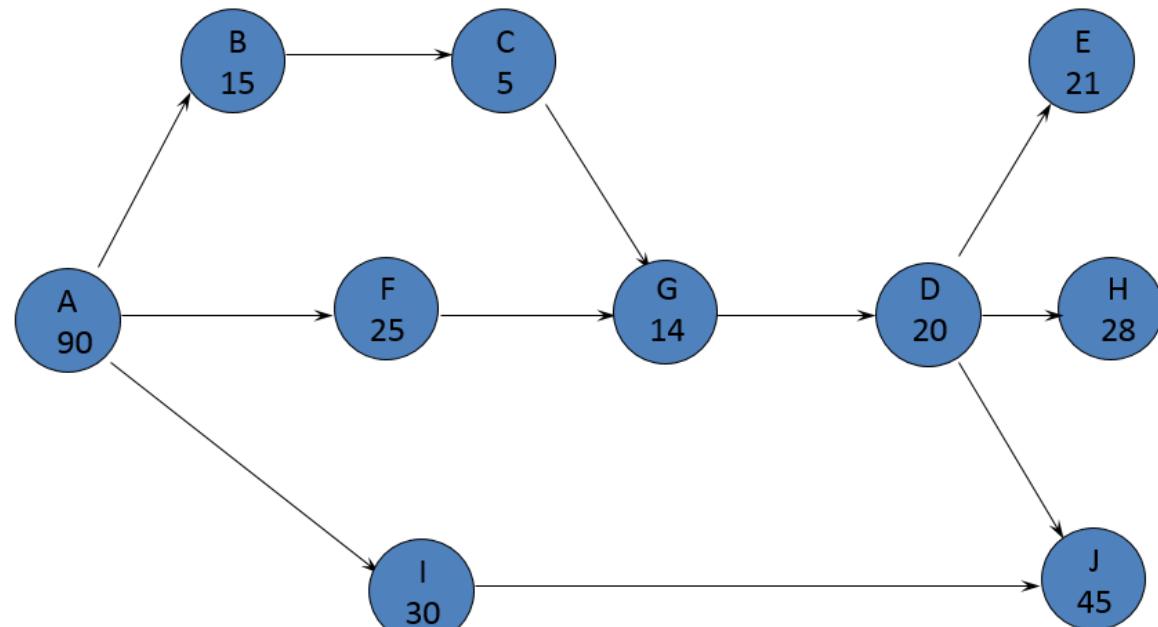
KLONE COMPUTERS, INC

**From the activity description chart,
we can determine immediate
predecessors for each activity.**



Activity A is an immediate predecessor of activity B, because it must be completed just prior to the commencement of B.

Precedence Relationships Chart



| Activity | Immediate Predecessor | Completion Time |
|----------|-----------------------|-----------------|
| A | None | 90 |
| B | A | 15 |
| C | B | 5 |
| D | G | 20 |
| E | D | 21 |
| F | A | 25 |
| G | C,F | 14 |
| H | D | 28 |
| I | A | 30 |
| J | D,I | 45 |

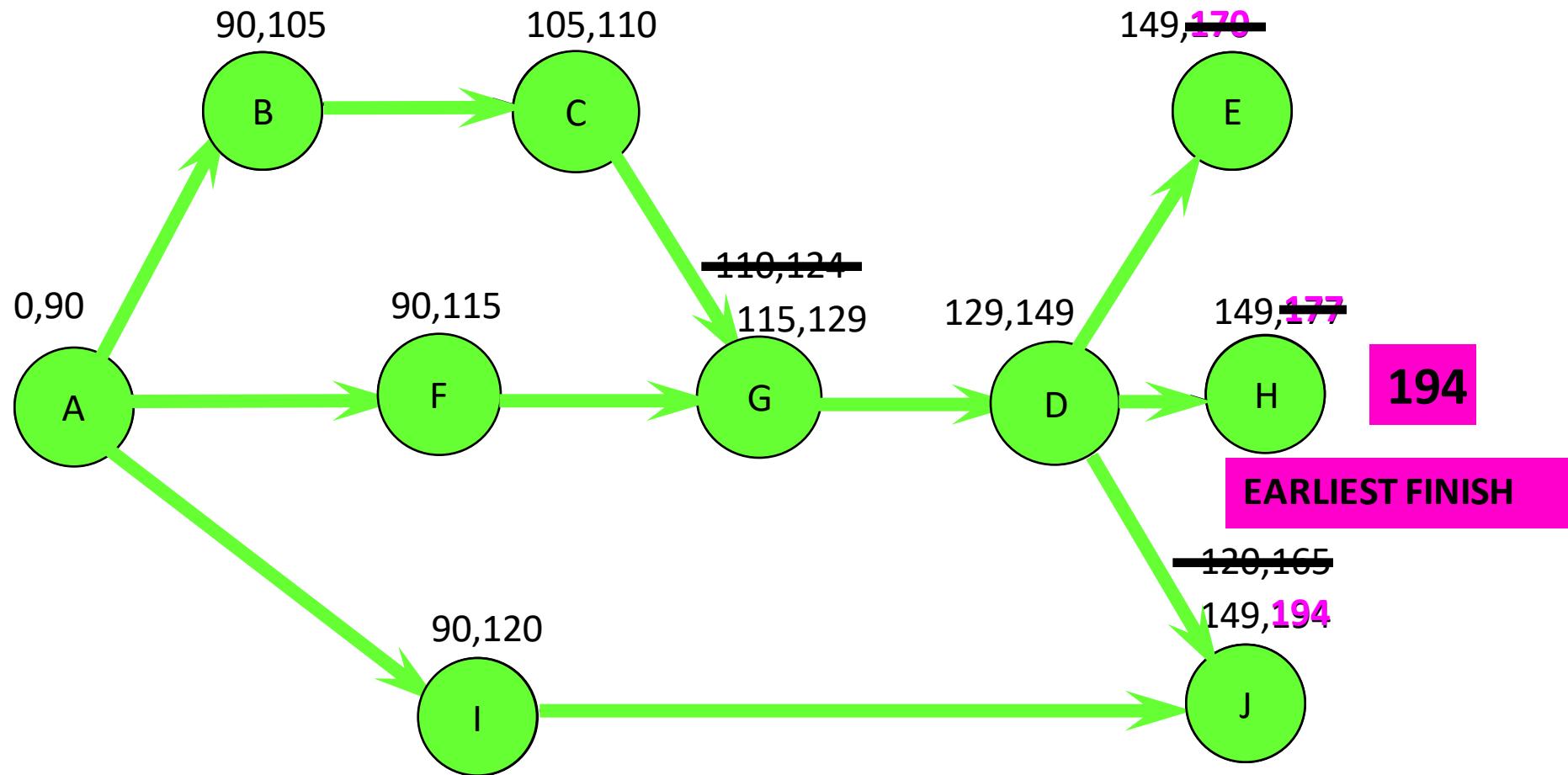
KLONE COMPUTERS, INC. - Continued

- Management at KLONE would like to schedule the activities so that the project is completed in minimal time.
- Management wishes to know:
 - The earliest and latest start times for each activity which will not alter the earliest completion time of the project.
 - The earliest finish times for each activity which will not alter this date.
 - Activities with rigid schedule and activities that have slack in their schedules.

Earliest Start Time / Earliest Finish Time

- Make a forward pass through the network as follows:
 - Evaluate all the activities which have no immediate predecessors.
 - The earliest start for such an activity is zero $ES = 0$.
 - The earliest finish is the activity duration $EF = \text{Activity duration}$.
 - Evaluate the ES of all the nodes for which EF of all the immediate predecessor has been determined.
 - $ES = \text{Max } EF$ of all its immediate predecessors.
 - $EF = ES + \text{Activity duration}$.
 - Repeat this process until all nodes have been evaluated
 - EF of the finish node is the earliest finish time of the project.

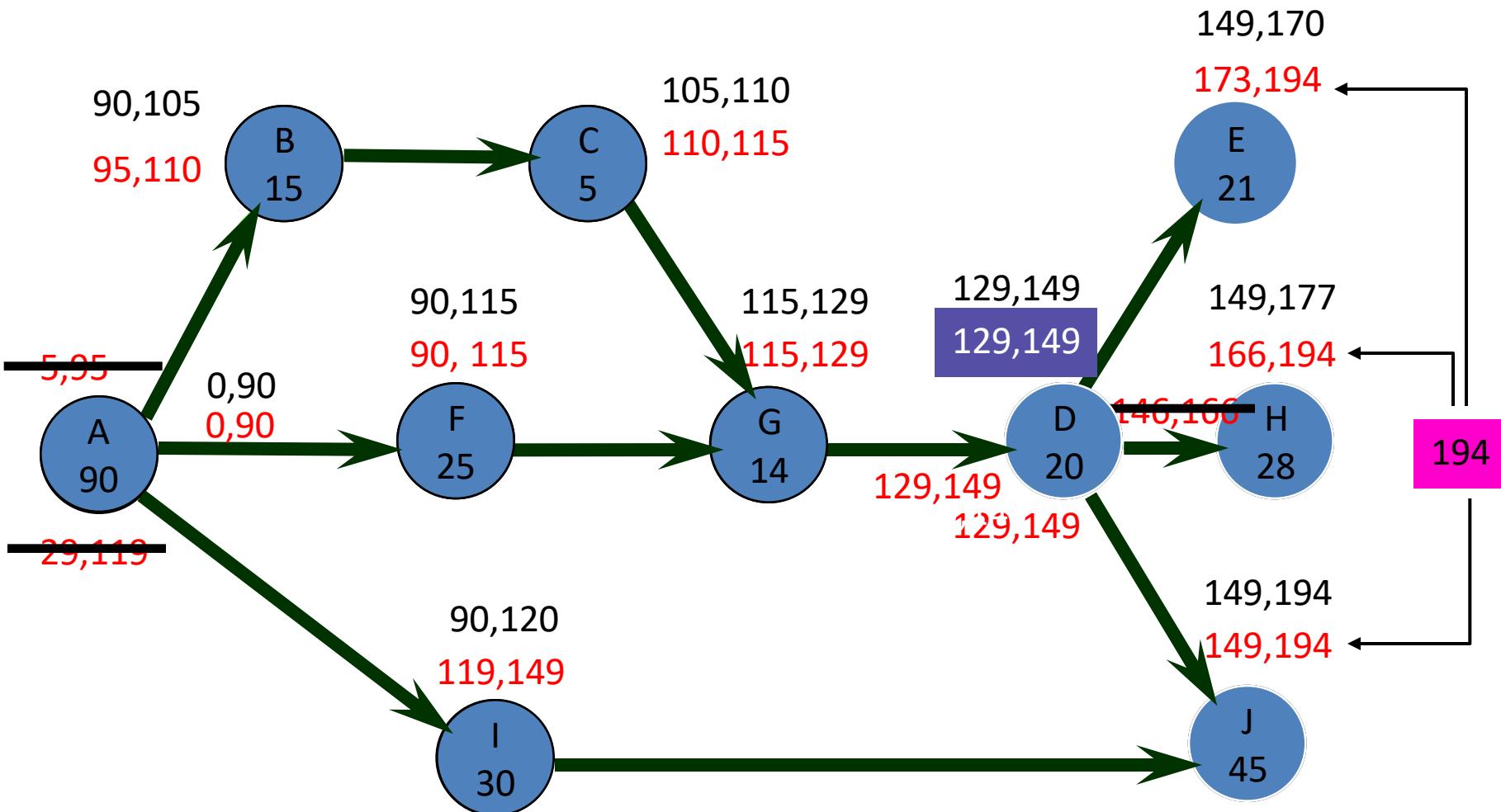
Earliest Start / Earliest Finish – Forward Pass



Latest start time / Latest finish time

- Make a backward pass through the network as follows:
 - Evaluate all the activities that immediately precede the finish node.
 - The latest finish for such an activity is $LF = \text{minimal project completion time}$.
 - The latest start for such an activity is $LS = LF - \text{activity duration}$.
 - Evaluate the LF of all the nodes for which LS of all the immediate successors has been determined.
 - $LF = \text{Min } LS \text{ of all its immediate successors.}$
 - $LS = LF - \text{Activity duration.}$
 - Repeat this process backward until all nodes have been evaluated.

Latest Start / Latest Finish – Backward Pass



Slack Times

- Activity start time and completion time may be delayed by planned reasons as well as by unforeseen reasons.
- Some of these delays may affect the overall completion date.
- To learn about the effects of these delays, we calculate the **slack time**, and form the **critical path**.

Slack Times

- Slack time is the amount of time an activity can be delayed without delaying the project completion date, assuming no other delays are taking place in the project.

Slack Time = LS - ES = LF - EF

Slack time in the Klonepalm 2000 Project

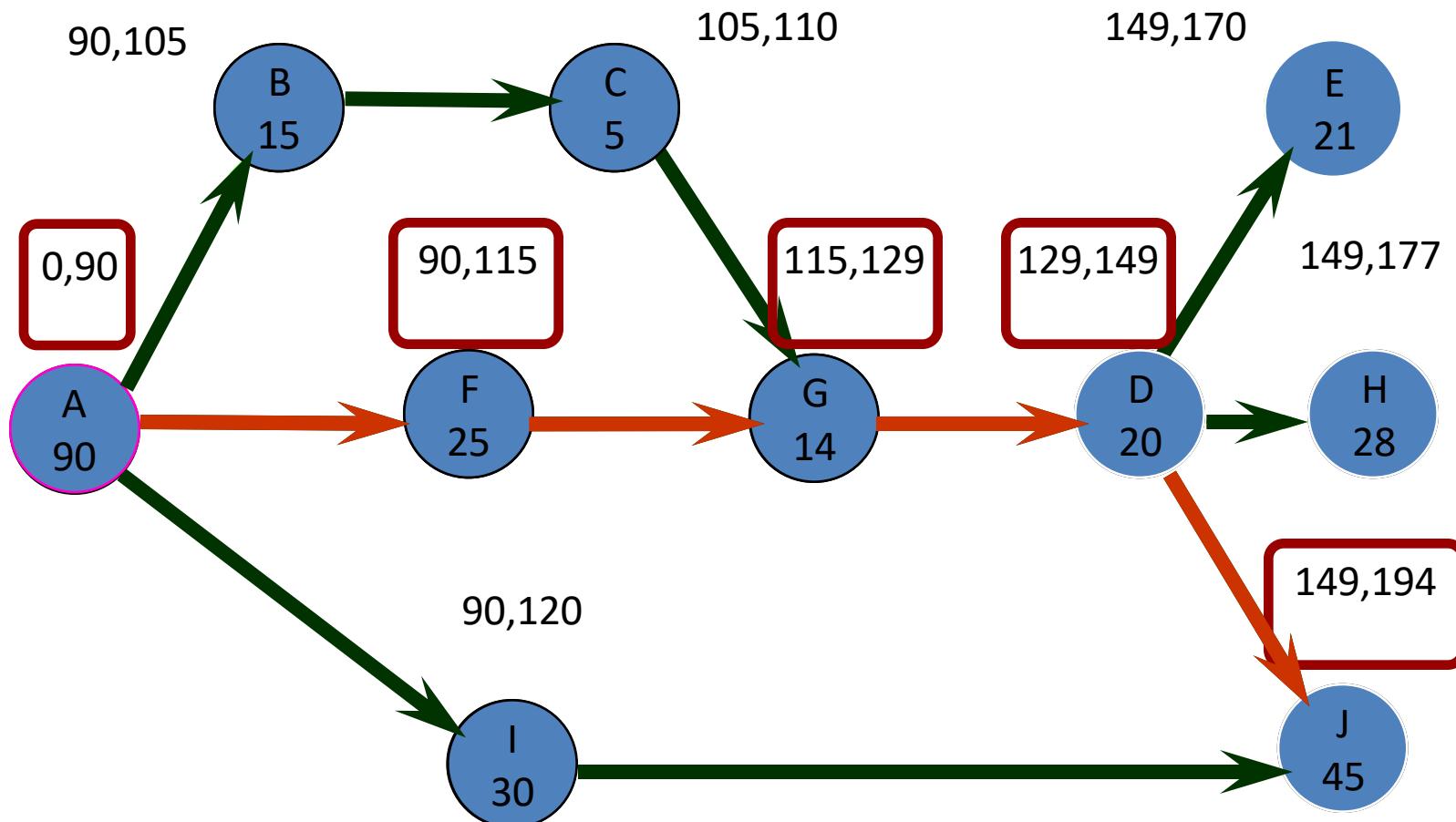
| Activity | LS - ES | Slack |
|----------|-----------|-------|
| A | 0 - 0 | 0 |
| B | 95 - 90 | 5 |
| C | 110 - 105 | 5 |
| D | 119 - 119 | 0 |
| E | 173 - 149 | 24 |
| F | 90 - 90 | 0 |
| G | 115 - 115 | 0 |
| H | 166 - 149 | 17 |
| I | 119 - 90 | 29 |
| J | 149 - 149 | 0 |

Critical activities
must be rigidly
scheduled

The Critical Path

- The critical path is a set of activities that have no slack, connecting the START node with the FINISH node.
- The critical activities (activities with 0 slack) form at least one critical path in the network.
- A critical path is the longest path in the network.
- The sum of the completion times for the activities on the critical path is the minimal completion time of the project.

The Critical Path



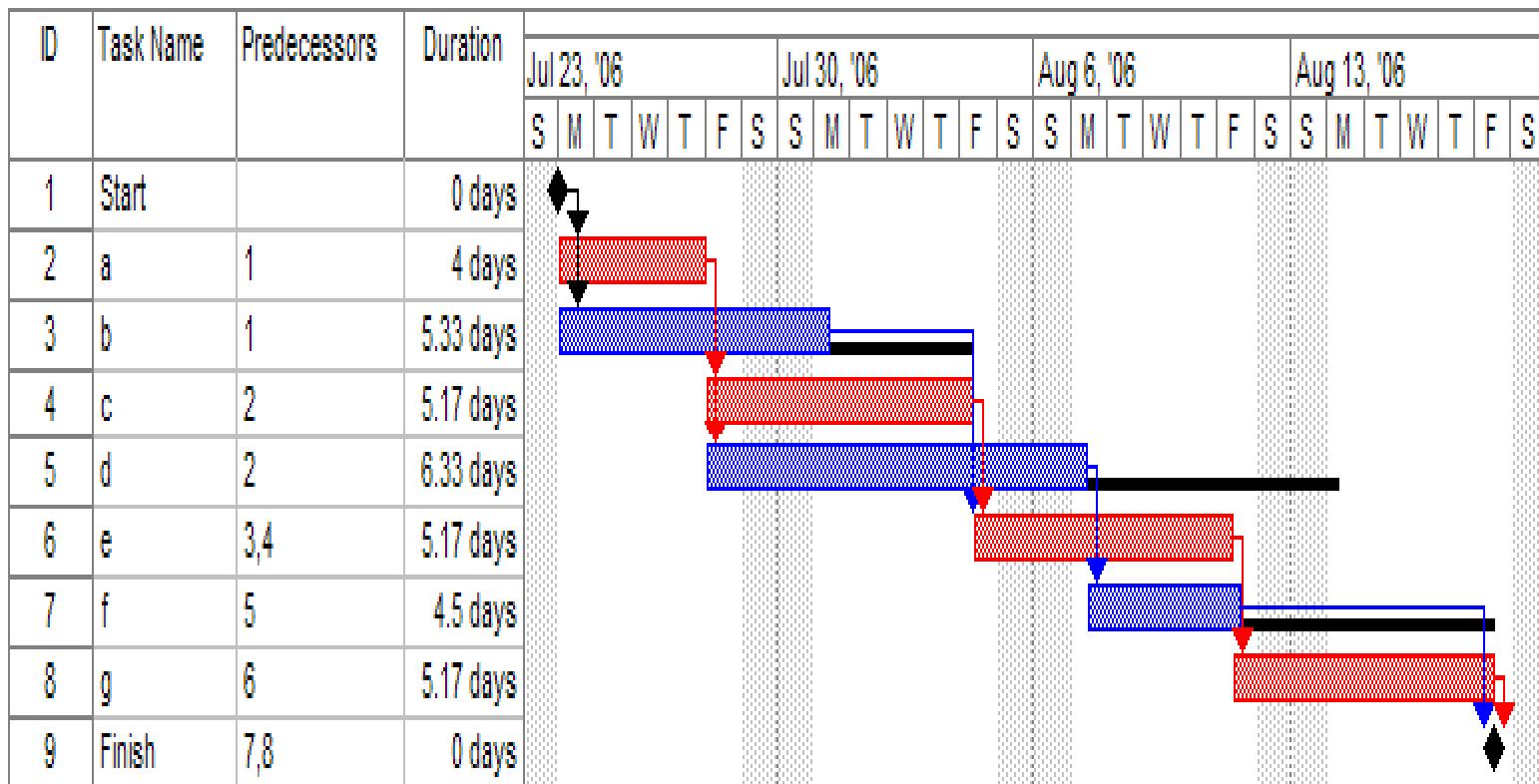
Gantt Chart

- Gantt charts illustrate the start and finish dates of the terminal elements and summary elements of a project
- Gantt charts also show the dependency relationships between activities.

Example - 1 on Gantt Chart

| ID | Task Name | Predecessors | Duration |
|----|-----------|--------------|-----------|
| 1 | Start | | 0 days |
| 2 | a | 1 | 4 days |
| 3 | b | 1 | 5.33 days |
| 4 | c | 2 | 5.17 days |
| 5 | d | 2 | 6.33 days |
| 6 | e | 3,4 | 5.17 days |
| 7 | f | 5 | 4.5 days |
| 8 | g | 6 | 5.17 days |
| 9 | Finish | 7,8 | 0 days |

Example - 1 on Gantt Chart with Critical Path

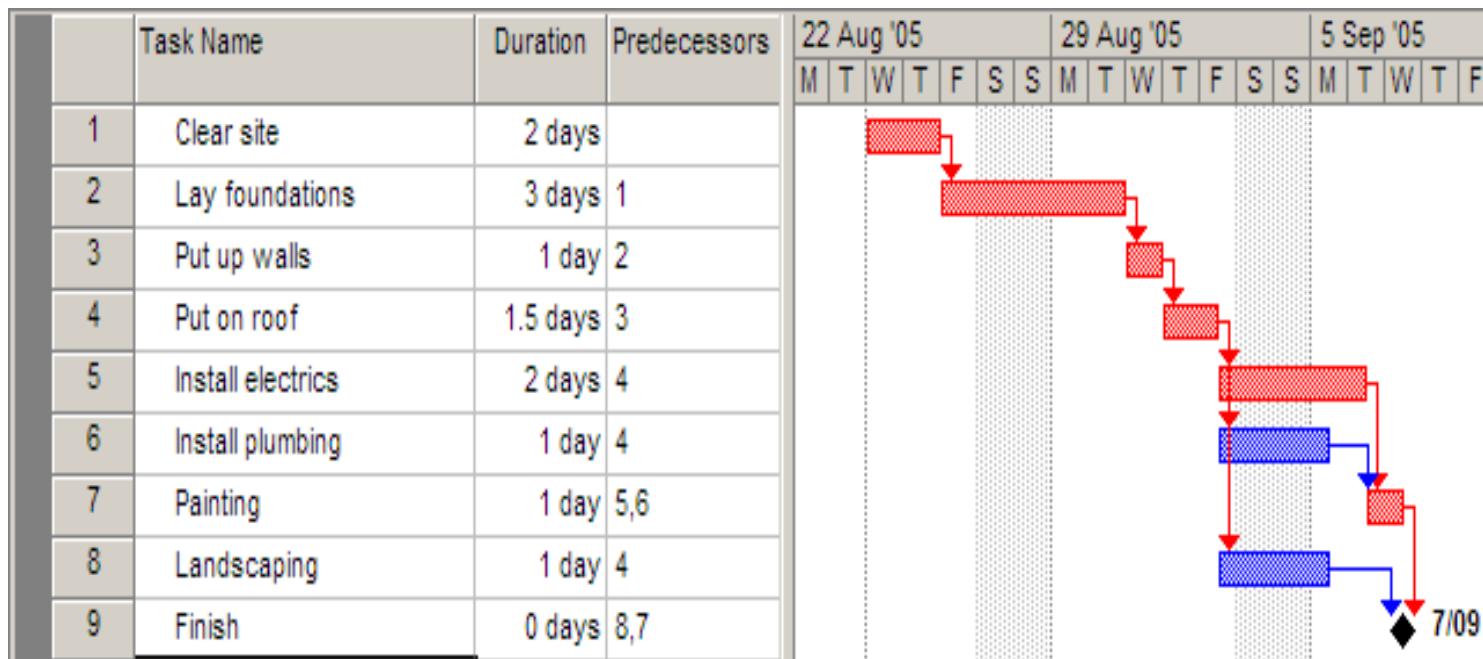


Example - 1 on Gantt Chart

| | Task Name | Duration | Predecessors |
|---|-------------------|----------|--------------|
| 1 | Clear site | 2 days | |
| 2 | Lay foundations | 3 days | 1 |
| 3 | Put up walls | 1 day | 2 |
| 4 | Put on roof | 1.5 days | 3 |
| 5 | Install electrics | 2 days | 4 |
| 6 | Install plumbing | 1 day | 4 |
| 7 | Painting | 1 day | 5,6 |
| 8 | Landscaping | 1 day | 4 |
| 9 | Finish | 0 days | 8,7 |

Example - 1 on Gantt Chart with Critical Path

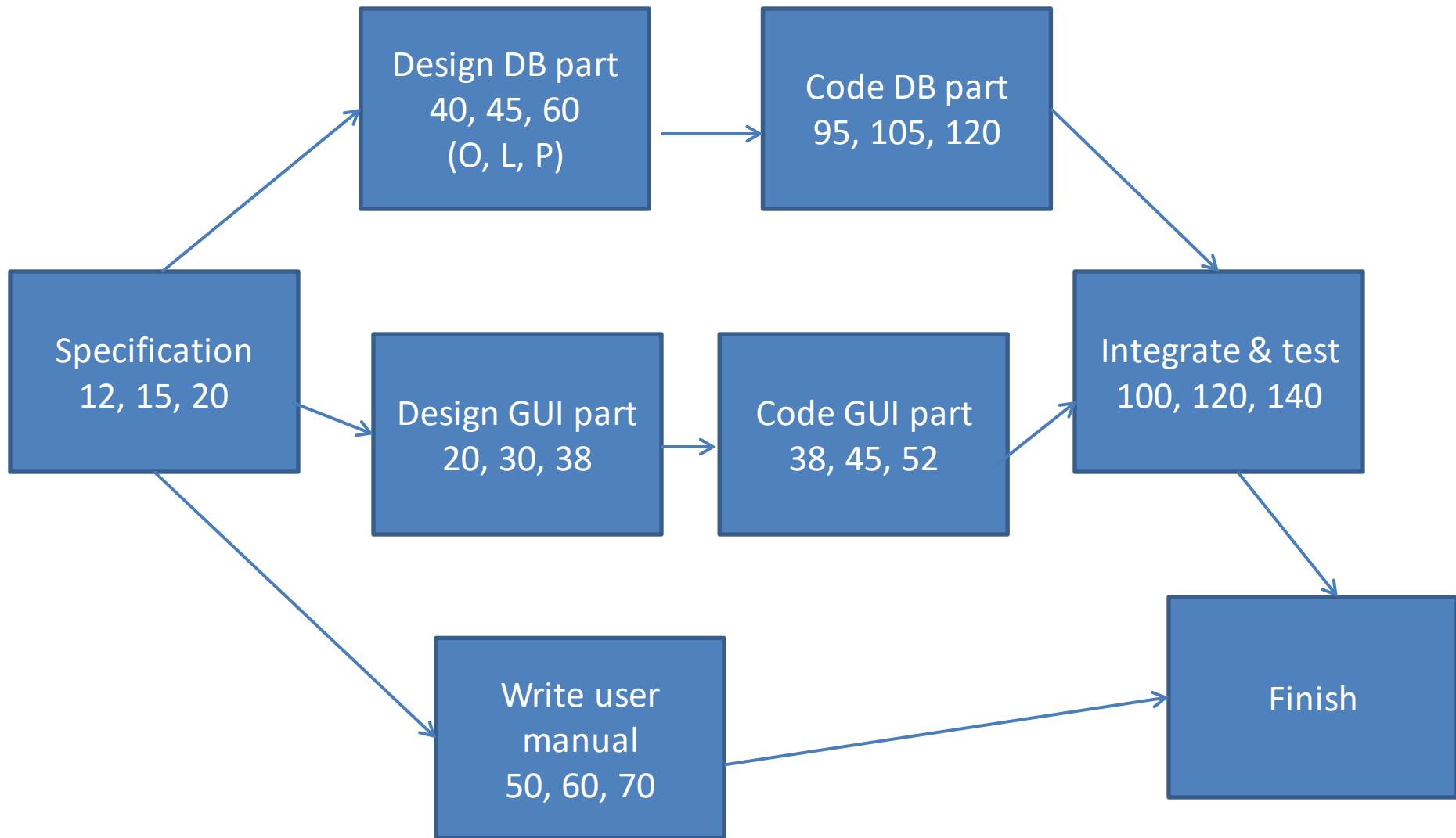
- The length of the critical path is the sum of the lengths of all critical tasks (the red tasks 1,2,3,4,5,7) which is $2+3+1+1.5+2+1 = 10.5$ days.



PERT Chart

- Program Evaluation and Review Technique (PERT)
- Consists of a network of boxes (activities) and arrows (task dependencies)
- Helps to identify parallel activities
- PERT estimation
 - Optimistic
 - Likely
 - Pessimistic

Example on PERT estimation



Gantt v/s PERT

- Gantt chart is represented as a bar graph, while PERT chart is represented as a flow chart.
- Gantt charts are limited to small projects and are not effective for projects with more than 30 activities.
- Generally Gantt is useful for resource planning, while PERT is used for monitoring the timely progress of activities
- Gantt chart do not efficiently represent the dependency of one task to another, while PERT charts can manage large projects that have numerous complex tasks a very high inter-task dependency

Additional Theory and Examples

CPM

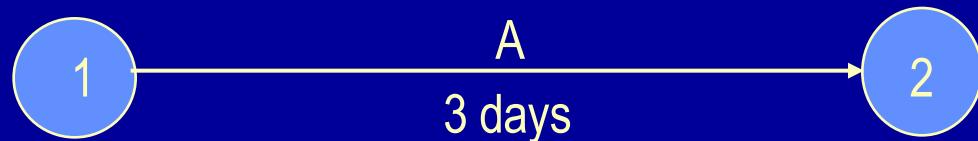
- Finding the critical path is a major part of controlling a project.
- The activities on the critical path represent tasks that will delay the entire project if they are delayed.
- Manager gain flexibility by identifying noncritical activities and replanning, rescheduling, and reallocating resources such as personnel and finances

Project Network

- A project network can be constructed to model the precedence of the activities.
- The arcs of the network represent the activities.
- The nodes of the network represent the start and the end of the activities.
- A critical path for the network is a path consisting of activities with zero slack. And it is always the longest path in the project network.

Drawing the project network (AOA)

- An activity carries the arrow symbol, \longrightarrow . This represent a task or subproject that uses time or resources
- A node (an event), denoted by a circle \bigcirc , marks the start and completion of an activity, which contain a number that helps to identify its location. For example activity A can be drawn as:



This means activity A starts at node 1 and finishes at node 2 and it will takes three days

Determining the Critical Path

- Step 1: Make a forward pass through the network as follows: For each activity i beginning at the Start node, compute:
 - Earliest Start Time (ES) = the maximum of the earliest finish times of all activities immediately preceding activity i . (This is 0 for an activity with no predecessors.). This is the earliest time an activity can begin without violation of immediate predecessor requirements.
 - Earliest Finish Time (EF) = (Earliest Start Time) + (Time to complete activity i). This represent the earliest time at which an activity can end.

The project completion time is the maximum of the Earliest Finish Times at the Finish node.

Determining the Critical Path

- Step 2: Make a backwards pass through the network as follows: Move sequentially backwards from the Finish node to the Start node. At a given node, j , consider all activities ending at node j . For each of these activities, (i,j) , compute:
 - Latest Finish Time (LF) = the minimum of the latest start times beginning at node j . (For node N , this is the project completion time.). This is the latest time an activity can end without delaying the entire project.
 - Latest Start Time (LS) = (Latest Finish Time) - (Time to complete activity (i,j)). This is the latest time an activity can begin without delaying the entire project.

Determining the Critical Path

- Step 3: Calculate the slack time for each activity by:

Slack = (Latest Start) - (Earliest Start), or
= (Latest Finish) - (Earliest Finish).

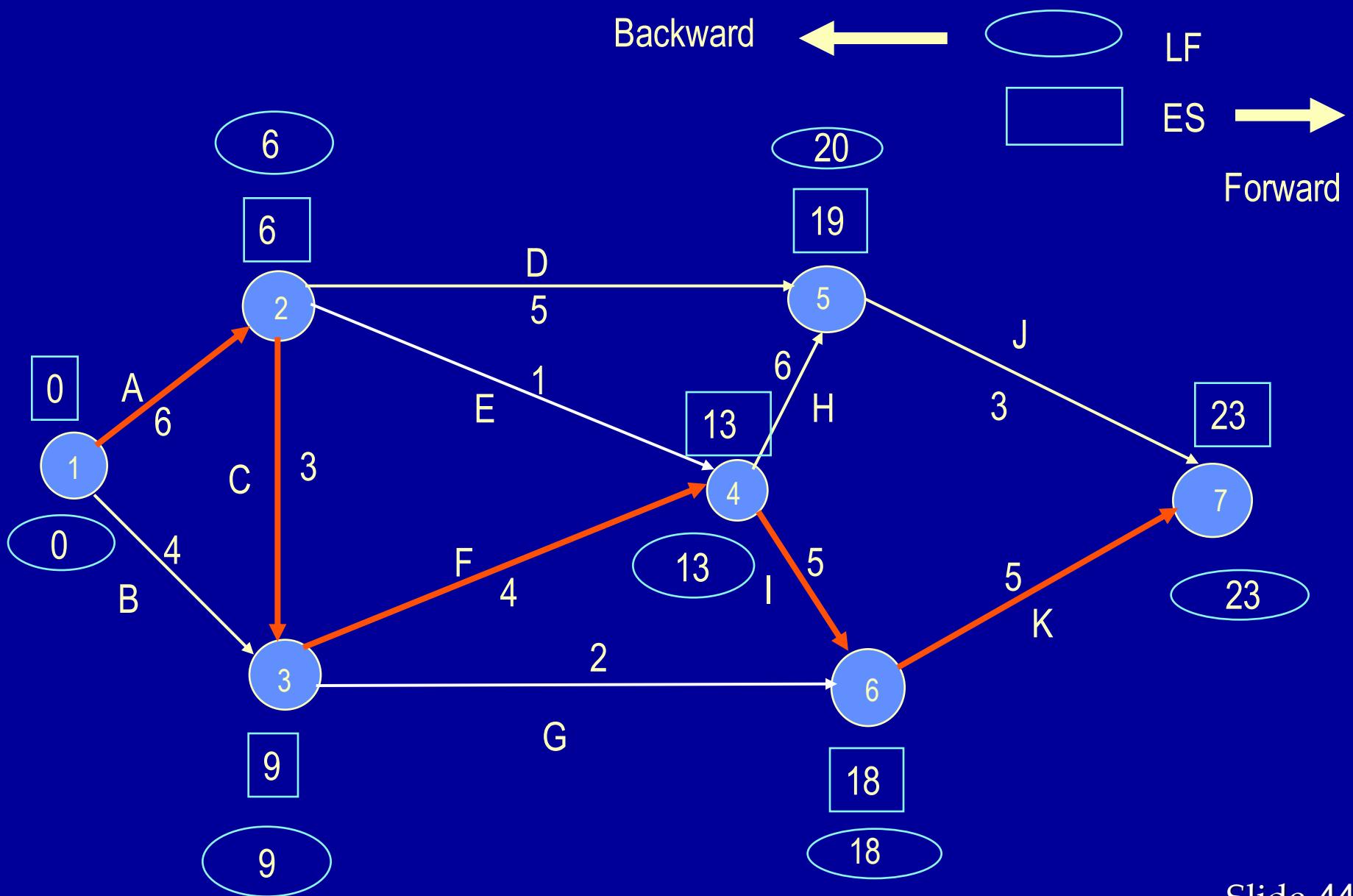
A critical path is a path of activities, from the Start node to the Finish node, with 0 slack times.

Example: ABC Associates

- Consider the following project:

| <u>Activity</u> | <u>Predecessor</u> | Immediate time (days) |
|-----------------|--------------------|--------------------------|
| A | -- | 6 |
| B | -- | 4 |
| C | A | 3 |
| D | A | 5 |
| E | A | 1 |
| F | B,C | 4 |
| G | B,C | 2 |
| H | E,F | 6 |
| I | E,F | 5 |
| J | D,H | 3 |
| K | G,I | 5 |

Example: network



Example: ABC Associates

■ Earliest/Latest Times

| | <u>Activity</u> | <u>time</u> | <u>ES</u> | EF | <u>LS</u> | <u>LF</u> | <u>Slack</u> |
|-------------------------------------|-----------------|-------------|-----------|----|-----------|-----------|--------------|
| | A | 6 | 0 | 6 | 0 | 6 | 0 *critical |
| $EF = ES + t$ | B | 4 | 0 | 4 | 5 | 9 | 5 |
| | C | 3 | 6 | 9 | 6 | 9 | 0 * |
| $LS = LF - t$ | D | 5 | 6 | 11 | 15 | 20 | 9 |
| $Where t is the$ $Activity time$ | E | 1 | 6 | 7 | 12 | 13 | 6 |
| | F | 4 | 9 | 13 | 9 | 13 | 0 * |
| | G | 2 | 9 | 11 | 16 | 18 | 7 |
| $Slack = LF - EF$ $= LS - ES$ | H | 6 | 13 | 19 | 14 | 20 | 1 |
| | I | 5 | 13 | 18 | 13 | 18 | 0 * |
| | J | 3 | 19 | 22 | 20 | 23 | 1 |
| | K | 5 | 18 | 23 | 18 | 23 | 0 * |

- The estimated project completion time is the Max EF at node 7 = 23.

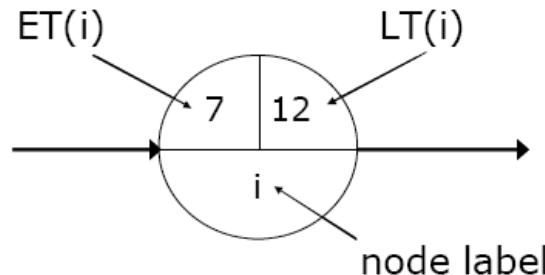
Additional Example

CPM – Critical Path Method

- It is determined by adding the times for the activities in each sequence.
- CPM determines the **total calendar time** required for the project.
- If activities outside the critical path speed up or slow down (within limits), the total project time does not change.
- The amount of time that a **non-critical** activity can be delayed without delaying the project is called **slack-time**.

CPM – Critical Path Method

- **ET** – Earliest node time for given activity duration and precedence relationships
- **LT** – Latest node time assuming no delays



- **ES** – Activity earliest start time
- **LS** – Activity latest start time
- **EF** – Activity earliest finishing time
- **LF** – Activity latest finishing time
- **Slack Time** – Maximum activity delay time

CPM – Critical Path Method

Step 1. Calculate ET for each node.

For each node i for which predecessors j are labelled with $ET(j)$, $ET(i)$ is given by:

$$ET(i) = \max_j [ET(j) + t(j,i)]$$

where $t(j,i)$ is the duration of task between nodes (j,i) .

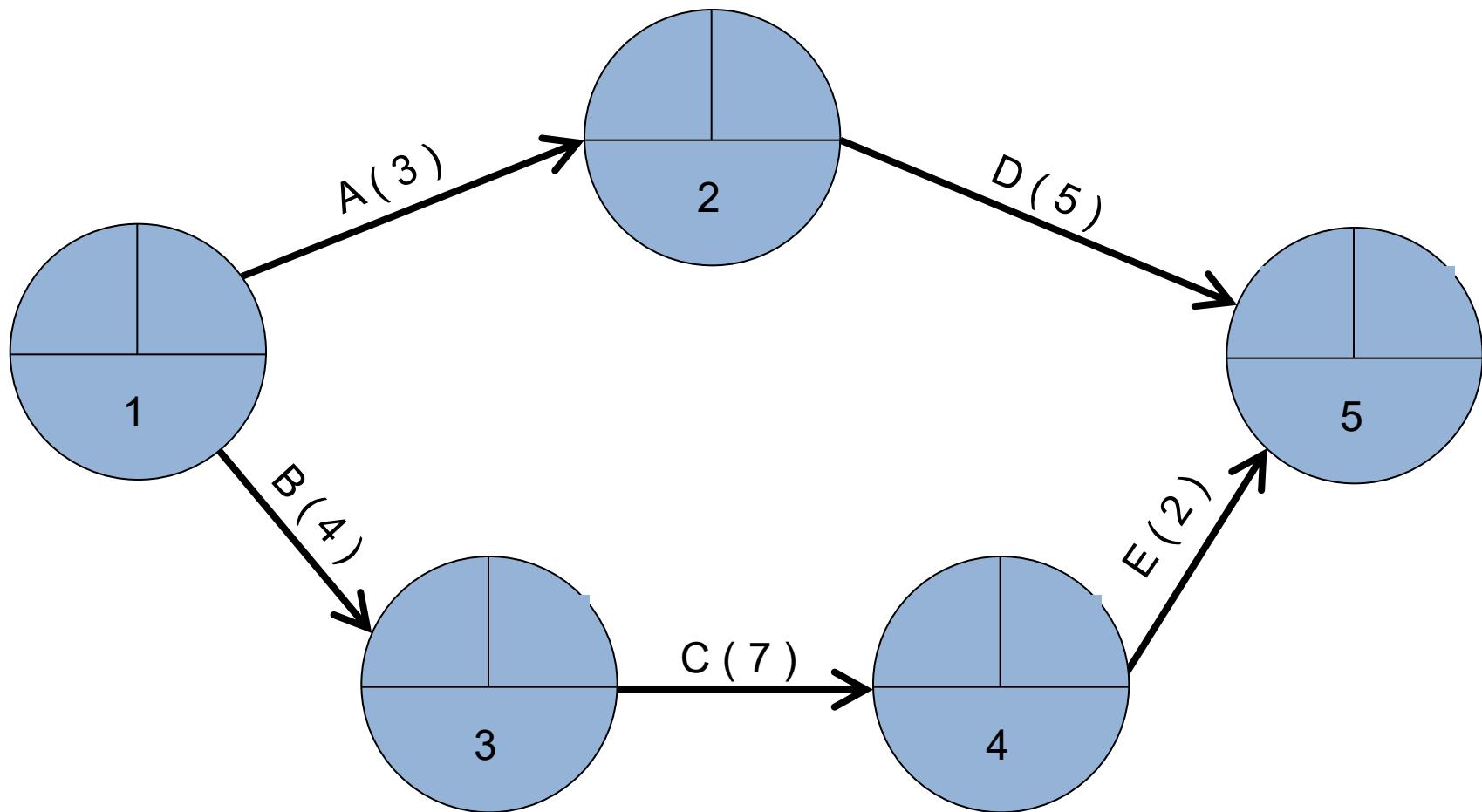
Step 2. Calculate LT for each node.

For each node i for which successors j are labelled with $LT(j)$, $LT(i)$ is given by:

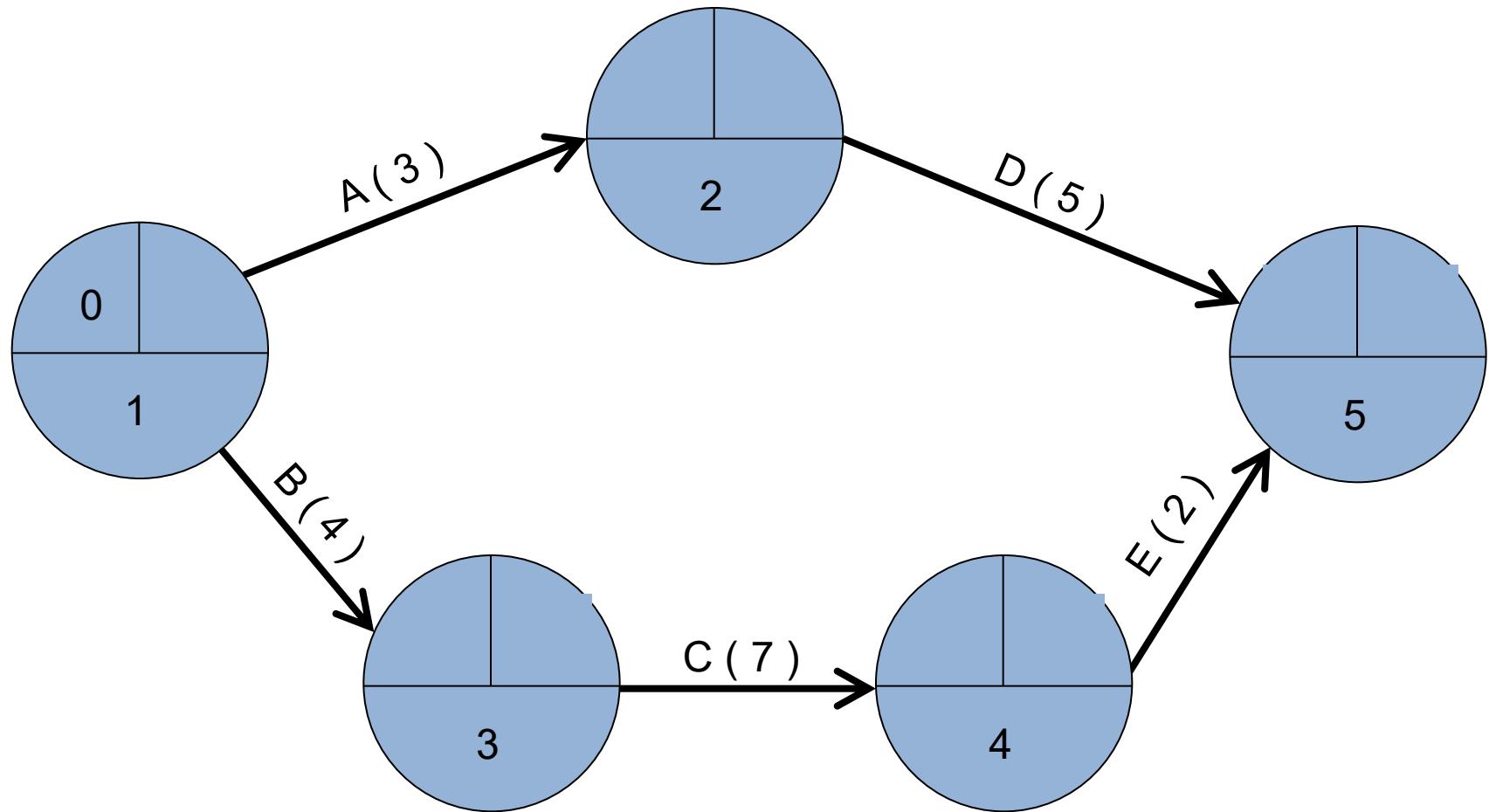
$$LT(i) = \min_j [LT(j) - t(i,j)]$$

where $t(j,i)$ is the duration of task between nodes (i,j) .

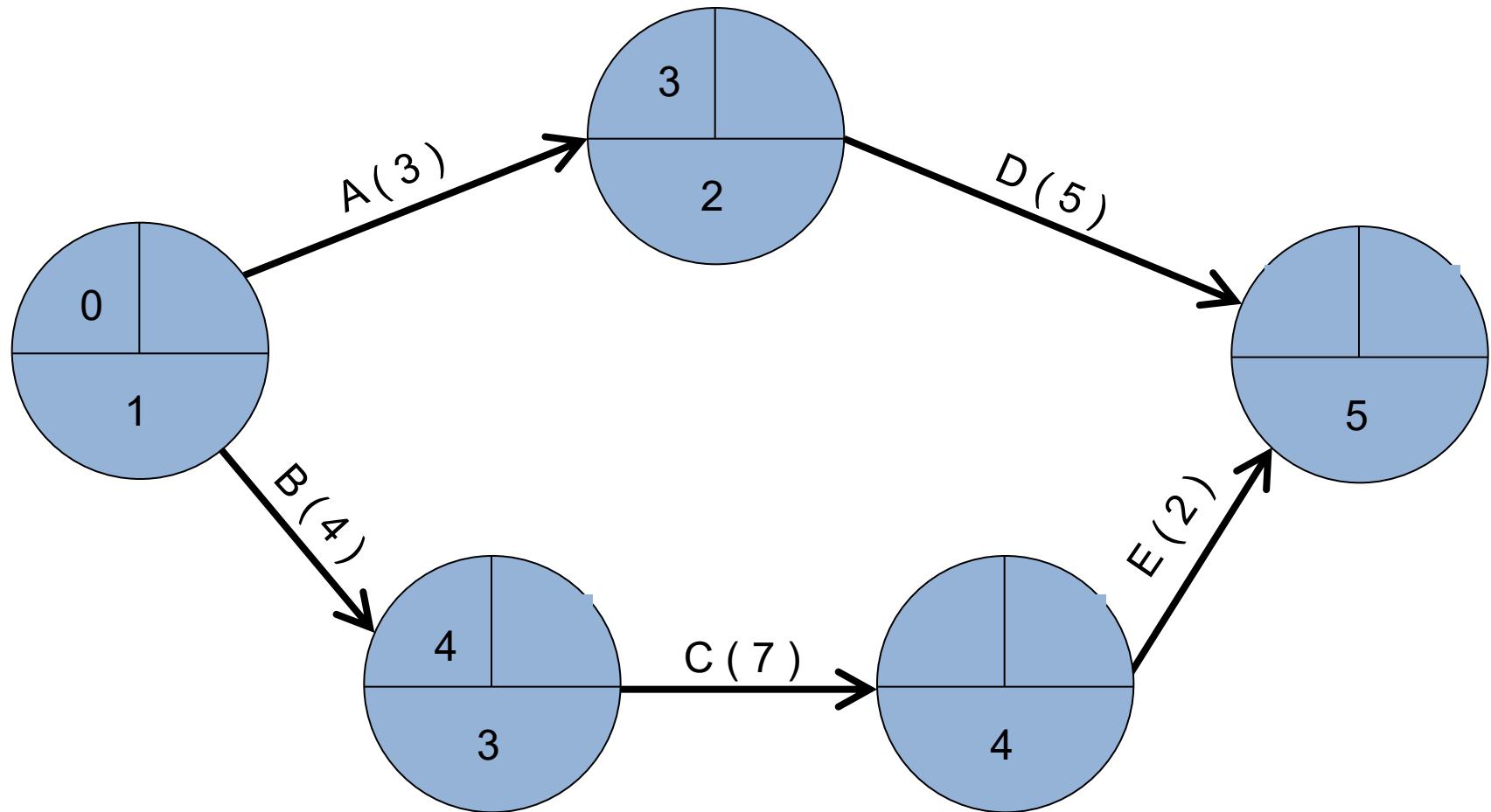
CPM – Critical Path Method



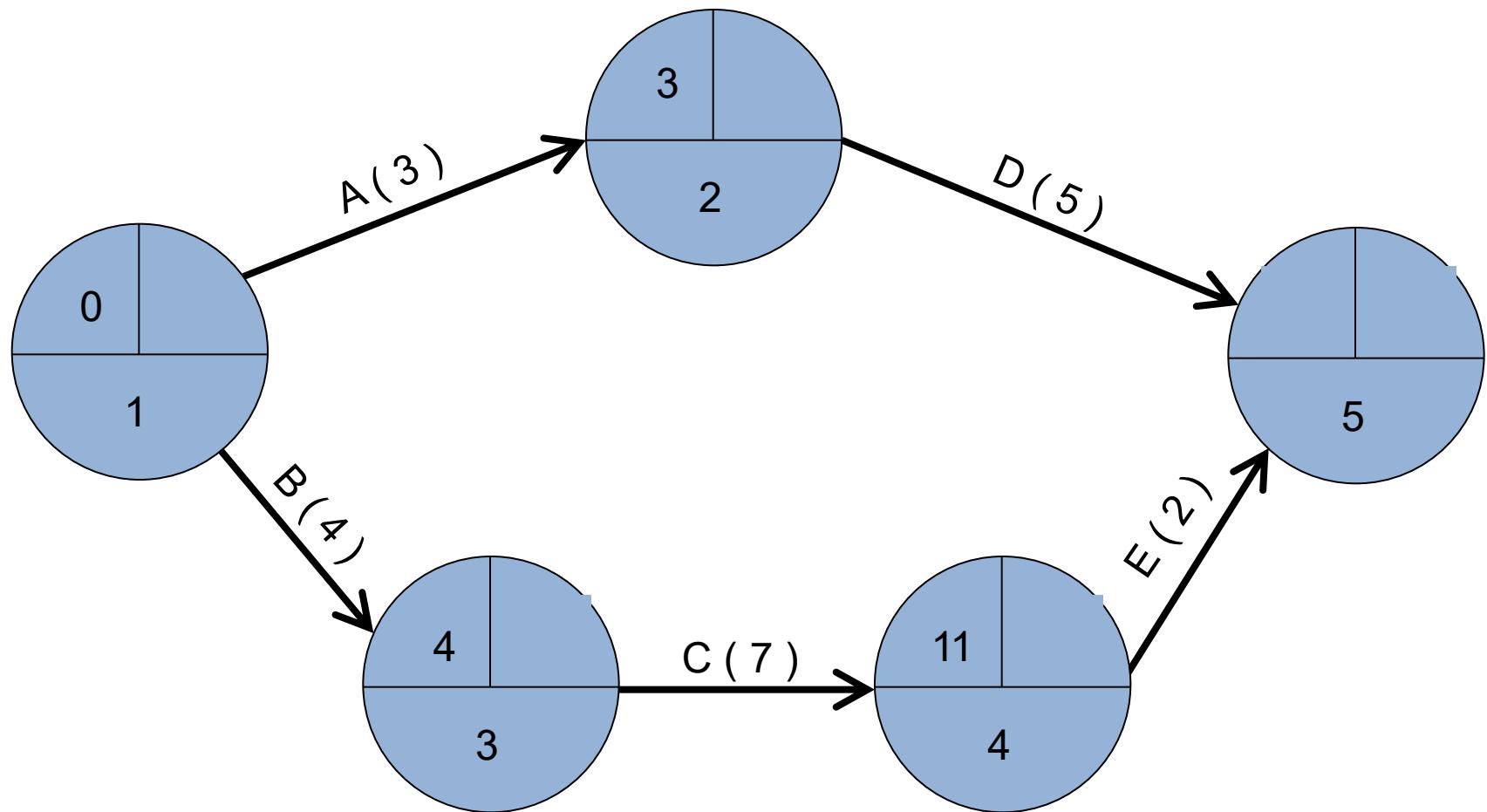
CPM – Critical Path Method



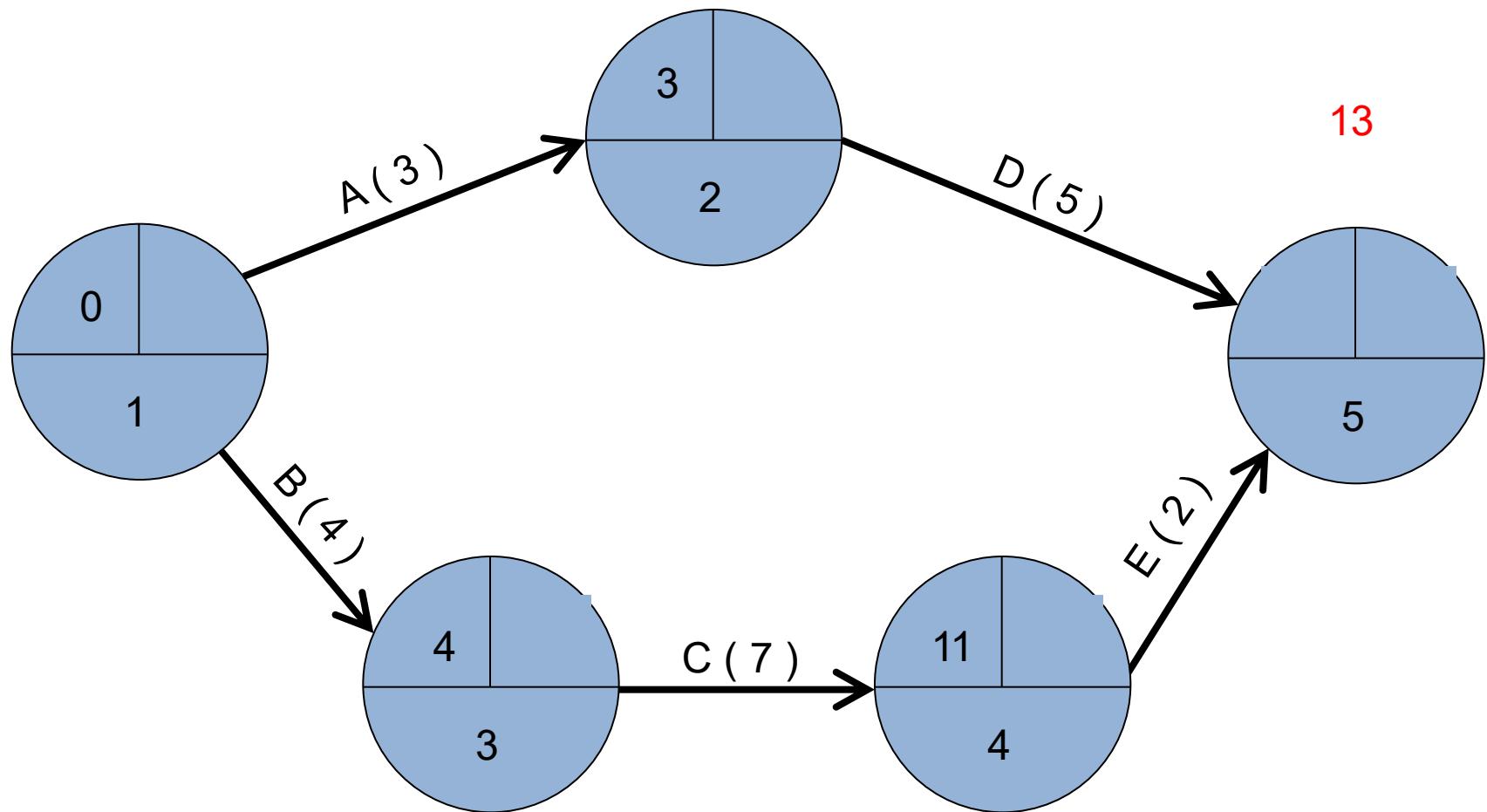
CPM – Critical Path Method



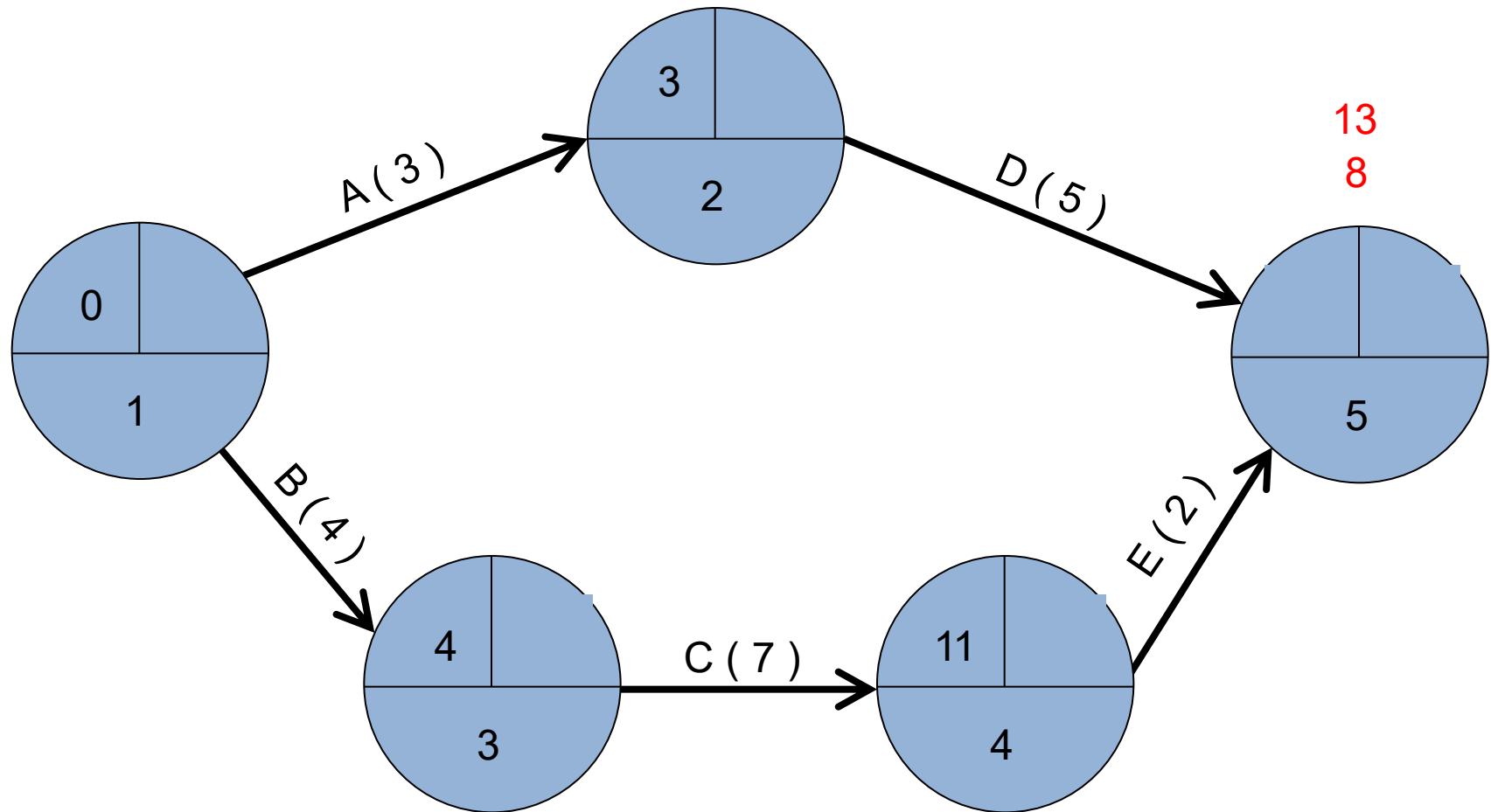
CPM – Critical Path Method



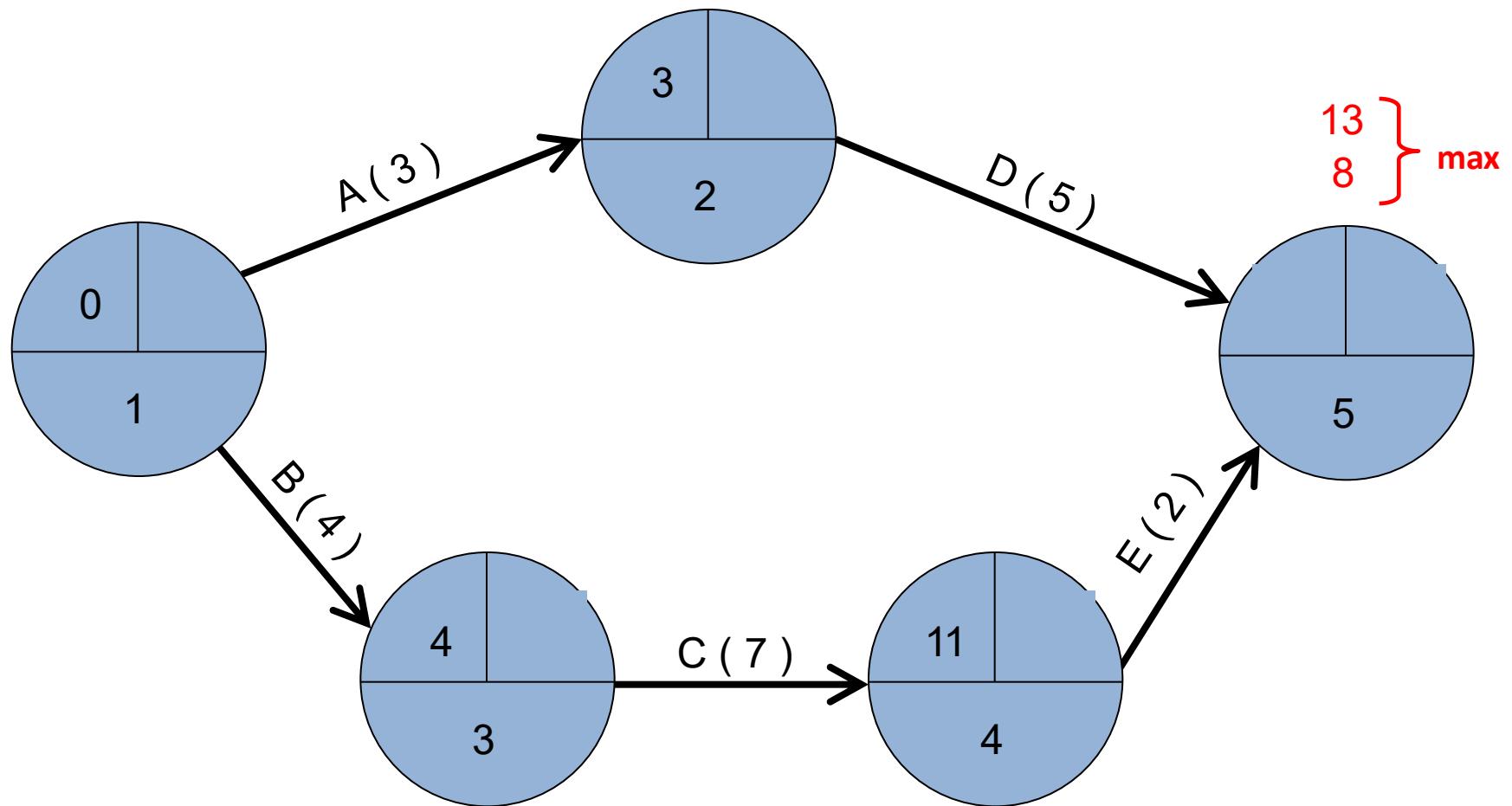
CPM – Critical Path Method



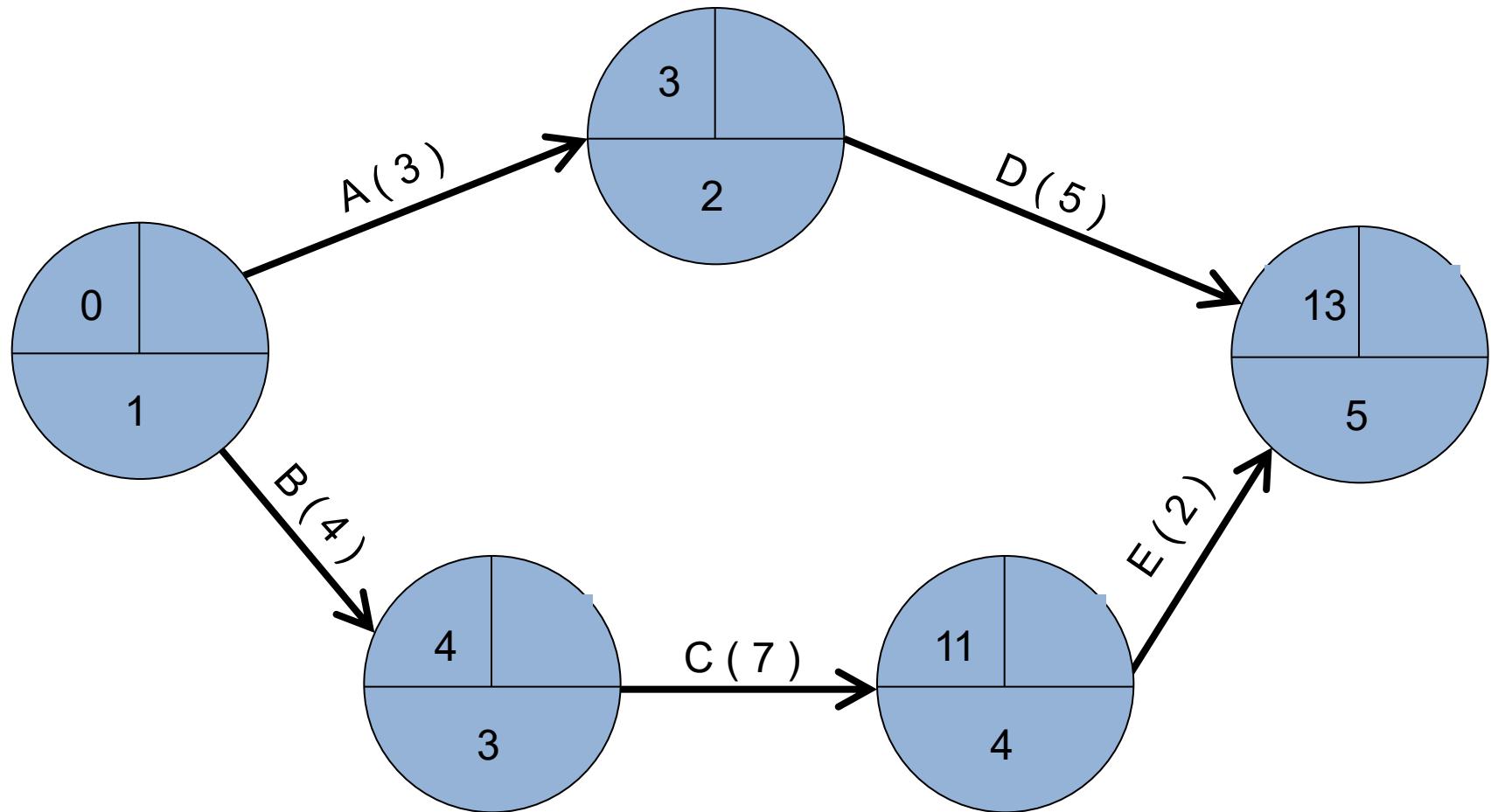
CPM – Critical Path Method



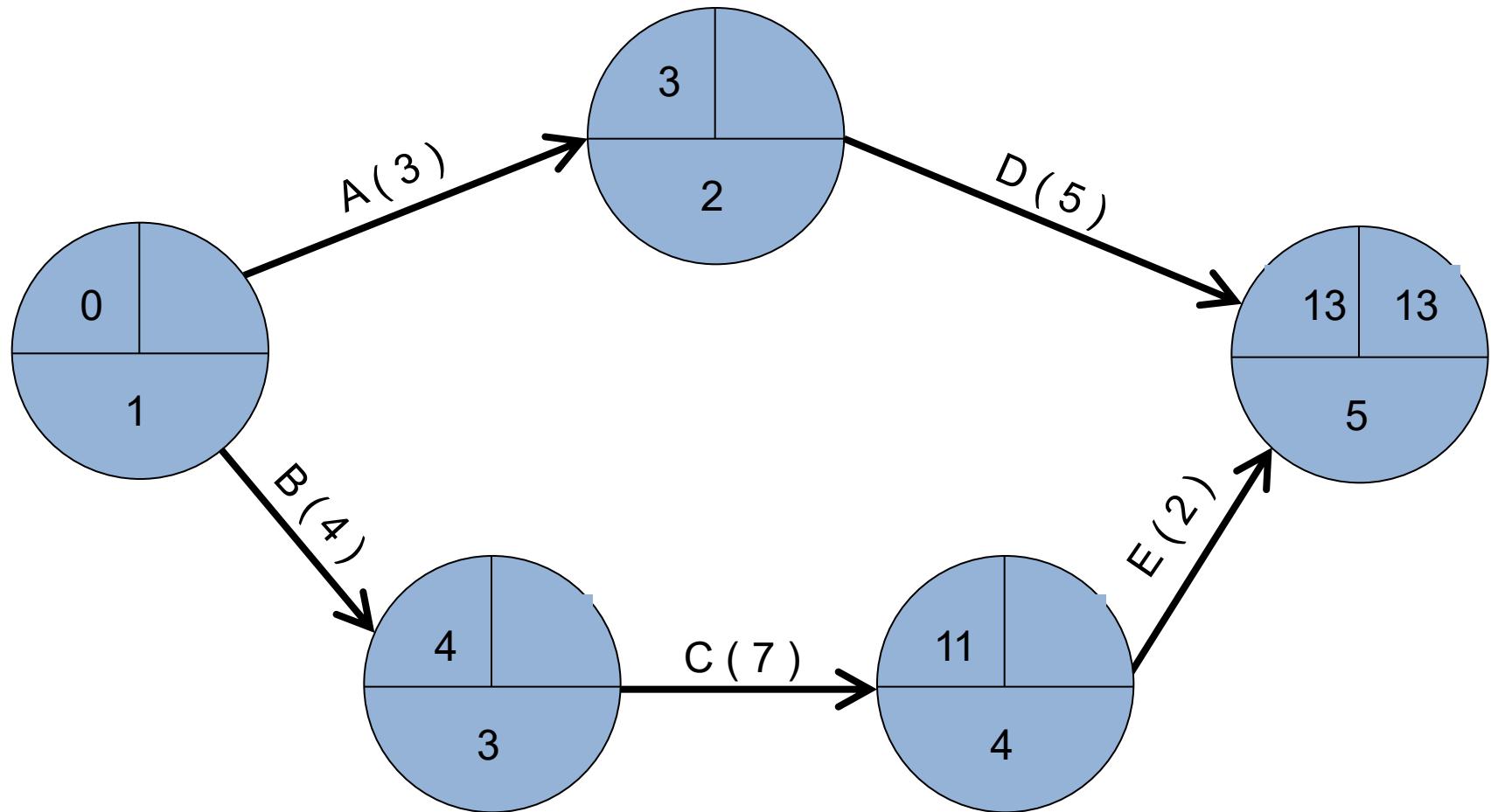
CPM – Critical Path Method



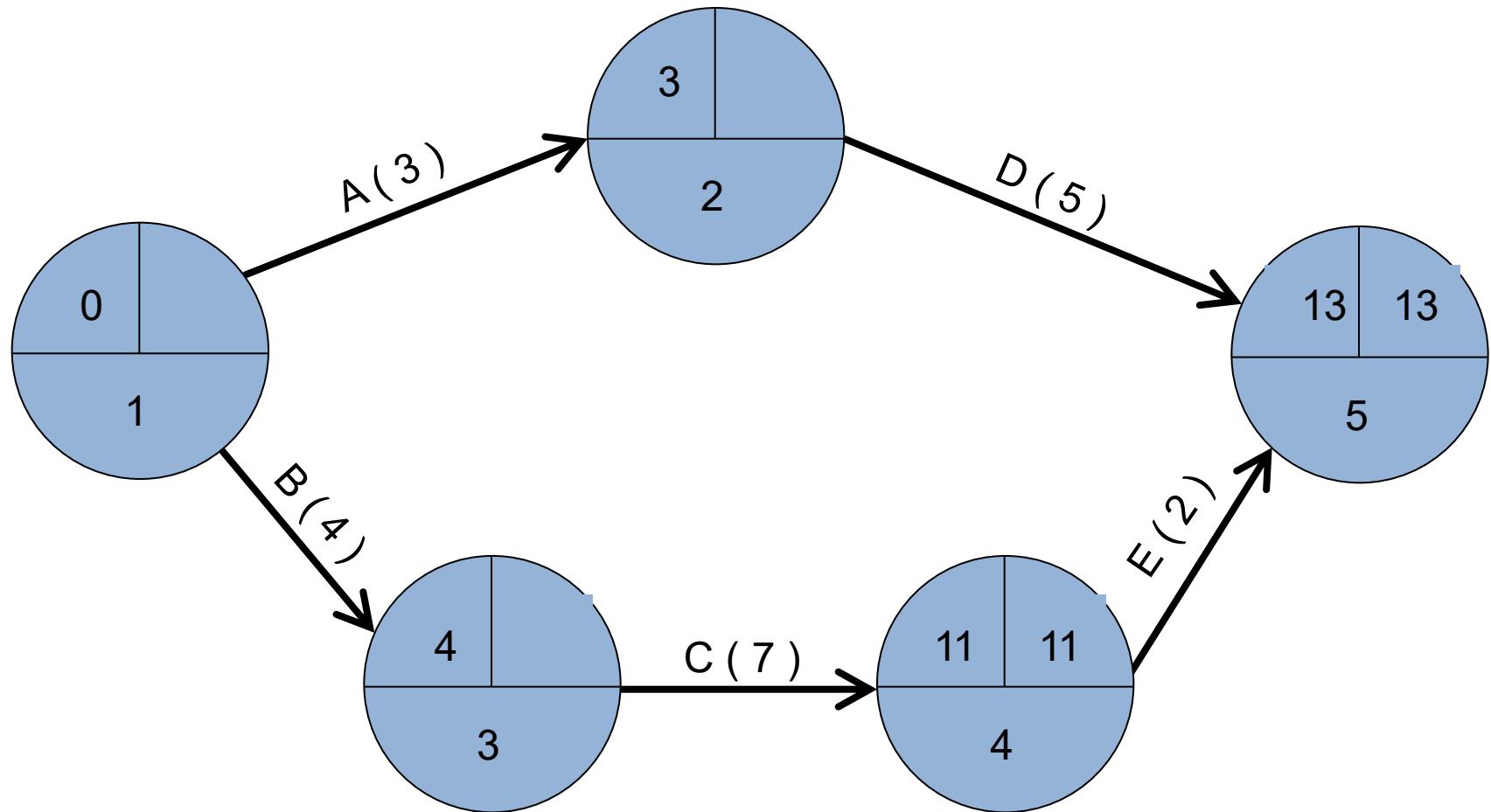
CPM – Critical Path Method



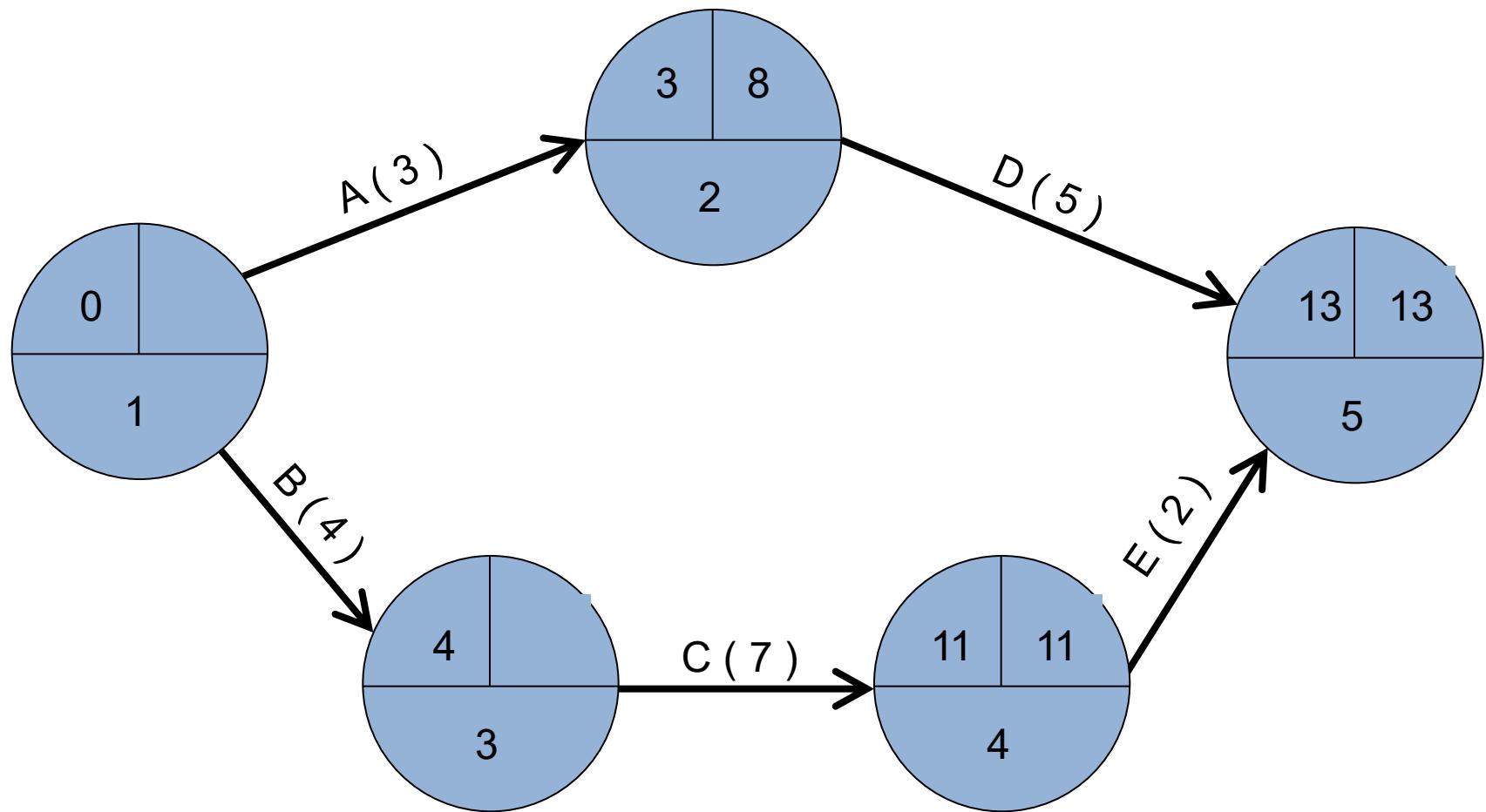
CPM – Critical Path Method



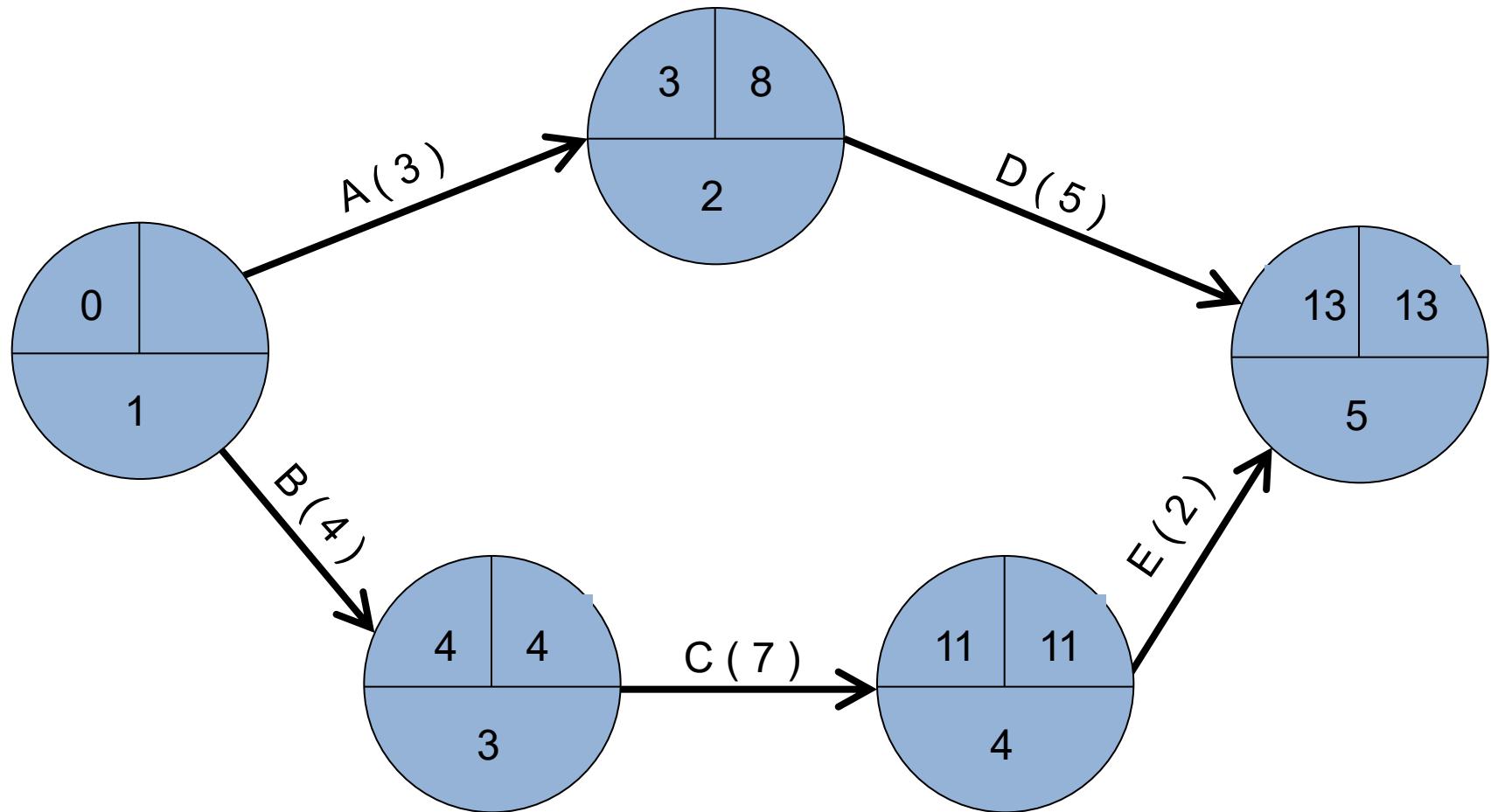
CPM – Critical Path Method



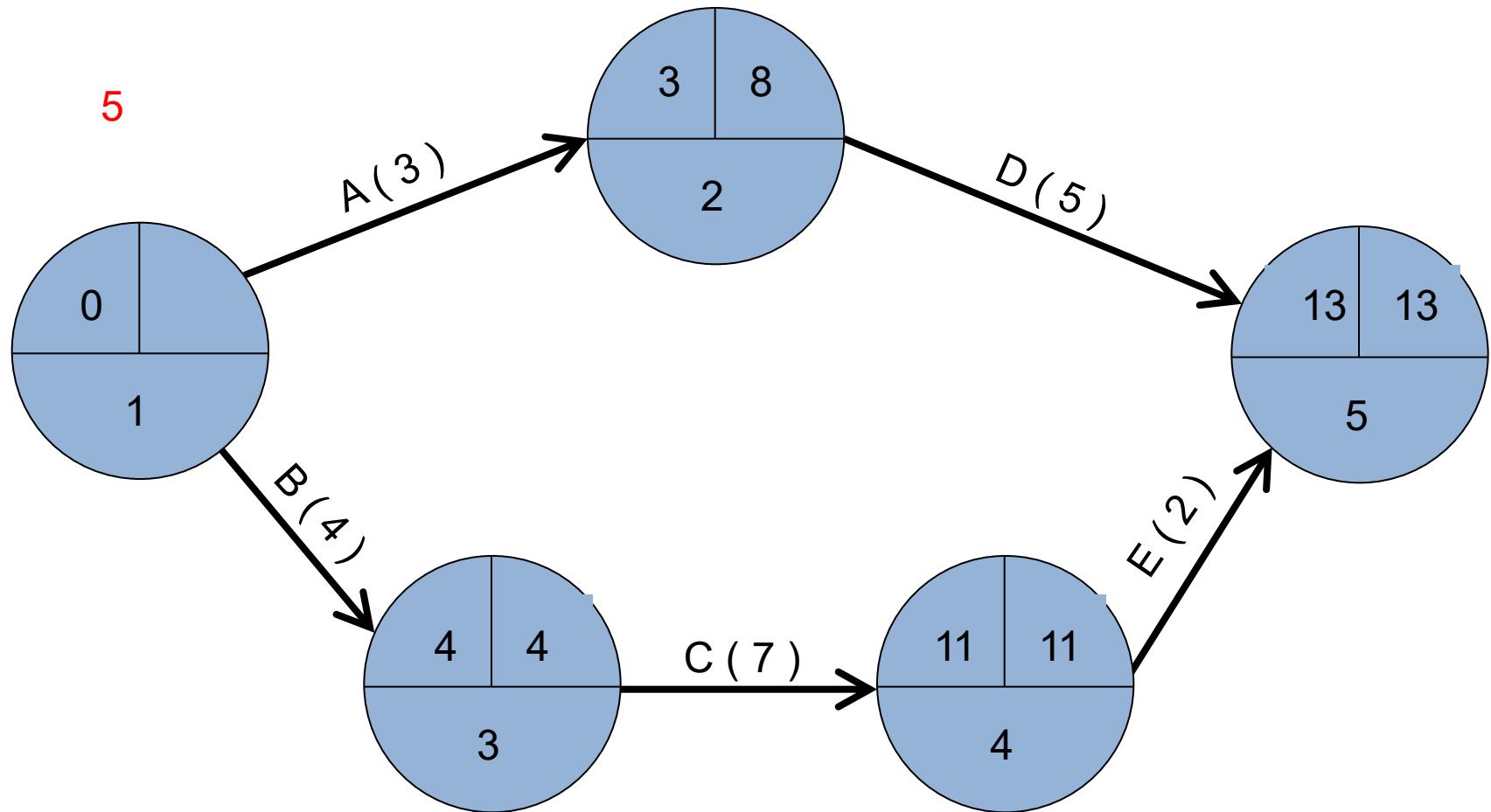
CPM – Critical Path Method



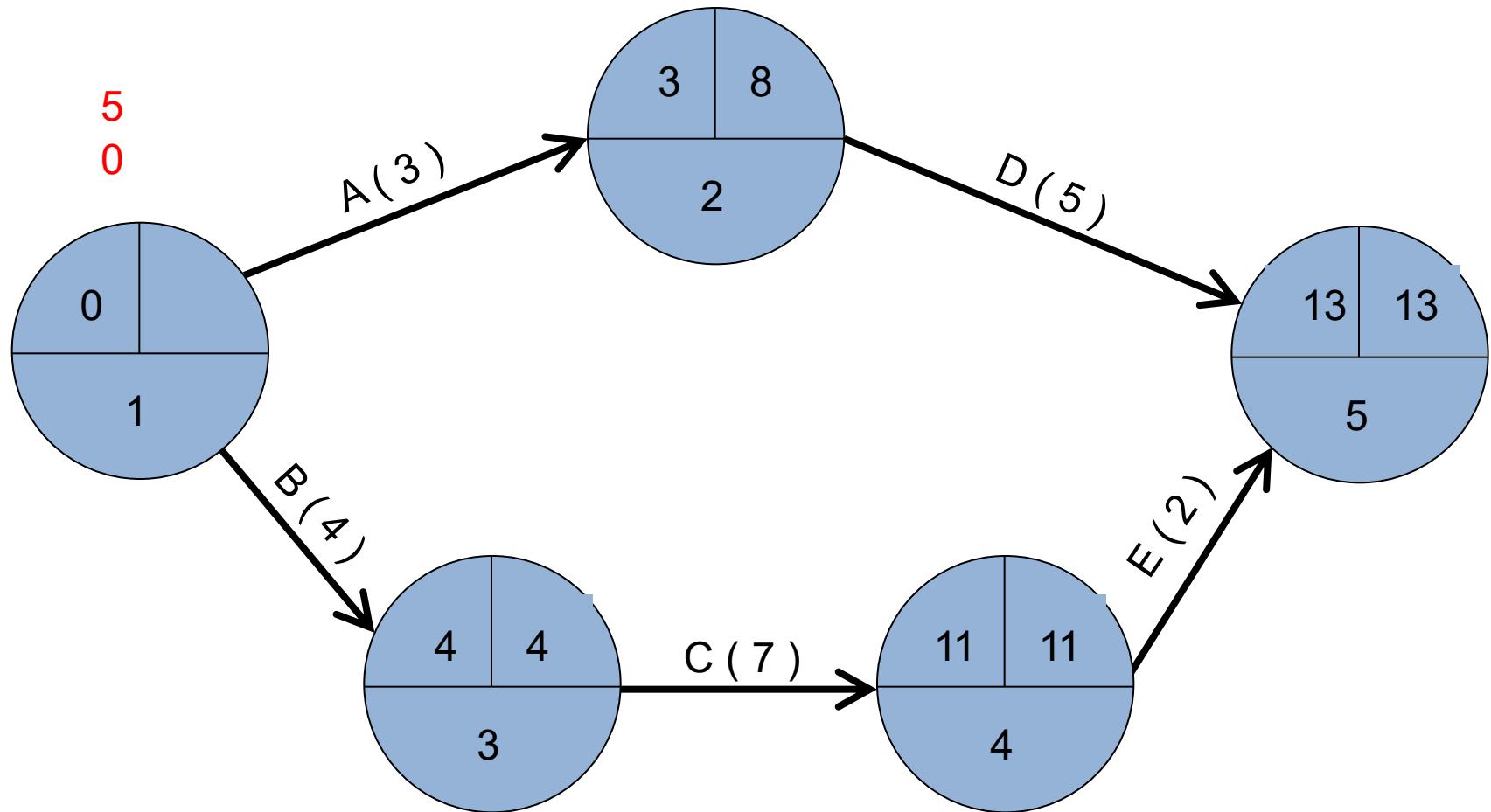
CPM – Critical Path Method



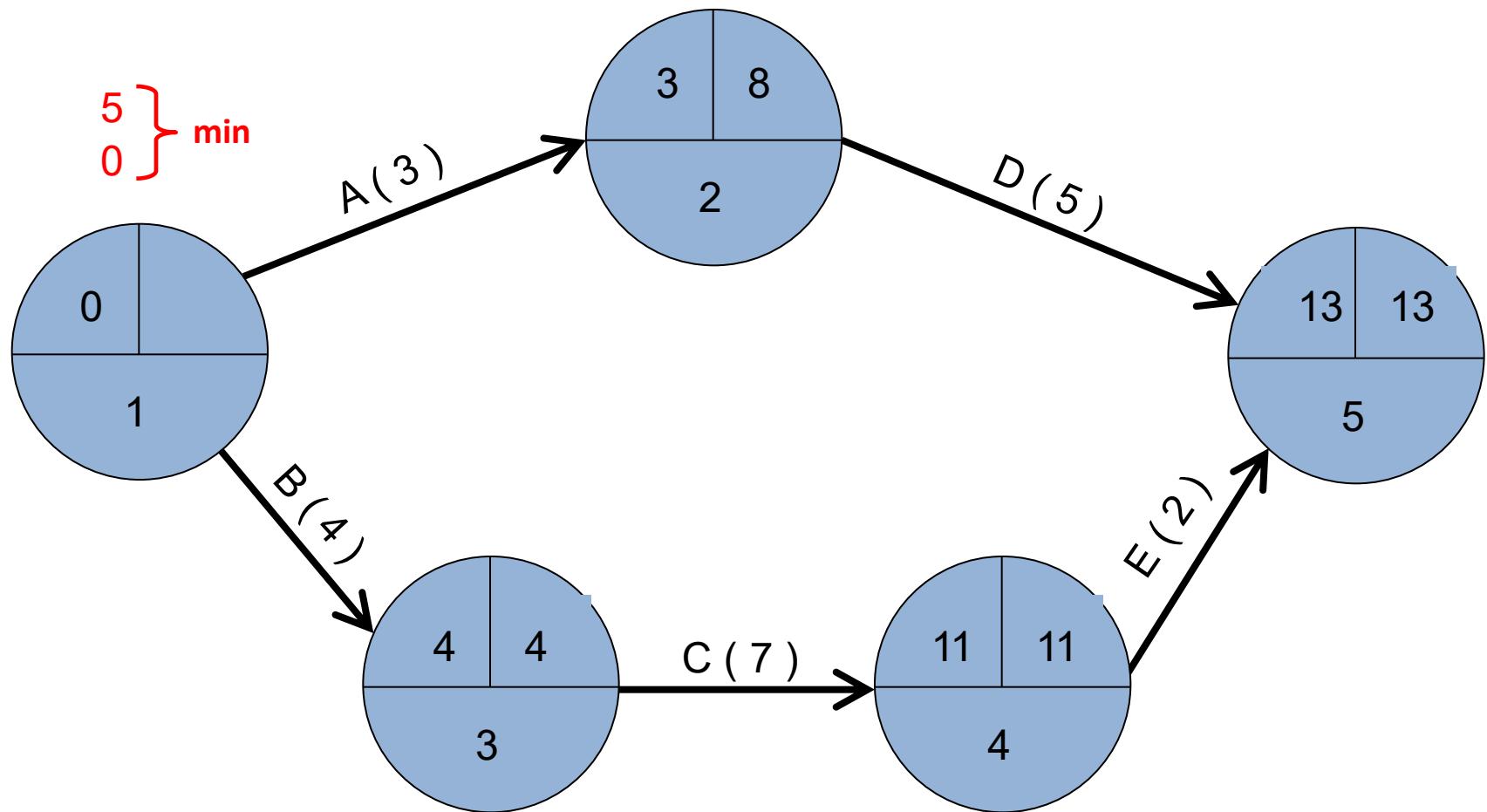
CPM – Critical Path Method



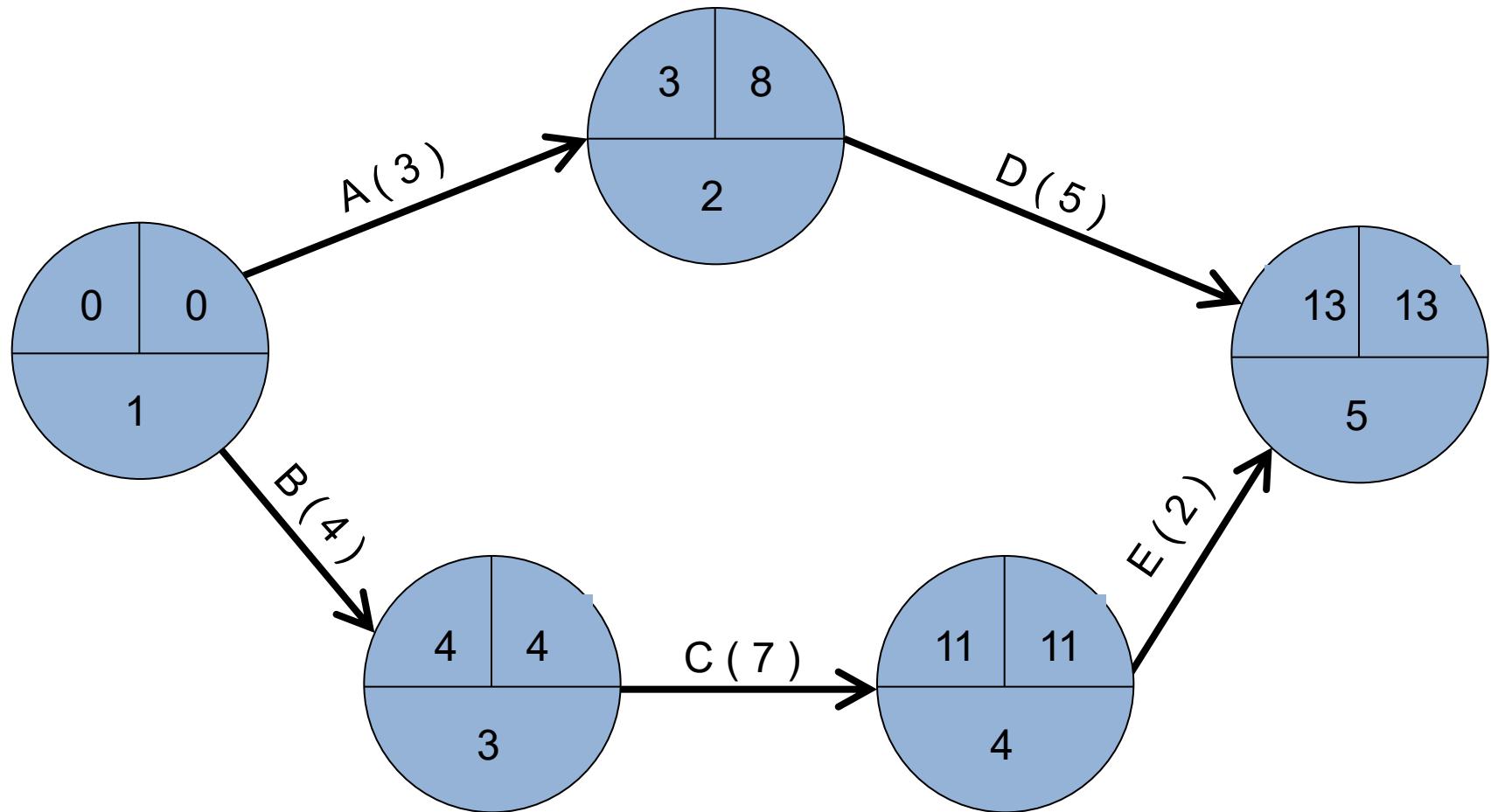
CPM – Critical Path Method



CPM – Critical Path Method



CPM – Critical Path Method

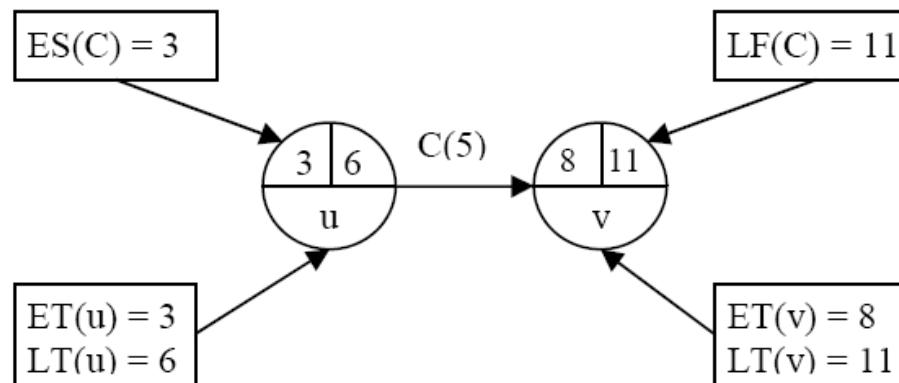


CPM – Critical Path Method

- An activity with zero slack time is a **critical activity** and *cannot be delayed* without causing a delay in the whole project.

$$\begin{aligned} EF(C) &= ES(C) + t(C) \\ &= 3 + 5 = 8 \end{aligned}$$

$$\begin{aligned} LS(C) &= LF(C) - t(C) \\ &= 11 - 5 = 6 \end{aligned}$$



$$\begin{aligned} \text{Slack Time (C)} &= LS(C) - ES(C) = 6 - 3 = 3 \\ &= LF(C) - EF(C) = 11 - 8 = 3 \end{aligned}$$

CPM – Critical Path Method

Step 3. Calculate processing times for each activity.

For each activity X with start node i and end node j:

$$ES(X) = ET(i)$$

$$EF(X) = ES(X) + t(X)$$

$$LF(X) = LT(j)$$

$$LS(X) = LF(X) - t(X)$$

$$\text{Slack Time (X)} = LS(X) - ES(X) = LF(X) - EF(X)$$

Where $t(X)$ is the duration of activity X.

An activity with zero slack time is a critical activity and cannot be delayed without causing a delay in the whole project.

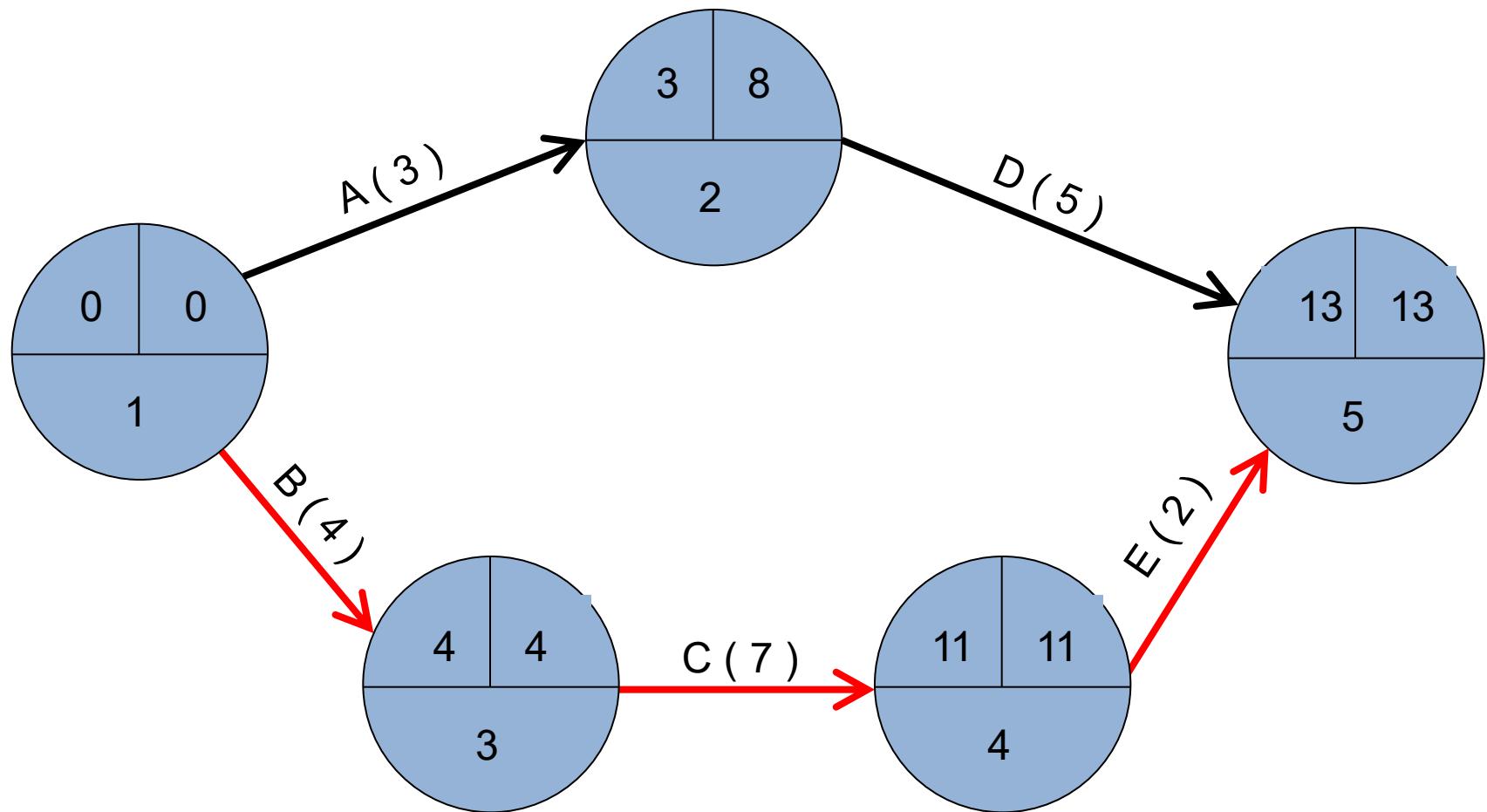
CPM – Critical Path Method

Step 3. Calculate processing times for each activity.

| Task | Duration | ES | EF | LS | LF | Slack | Critical Task |
|------|----------|----|----|----|----|-------|---------------|
| A | 3 | 0 | 3 | 5 | 8 | 5 | No |
| B | 4 | 0 | 4 | 0 | 4 | 0 | Yes |
| C | 7 | 4 | 11 | 4 | 11 | 0 | Yes |
| D | 5 | 3 | 8 | 8 | 13 | 5 | No |
| E | 2 | 11 | 13 | 11 | 13 | 0 | Yes |

Reading: (Kendall&Kendall, chapter 3), (Dennis &Wixom, chapter 3)

CPM – Critical Path Method



RISK MANAGEMENT

- INTRODUCTION
- RISK IDENTIFICATION
- RISK PROJECTION (ESTIMATION)
- RISK MITIGATION, MONITORING, AND MANAGEMENT

INTRODUCTION

DEFINITION OF RISK

- A risk is a potential problem – it might happen and it might not
- Risk Management: Series of steps that help a software team to understand and manage uncertainty.
- Conceptual definition of risk
 - Risk concerns future happenings
 - Risk involves change in mind, opinion, actions, places, etc.
 - Risk involves choice and the uncertainty that choice entails
- 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 – APPROACH #1

- 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

RISK CATEGORIZATION – APPROACH #1 (CONTINUED)

- 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

RISK CATEGORIZATION – APPROACH #2

- 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

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
 - 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

- 1) Identify possible risks; recognize what can go wrong
- 2) Analyze each risk to estimate the probability that it will occur and the impact (i.e., damage) that it will do if it does occur
- 3) Rank the risks by probability and impact
 - Impact may be negligible, marginal, critical, and catastrophic
- 4) Develop a contingency plan to manage those risks having high probability and high impact

RISK IDENTIFICATION

BACKGROUND

- 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 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?"

RISKITEM 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

KNOWN AND PREDICTABLE RISK CATEGORIES

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

QUESTIONNAIRE ON PROJECT RISK

(Questions are ordered by their relative importance to project success)

- 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 fully in the definition of requirements?
- 5) Do end-users have realistic expectations?
- 6) Is the project scope stable?

QUESTIONNAIRE ON PROJECT RISK (CONTINUED)

- 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 customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

RISK COMPONENTS AND DRIVERS

- The project manager identifies the risk drivers 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)

BACKGROUND

- 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

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

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 |
|--------------|---------------|-------------|--------------|------|
| | | | | |
| | | | | |
| | | | | |

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

| <u>Risks</u> | <u>Category</u> | <u>Probability</u> | <u>Impact</u> | <u>RMMM</u> |
|--|-----------------|--------------------|---------------|-------------|
| <u>Estimated size of project in LOC or FP</u> | <u>PS</u> | <u>80%</u> | <u>2</u> | <u>**</u> |
| <u>Lack of needed specialization increases defects and reworks</u> | <u>ST</u> | <u>50%</u> | <u>2</u> | <u>**</u> |
| <u>Unfamiliar areas of the product take more time than expected to design and implement</u> | <u>DE</u> | <u>50%</u> | <u>2</u> | <u>**</u> |
| <u>Does the environment make use of a database</u> | <u>DE</u> | <u>35%</u> | <u>3</u> | <u>-</u> |
| <u>Components developed separately cannot be integrated easily, requiring redesign</u> | <u>DE</u> | <u>25%</u> | <u>3</u> | <u>-</u> |
| <u>Development of the wrong software functions requires redesign and implementation</u> | <u>DE</u> | <u>25%</u> | <u>3</u> | <u>-</u> |
| <u>Development of extra software functions that are not needed</u> | <u>DE</u> | <u>20%</u> | <u>3</u> | <u>-</u> |
| <u>Strict requirements for compatibility with existing system require more testing, design, and implementation than expected</u> | <u>DE</u> | <u>20%</u> | <u>3</u> | <u>-</u> |
| <u>Operation in unfamiliar software environment causes unforeseen problems</u> | <u>EV</u> | <u>25%</u> | <u>4</u> | <u>-</u> |
| <u>Team members do not work well together</u> | <u>ST</u> | <u>20%</u> | <u>4</u> | <u>21</u> |
| <u>Key personnel are available only part-time</u> | <u>ST</u> | <u>20%</u> | <u>4</u> | <u>-</u> |

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 $RE = P \times C$
 - 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
 - $RE = .80 \times \$25,000 = \$20,000$

RISK MITIGATION, MONITORING, AND MANAGEMENT

BACKGROUND

- 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

BACKGROUND (CONTINUED)

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

BACKGROUND (CONTINUED)

- 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

- Risks are also associated with software failures that occur in the field after the development project has ended.
- Computers control many mission critical applications today (weapons systems, flight control, industrial processes, etc.).
- 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 information sheet

Risk ID: P02-4-32

Date: 5/9/02

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 7/1/02

Current status:

5/12/02: Mitigation steps initiated.

Originator: D. Gagne

Assigned: B. Laster

RISK MANAGEMENT

Risk for our project is: planned budget increase
For risk management using RMMM plan.

| Risk Information Sheet | | | |
|--|--|-------------------------|-----------------------|
| Project Name: Library Management System | | | |
| Risk ID: 007 | Date: 13/03/2015 | Probability: 67% | Impact: Medium |
| Origin: Viraj Kelkar | Assigned to: Sanket Kudalkar | | |
| Description: More bugs are arising in the system. System is unstable. Need to be updated as the technique gets modern. | | | |
| Refinement/ Context: System is unstable due to the unexpected developer. Because of such problem the solution to the critical problem is harder to find. | | | |
| Mitigation / Monitoring Bugs must be found and simultaneously solutions to those bugs should also be found. Testing of each module should be done. | | | |
| Contingency plan and trigger Experts should be hire in case of emergency or in case of issue. As computed the risk exposure to be Rs.45,000. Allocate this amount within the project management cost. Develop revised schedule and allocate more skilled staff accordingly. | | | |
| Status / date Mitigation step initiated | | | |
| Approval Tejas Kondhalkar | Closing date 13/3/2015 | | |

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

SUMMARY

- Whenever much is riding on a software project, common sense dictates risk analysis
 - Yet, most project managers do it informally and superficially, if at all
- However, the time spent in risk management results in
 - Less upheaval during the project
 - A greater ability to track and control a project
 - The confidence that comes with planning for problems before they occur
- Risk management can absorb a significant amount of the project planning effort...but the effort is worth it



Project Management....



Work Smart !!!

Project...

A collection of linked activities, carried out in an organized manner, with a clearly defined START POINT and END POINT to achieve some specific results desired to satisfy the needs of the organization at the current time.

Project Management

- A dynamic process that utilizes the appropriate resources of the organization in a controlled and structured manner, to achieve some clearly defined objectives identified as needs.
- It is always conducted within a defined set of constraints

What does Project Management Entail?

- **Planning:** is the most critical and gets the least amount of our time

Beginning with the End in mind-Stephen Covey

- **Organizing:** Orderly fashion
(Contingent/Prerequisites)
- **Controlling:** is critical if we are to use our limited resources wisely
- **Measuring:** To determine if we accomplished the goal or met the target?



Why is Project Management Important?

- Enables us to map out a course of action or work plan
- Helps us to think systematically and thoroughly
- Unique Task
- Specific Objective
- Variety of Resources
- Time bound

The Management Spectrum

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

SEI has developed a *people management capability maturity model (PM-CMM)*:

- to enhance the readiness of software organizations
- to undertake increasingly complex applications by helping
- to attract, grow, motivate, deploy, and retain the talented needed
- to improve their software development capability

The People

The people management maturity model defines:

recruiting, selection, performance
management, training, compensation,
career development, organization and work
design, and team/culture development etc.

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.

People

- Stakeholders
- The Players
- Team Leaders
- The Software Team
- Coordination and Communication Issues

The Stakeholders/Players

Five categories:

1. Senior Managers: defines business issues
2. Project (technical) managers: plan, motivate, organize, and control the practitioners
3. Practitioners: deliver the technical skills
4. Customers: specify the requirements for the software
5. End-Users: interact with the software once it is released

Team Leaders

- Project management is a people-intensive activity → need “people skill”
- MOI model for Leadership:
 - Motivation
 - Organization
 - Ideas for innovation
- Characteristics of Effective Project Manager:
Problem Solving, Managerial Identity, Achievement, Influence and Team Building

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

Organizational Paradigms for Team

- A Closed Paradigm
- A Random Paradigm
- An Open Paradigm
- A Synchronous Paradigm

The Software Team

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)

Coordination and Communication Issues

Many reasons that software projects get into trouble:

- Scale
- Uncertainty
- Interoperability

Therefore, must establish methods for coordinating the people.

Coordination and Communication Issues

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, forums
- Interpersonal networking: interpersonal discussion with outsiders.

The Product

- Dilemma: Quantitative estimates, but no solid information
 1. Software scope:
 2. Problem decomposition

Software Scope

- Context:
- Information objectives
- Function and performance

Software scope must be unambiguous and understandable.

Problem Decomposition

- Sometimes call partitioning or problem elaboration
- The core of software requirement analysis
 1. Functionality
 2. Process

The Process

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

The Process

Must decide which model is most appropriate for

1. The customers
2. The characteristics of the product
3. The project environment

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

The Project

Ten signs (cont..)

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

The Project

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 W⁵HH 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?
- Who is responsible for a function?
- Where are they organizationally located?
- How will the job be done technically and managerially?
- How much of each resources is needed?

CHANGE MANAGEMENT

- Introduction
- SCM repository
- The SCM process

INTRODUCTION

What is Change Management

- Also called software configuration management (SCM)
- It is an **umbrella activity** that is applied throughout the software process
- Its goal is to maximize productivity by minimizing mistakes caused by confusion when coordinating software development
- SCM identifies, organizes, and controls modifications to the software being built by a software development team
- SCM activities are formulated to identify change, control change, ensure that change is being properly implemented, and report changes to others who may have an interest

What is Change Management (continued)

- SCM is initiated when the project begins and terminates when the software is taken out of operation
- View of SCM from various roles
 - *Project manager -> an auditing mechanism*
 - *SCM manager -> a controlling, tracking, and policy making mechanism*
 - *Software engineer -> a changing, building, and access control mechanism*
 - *Customer -> a quality assurance and product identification mechanism*

Software Configuration

- The Output from the software process makes up the software configuration
 - Computer programs (both source code files and executable files)
 - Work products that describe the computer programs (documents targeted at both technical practitioners and users)
 - Data (contained within the programs themselves or in external files)
- The major danger to a software configuration is change
 - *First Law of System Engineering: "No matter where you are in the system life cycle, the system will change, and the desire to change, it will persist throughout the life cycle"*

Origins of Software Change

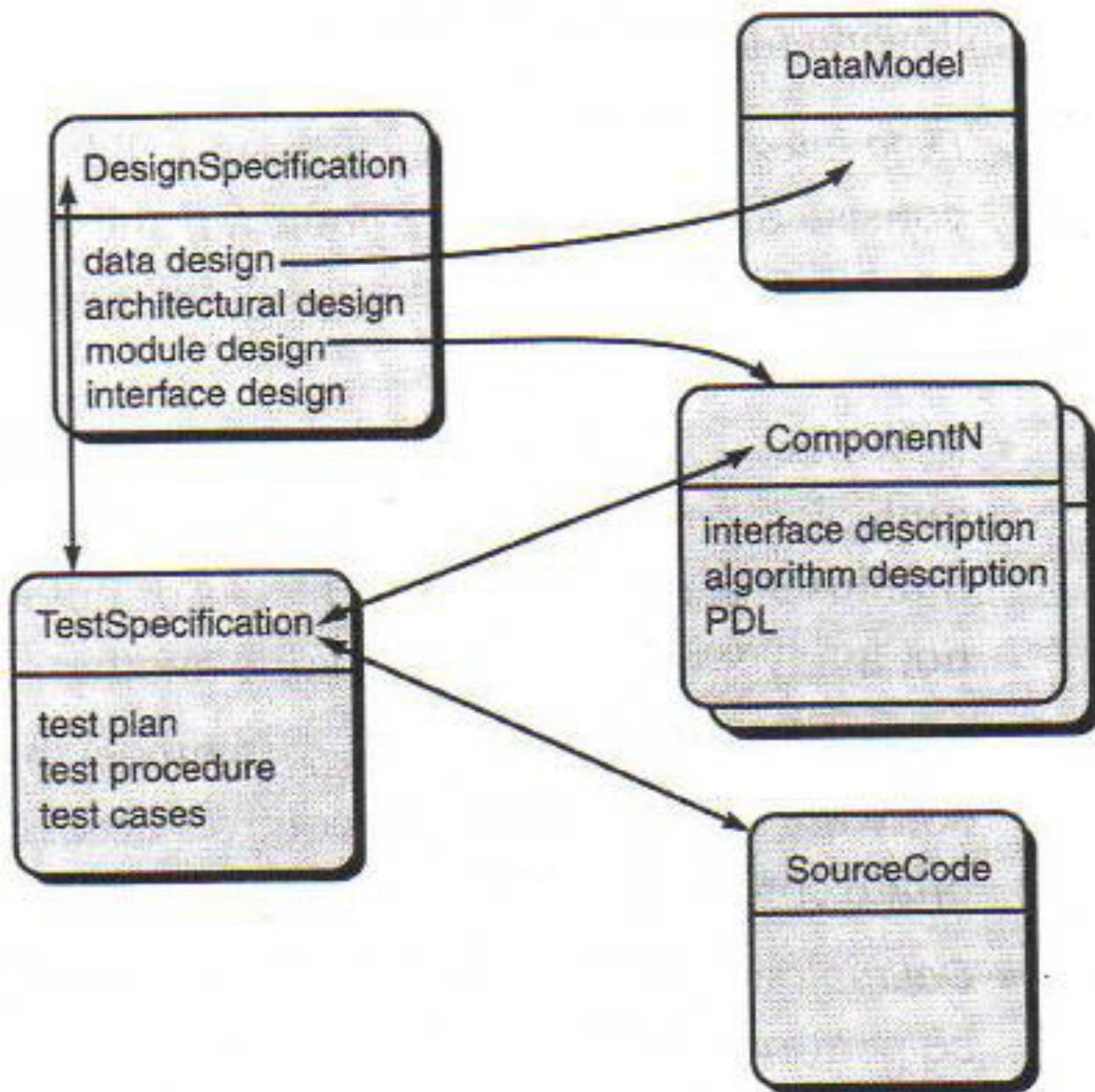
- Errors detected in the software need to be corrected
- New business or market conditions dictate changes in product requirements or business rules
- New customer needs demand modifications of data produced by information systems, functionality delivered by products, or services delivered by a computer-based system
- Reorganization or business growth/downsizing causes changes in project priorities or software engineering team structure
- Budgetary or scheduling constraints cause a redefinition of the system or product

Elements of a Configuration Management System

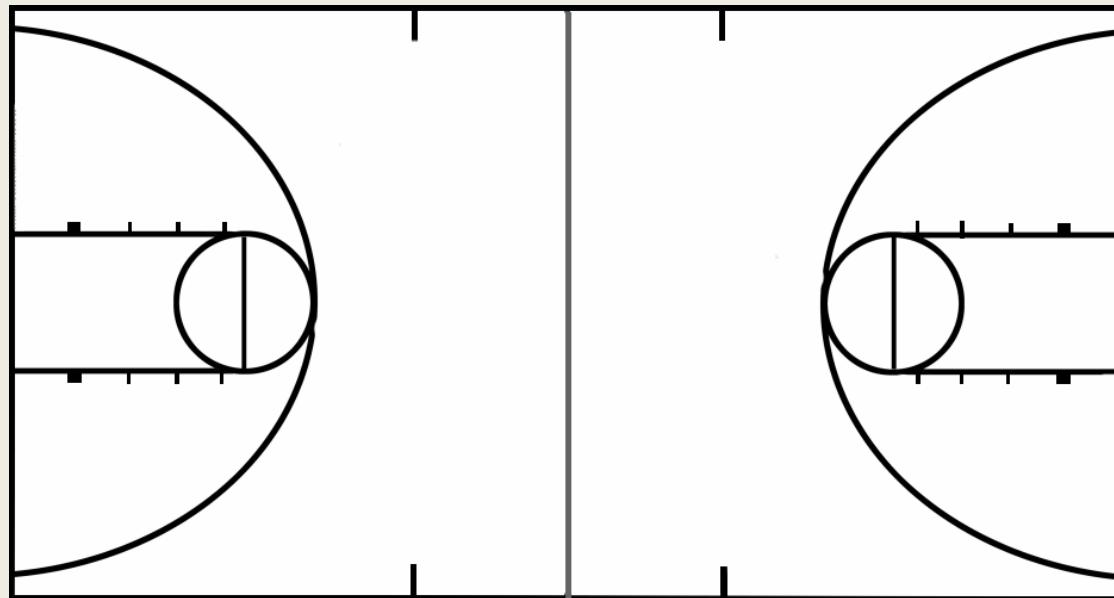
- Configuration elements
 - A set of tools coupled with a file management (e.g., database) system that enables access to and management of each software configuration item
- Process elements
 - A collection of procedures and tasks that define an effective approach to change management for all participants
- Construction elements
 - A set of tools that automate the construction of software by ensuring that the proper set of valid components (i.e., the correct version) is assembled
- Human elements
 - A set of tools and process features used by a software team to implement effective SCM

Software Configuration Items (SCIs)

- Element of information that can be as small as single UML diagram or as large as the complete design document.
- Can be considered as a single section of a large specification or one test case in a large suit of test cases.
- SCIs are stored in the database as a Configuration Object with name, attributes and is connected to other objects by relationships.



Have you established a baseline yet?

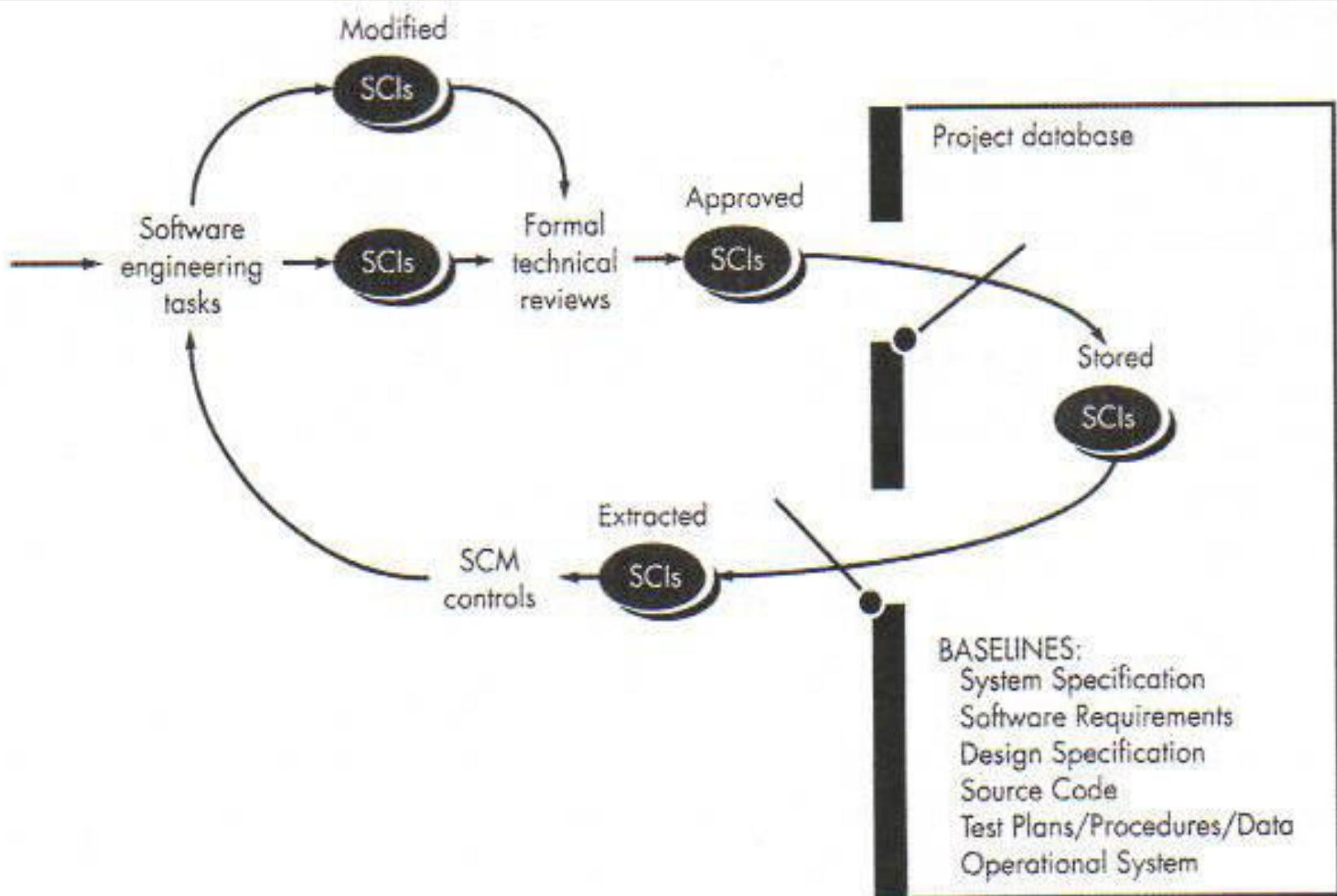


Baseline

- An SCM concept that helps practitioners to control change without seriously impeding justifiable change
- IEEE Definition: A specification or product that has been formally reviewed and agreed upon, and that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures
- It is a milestone in the development of software and is marked by the delivery of one or more computer software configuration items (SCIs) that have been approved as a consequence of a formal technical review
- A SCI may be such work products as a document (as listed in MIL-STD-498), a test suite, or a software component

Baselining Process

- 1) A series of software engineering tasks produces a SCI
- 2) The SCI is reviewed and possibly approved
- 3) The approved SCI is given a new version number and placed in a project database (i.e., software repository)
- 4) A copy of the SCI is taken from the project database and examined/modified by a software engineer
- 5) The baselining of the modified SCI goes back to Step #2



THE SCM REPOSITORY

Paper-based vs. Automated Repositories

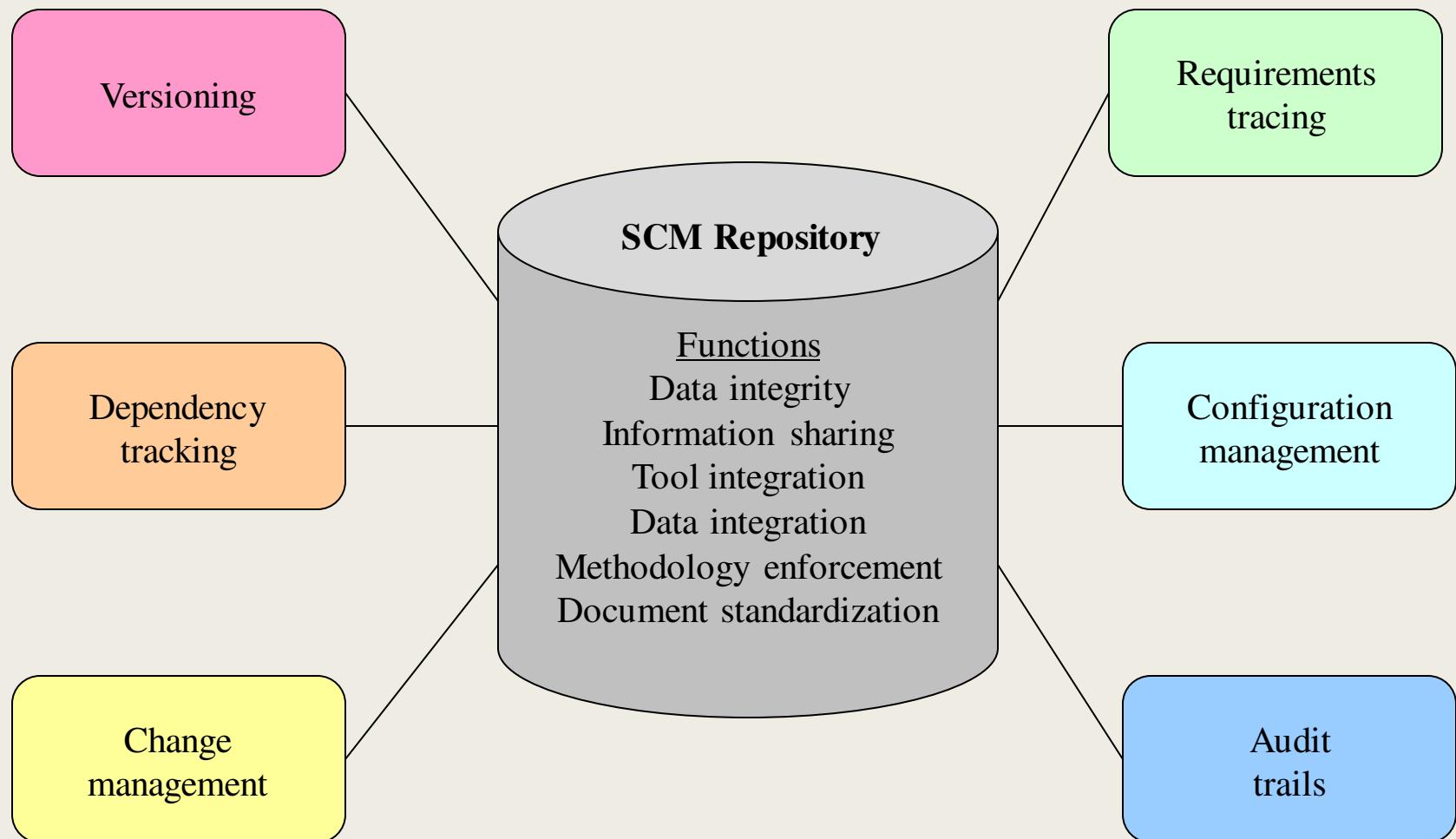
Problems with paper-based repositories (i.e., file cabinet containing folders)

- *Finding a configuration item when it was needed was often difficult*
- *Determining which items were changed, when and by whom was often challenging*
- *Constructing a new version of an existing program was time consuming and error prone*
- *Describing detailed or complex relationships between configuration items was virtually impossible*

Today's automated SCM repository

- *It is a set of mechanisms and data structures that allow a software team to manage change in an effective manner*
- *It acts as the center for both accumulation and storage of software engineering information*
- *Software engineers use tools integrated with the repository to interact with it*

Automated SCM Repository (Functions and Tools)



Functions of an SCM Repository

- Data integrity
 - *Validates entries, ensures consistency, cascades modifications*
- Information sharing
 - *Shares information among developers and tools, manages and controls multi-user access*
- Tool integration
 - *Establishes a data model that can be accessed by many software engineering tools, controls access to the data*
- Data integration
 - *Allows various SCM tasks to be performed on one or more CSCIs*
- Methodology enforcement
 - *Defines an entity-relationship model for the repository that implies a specific process model for software engineering*
- Document standardization
 - *Defines objects in the repository to guarantee a standard approach for creation of software engineering documents*

Toolset Used on a Repository

- Versioning
 - *Save and retrieve all repository objects based on version number*
- Dependency tracking and change management
 - *Track and respond to the changes in the state and relationship of all objects in the repository*
- Requirements tracing
 - *(Forward tracing) Track the design and construction components and deliverables that result from a specific requirements specification*
 - *(Backward tracing) Identify which requirement generated any given work product*
- Configuration management
 - *Track a series of configurations representing specific project milestones or production releases*
- Audit trails
 - *Establish information about when, why, and by whom changes are made in the repository*

THE SCM PROCESS

Primary Objectives of the SCM Process

- Identify all items that collectively define the software configuration
- Manage changes to one or more of these items
- Facilitate construction of different versions of an application
- Ensure the software quality is maintained as the configuration evolves over time
- Provide information on changes that have occurred

SCM Questions

How does a software team identify the discrete elements of a software configuration?

How does an organization manage the many existing versions of a program (and its documentation) in a manner that will enable change to be accommodated efficiently?

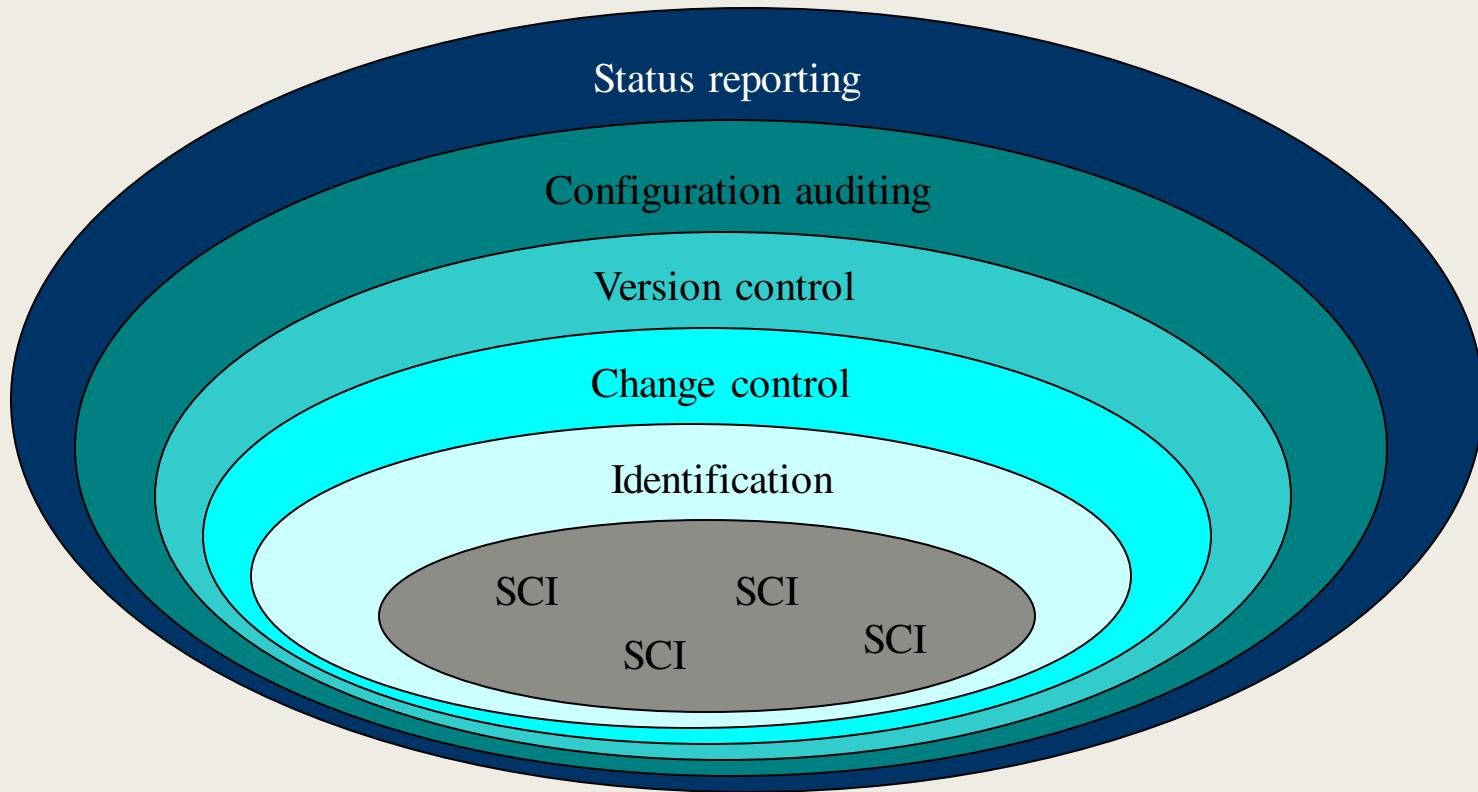
How does an organization control changes before and after software is released to a customer?

Who has responsibility for approving and ranking changes?

How can we ensure that changes have been made properly?

What mechanism is used to appraise others of changes that are made?

SCM Tasks



SCM Tasks (continued)

- Concentric layers (from inner to outer)
 - *Identification*
 - *Change control*
 - *Version control*
 - *Configuration auditing*
 - *Status reporting*
- SCIs flow outward through these layers during their life cycle
- SCIs ultimately become part of the configuration of one or more versions of a software application or system

Identification Task

- Identification separately names each SCI and then organizes it in the SCM repository using an object-oriented approach
- Objects start out as basic objects and are then grouped into aggregate objects
- Each object has a set of distinct features that identify it
 - A name that is *unambiguous to all other objects*
 - A description that contains the SCI type, a project identifier, and change and/or version information
 - List of resources needed by the object
 - The object realization (i.e., the document, the file, the model, etc.)

Change Control Task

- Change control is a procedural activity that ensures quality and consistency as changes are made to a configuration object
- A change request is submitted to a configuration control authority, which is usually a change control board (CCB)
 - *The request is evaluated for technical merit, potential side effects, overall impact on other configuration objects and system functions, and projected cost in terms of money, time, and resources*
- An engineering change order (ECO) is issued for each approved change request
 - *Describes the change to be made, the constraints to follow, and the criteria for review and audit*
- The baselined SCI is obtained from the SCM repository
 - *Access control governs which software engineers have the authority to access and modify a particular configuration object*
 - *Synchronization control helps to ensure that parallel changes performed by two different people don't overwrite one another*

Version Control Task

- Version control is a set of procedures and tools for managing the creation and use of multiple occurrences of objects in the SCM repository
- Required version control capabilities
 - An SCM repository that stores all relevant configuration objects
 - A version management capability that stores all versions of a configuration object (or enables any version to be constructed using differences from past versions)
 - A make facility that enables the software engineer to collect all relevant configuration objects and construct a specific version of the software
 - Issues tracking (bug tracking) capability that enables the team to record and track the status of all outstanding issues associated with each configuration object
- The SCM repository maintains a change set
 - Serves as a collection of all changes made to a baseline configuration
 - Used to create a specific version of the software
 - Captures all changes to all files in the configuration along with the reason for changes and details of who made the changes

Configuration Auditing Task

- Configuration auditing is an SQA activity that helps to ensure that quality is maintained as changes are made
- It complements the formal technical review and is conducted by the SQA group
- It addresses the following questions
 - *Has the change specified in the ECO been made? Have any additional modifications been incorporated?*
 - *Has a formal technical review been conducted to assess technical correctness?*
 - *Has the software process been followed, and have software engineering standards been properly applied?*
 - *Has the change been "highlighted" and "documented" in the SCI? Have the change data and change author been specified? Do the attributes of the configuration object reflect the change?*
 - *Have SCM procedures for noting the change, recording it, and reporting it been followed?*
 - *Have all related SCIs been properly updated?*
- A configuration audit ensures that
 - *The correct SCIs (by version) have been incorporated into a specific build*
 - *That all documentation is up-to-date and consistent with the version that has been built*

Status Reporting Task

- Configuration status reporting (CSR) is also called status accounting
- Provides information about each change to those personnel in an organization with a need to know
- Answers what happened, who did it, when did it happen, and what else will be affected?
- Sources of entries for configuration status reporting
 - *Each time a SCI is assigned new or updated information*
 - *Each time a change is approved by the CCB and an ECO is issued*
 - *Each time a configuration audit is conducted*
- The configuration status report
 - *Placed in an on-line database or on a website for software developers and maintainers to read*
 - *Given to management and practitioners to keep them apprised of important changes to the project SCIs*

Summary

- Introduction
- SCM Repository
- SCM Process
 - *Identification*
 - *Change control*
 - *Version control*
 - *Configuration auditing*
 - *Status reporting*



Software Requirements Specification Document

Systems Requirements Specification

Table of Contents

- I. Introduction
- II. General Description
- III. Functional Requirements
- IV. Non Functional Requirements
- V. System Architecture
- VI. System Models
- VII. Appendices

Systems Requirements Specification

I. Introduction

A Purpose

B Scope

C Definition, Acronyms, or Abbreviations

D References

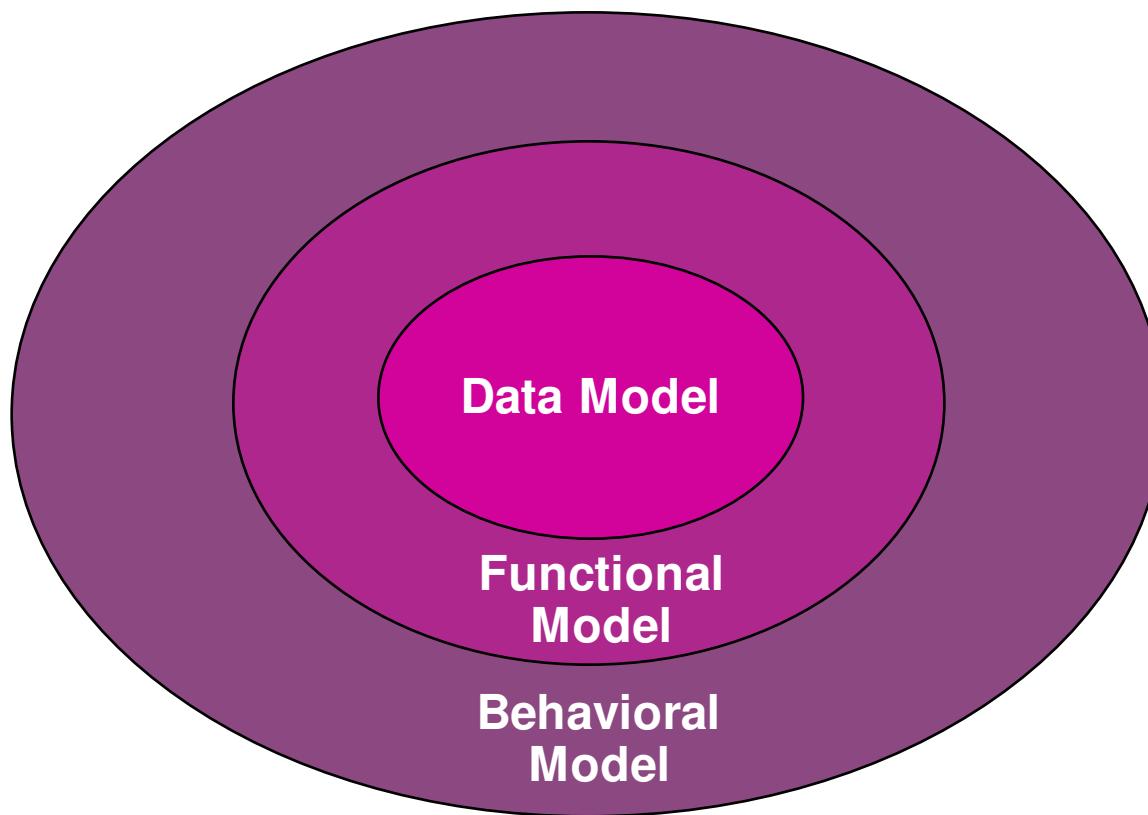
E Overview

Systems Requirements Specification

II. General Description

- A Product Perspective**
- B Product Functions**
- C User Characteristics**
- D General Constraints**
- E Assumptions**

Systems Requirements Specification



The SRS is composed of the outer layer of the behavioral model, the functional model, then the data model.

Systems Requirements Specification

I. Introduction

A Purpose

- The purpose of this Software
- Requirements Specification document
- Intended audience of this document

Systems Requirements Specification

I. Introduction

A Purpose

The purpose of the Software Requirements Specification document is to clearly define the system under development, namely the Video Rental System (VRS). The intended audience of this document includes the owner of the video store, the clerks of the video store, and the end users of the VRS. Other intended audience includes the development team such as the requirements team, requirements analyst, design team, and other members of the developing organization.

Systems Requirements Specification

I. Introduction

B. Scope

- Origin of need
- High-level description of the system functionality
- Goals of proposed system

Systems Requirements Specification

I. Introduction

B. Scope

Origin of the need

- who and what triggered the request for this software development activity
- gives developers an understanding of the goals for the proposed system

Systems Requirements Specification

I. Introduction

B. Scope

High-level functionality

- defined for the system
- usually in list separated by commas

Systems Requirements Specification

I. Introduction

B. Scope

Goals are general purposes of a system. They are fuzzy and non measurable.

A typical goal would be things like

- Increase customer satisfaction
- Make xyz easier for the customer
- Improve customer relationships

Systems Requirements Specification

I. Introduction

B. Scope

The owner of a local video store wanted to create a new business plan where everything about renting a video (except the picking up and returning of videos) was done online. Therefore, the new VRS will allow the following functionality online: to search for videos, to become members, to rent videos, to modify membership information, and to pay overdue fees. The store personnel may use the VRS to process the rented or returned videos, to add or remove videos to/from his store's video inventory and to update video information. The VRS is intended to increase the owner's profit margin by increasing video sales with this unique business approach and by allowing him to reduce the staffing needed in his stores.

Systems Requirements Specification

I. Introduction

C. Definitions, Acronyms..

As you begin to define a system, you will encounter words which need definition and general usage acronyms. These should be documented for new personnel and for clarity of all concerned parties.

Systems Requirements Specification

I. Introduction C Definitions, Acronyms..

FSU – Florida State University

CS - Computer Science

MSES - Masters in Software Engineering Science

DOE - Department of Education

....

Systems Requirements Specification

I. Introduction

D. References

Many references may be used to define existing systems, procedures (both new and old), documents and their requirements, or previous system endeavors. These references are listed here for others.

If any of these references are provided in the appendices, it should be noted here.

Systems Requirements Specification

I. Introduction D References

Clerk - Personnel staff who is working in a video store

Customer - Anyone who interacts with the VRS without becoming a member

Functional requirement - A service provided by the software system

Member - Anyone who registers with the VRS to acquire membership in the video store

Systems Requirements Specification

I. Introduction

E. Overview

This section defines the organization of the entire document. It will lay the framework for reading the document.

Systems Requirements Specification

I. Introduction E Overview

Section 2 of the SRS describes the product in more detail. Section 3 provides a complete list of the functional requirements of the intended system. Section 4 provides the non-functional requirements. Section 5 shows the class diagram, and Section 6 the use case diagram. The appendices appear next.

Section I of SRS

| | |
|--|-----------------------------|
| I.A Purpose | Paragraph form |
| I.B Scope of the System Specified | Paragraph form |
| I.C Definitions, Acronyms, and Abbreviations | Table form or bulleted list |
| I.D References to Supporting Documents | Bulleted list |
| I.E Overview of rest of SRS | Paragraph form |

Systems Requirements Specification

II. General Description

- A Product Perspective
- B Product Functions
- C User Characteristics
- D General Constraints
- E Assumptions

Systems Requirements Specification

II General Description

A Product Perspective

This defines the relationship this product has in the entire spectrum of products.

It defines who will be responsible for the product and what business purpose it serves.

It also defines what interfaces it may have to other systems.

Systems Requirements Specification

II General Description

A Product Perspective

The VRS is a web-based system. The system interfaces with two other systems, the owner's email system, the video distributor's video system, and the browsers used by VRS customers. The system provides a secure environment for all financial transactions and for the storing and retrieving of confidential member information.

Systems Requirements Specification

II. General Description B Product Functions

This section lists the major functions of the system.

It provides a summary of all the functions of the software. The functions should be organized in a way that makes the list of functions understandable to the customer or to anyone else reading the document for the first time.

This section should be consistent with the functional requirements defined in Section III.

Systems Requirements Specification

II. General Description – B Product Functions

The VRS allows customers to search the video inventory provided by this video store. To rent videos through the VRS, one must register as a member using the VRS. Upon becoming a member and logging into the VRS, the VRS provides the functionality for renting videos, modifying membership information, and paying overdue fines.

The clerks of the video store use VRS to process the return of rented videos. The owner of the video store uses VRS to add new videos into the system, remove videos from the system, and modify video information.

The VRS sends emails to members concerning video rentals. One day before a rented video is due to be returned, VRS emails the member a reminder of the due date for the video(s). For any overdue videos, VRS emails the member every 3rd day with overdue notices. At the 60-day limit for outstanding videos, VRS debits the member's credit card with the appropriate charge and notifies the member of this charge.

Systems Requirements Specification

II. General Description – C User Characteristics

List the users involved with the proposed system including the general characteristics of eventual users (for example, educational background, amount of product training).

List the responsibility of each type of user involved, if needed.

Systems Requirements Specification

II. General Description – C User Characteristics

The three main groups of VRS users are customers, members, and store personnel. A customer is anyone who is not a member. The customer can only search through the video inventory. The amount of product training needed for a customer is none since the level of technical expertise and educational background is unknown. The only skill needed by a customer is the ability to browse a website.

Member is someone who has registered with VRS. A member can rent videos and pay fees online. As with a customer, these activities require no product training since the level of technical expertise and educational background of a member is unknown. The only skill needed by a member is the ability to browse a website.

The store personnel are divided into two groups: the clerk-level personnel and owner-level personnel. Their educational level is unknown and both group needs little to no training.

Systems Requirements Specification

II. General Description – D General Constraints

D General Constraints

In this section, the constraints of the system are listed. They include hardware, network, system software, and software constraints. It also includes user constraints, processing constraints, timing constraints, and control limits.

Systems Requirements Specification

II. General Description – D General Constraints

This system provides web access for all customer and member functions. The user interface will be intuitive enough so that no training is required by customers, members, or store personnel. All online financial transactions and the storage of confidential member information will be done in a secure environment. Persistent storage for membership, rental, and video inventory information will be maintained.

Systems Requirements Specification

II General Description – D Assumptions and Dependencies

This includes assumptions made at the beginning of the development effort as well as those made during the development.

List and describe each of the factors that affect the requirements stated in the SRS. These factors are not design constraints on the software but any changes to them can affect the requirements in the SRS.

For example, an assumption might be that a specific operating system will be available on the hardware designated for the software product. If, in fact, the operating system is not available, the SRS would then have to change.

Section II of SRS

| | |
|-----------------------------------|----------------|
| II.A Product Perspective | Paragraph form |
| II.B Product Functions | Paragraph form |
| II.C User Characteristics | Paragraph form |
| II.D General Constraints | Paragraph form |
| II.E Assumptions and Dependencies | Paragraph form |

Systems Requirements Specification

III Functional Requirements

Functional requirements are those business functions which are included in this software under development. It describes the features of the product and the needed behavior.

The functional requirements are going to be written in narrative form identified with numbers. Each requirement is something that the system SHALL do. Thus, it has a common name of a *shall* list. You may provide a brief design rationale for any requirement which you feel requires explanation for how and/or why the requirement was derived.

Systems Requirements Specification

IV Non Functional Requirements

Non functional requirements are properties that the system must have such as performance, reusability, usability, user friendliness, etc.

The same format as the functional requirements is to be used for the non-functional requirements. You may provide a brief design rationale for any requirement which you feel requires explanation for how and/or why the requirement was derived.

Systems Requirements Specification

V System Architecture

This section presents a high-level overview of the anticipated system architecture using a class diagram. It shows the fundamental objects/classes that must be modeled with the system to satisfy its requirements. Each class on the diagram must include the attributes related to the class. All the relationships between classes and their multiplicity must be shown on the class diagram. The classes specified in this document only are those directly derived from the application domain.

Systems Requirements Specification

VI System Models

This section presents the use case diagram for the system under development. The use case diagram should be a complete version containing all the use cases needed to describe the functionality to be developed.

Systems Requirements Specification

VII Appendixes

Appendix A. Data dictionary

Appendix B. Raw use case point analysis

Appendix C. Screens and reports with navigation matrix.

Appendix D. Scenario analysis tables

Appendix E. Screens/reports list

Appendix F and following. Other items needed