

Introduction to Software Engineering

- **Nature of Software**
- **Unique Nature of Web Apps**
- **Software Engineering**
- **Software Engineering Practices**
- **Software Myths**

Nature of Software

- Software is an **information transformer**—producing, managing, acquiring, modifying, displaying, or transmitting information
- Software delivers the most important product of our time—**information**
- **programmer are the same questions** that are asked when modern computer-based systems are built
 - Why does it take so long to get software finished?
 - Why are development costs so high?
 - Why can't we find all errors before we give the software to our customers?
 - Why do we spend so much time and effort maintaining existing programs?
 - Why do we continue to have difficulty in measuring progress as software is being developed and maintained?

What is Software?

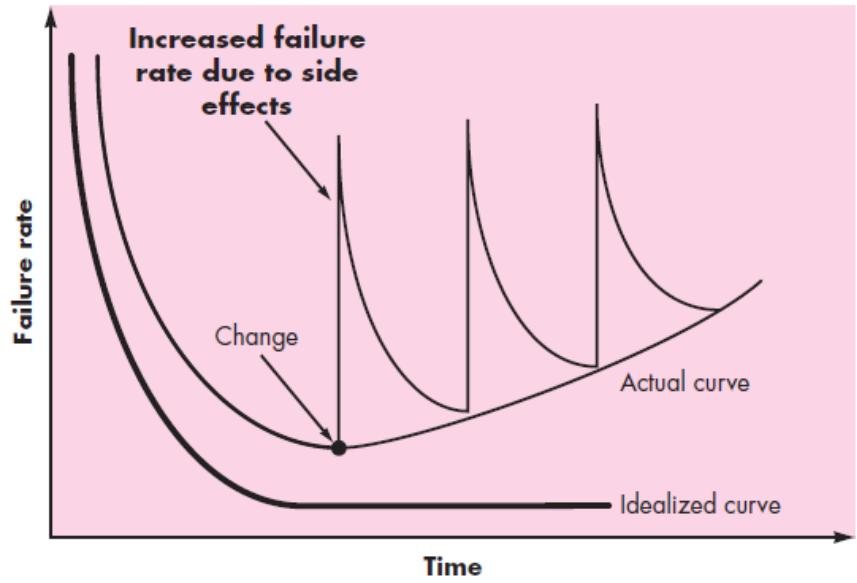
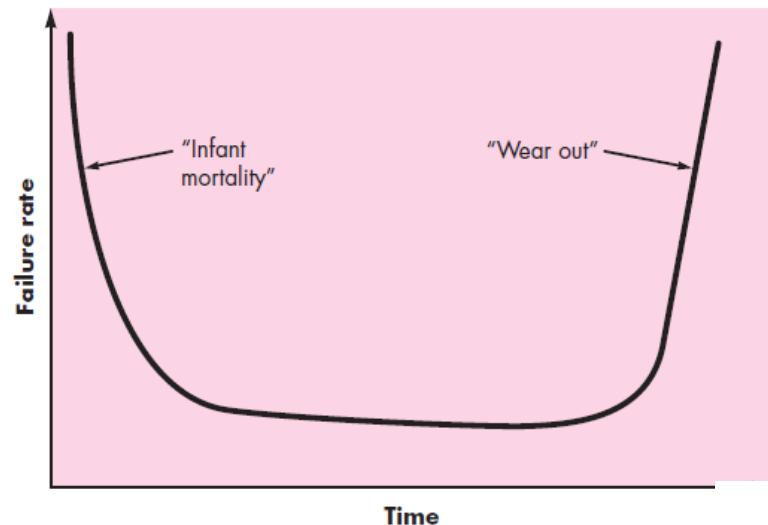
Software is:

- (1) *instructions*** (computer programs) that when executed provide desired features, function, and performance;
- (2) *data structures*** that enable the programs to adequately manipulate information and
- (3) *documentation*** that describes the operation and use of the programs.

Features of Software?

- Features of such logical system:
 - Software is **developed or engineered**; it is not manufactured in the classical sense which has quality problem.
 - Software **doesn't "wear out."** but it deteriorates (due to change). Hardware has bathtub curve of failure rate (high failure rate in the beginning, then drop to steady state, then cumulative effects of dust, vibration, abuse occurs).
 - Although the industry is moving toward component-based construction (e.g. standard screws and off-the-shelf integrated circuits), most software continues to be **custom-built**. Modern reusable components encapsulate data and processing into software parts to be reused by different programs. E.g. graphical user interface, window, pull-down menus in library etc.

Wear vs. Deterioration



Software Applications

- **System software:** to service other programs
 - E.g. Compilers, editors, file managers, OS, drivers, networking s/w etc
- **Application software:** to solve specific business need
 - E.g. Transaction Processing, Manufacturing process control etc
- **Engineering/scientific software :** to develop algorithms for real time
 - E.g. CAD, System Simulation
- **Embedded software :** Software+Hardware like interface
 - E.g. Digital functionalities in automobile, keypad for microwave
- **Product-line software:** to provide specific capability
 - E.g. Word, Multimedia applications, database etc
- **WebApps (Web applications) :** Online
- **Artificial Intelligence software :** to solve complex problems
 - E.g. Pattern Recognition, Artificial Neural Network, Game Playing etc.

Software—New Categories

- **Open world computing**—pervasive, distributed computing
- **Ubiquitous computing**— wireless networks
- **Net-sourcing**—the Web as a computing engine
- **Open source**— “free” source code open to the computing community (a blessing, but also a potential curse!)
- Also ...
 - **Data mining**
 - **Grid computing**
 - **Cognitive machines**
 - **Software for nanotechnologies**

Legacy Software : Older Programs

- Software developed **decades ago** and continually modified to meet changes in business requirements and computing platforms
- Software **difficult and costly to maintain**, risky to evolve

The Quality of Legacy Software

Why must it change?

- software must be **adapted** to meet the needs of new computing environments or technology.
- software must be **enhanced** to implement new business requirements.
- software must be **extended to make it interoperable** with other more modern systems or databases.
- software must be **re-architected** to make it viable within a network environment.

Unique Nature of Web Apps

- The following attributes are encountered in the vast majority of Web Apps:
 - Network intensiveness
 - Concurrency
 - Unpredictable load
 - Performance
 - Availability
 - Data driven
 - Content sensitive
 - Continuous evolution
 - Immediacy
 - Security
 - Aesthetics

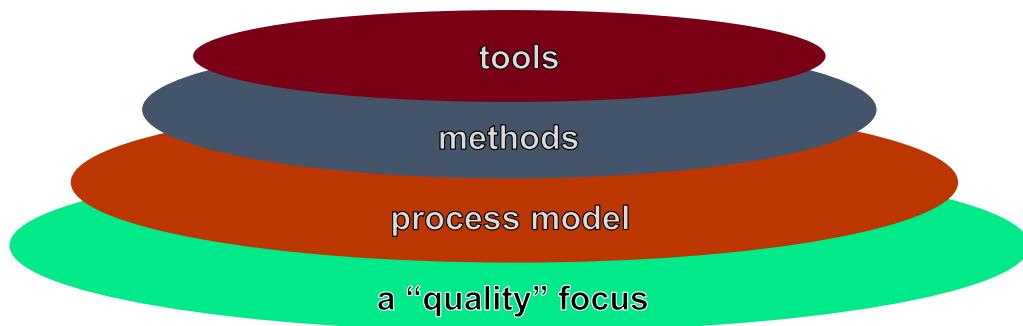
Key Points of Software Engineering

- It follows that a concerted effort should be made to **understand the problem before a software solution** is developed.
- It follows that **design becomes a pivotal activity** .
- It follows that **software should exhibit high quality** .
- It follows that **software should be maintainable** .
- Software in all of its forms and across all of its application domains **should be engineered**.

Software Engineering

- The IEEE definition:
 - *Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as mentioned in (1).*

A Layered Technology



Software Engineering

Definition of Software Process

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

Generic Process Model

- Communication
- Planning
- Modeling
 - Analysis of requirements
 - Design
- Construction
 - Code generation
 - Testing
- Deployment

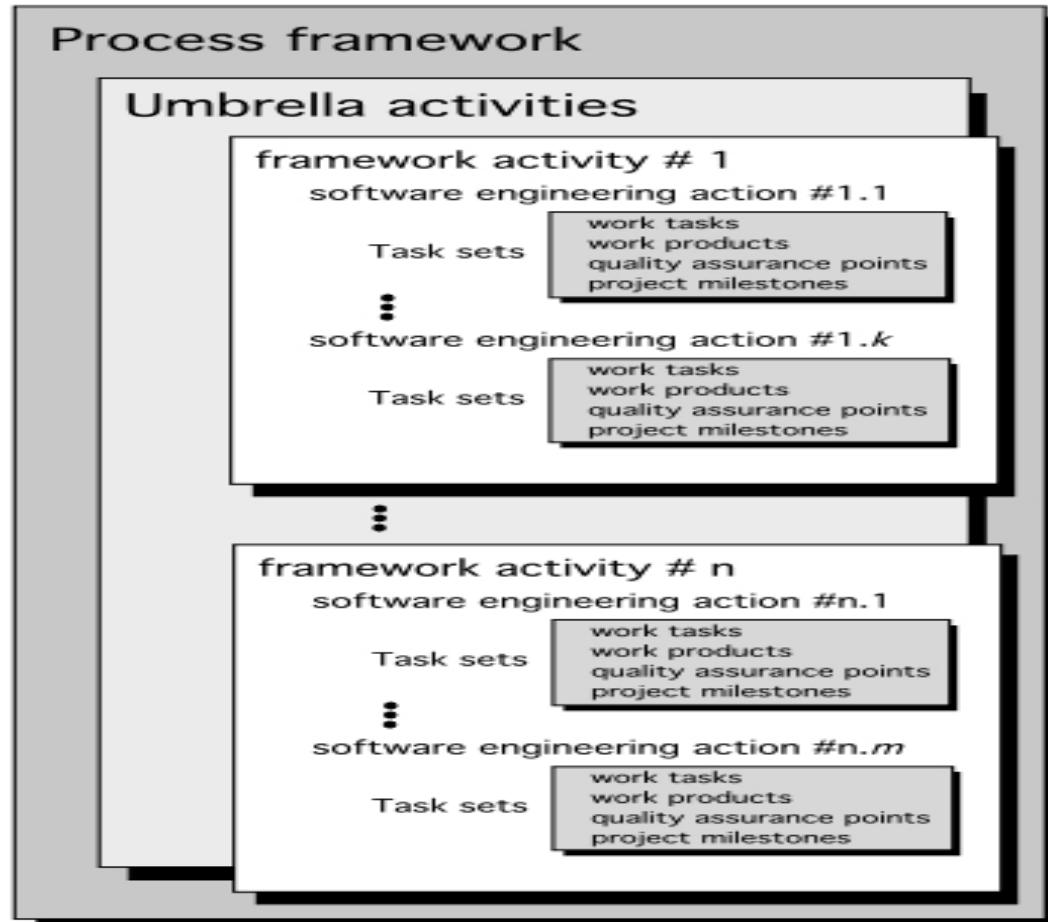
Umbrella Activities

Complement the five process framework activities and help team [manage and control](#) progress, quality, change, and risk.

- **Software project tracking and control:** assess progress against the plan and take actions to maintain the schedule.
- **Risk management:** assesses risks that may affect the outcome and quality.
- **Software quality assurance:** defines and conduct activities to ensure quality.
- **Technical reviews:** assesses work products to uncover and remove errors before going to the next activity.
- **Measurement:** define and collects process, project, and product measures to ensure stakeholder's needs are met.
- **Software configuration management:** manage the effects of change throughout the software process.
- **Reusability management:** defines criteria for work product reuse and establishes mechanism to achieve reusable components.
- **Work product preparation and production :** create work products such as models, documents, logs, forms and lists.

A Process Framework

Software process



Identifying a Task Set

Before you can proceed with the process model, a key question: what **actions** are appropriate for a framework **activity** given the nature of the problem, the characteristics of the people and the stakeholders?

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

A list of the task to be accomplished

A list of the work products to be produced

A list of the quality assurance filters to be applied

Identifying a Task Set

For example, a small software project requested by one person with simple requirements, the communication activity might encompass little more than a phone call with the stakeholder. Therefore, the only necessary action is phone conversation, the work tasks of this action are:

- 1. Make the contact with stakeholder via telephone.**
- 2. Discuss requirements and take notes.**
- 3. Organize notes into a brief written statement of requirements.**
- 4. E-mail to stakeholder for review and approval.**

Example of a Task Set for Elicitation

The task sets for Requirements gathering action for a **simple** project may include:

1. Make a list of stakeholders for the project.
2. Invite all stakeholders to an informal meeting.
3. Ask each stakeholder to make a list of features and functions required.
4. Discuss requirements and build a final list.
5. Prioritize requirements.
6. Note areas of uncertainty.

Example of a Task Set for Elicitation

The task sets for Requirements gathering action for a **big** project may include:

1. Make a list of stakeholders for the project.
2. Interview each stakeholders separately to determine overall wants and needs.
3. Build a preliminary list of functions and features based on stakeholder input.
4. Schedule a series of facilitated application specification meetings.
5. Conduct meetings.
6. Produce informal user scenarios as part of each meeting.
7. Refine user scenarios based on stakeholder feedback.
8. Build a revised list of stakeholder requirements.
9. Use quality function deployment techniques to prioritize requirements.
10. Package requirements so that they can be delivered incrementally.
11. Note constraints and restrictions that will be placed on the system.
12. Discuss methods for validating the system.

**Both do the same work with different depth and formality.
Choose the task sets that achieve the goal and still²¹ maintain quality and agility.**

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

Adapting a Process Model

The process should be **agile and adaptable** to problems. Process adopted for one project might be significantly different than a process adopted from another project (**to the problem, the project, the team, organizational culture**). Among the differences are:

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

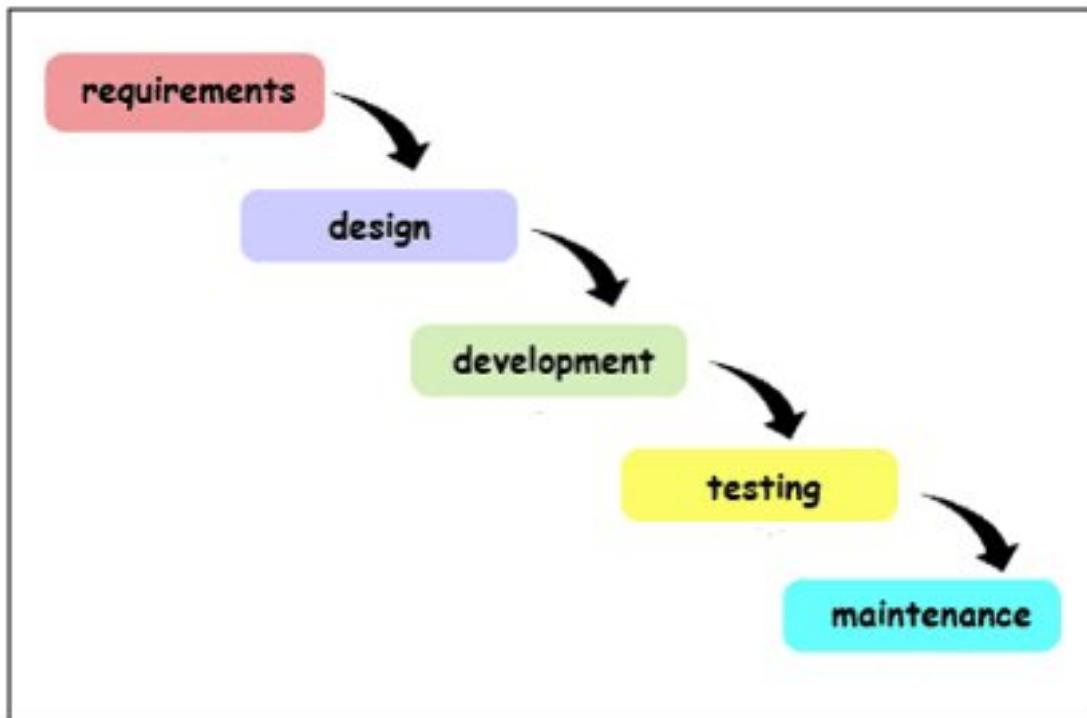
Prescriptive and Agile Process Models

- The **prescriptive process** models stress detailed definition, identification, and application of process activates and tasks. Intent is to improve system quality, make projects more manageable, make delivery dates and costs more predictable, and guide teams of software engineers as they perform the work required to build a system.
- Unfortunately, there have been times when these objectives were not achieved. If prescriptive models are applied dogmatically and without adaptation, they can increase the level of bureaucracy.
- **Agile process models** emphasize project “agility” and follow a set of principles that lead to a more informal approach to software process. It emphasizes maneuverability and adaptability. It is particularly useful when Web applications are engineered.

Software Engineering Practice

- Essence of practice
 - *Understand the problem* (communication and analysis).
 - *Plan a solution* (modeling and software design).
 - *Carry out the plan* (code generation).
 - *Examine the result for accuracy* (testing and quality assurance).
- Hooker's General Principles :
 - **The First Principle:** *The Reason It All Exists*
 - **The Second Principle:** *KISS (Keep It Simple, Stupid!)*
 - **The Third Principle:** *Maintain the Vision*
 - **The Fourth Principle:** *What You Produce, Others Will Consume*
 - **The Fifth Principle:** *Be Open to the Future*
 - **The Sixth Principle:** *Plan Ahead for Reuse*
 - **The Seventh principle:** *Think!*

Phases in Software Development



The Primary Goal of Any Software Process: High Quality

Remember:

High quality ⇒ project timeliness

Why?

Less rework!

Software Myths

- **Software Management Myths**
- **Software Customer Myths**
- **Developer Myths**

Software Management Myths

- **Myth:** *We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?*
- **Reality:** The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it adaptable? Is it streamlined to improve time-to-delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."

Software Management Myths

- **Myth:** *If we get behind schedule, we can add more programmers and catch up (sometimes called the “Mongolian horde” concept).*
- **Reality:** Software development is not a mechanistic process like manufacturing. In the words of Brooks [Bro95]: “adding people to a late software project makes it later.” At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well coordinated manner.

Software Management Myths

- **Myth:** *If I decide to outsource the software project to a third party, I can just relax and let that firm build it.*
- **Reality:** If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

Customer Myths

- **Myth:** *A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.*
- **Reality:** Although a comprehensive and stable statement of requirements is not always possible, an ambiguous “statement of objectives” is a recipe for disaster. Unambiguous requirements (usually derived iteratively) are developed only through effective and continuous communication between customer and developer

Customer Myths

- **Myth:** *Software requirements continually change, but change can be easily accommodated because software is flexible.*
- **Reality:** It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small.¹⁶ However, as time passes, the cost impact grows rapidly—resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification.

Practitioner's myths

- **Myth:** Once we *write the program and get it to work, our job is done.*
- **Reality:** Someone once said that “the sooner you begin ‘writing code,’ the longer it’ll take you to get done.” Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Practitioner's myths

- **Myth:** *Until I get the program “running” I have no way of assessing its quality.*
- **Reality:** One of the most effective software quality assurance mechanisms can be applied from the inception of a project—*the technical review*. Software reviews (described in Chapter 15) are a “quality filter” that have been found to be more effective than testing for finding certain classes of software defects.

Practitioner's myths

- **Myth:** *The only deliverable work product for a successful project is the working program.*
- **Reality:** A working program is only one part of a software configuration that includes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering and, more important, guidance for software support.

Practitioner's myths

- **Myth:** *Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.*
- **Reality:** Software engineering is not about creating documents. It is about creating a quality product. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

Requirements Engineering

- Problems with requirements practices
- Requirement engineering phases

A Customer walks into your office and says;

“I know you think you understand what I said, but what you don’t understand is what I said is not what I mean”.

Invariably this happens late in the project, after deadline commitments have been made, reputations are on the line, and serious money is at stake.

Requirement Engineering

- Requirements engineering (RE) refers to the process of defining, documenting and maintaining requirements.
- Requirement Engineering (RE) is the science and discipline concerned with analyzing and documenting requirements.

Starter Questions

- What is a "requirement"?
- How do we determine the requirements?



Types of Requirements

- Functional
 - ex - it must email the sales manager when an inventory item is "low"
- Non-Functional
 - ex - it must require less than one hour to run
- Explicit
 - ex – required features
- Implied
 - ex – software quality
- Forgotten
 - ex – exists in current process
- Unimagined



Questions

- What makes a particular requirement good or bad?
- Why is requirements engineering difficult?



Requirements of Requirements

- **Clear**
- **Measurable**
- **Feasible**
- **Necessary**
- **Prioritized**
- **Concise**

The Problems with our Requirements Practices

- We have trouble in understanding the requirements that we do acquire from the customer
- We often record requirements in a disorganized manner
- We spend far too little time verifying what we do record
- We allow change to control us, rather than establishing mechanisms to control change
- Most importantly, we fail to establish a solid foundation for the system or software that the user wants built

The Problems with our Requirements Practices (contd..)

- Many software developers argue that
 - Building software is so compelling that we want to jump right in (before having a clear understanding of what is needed)
 - Things will become clear as we build the software
 - Project stakeholders will be able to better understand what they need only after examining early iterations of the software
 - Things change so rapidly that requirements engineering is a waste of time
 - The bottom line is producing a working program and that all else is secondary
- All of these arguments contain some truth, especially for small projects that take less than one month to complete
- However, as software grows in size and complexity, these arguments begin to break down and can lead to a failed software project

A Solution: Requirements Engineering

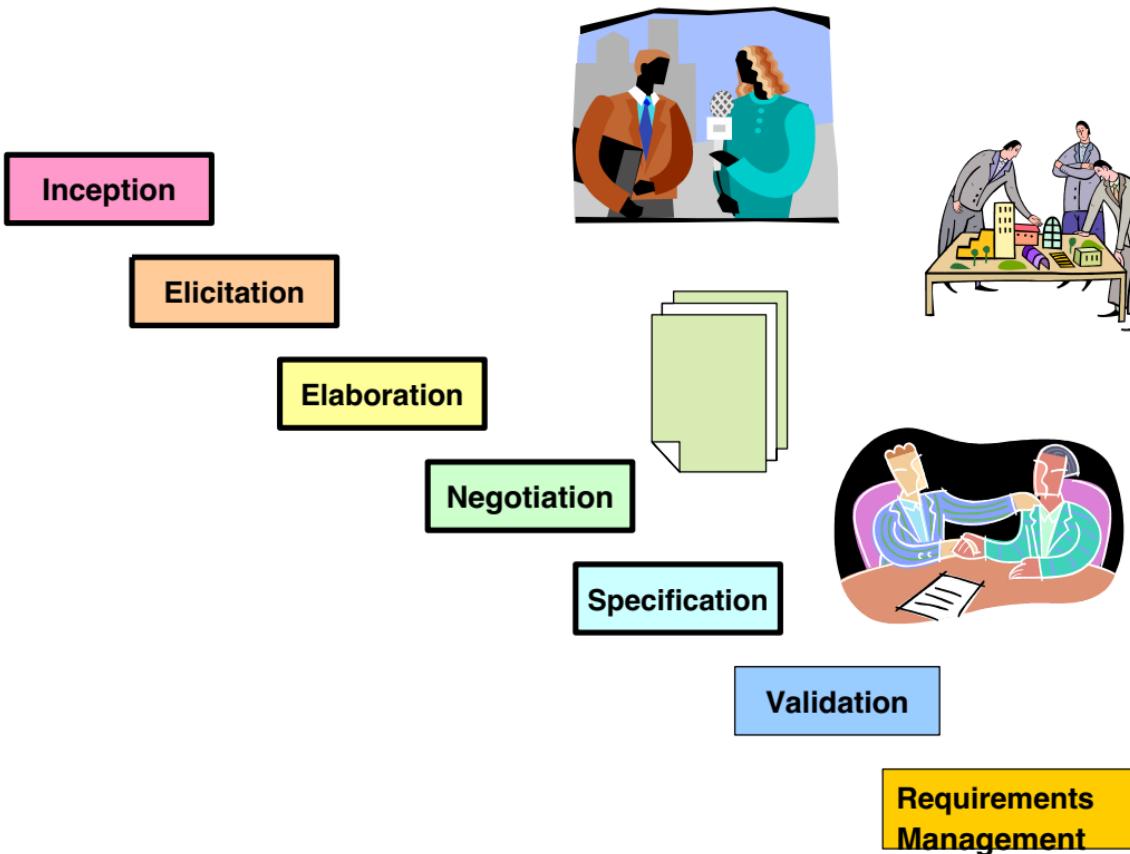
- Begins during the communication activity and continues into the modeling activity
- Builds a bridge from the system requirements into software design and construction
- Allows the requirements engineer to examine
 - the context of the software work to be performed
 - the specific needs that design and construction must address
 - the priorities that guide the order in which work is to be completed
 - the information, function, and behavior that will have a profound impact on the resultant design

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.

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.
- Seven distinct tasks: Inception, Elicitation, Elaboration, Negotiation, Specification, Validation, Requirements Management



Inception Task

- During inception, the requirements engineer asks a set of questions to establish...
 - A basic understanding of the problem
 - The people who want a solution
 - The nature of the solution that is desired
 - The effectiveness of preliminary communication and collaboration between the customer and the developer
- Through these questions, the requirements engineer needs to...
 - Identify the stakeholders
 - Recognize multiple viewpoints
 - Work toward collaboration
 - Break the ice and initiate the communication

The First Set of Questions

These questions focus on the customer, other stakeholders, the overall goals, and the benefits

- 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 Next Set of Questions

These questions enable the requirements engineer to gain a better understanding of the problem and allow the customer to voice his or her perceptions about a solution

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

The Final Set of Questions

These questions focus on the effectiveness of the communication activity itself

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

Elicitation Task

- Questions asked in a manner that no further doubt remains
- Elicitation of requirements is difficult because
 - Problems of scope in identifying the boundaries of the system or specifying too much technical detail rather than overall system objectives
 - Problems of understanding what is wanted, what the problem domain is, and what the computing environment can handle (Information that is believed to be "obvious" is often omitted)
 - Problems of volatility because the requirements change over time
- Elicitation may be accomplished through two activities
 - Collaborative requirements gathering
 - Quality function deployment

Collaborative Requirement Gathering

- Team Oriented Approach
- Team of stakeholders and developers meet
 - Identify problem
 - Propose a solution
 - Negotiate different approaches
 - Specify preliminary solution
- Many approaches but all follow basic guidelines

Guidelines

Meetings are conducted and attended by both s/w engineers and customers

Rules for preparation and participation should be established

Agenda for formal and informal discussions

Facilitator- controls the meetings, can be anyone from participants

Definition mechanism- Worksheets, Flip charts, Chat rooms, Virtual Forum etc

Goal- Identify problem and solution

Scenario

- Initial meetings in inception phase – One or two pages of Product Request (PR)
- Meeting place, time & date should be selected for further meetings and facilitator need to be identified
- Members of s/w team and other stakeholders are invited to attend
 - PR is distributed to all members for approval
- While reviewing PR each attendee asked to make **object list** before attending future meeting
 - Objects regarding environmental aspects
 - Objects produced by system
 - Objects used by system
- Each attendee asked to make **list of services**
 - Process and Functions
- Each attendee has to make **list of constraints** and **performance criteria**

Example: SafeHome Project (Home Security Function)

Our research indicates that the market for home management systems is growing at a rate of 40% per year. The first SafeHome function we bring to market should be the home security function. Most people are familiar with “Alarm Systems” so this would be an easy sell.

The home security function would protect against and/or recognize a variety of undesirable “situations” such as illegal entry, fire, flooding, carbon monoxide levels and others. It will use our wireless sensors to detect each situation, can be programmed by homeowner and will automatically telephone a monitoring agency when a situation is detected.

- List of Objects:
 - Control Panel
 - Smoke detector
 - Window & door sensors
 - Motion Detector
 - An Alarm
 - An Event
 - A Display
 - A PC
 - Telephone numbers
 - Telephone call
- List of Services:
 - Configuring the system
 - Setting the alarm
 - Monitoring the sensors
 - Dialing phone
 - Programming control panel
 - Reading display
- List of Constraints:
 - System should detect and recognize failure of sensors
 - User Friendly display
 - Performance of sensor recognition

Elicitation Meeting

- Need to combine all documents from all attendees and prepare agreement
- Take one topic and prepare list of points
- Start with mini specifications
 - E.g. control panel: mounted unit with some size with sensors and PC connected. User interaction through keyboard. LCD display for feedback.
- Each mini specification will be discussed: addition/deletion/further elaboration
- Uncover new objects, services and constraints
- After each meeting **Issue List** will be prepared and maintained.
- Mention **Validation Criteria** for all requirements
- Finally call outsider to write **SRS (Software Requirement Specifications)**

Quality Function Deployment (QFD)

- Technique to translate needs of the customer into technical requirements
- Concentrates on maximizing customer satisfaction from s/w engineering process
- Three types of Requirements:
 - Normal- Objectives and Goals
 - E.g. Graphical Display, Specific functions, performance
 - Expected- implicit to the product, customer does not explicitly state them
 - E.g. GUI, Reliability, Ease of S/W installation
 - Exciting- Go beyond customers expectations

Deployment Tasks

- Function Deployment: Value each function i.e. requirement by setting priority
- Information Deployment: Identifies data, objects and events
- Task Deployment: Behavior of the system
- Value Analysis: Determine relative priority of the requirements

QFD Process

- QFD uses raw data for requirement gathering:
 - Customer Requirements
 - Observations
 - Surveys
 - Examination of historical data
- The above all data will be translated into a table of requirements- Customer Voice Table – reviewed with customer
- Variety of diagrams, matrices and evaluation methods are used to extract the expected and excited requirements

Elaboration Task

- During elaboration, the software engineer takes the information obtained during inception and elicitation and begins to expand and refine it
- Elaboration focuses on developing a refined technical model of software functions, features, and constraints
- It is an analysis modeling task
 - Use cases are developed
 - Domain classes are identified along with their attributes and relationships
 - State machine diagrams are used to capture the life on an object
- The end result is an analysis model that defines the functional, informational, and behavioral domains of the problem

Developing Use Cases

- Step One – Define the set of actors that will be involved in the story
 - Actors are people, devices, or other systems that use the system or product within the context of the function and behavior that is to be described
 - Actors are anything that communicate with the system or product and that are external to the system itself
- Step Two – Develop use cases, where each one answers a set of questions

Questions Commonly Answered by a Use Case

- Who is the primary actor(s), the secondary actor(s)?
- What are the actor's goals?
- What preconditions should exist before the scenario begins?
- What main tasks or functions are performed by the actor?
- What exceptions might be considered as the scenario is described?
- What variations in the actor's interaction are possible?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

Elements of the Analysis Model

- Scenario-based elements
 - Describe the system from the user's point of view using scenarios that are depicted in use cases and activity diagrams
- Class-based elements
 - Identify the domain classes for the objects manipulated by the actors, the attributes of these classes, and how they interact with one another; they utilize class diagrams to do this
- Behavioral elements
 - Use state diagrams to represent the state of the system, the events that cause the system to change state, and the actions that are taken as a result of a particular event; can also be applied to each class in the system
- Flow-oriented elements
 - Use data flow diagrams to show the input data that comes into a system, what functions are applied to that data to do transformations, and what resulting output data are produced

Negotiation Task

- During negotiation, the software engineer reconciles the conflicts between what the customer wants and what can be achieved given limited business resources
- Requirements are ranked (i.e., prioritized) by the customers, users, and other stakeholders
- Risks associated with each requirement are identified and analyzed
- Rough guesses of development effort are made and used to assess the impact of each requirement on project cost and delivery time
- Using an iterative approach, requirements are eliminated, combined and/or modified so that each party achieves some measure of satisfaction

The Art of Negotiation

- Recognize that it is not competition
- Map out a strategy
- Listen actively
- Focus on the other party's interests
- Don't let it get personal
- Be creative
- Be ready to commit

Specification Task

- A specification is the final work product produced by the requirements engineer
- It is normally in the form of a software requirements specification
- It serves as the foundation for subsequent software engineering activities
- It describes the function and performance of a computer-based system and the constraints that will govern its development
- It formalizes the informational, functional, and behavioral requirements of the proposed software in both a graphical and textual format

Typical Contents of a Software Requirements Specification

- Requirements
 - Required states and modes
 - Software requirements grouped by capabilities (i.e., functions, objects)
 - Software external interface requirements
 - Software internal interface requirements
 - Software internal data requirements
 - Other software requirements (safety, security, privacy, environment, hardware, software, communications, quality, personnel, training, logistics, etc.)
 - Design and implementation constraints
- Qualification provisions to ensure each requirement has been met
 - Demonstration, test, analysis, inspection, etc.
- Requirements traceability
 - Trace back to the system or subsystem where each requirement applies

Validation Task

- During validation, the work products produced as a result of requirements engineering are assessed for quality
- The specification is examined to ensure that
 - all software requirements have been stated unambiguously
 - inconsistencies, omissions, and errors have been detected and corrected
 - the work products conform to the standards established for the process, the project, and the product
- The formal technical review serves as the primary requirements validation mechanism
 - Members include software engineers, customers, users, and other stakeholders

Questions to ask when Validating Requirements

- Is each requirement consistent with the overall objective for the system/product?
- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?

Questions to ask when Validating Requirements (continued)

- Do any requirements conflict with other requirements?
- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?
 - Approaches: Demonstration, actual test, analysis, or inspection
- Does the requirements model properly reflect the information, function, and behavior of the system to be built?
- Has the requirements model been “partitioned” in a way that exposes progressively more detailed information about the system?

Requirements Management Task

- During requirements management, the project team performs a set of activities to identify, control, and track requirements and changes to the requirements at any time as the project proceeds
- Each requirement is assigned a unique identifier
- The requirements are then placed into one or more traceability tables
- These tables may be stored in a database that relate features, sources, dependencies, subsystems, and interfaces to the requirements
- A requirements traceability table is also placed at the end of the software requirements specification

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
2	Sno	Req ID	Req Desc	TC ID	TC Desc	Test Design	Test Designer	UAT Test Req?	Test Execution			Defects?	Defect ID	Defect Status	Req Coverage Status	
3									Test Env	UAT Env	Prod Env					
4					Login with Invalid Username and valid password			No	Passed	No Run	No Run	None	None	N/A		
5	1	Req01	Login to the Application	TC01	Completed	XYZ									Partial	
6	2			TC02	Login with Valid Username and invalid password			No	Passed	No Run	No Run	None	None	N/A	Partial	
7	3			TC03	Login with valid credentials	Completed	YZA	Yes	Passed	Passed	No Run	Yes	DFCT001	Test OK	Partial	
8																

Requirements Management...

<https://www.guru99.com/traceability-matrix.html>

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

https://www.youtube.com/watch?time_continue=15&v=cm-cSW66lsc&feature=emb_logo

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)

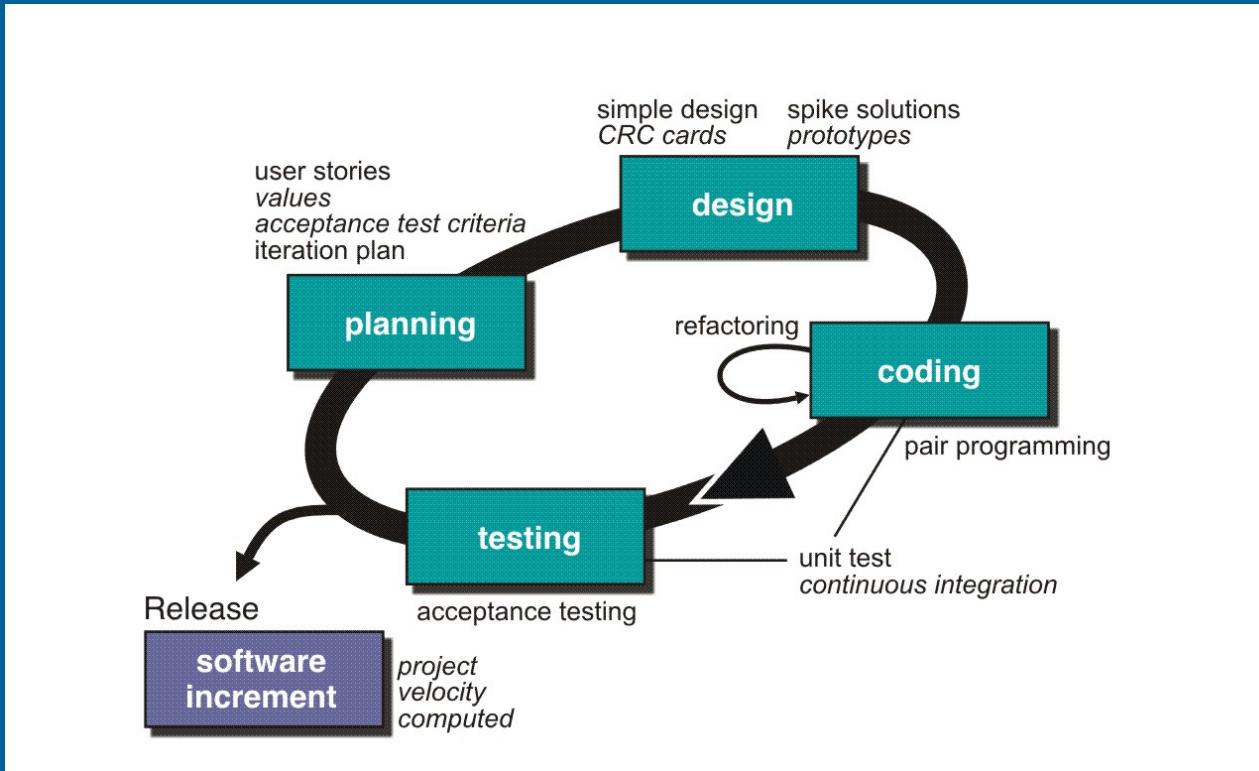
The most widely used agile process, originally proposed by Kent Beck [BEC99]

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



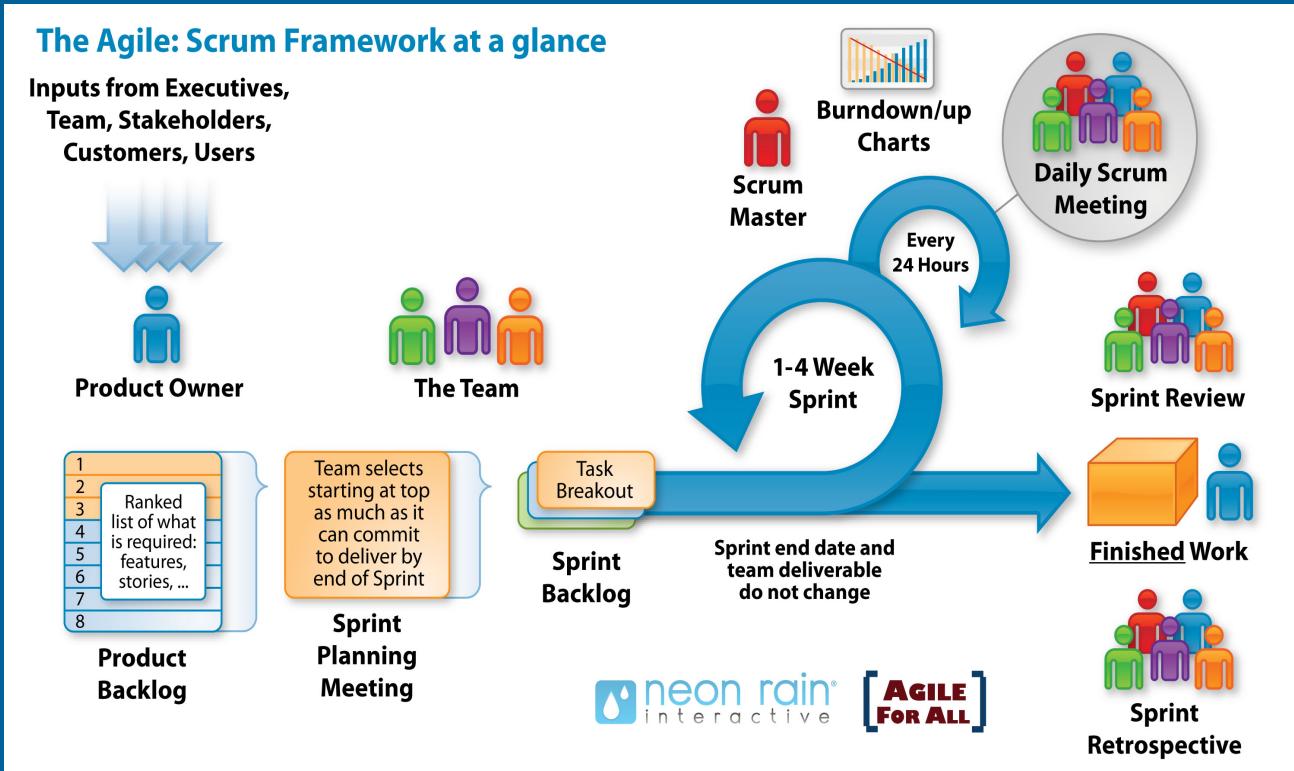
Scrum

A software development method Originally proposed by Schwaber and Beedle (an activity occurs during a rugby match) in early 1990.

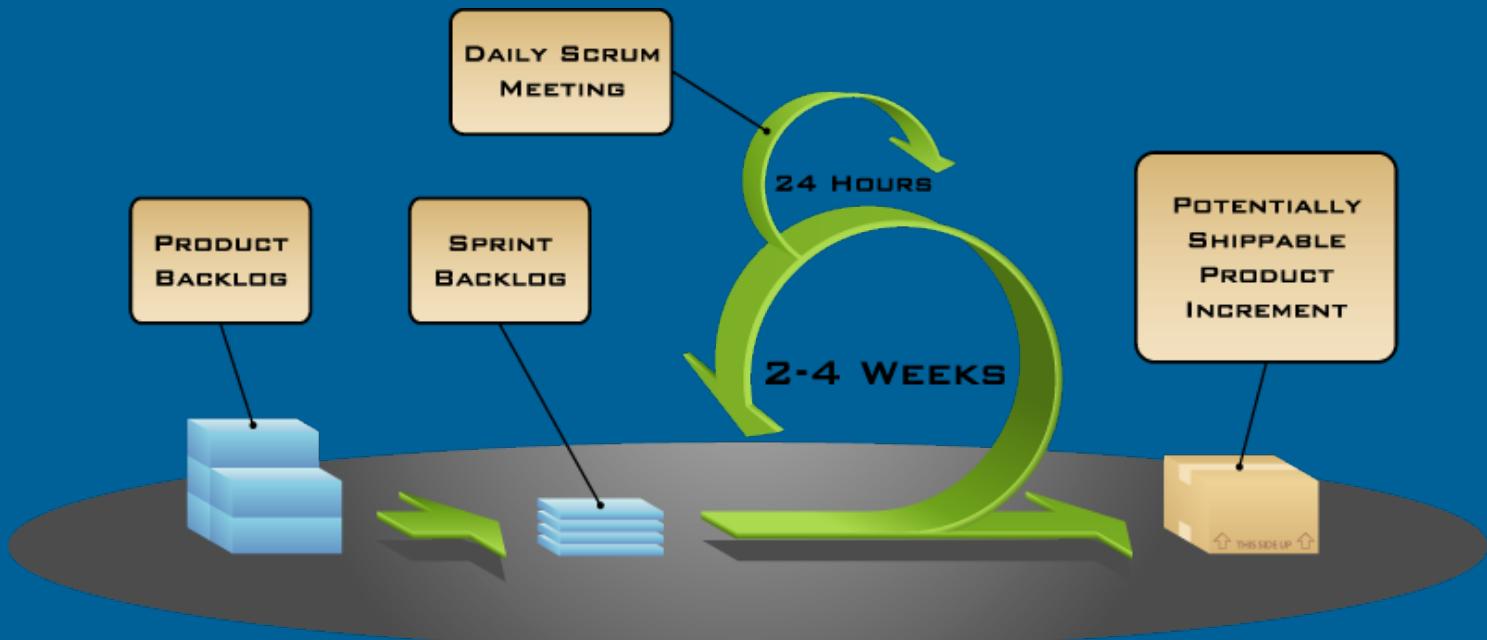
Scrum—distinguishing features

- 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

Scrum's core values

Commitment. Because we have great control over our own destiny, we become more committed to success.

Focus. Because we focus on only a few things at a time, we work well together and produce excellent work. We deliver valuable items sooner.

Openness. As we work together, we practice expressing how we're doing and what's in our way. We learn that it is good to express concerns so that they can be addressed.

Respect. As we work together, sharing successes and failures, we come to respect each other and to help each other become worthy of respect.

Courage. Because we are not alone, we feel supported and have more resources at our disposal. This gives us the courage to undertake greater challenges

Other Agile Processes

Adaptive Software Development (ASD)

**Dynamic Systems Development Method
(DSDM)**

Crystal

Feature Driven Development

Agile Modeling (AM)

Process Models

Perspective Models: The Waterfall

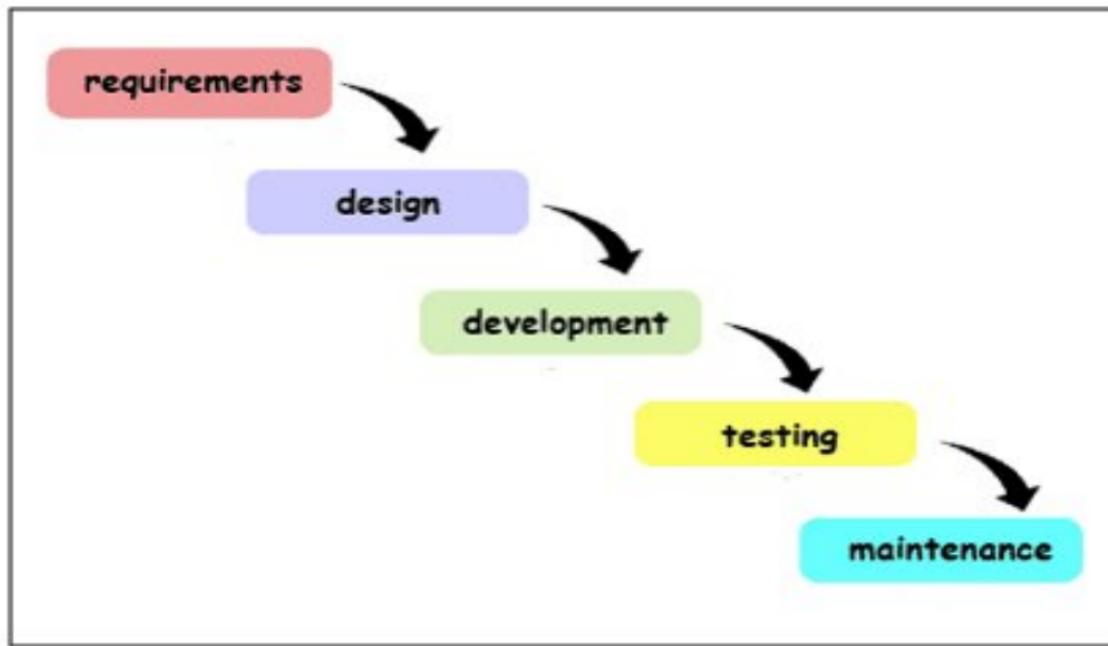
Incremental Models: Increment and RAD

Evolutionary Model: Prototype and Spiral

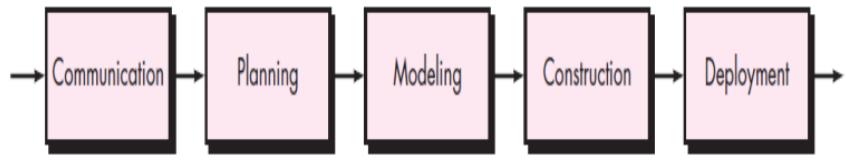
Specialized Process Models: Component, Formal Methods, AOSD

Unified Process Model

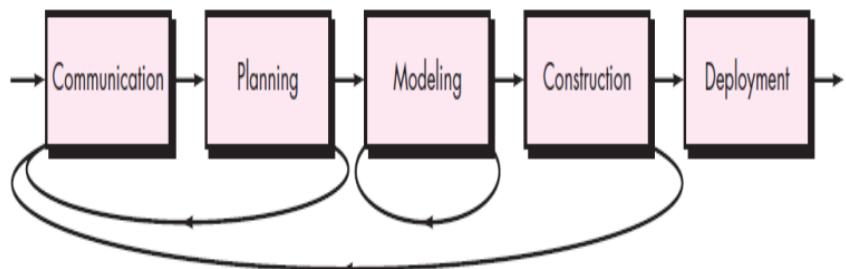
Software Development Life Cycle (SDLC) Phases



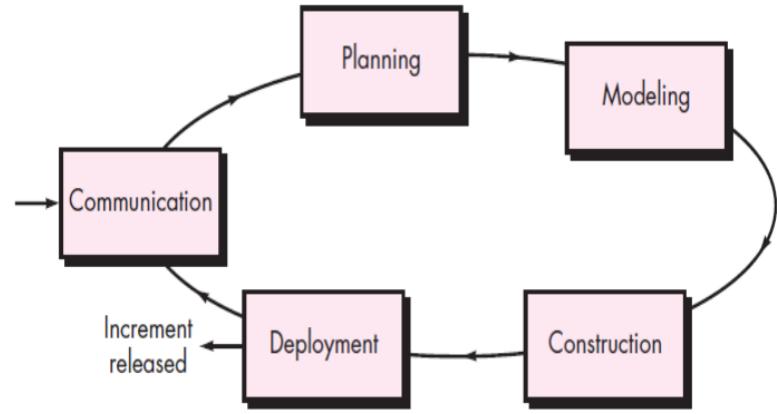
Process Flow



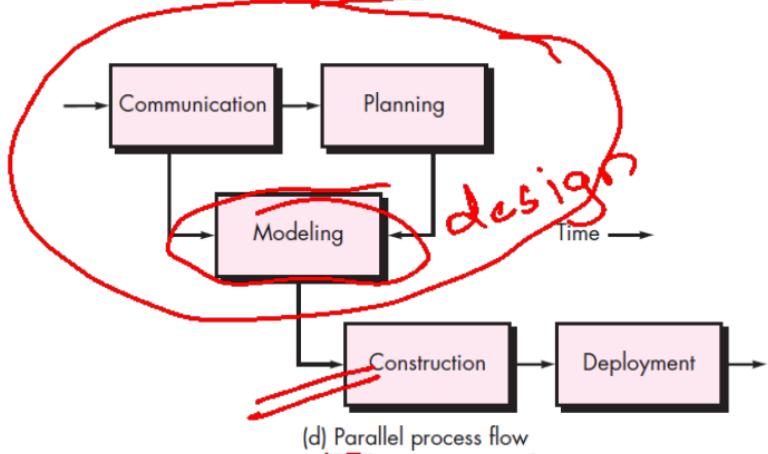
(a) Linear process flow



(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** *sequential*
- Various process models differ in their emphasis on different activities and workflow

The Waterfall Model

Sometimes called the *classic life cycle*

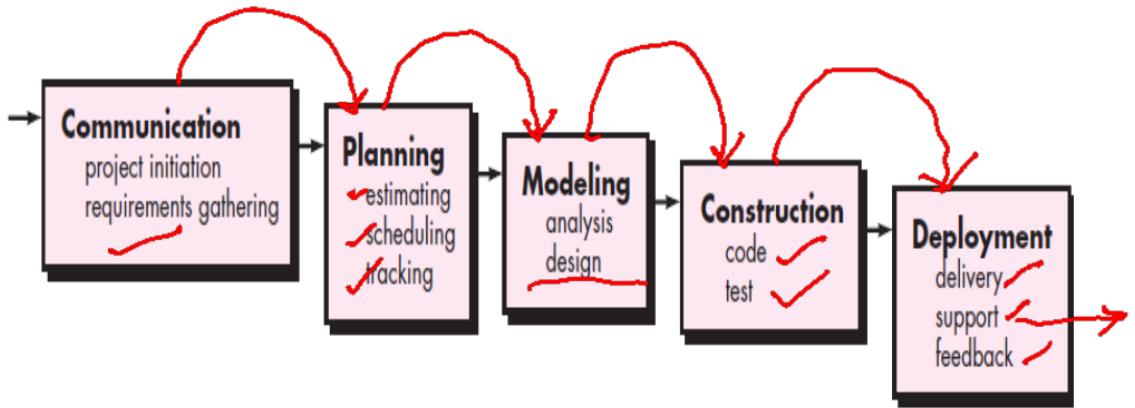
Suggests a systematic, sequential (or linear) approach to s/w development

The oldest paradigm for s/w engineering

Works best when –

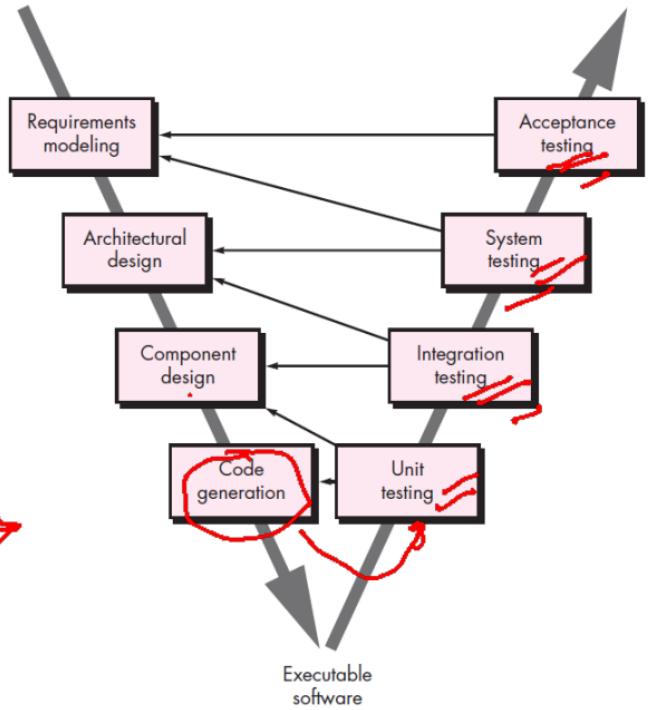
- Requirements of a problem are reasonably well understood
- Well-defined adaptations or enhancements to an existing system must be made
- Requirements are well-defined and reasonably stable
- Technology is understood
- There are no ambiguous requirements
- Ample resources with required expertise are 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

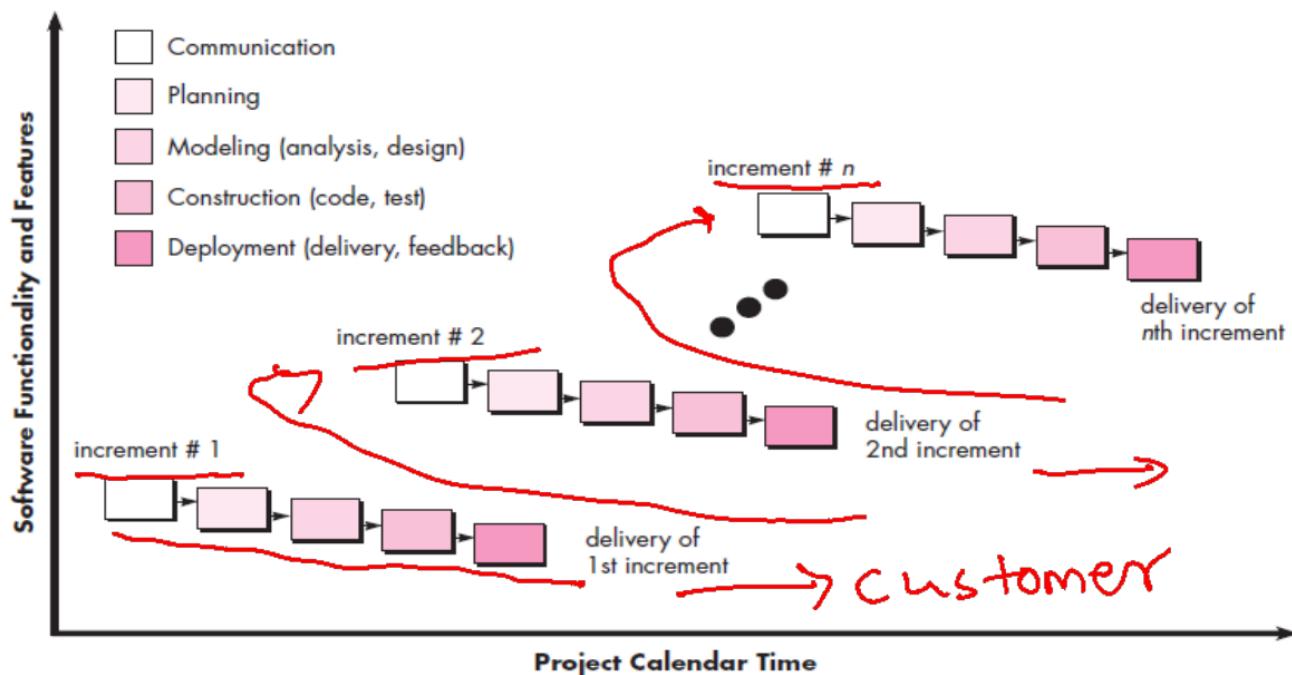
- Has difficulty accommodating the natural uncertainty that exists at the beginning of many projects

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**

→ Basic product

→ increment

The RAD Model

Rapid Application Development is Incremental Process Model

Emphasizes on short development cycle

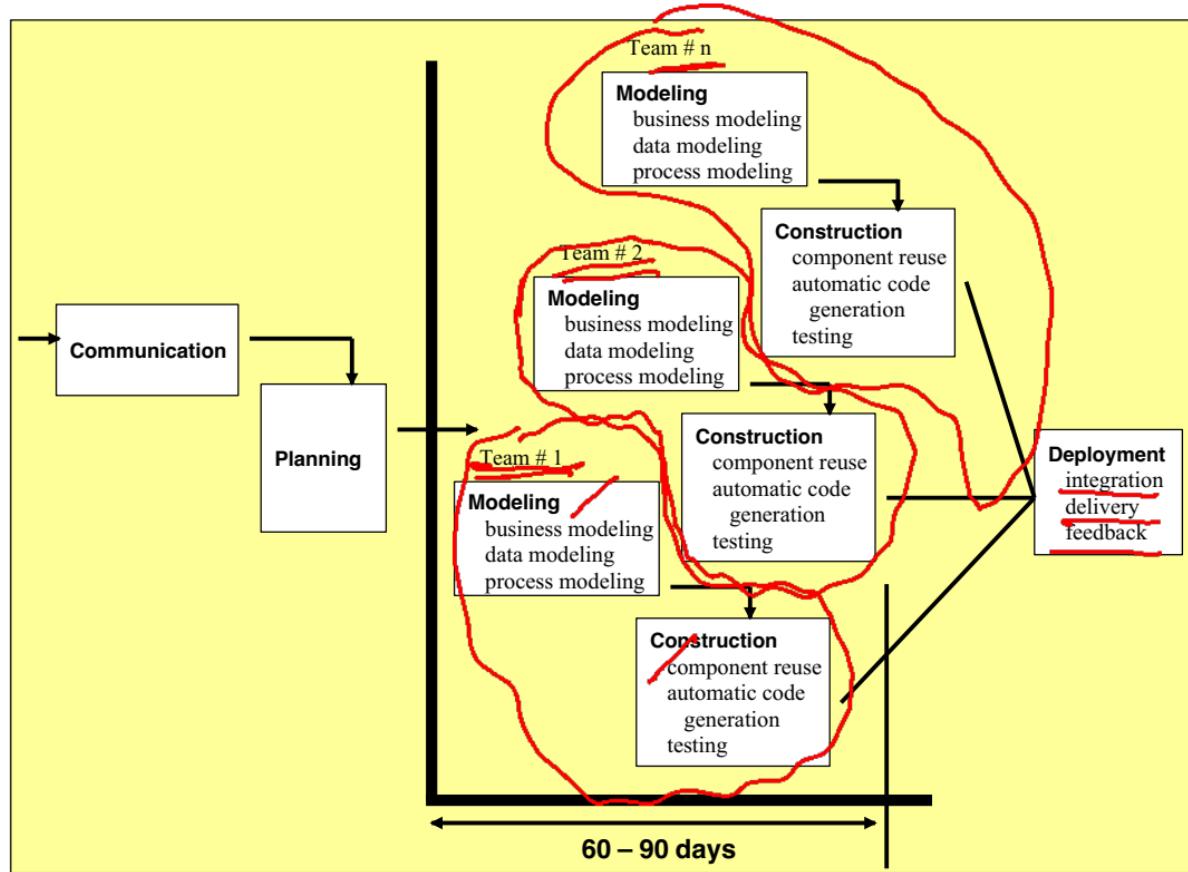
A “high speed” adaptation of the waterfall model

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

Reusability

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

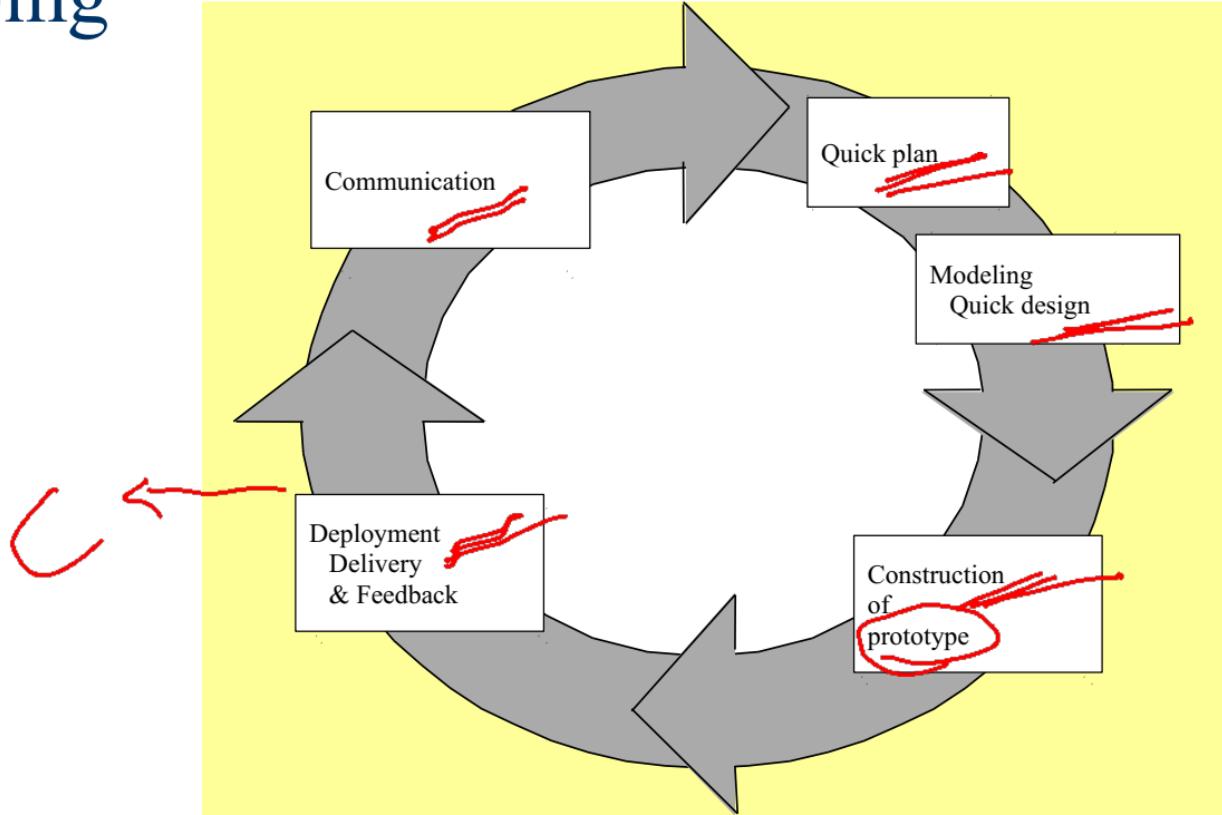
Evolutionary models are iterative

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

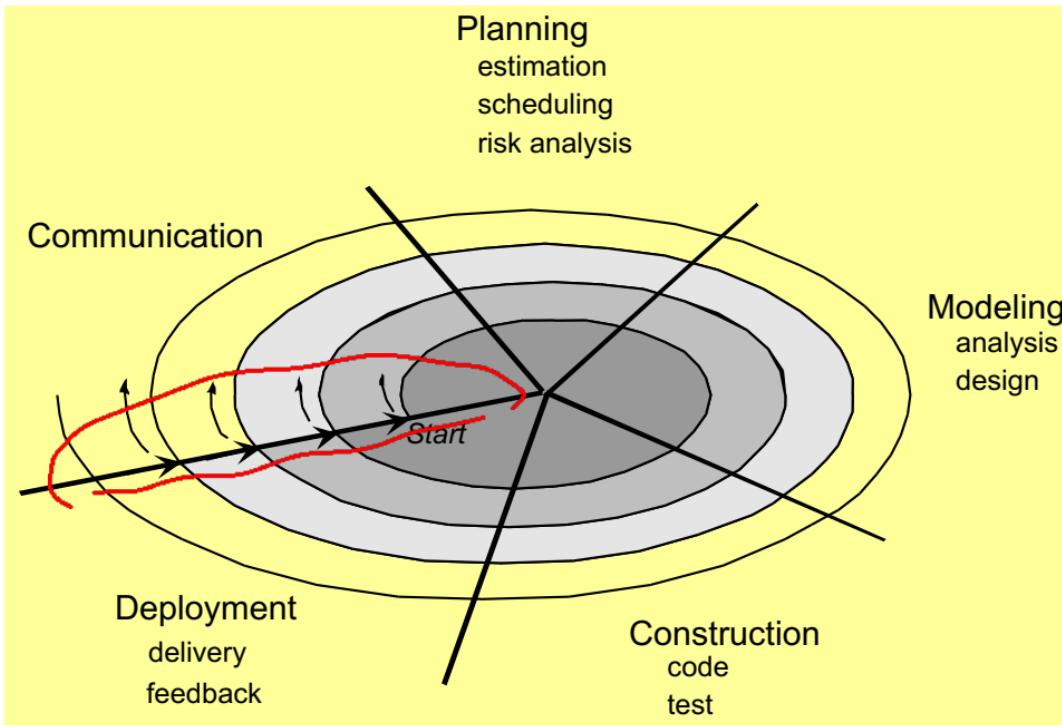
It provides the potential for rapid development of increasingly more complete versions of the software

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

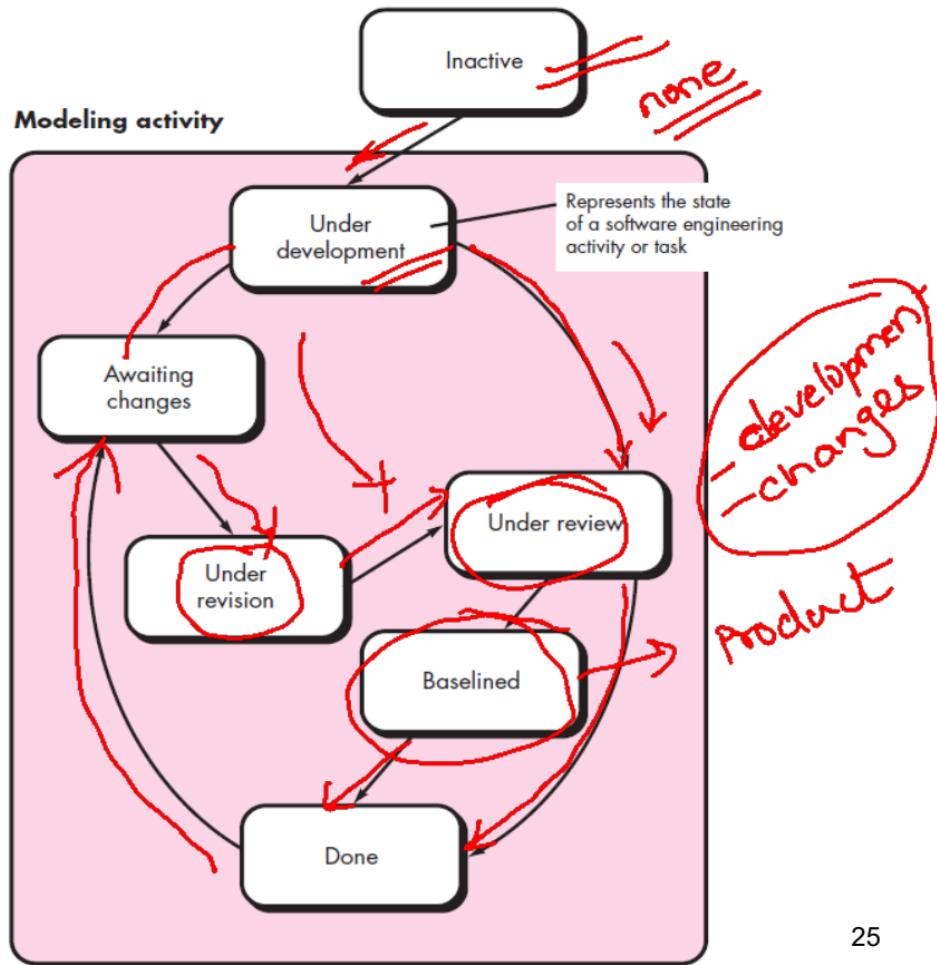
Defines a network of activities

Events generated at one point in the process network trigger transitions among the states



events, actions

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
- 

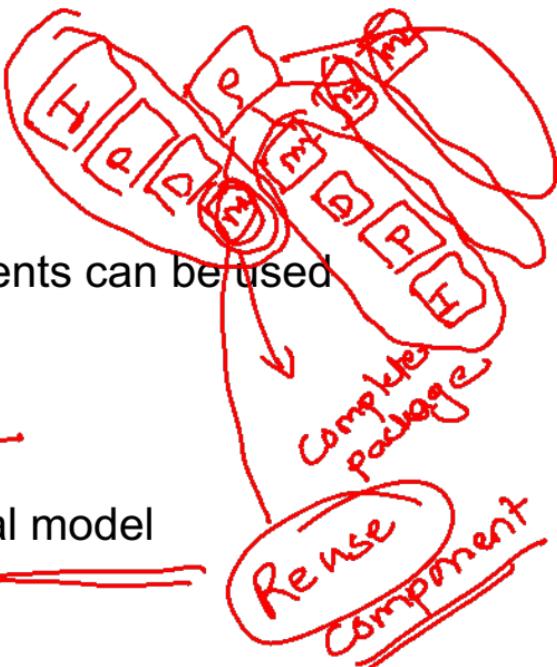
Component-Based Development

Commercial off-the-shelf (COTS) software components can be used

Components should have well-defined interfaces

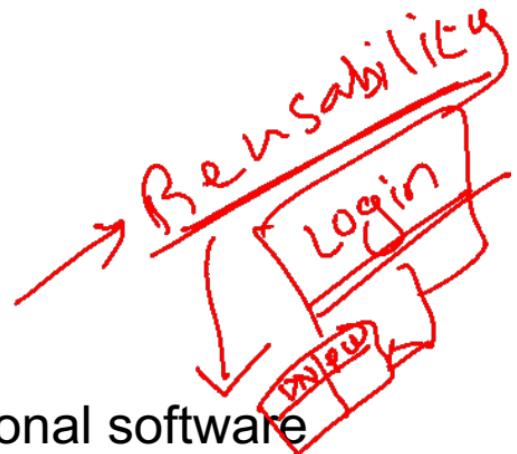
Incorporates many of the characteristics of the spiral model

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

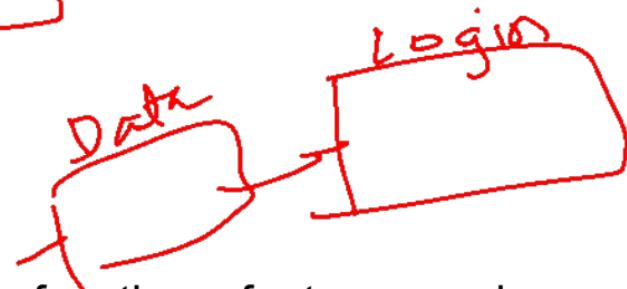
It is difficult to use the models as a communication mechanism for technically unsophisticated customers

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)

Rational Unified Process (RUP)
Object-oriented

It is a use-case driven, architecture-centric, iterative and incremental software process

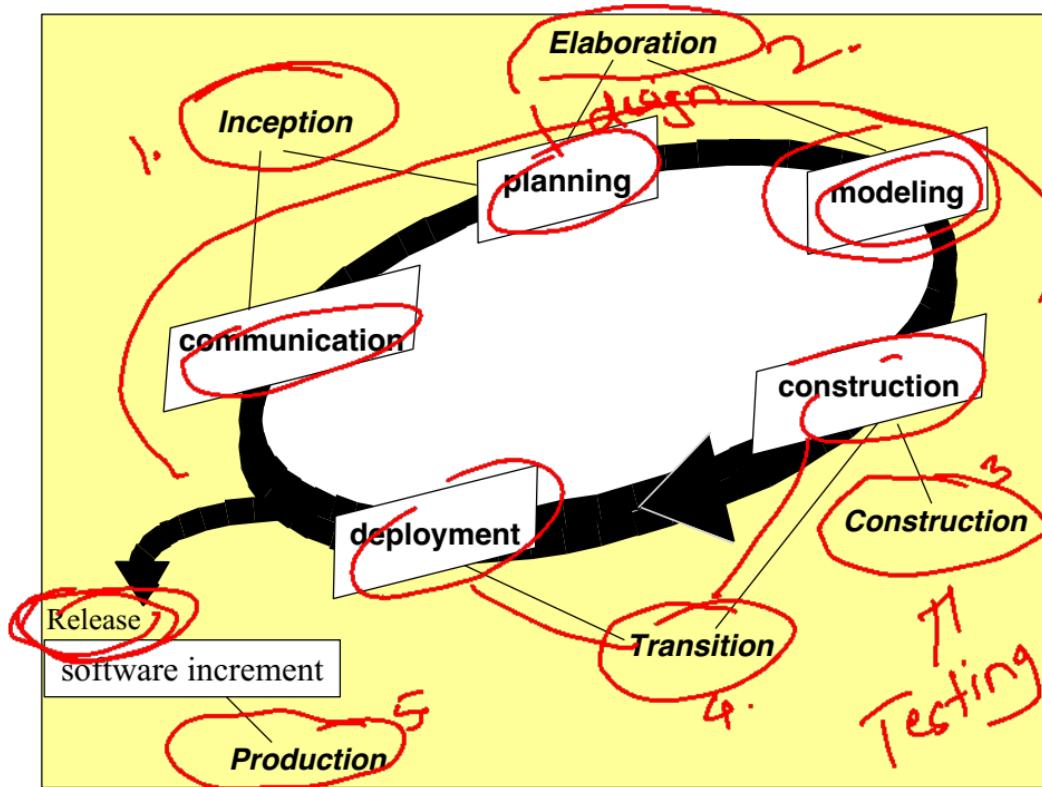
UP is an attempt to draw on the best features and characteristics of conventional s/w process models

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)

design

Phases of the Unified Process



Phases of UP - Inception

→ Use-cases
requirements

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

Unit

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

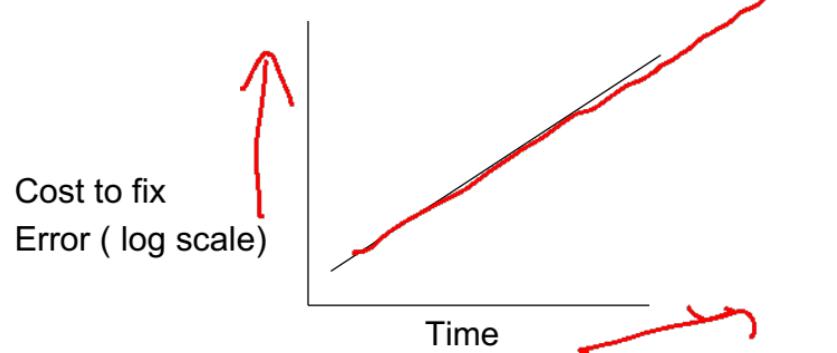
- Req. - 20%
- Design - 30%
- Coding - 50%



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



Class Based Modeling

Approaches for Identifying Classes

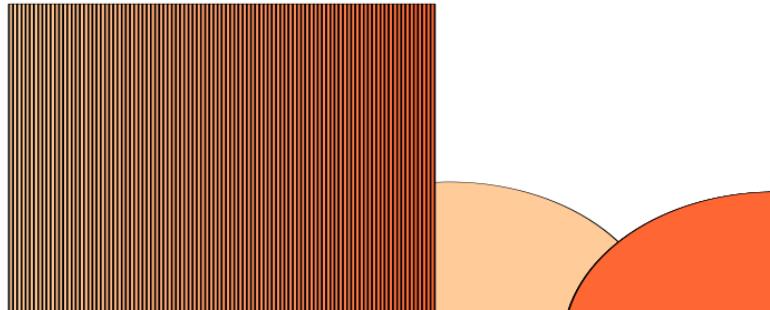
- The noun phrase approach.
- The common class patterns approach.
- The class responsibilities collaboration (CRC) approach.
- The use-case driven approach.

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:

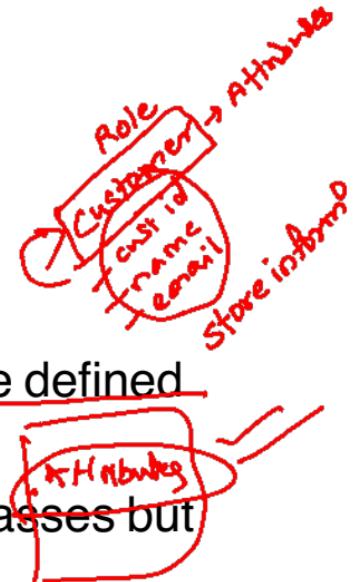
- Does the object represented by the noun behave differently when the adjective is applied to it?
- If the use of the adjective signals that the behavior of the object is different, then make a new class.
- For example, If Adult Membership and Youth Membership behave differently, than they should be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

Actor



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.

No. of copies

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 phases the 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

member
Book
SE Pressman

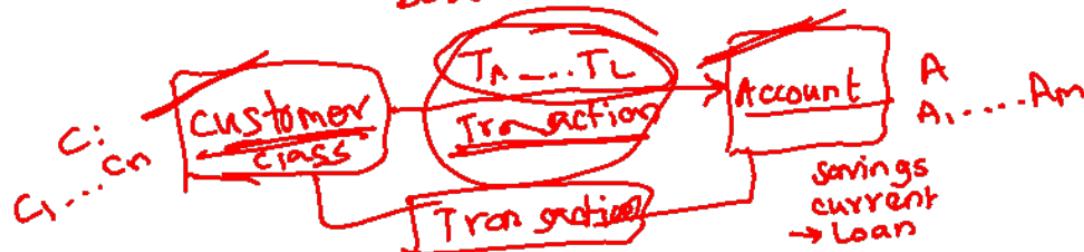
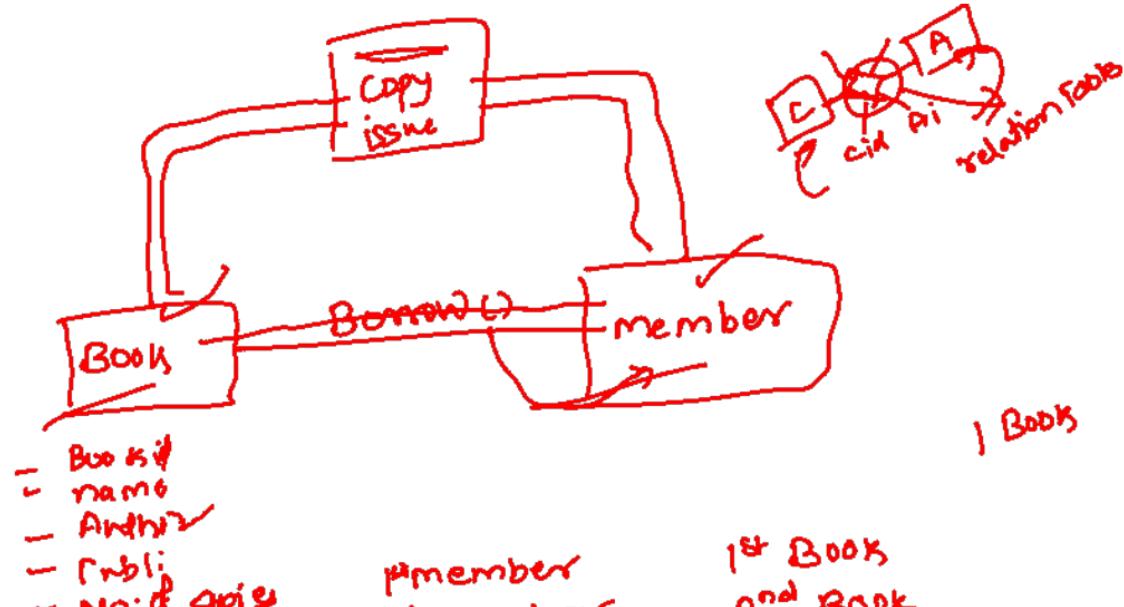
4/20

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
patient
type of ward
male ward
female ward
ward
sex
capacity (maximum number of patients, number of beds)
name of ward
Administration department
information

doctor
team
team code
Orthopaedics A
Paediatrics
grade 1
consultant doctor (consultant)
junior doctor
record of teams and doctors care
number of doctors
number of patients

Potential classes

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

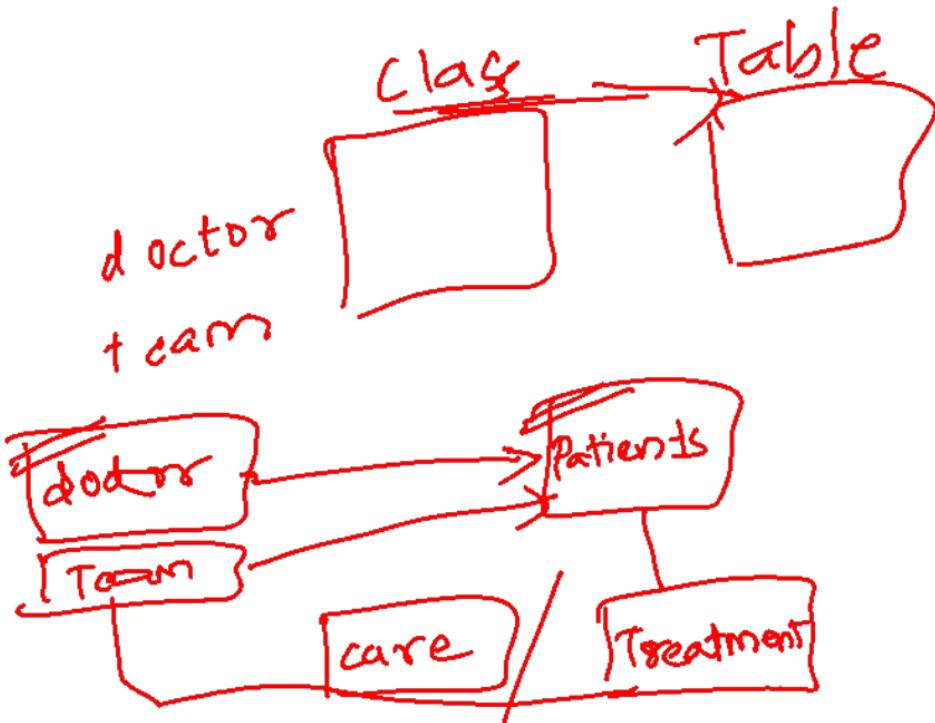
patient

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.



CRC Modeling

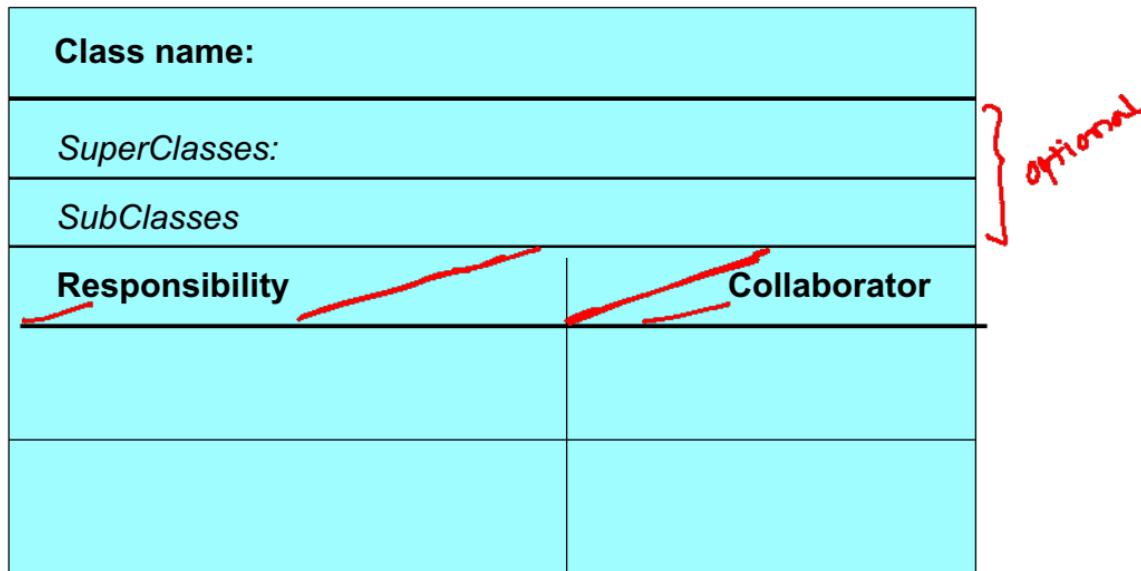
Class Responsibility Collaborator (CRC)

- CRC (class responsibility collaborator) modeling process for identifying user requirements.
- CRC modeling is an effective, low-tech method for developers and users to work closely together to identify and understand business requirements.
- A CRC card is an index card that is used to represent the responsibilities of classes and the interaction between the classes.
- CRC cards are an informal approach to object oriented modeling.
- The cards are created from use-case scenarios, based on the system requirements.

The CRC card Session

- Groups consisting of five or six people.
- Each group typically consists of **developers, domain experts** and an **OO technology facilitator**.

A CRC card



Inventory Item	
Item number	
Name	
Description	
Unit Price	
Give price	
R	C

Order	
Order number	Order Item
Date ordered	Customer
Date shipped	
Order items	
Calculate order total	
Print invoice	
Cancel	

Order Item	
Quantity	Inventory item
Inventory item	
Calculate total	?

Customer	
Name	Order
Phone number	Surface Address
Customer number	
Make order	G
Cancel order	
Make payment	
R	

Surface Address	
Street	
City	
State	
Zip	
Print label	

Class

- A class represents a collection of similar objects. An object is a person, place, thing, event, concept, screen, or report that is relevant to the system at hand.
- For example, a shipping/inventory control system with the classes such as Inventory Item, Order, Order Item, Customer, and Surface Address.
- The name of the class appears across the top of the card.

Responsibility

- A responsibility is anything that a class knows or does. For example, customers have names, customer numbers, and phone numbers.
- These are the things that a customer knows. Customers also order products, cancel orders, and make payments.
- These are the things that a customer does.
- The things that a class knows and does constitute its responsibilities.
- Responsibilities are shown on the left hand column of a CRC card.

Collaborator

- Sometimes a class will have a responsibility to fulfill, but will not have enough information to do it.
- When this happens it has to collaborate with other classes to get the job done.
- For example, an **Order** object has the responsibility to calculate its total. Although it knows about the **Order Item** objects that are a part of the order, it doesn't know how many items were ordered (**Order Item** knows this) nor does it know the price of the item (**Inventory Item** knows this). To calculate the order total, the **Order** object collaborates with each **Order Item** object to calculate its own total, and then adds up all the totals to calculate the overall total.
- For each **Order Item** to calculate its individual total, it has to collaborate with **Inventory Item** to determine the cost of the ordered item, multiplying it by the number ordered (which it does know). The collaborators of a class are shown in the right-hand column of a CRC card.

CRC Models

- A CRC model is a collection of CRC cards that represent whole or part of an application or problem domain.
- The most common use for CRC models, the one that this white paper addresses, is to gather and define the user requirements for an object-oriented application.
- Example CRC model for a shipping/inventory control system, showing the CRC cards as they would be placed on a desk or work table.
- Note the placement of the cards: Cards that collaborate with one another are close to each other, cards that don't collaborate are not near each other.
- CRC models are created by groups of business domain experts, led by a CRC facilitator who is assisted by one or two scribes.
- The CRC facilitator is responsible for planning and running the CRC modeling session.

CRC Cards

- **Step 1:** Identify the classes in the problem domain.
 - Use the problem statement or **requirements document** to find all of the nouns and verbs in the problem statement.
 - The nouns represent the object/classes in the system; the verbs may show what their responsibilities are.

CRC Cards

- **Step 2:** Take at least one card per person. Each person should be responsible for at least one class.
- **Step 3:** Add the class name and add responsibilities that are obvious from the requirements. Attributes typically are not added at this time. Add super or subclasses that are obvious.

CRC Cards

- **Step 4:** Walk-through a scenario that represents an important system function in the requirements document.

Decide which class (es) is responsible for this function. The owner of the class then picks up her card and announces that she needs to fulfill this responsibility.

The responsibility may be refined into smaller tasks if possible.

These smaller tasks can be fulfilled by the same object or they can be fulfilled by interacting with other objects.

If no appropriate class (to ~~fulfill this responsibility~~) exists, you may need to make a class.

Class: Book

Responsibilities

- knows whether on loan
- knows due date
- knows its title
- (1)** - knows its author(s)
- knows its registration code
- knows if late
- check out

Collaborators

Date

Class: Borrower

Responsibilities

- knows its name
- keeps track of borrowed items
- keeps track of overdue fines

Collaborators

(3)

Class: Librarian

Responsibilities

- check in book
- check out book
- search for book
- knows all books
- search for borrower
- knows all borrowers

Collaborators

- Book
- Book, Borrower
- Book
- Borrower

(2)



(4)

Class: Date

Responsibilities

- knows current date
- can compare two dates
- can compute new dates

Collaborators

Example-Vending Machine

Let us take an example application and examine the process of identifying the objects/classes and describe their responsibilities using CRC cards.

The example application is a **Vending Machine** that allows users to **buy snack items**. In addition, a user can find out the caloric content of her choice.

Example-Vending Machine

Specification :

A Vending machine holds a number of snack items and displays the list of snack items and their prices through an user interface with a display screen and buttons for making selections. In addition, the vending machine has a receptacle for money and an item dispenser.

A user can make a selection and query for the number of calories of a snack item. The calories are displayed on pressing a button. A user can place the money in the receptacle and select an item.

Example-Vending Machine

- Let us select the nouns and the verbs in the specification.
- Nouns are in blue and the verbs are in green.
- A *Vending machine* holds a number of *snack items* and displays the *list of snack items* and their *prices* through an *user interface* with a *display screen* and *buttons* for *making selections*. In addition, the vending machine has a *receptacle* for *money* and an *item dispenser*.
- A *user* can *make a selection* and *query* for the number of *calories* of a snack item. The calories are *displayed* on *pressing a button*. A user can *place* the money in the *receptacle* and select an item.

Example-Vending Machine

- Most of the nouns are objects/classes.
- Some nouns are attributes of these classes.
- The verbs are actions that can be attached to these objects.
- In order to focus on the problem-domain objects, let us separate the object/classes into presentation-specific (user-interface related) and problem-specific classes.
- We will then select two problem specific classes and write the CRC cards for them.

Example-Vending Machine

Problem-specific classes:

- Vending Machine
- Snack item
- *Price*
- *Calories*
- *Selection*
- *User*

Presentation-specific classes:

- Display screen
- Selection Buttons
- Item Dispenser
- Money receptacle

A CRC card for class **SnackItem**

Class name: SnackItem	
Responsibility	Collaborator
Knows its price and calories	

A CRC card for the Vending Machine

Class name: VendingMachine	
Responsibility	Collaborator
Maintains a collection of SnackItems. Allows addition and removal of SnackItems	SnackItem

Showing the Collaboration of a `VendingMachine` and `SnackItem`



Exercise

Problem Statement

An **Automated Teller Machine (ATM)** allows **bank customers** to perform a number of financial transactions: to **withdraw** and **deposit funds** to an account, query the balance of any account. The ATM offers an user interface with a display screen, keypad, cash dispenser, deposit slot and a card reader.

Once a customer's card is verified, the customer can **query to see the balance** in all her account (s), **deposit**, **withdraw** or **transfer** money from one account into another.

Using Object-Oriented analysis and design techniques, build an object model for the ATM machine.

Exercise

- Identify the **classes** from the problem statement and describe their responsibilities using CRC cards.

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 shows

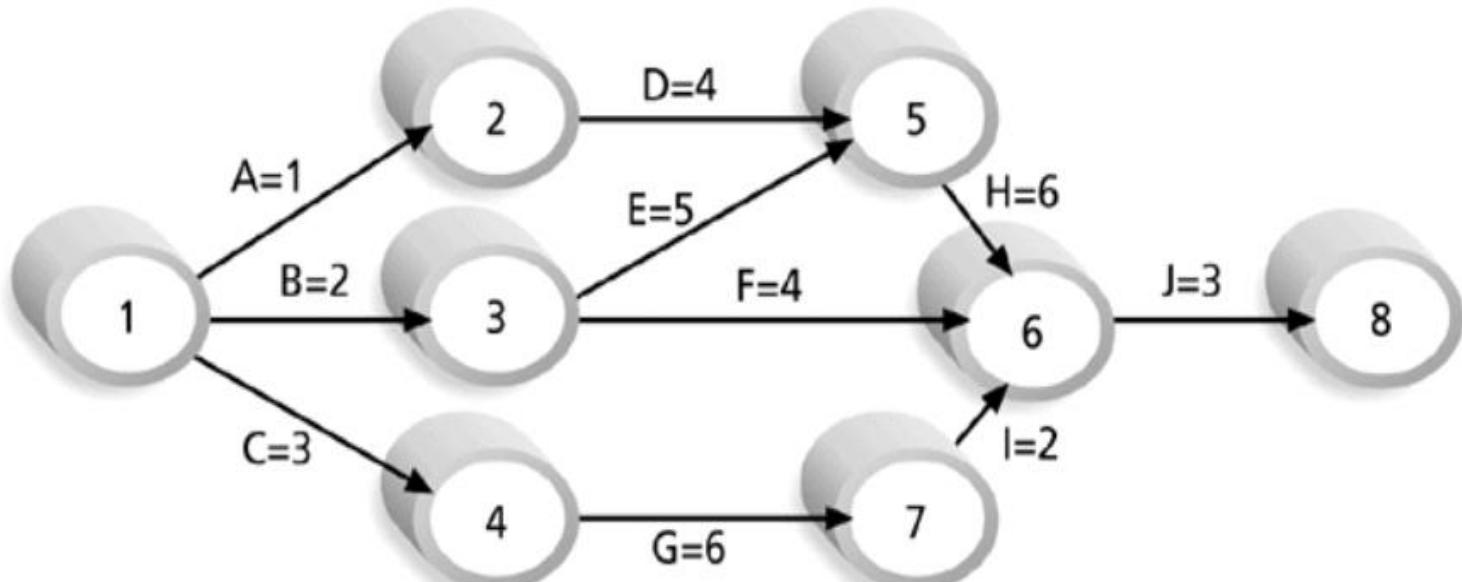
Activity Network

-  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

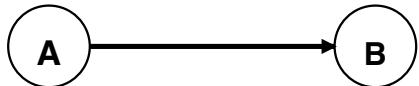
- There are three major tasks to perform:
 - Manufacture the new computer.
 - Train staff and vendor representatives.
 - Advertise the new computer.
- KLONE needs to develop a precedence relations chart.
- The chart gives a concise set of tasks and their immediate predecessors.

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

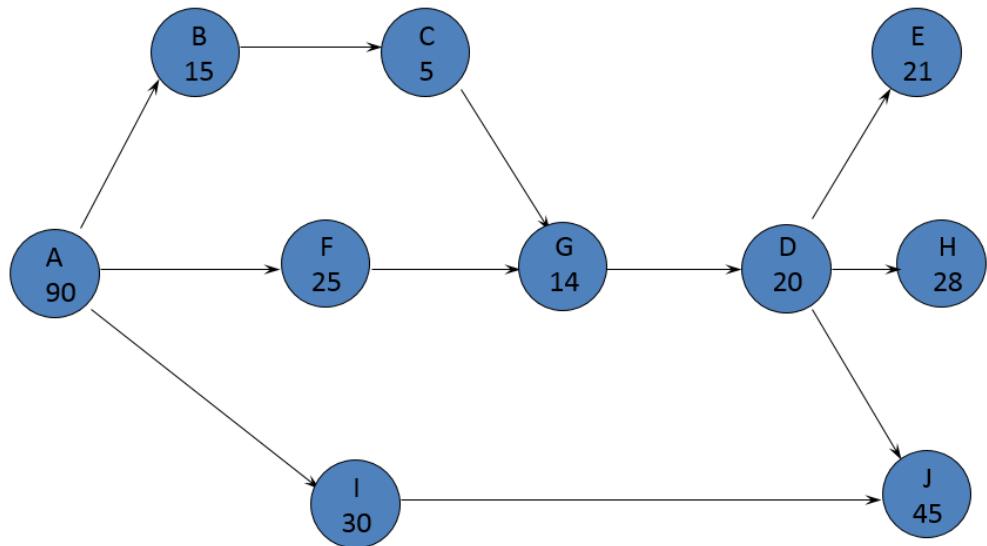
KLONE COMPUTERS, INC

**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



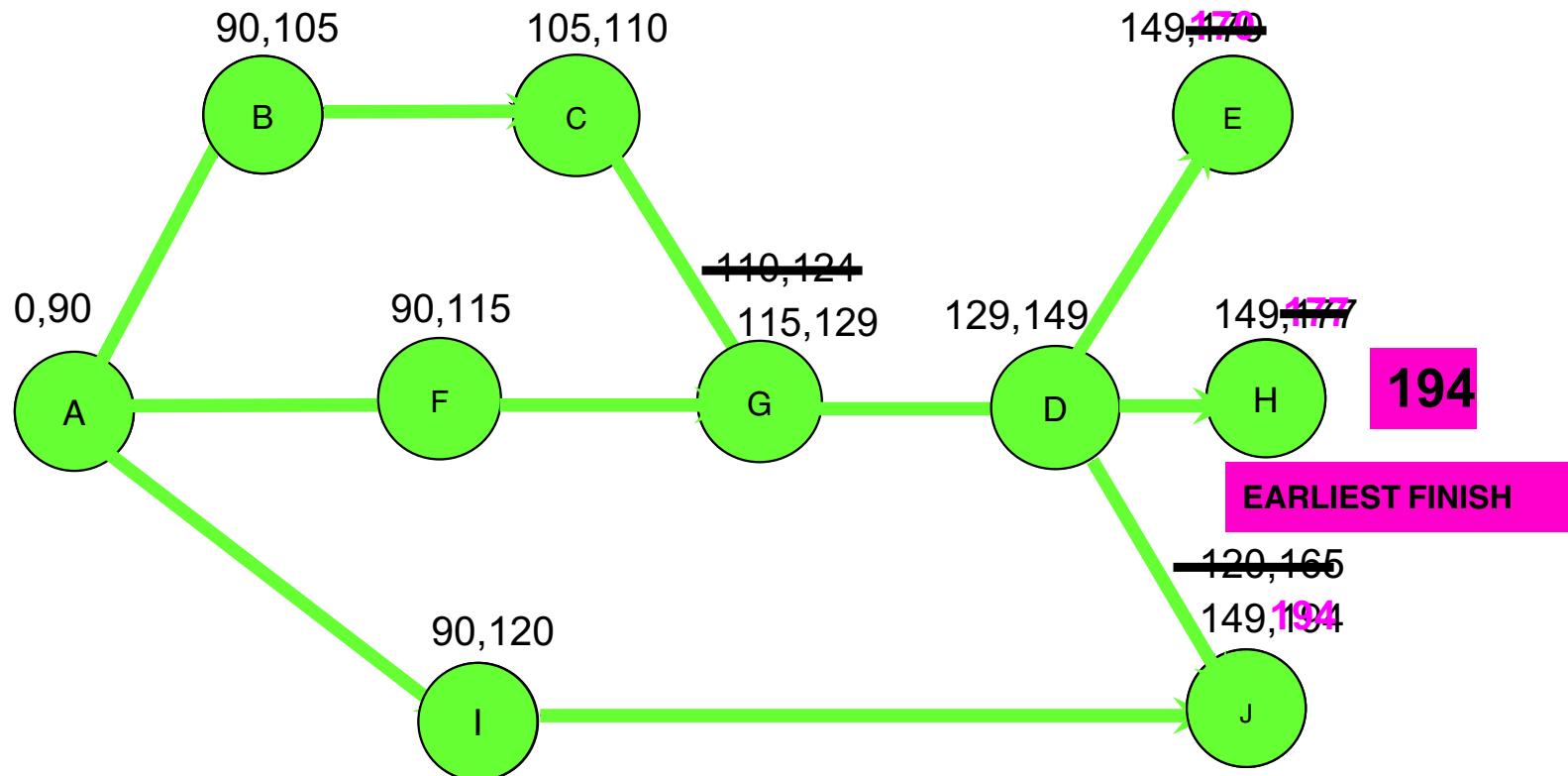
KLONE COMPUTERS, INC. - Continued

- 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.
 - $ES = \text{Max } EF \text{ of all its immediate predecessors.}$
 - $EF = ES + \text{Activity duration.}$
 - Repeat this process until all nodes have been evaluated
 - EF of the finish node is the earliest finish time of the project.

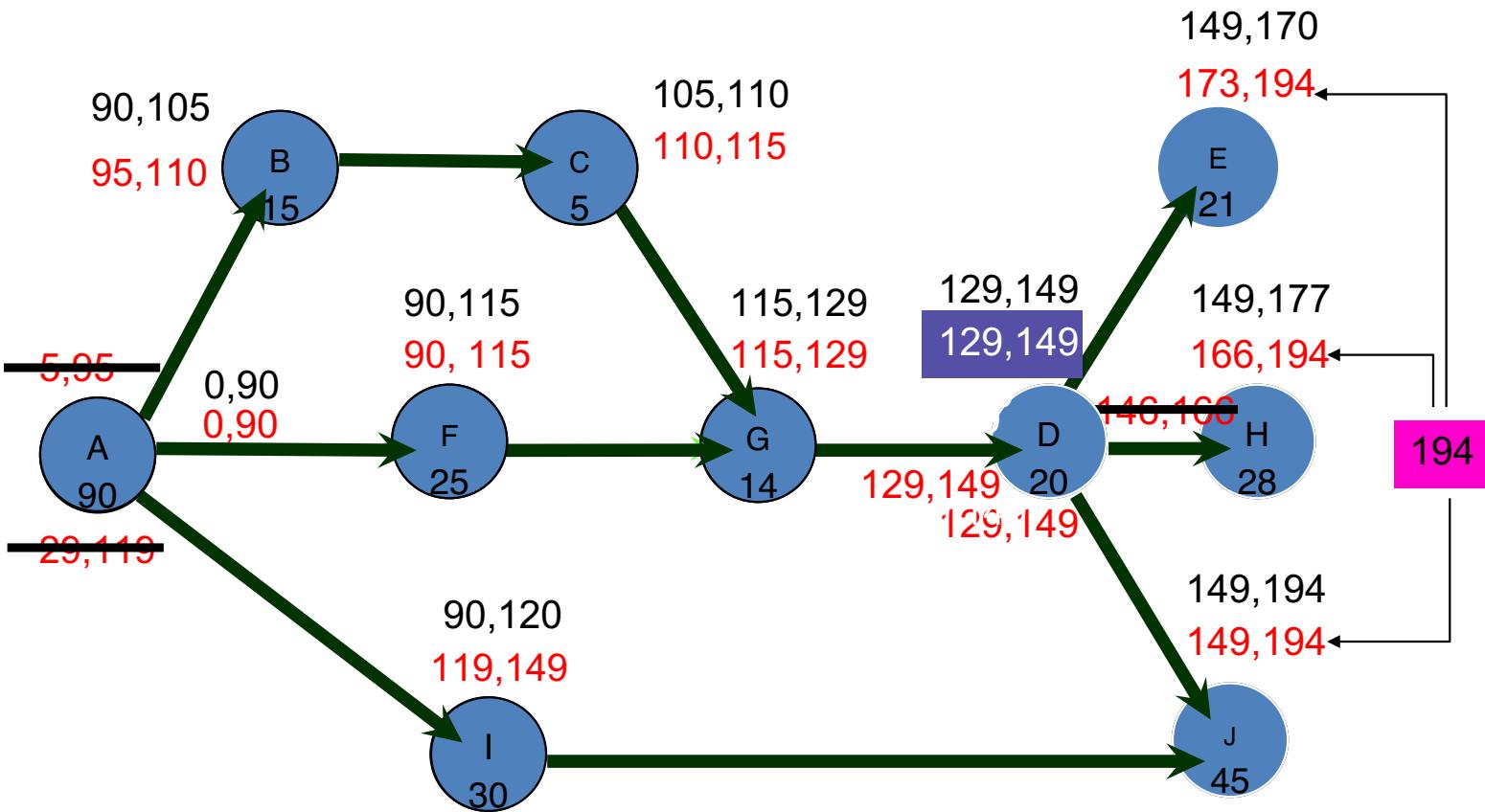
Earliest Start / Earliest Finish – Forward Pass



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

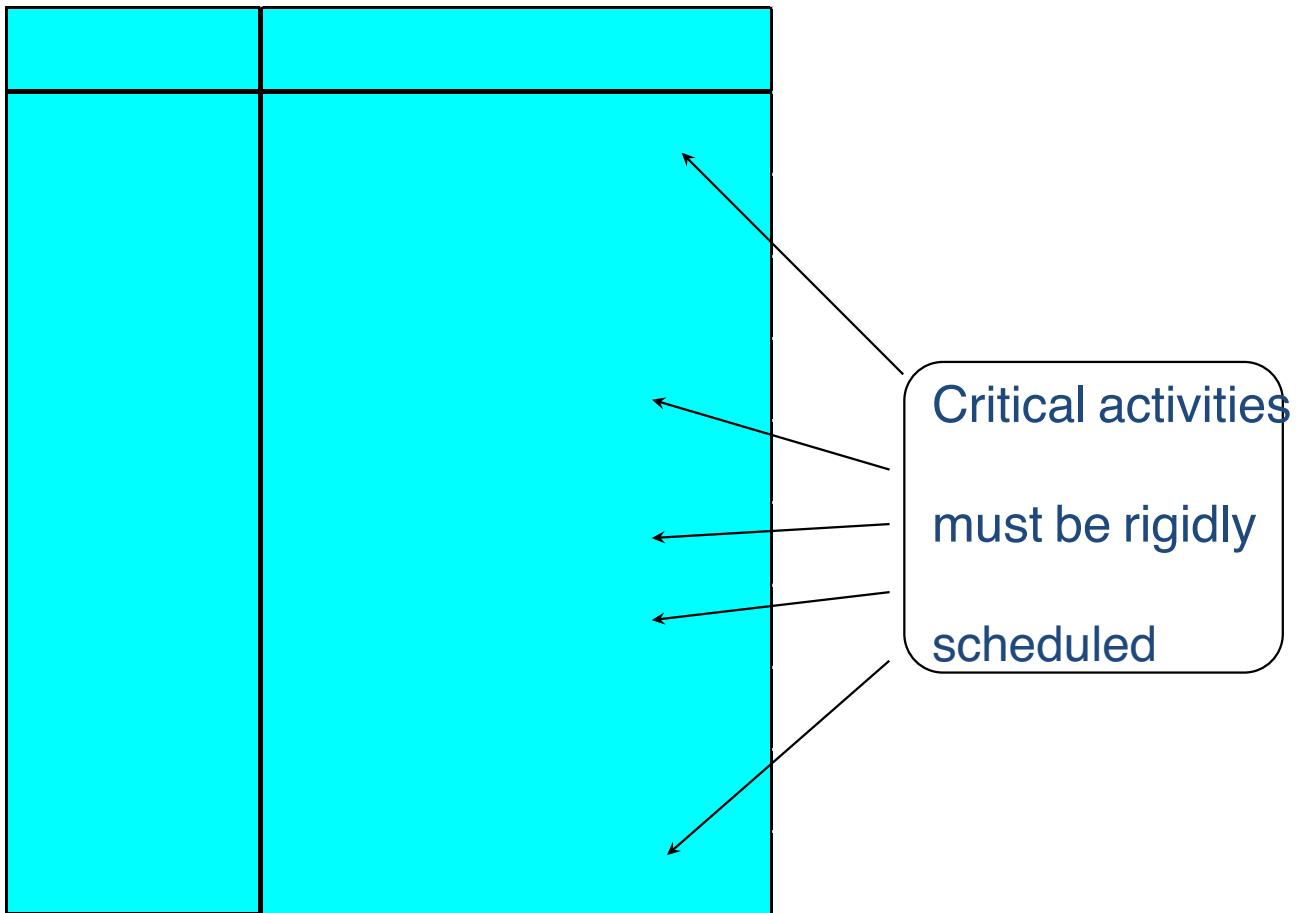
- Activity start time and completion time may be delayed by planned reasons as well as by unforeseen reasons.
- Some of these delays may affect the overall completion date.
- To learn about the effects of these delays, we calculate the **slack time**, and form the **critical path**.

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.

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

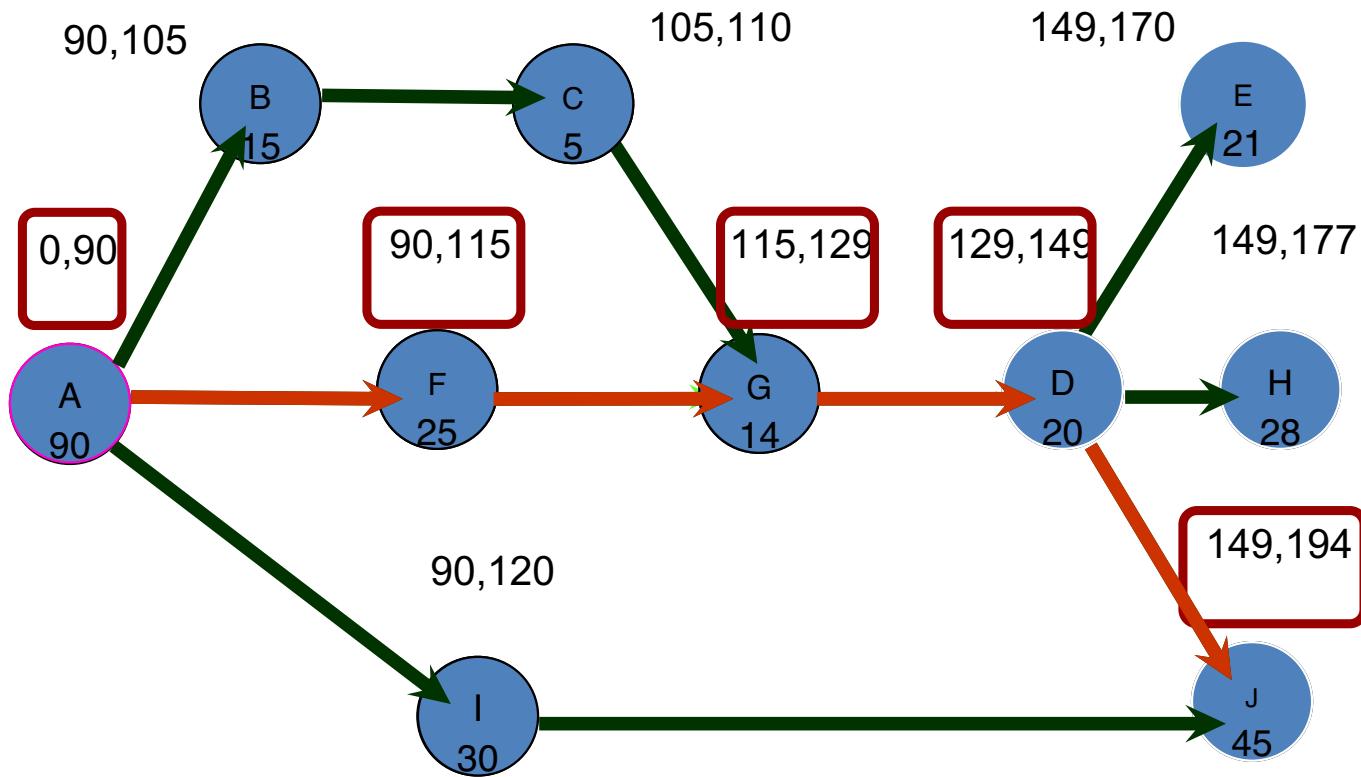
Slack time in the Klonepalm 2000 Project



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



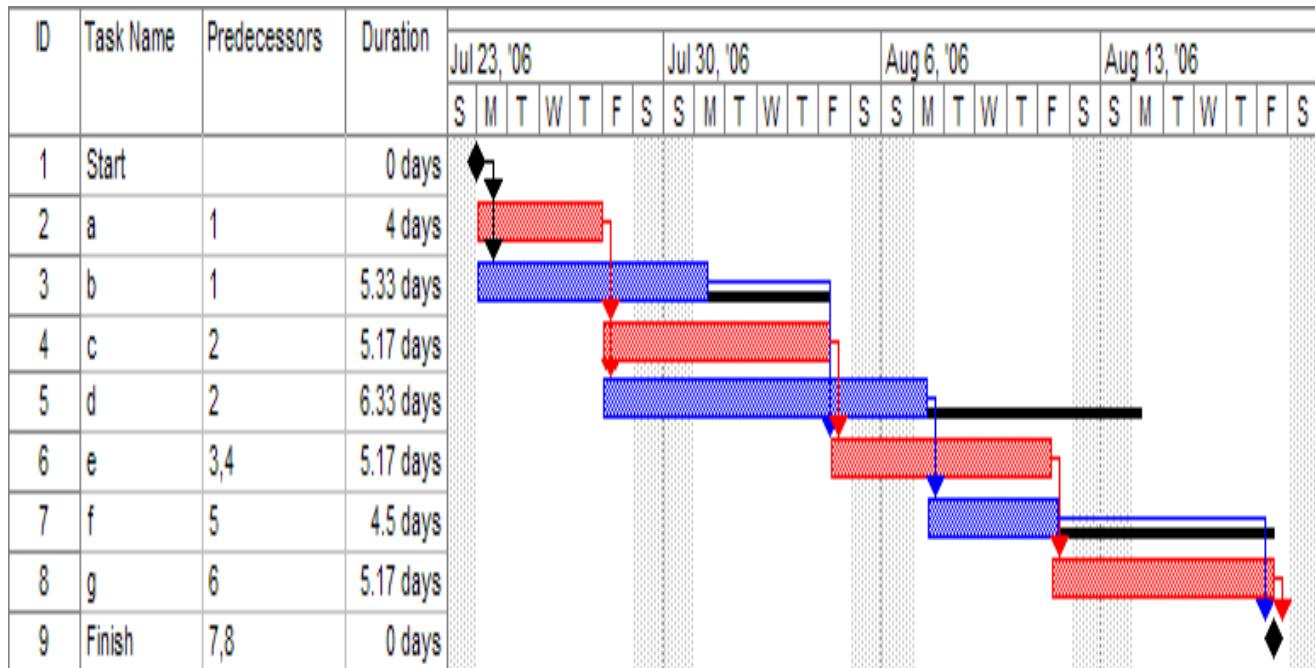
Gantt Chart

- 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

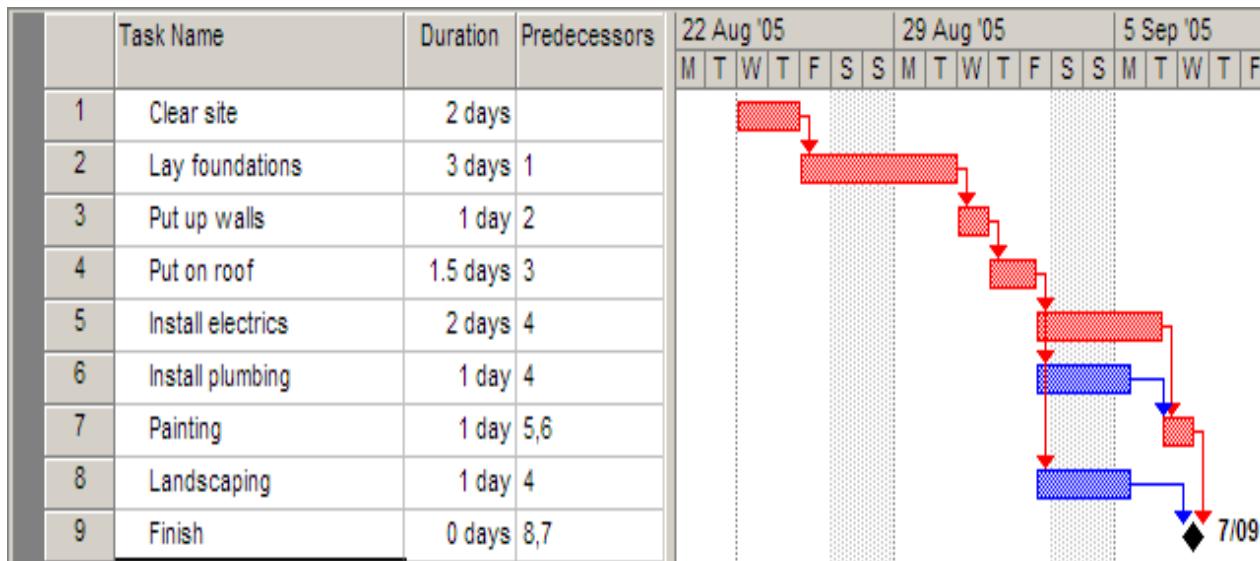


Example - 1 on Gantt Chart

	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

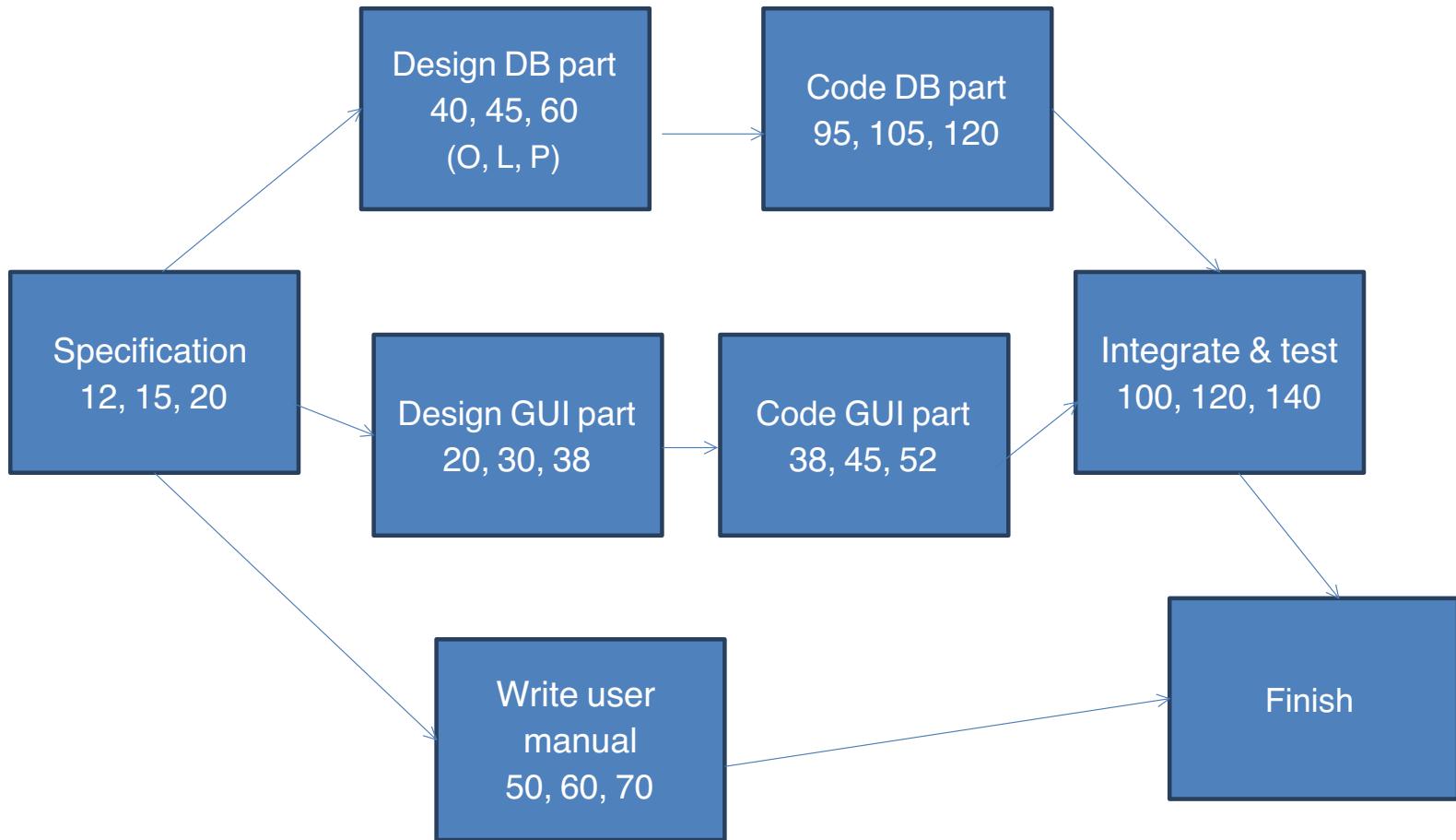
- The length of the critical path is the sum of the lengths of all critical tasks (the red tasks 1,2,3,4,5,7) which is $2+3+1+1.5+2+1 = 10.5$ days.



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

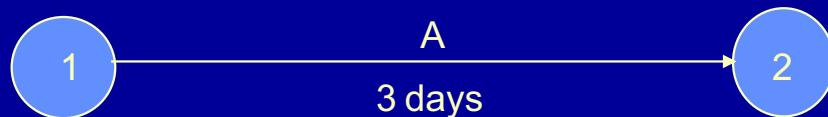
- n Finding the critical path is a major part of controlling a project.
- n The activities on the critical path represent tasks that will delay the entire project if they are delayed.
- n Manager gain flexibility by identifying noncritical activities and replanning, rescheduling, and reallocating resources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

Project Network

- n A project network can be constructed to model the precedence of the activities.
- n The arcs of the network represent the activities.
- n The nodes of the network represent the start and the end of the activities.
- n 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)

- n An activity carries the arrow symbol, →. This represent a task or subproject that uses time or resources
- n A node (an event), denoted by a circle ●, marks the start and completion of an activity, which contains 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 take three days

Determining the Critical Path

n 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

n 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

- n 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

n Consider the following project:

Immediate

<u>Activity</u>	<u>Predecessor</u>	time (days)
-----------------	--------------------	-------------

A	--	6
---	----	---

B	--	4
---	----	---

C	A	3
---	---	---

D	A	5
---	---	---

E	A	1
---	---	---

F	B,C	4
---	-----	---

G	B,C	2
---	-----	---

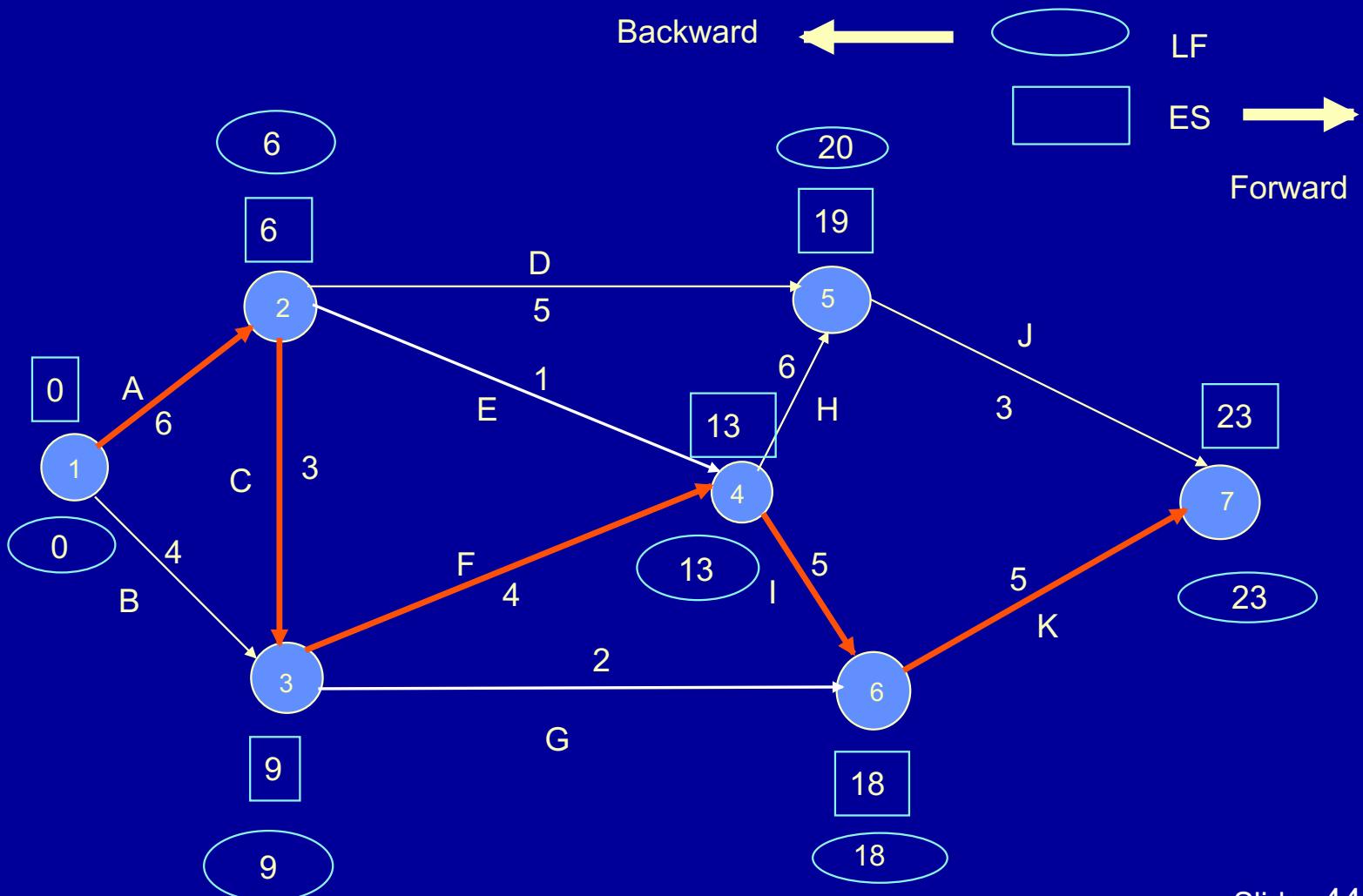
H	E,F	6
---	-----	---

I	E,F	5
---	-----	---

J	D,H	3
---	-----	---

K	G,I	5
---	-----	---

Example: network



Example: ABC Associates

a Network Diagram

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

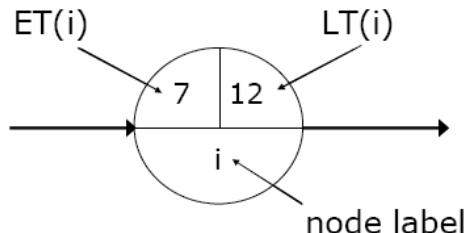
Additional Example

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

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



- O ES – Activity earliest start time
- O LS – Activity latest start time
- O EF – Activity earliest finishing time
- O LF – Activity latest finishing time
- O 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) .

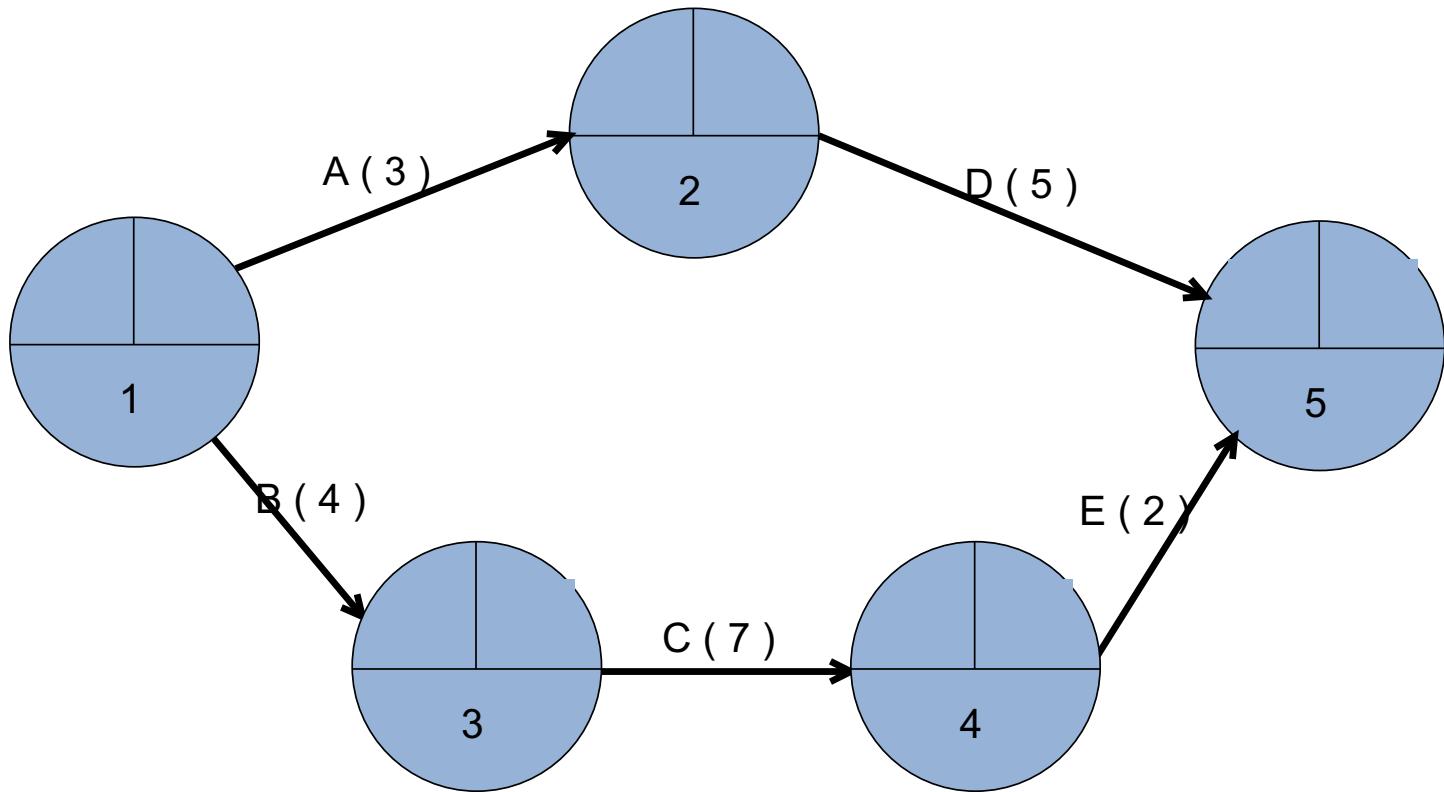
Step 2. Calculate LT for each node.

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

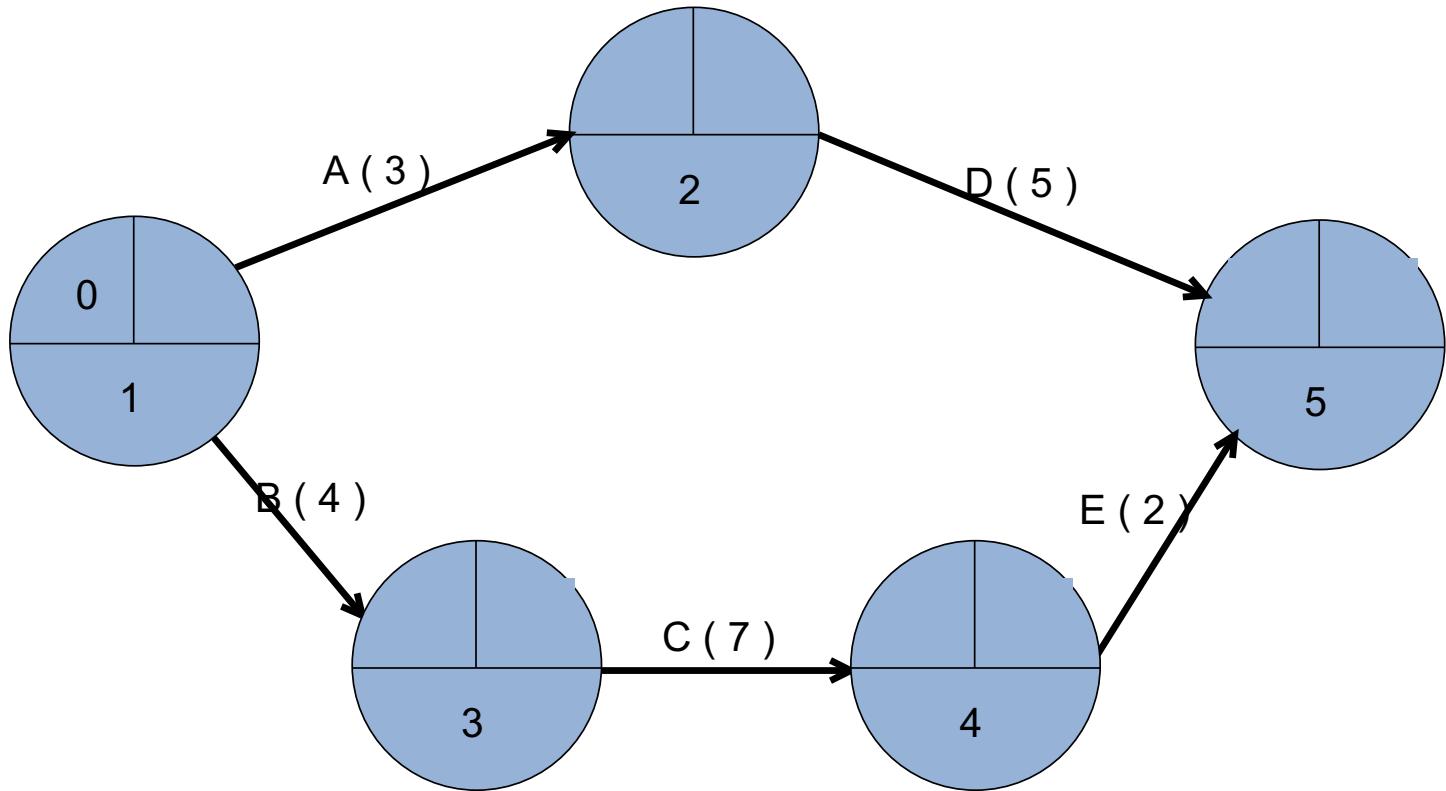
$$LT(i) = \min_j [LT(j) - t(i,j)]$$

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

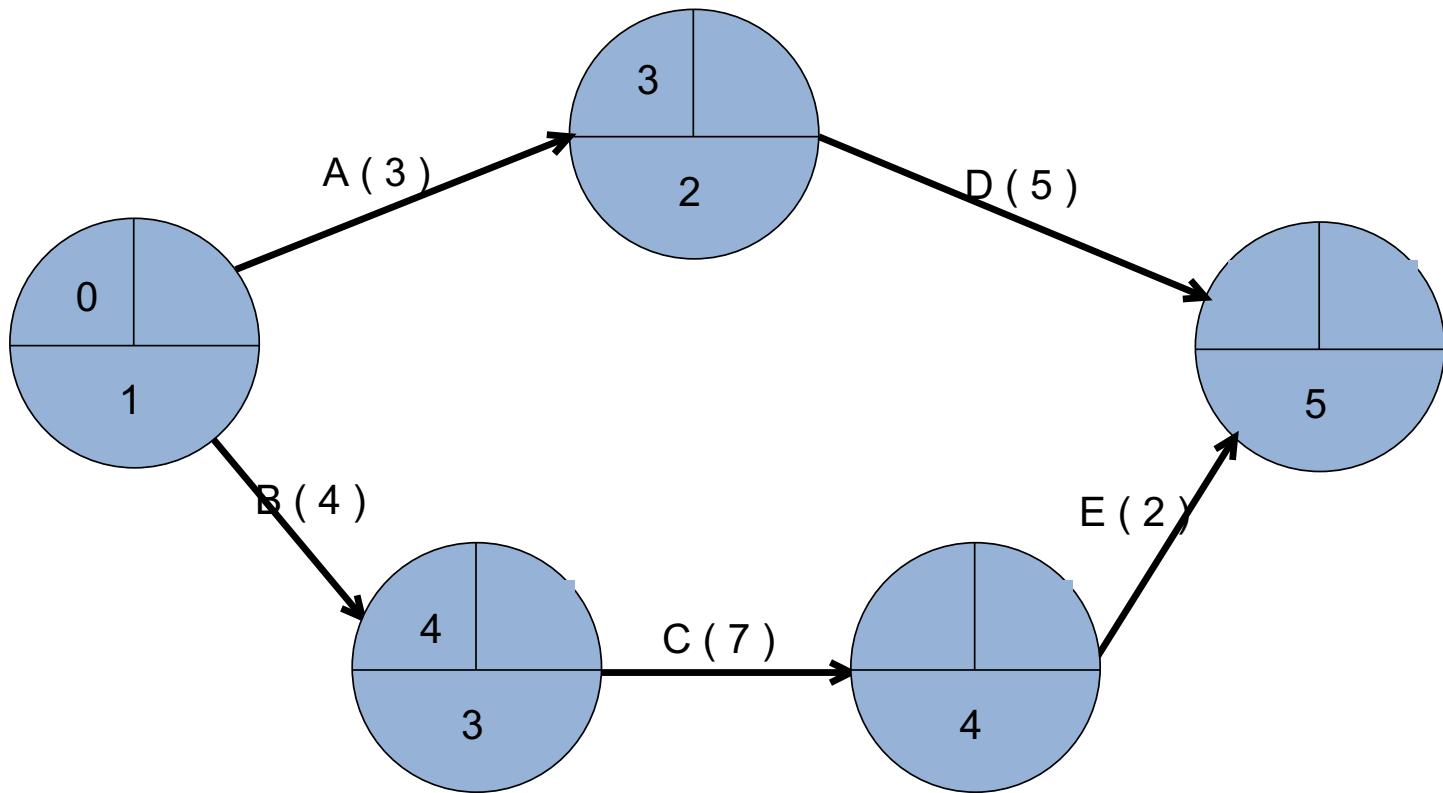
CPM – Critical Path Method



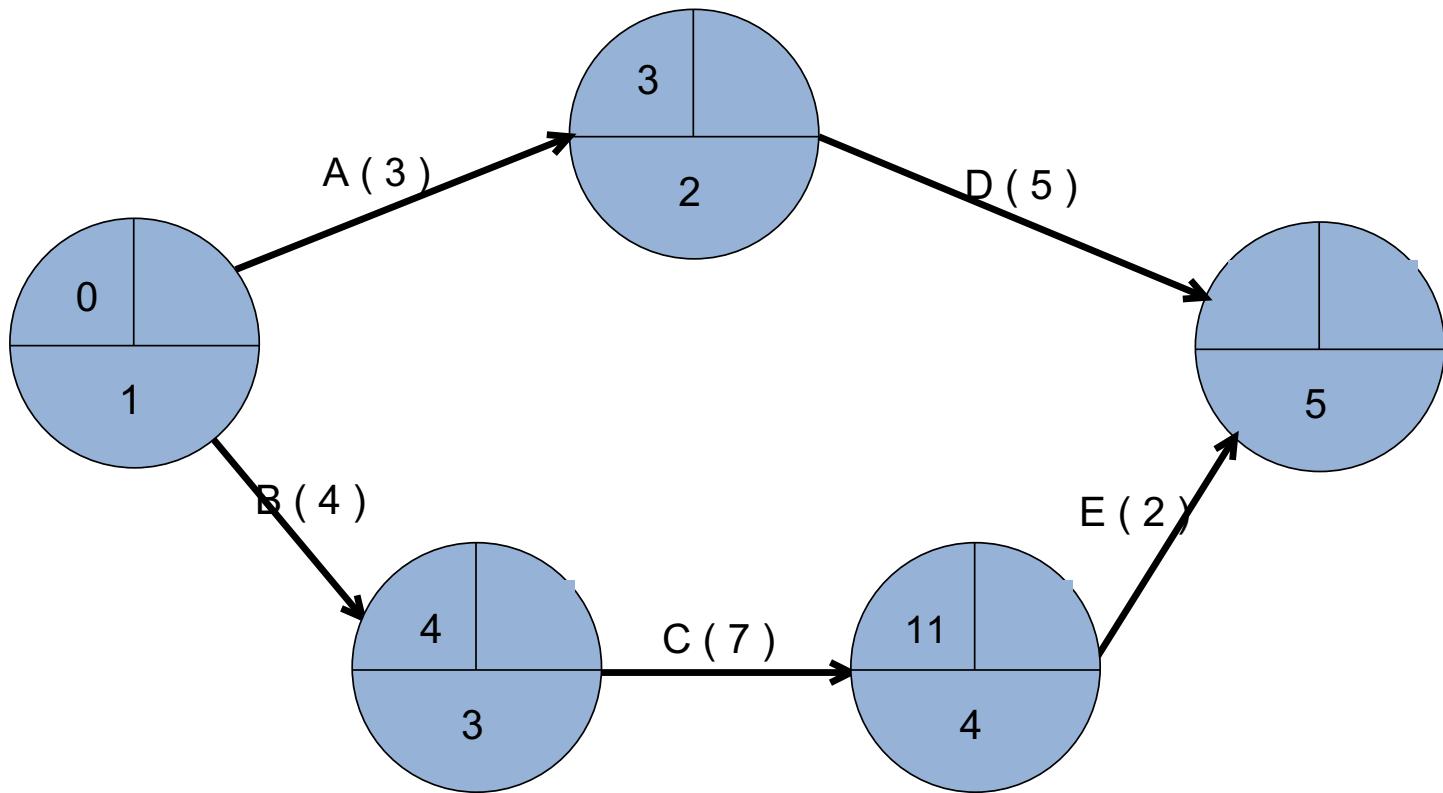
CPM – Critical Path Method



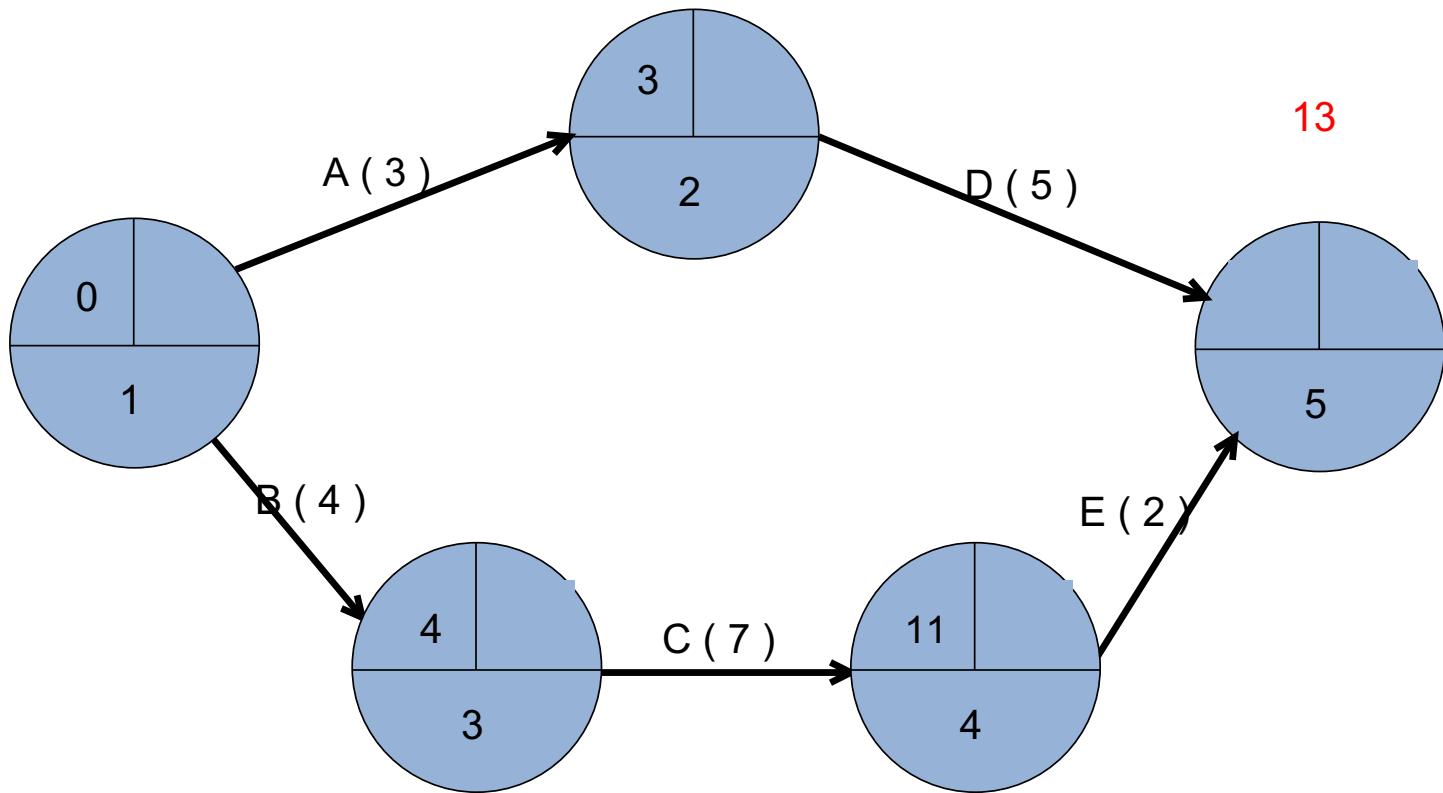
CPM – Critical Path Method



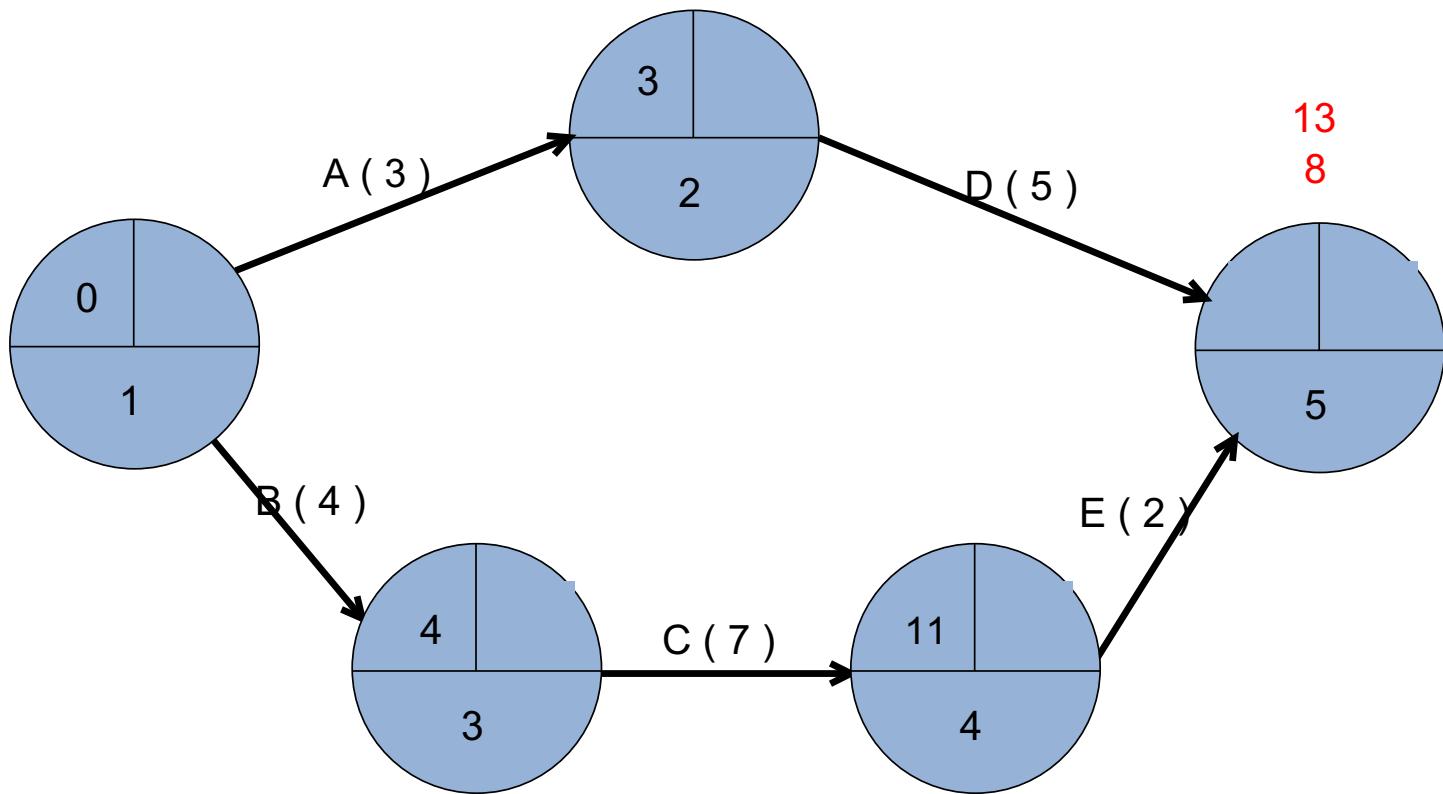
CPM – Critical Path Method



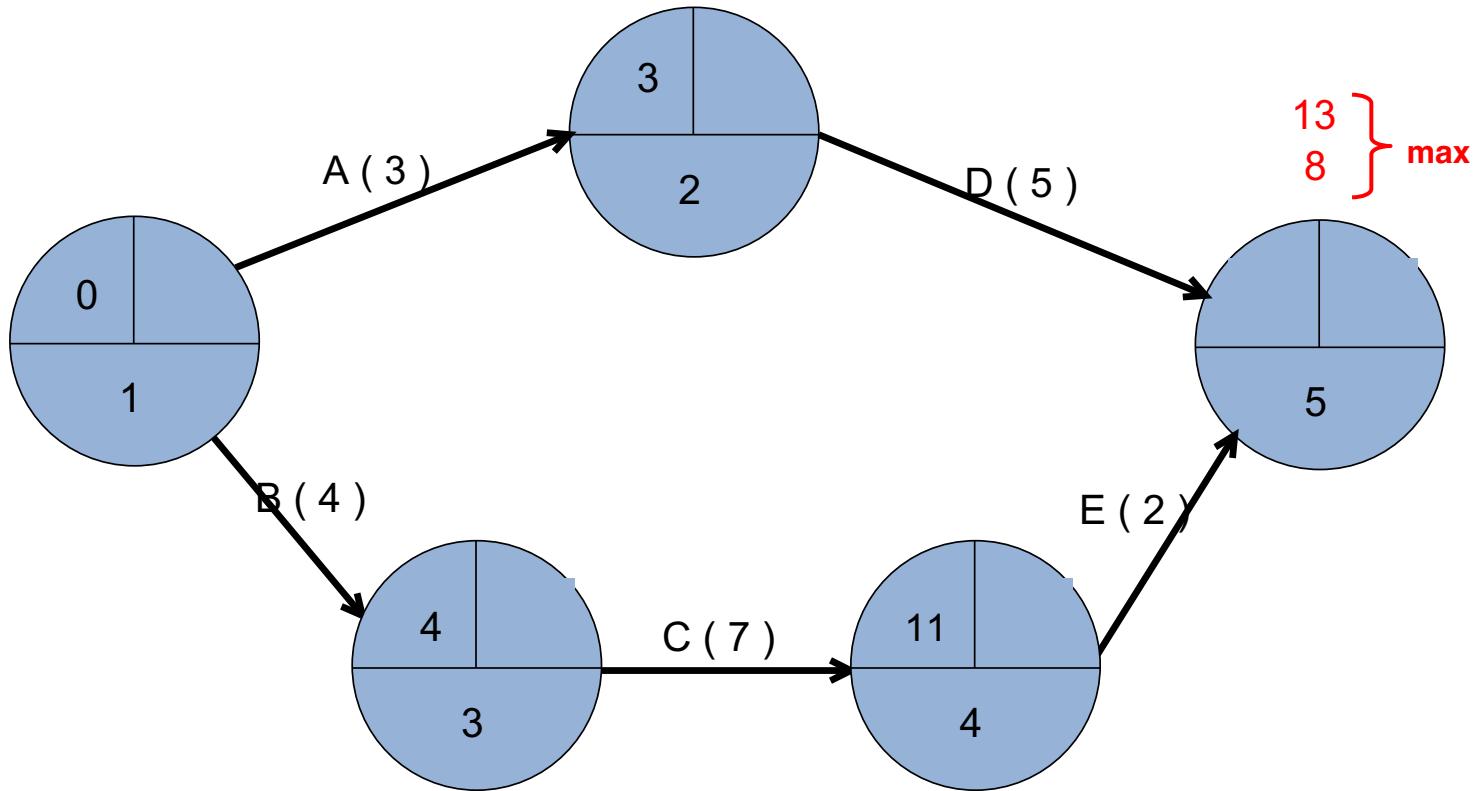
CPM – Critical Path Method



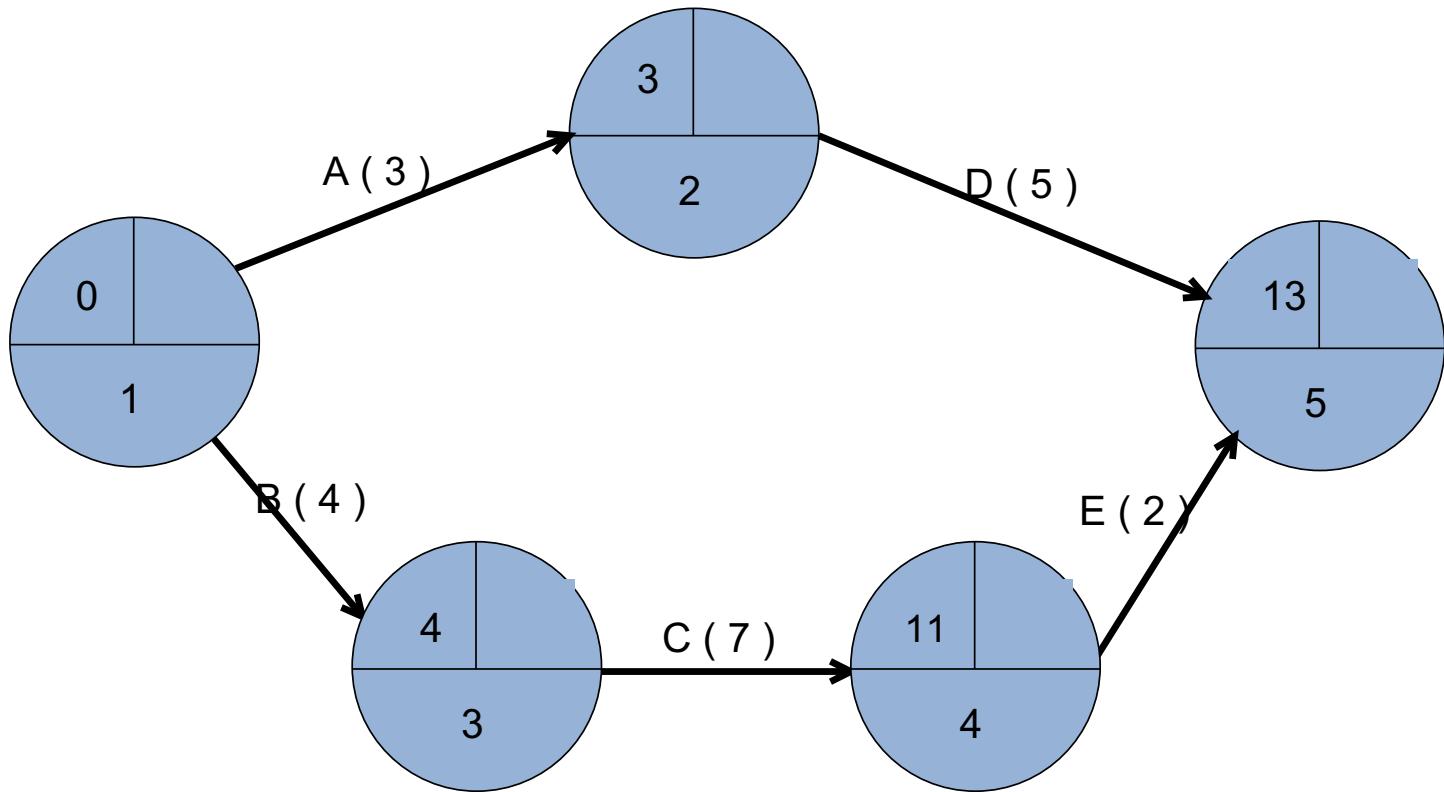
CPM – Critical Path Method



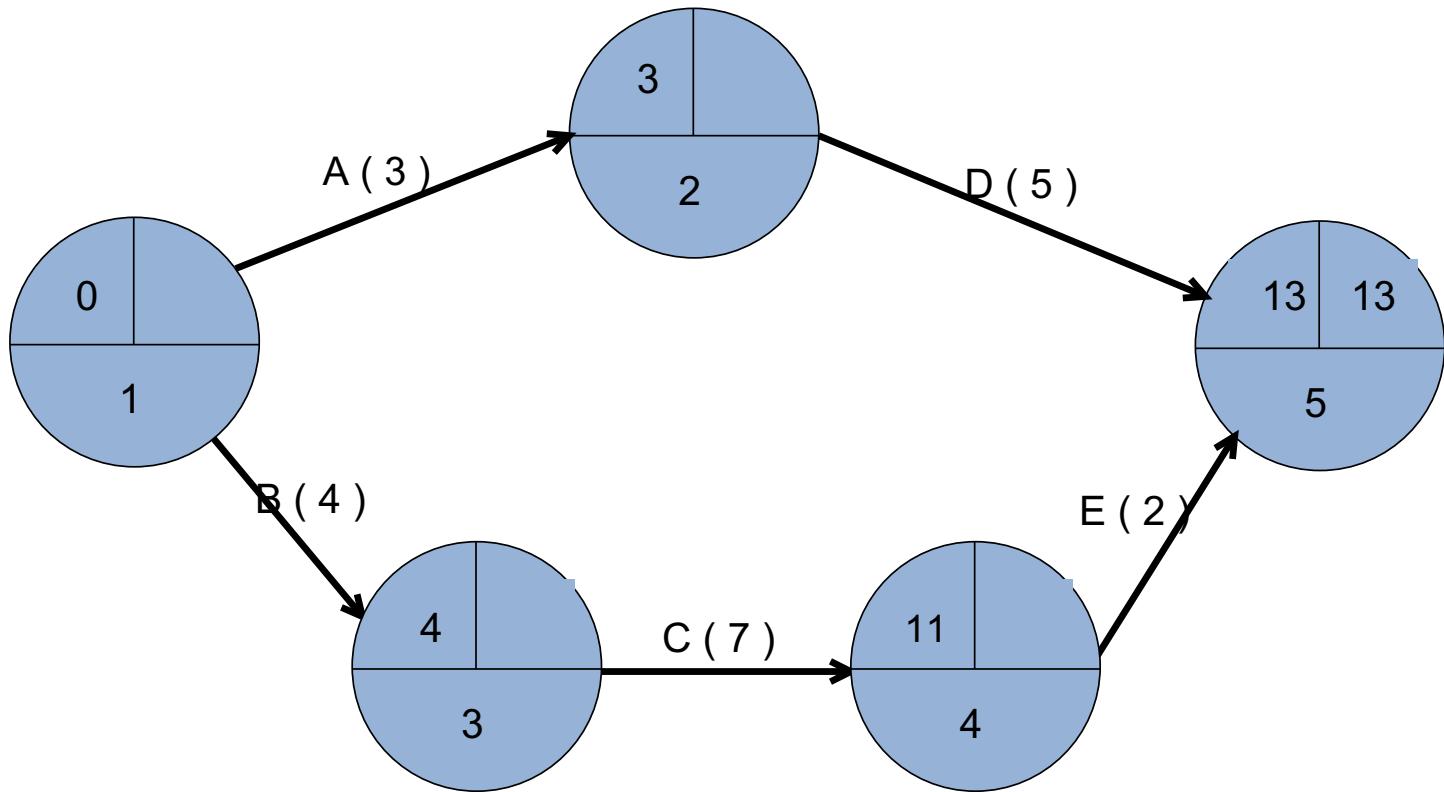
CPM – Critical Path Method



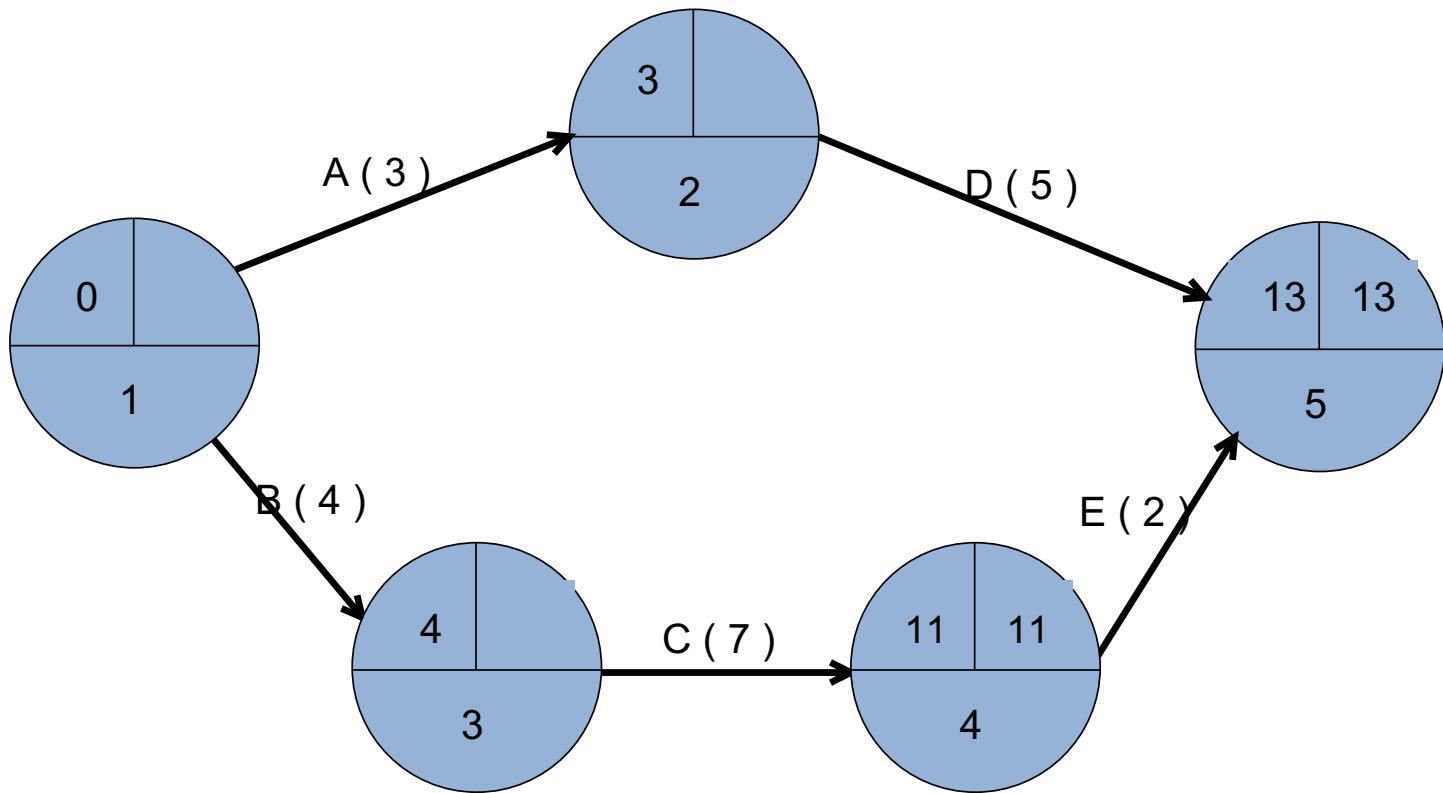
CPM – Critical Path Method



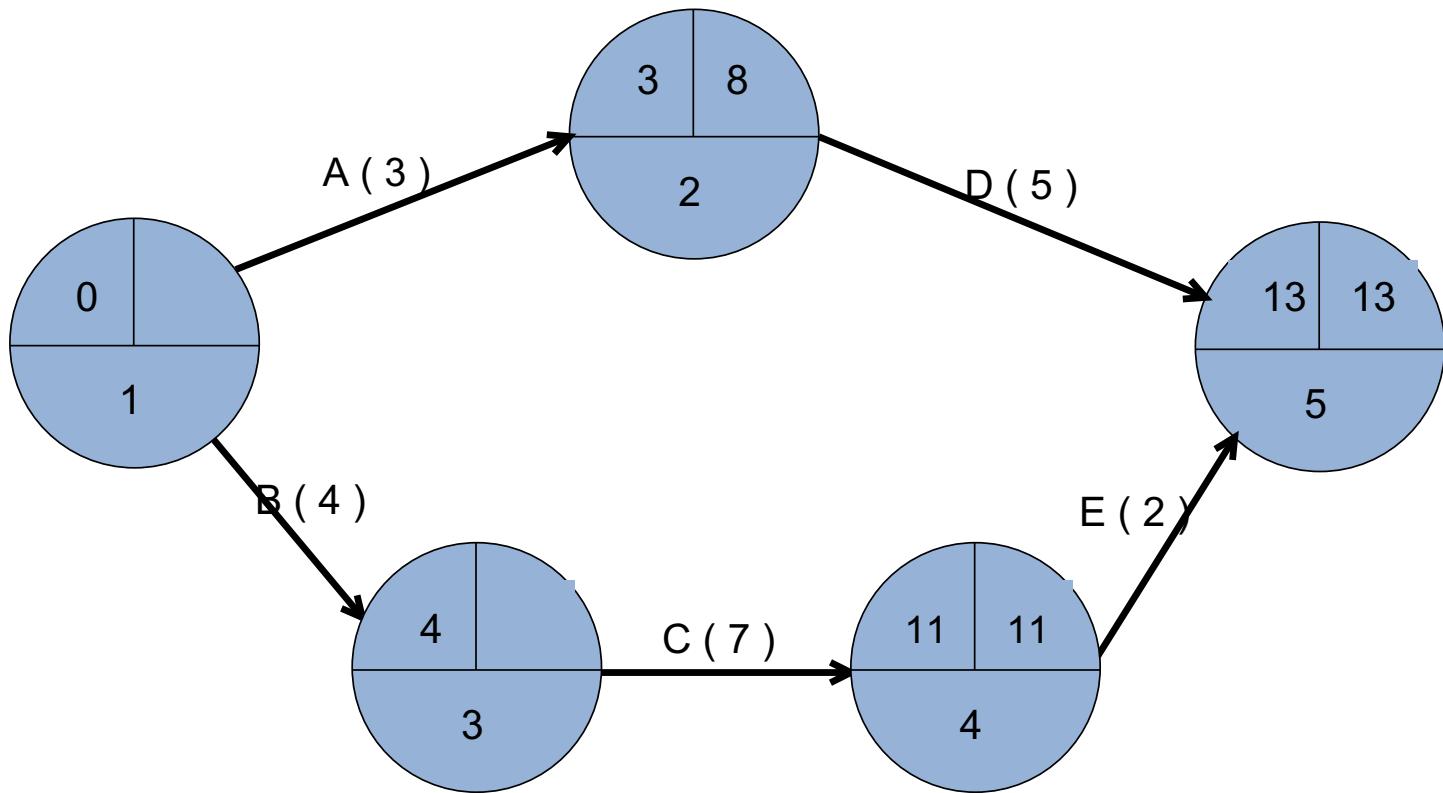
CPM – Critical Path Method



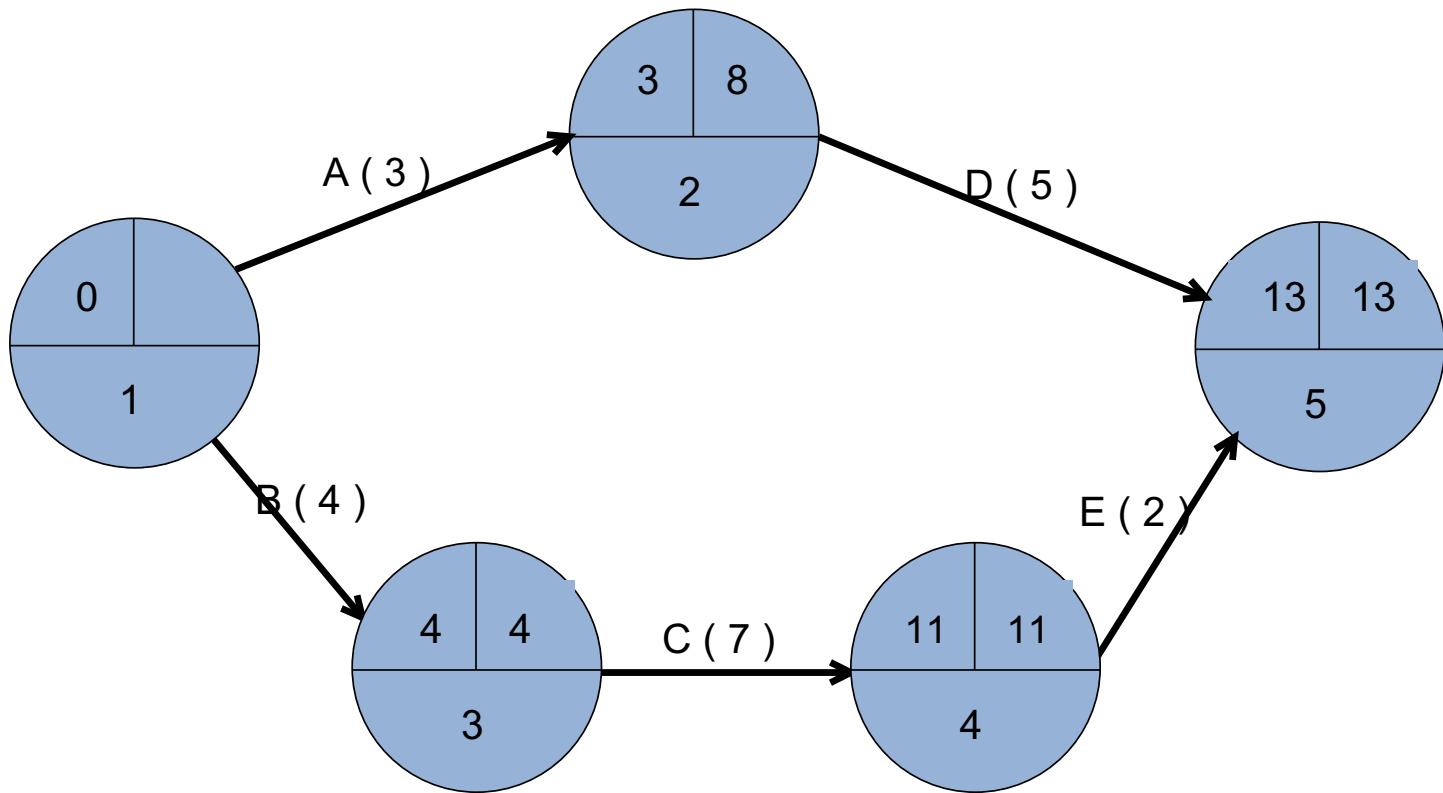
CPM – Critical Path Method



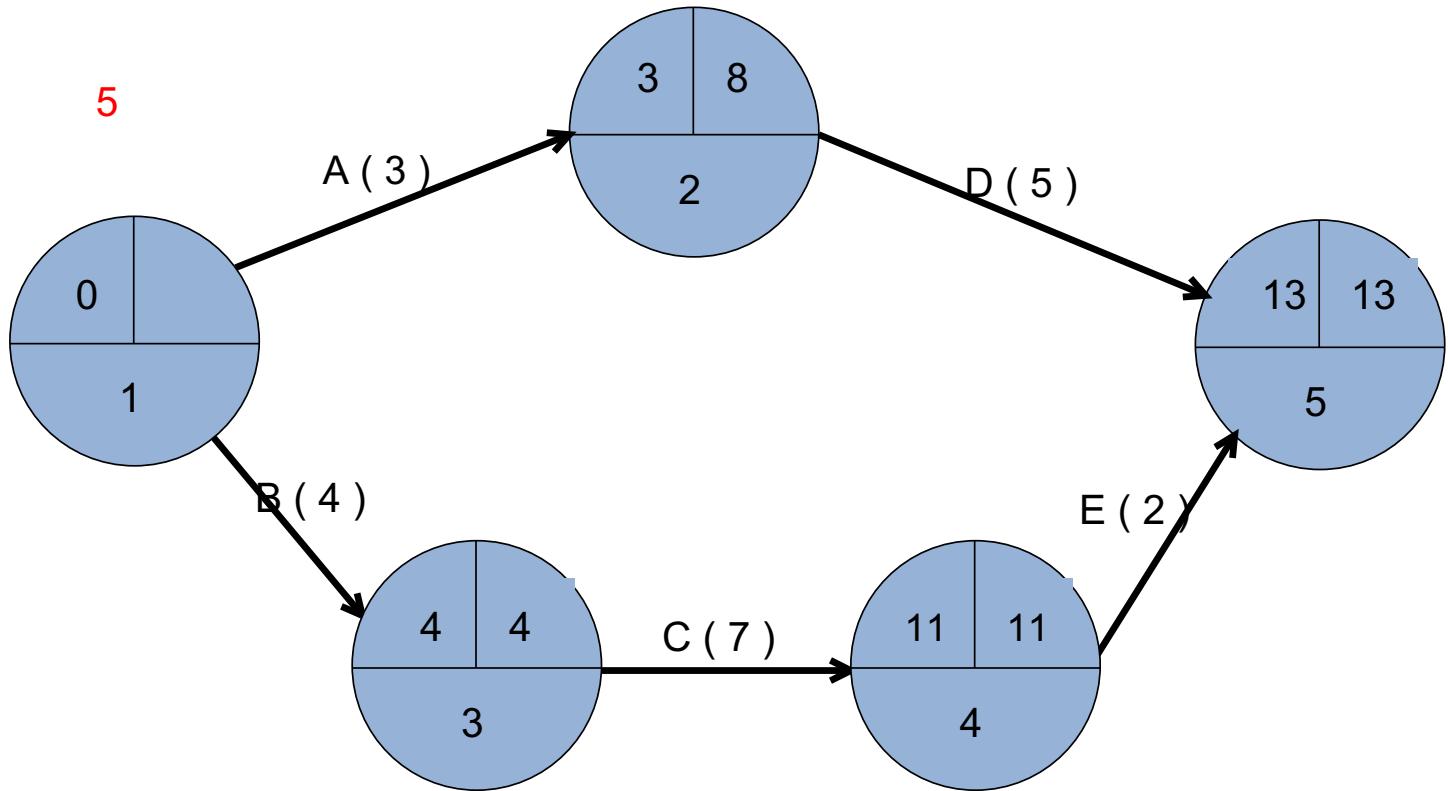
CPM – Critical Path Method



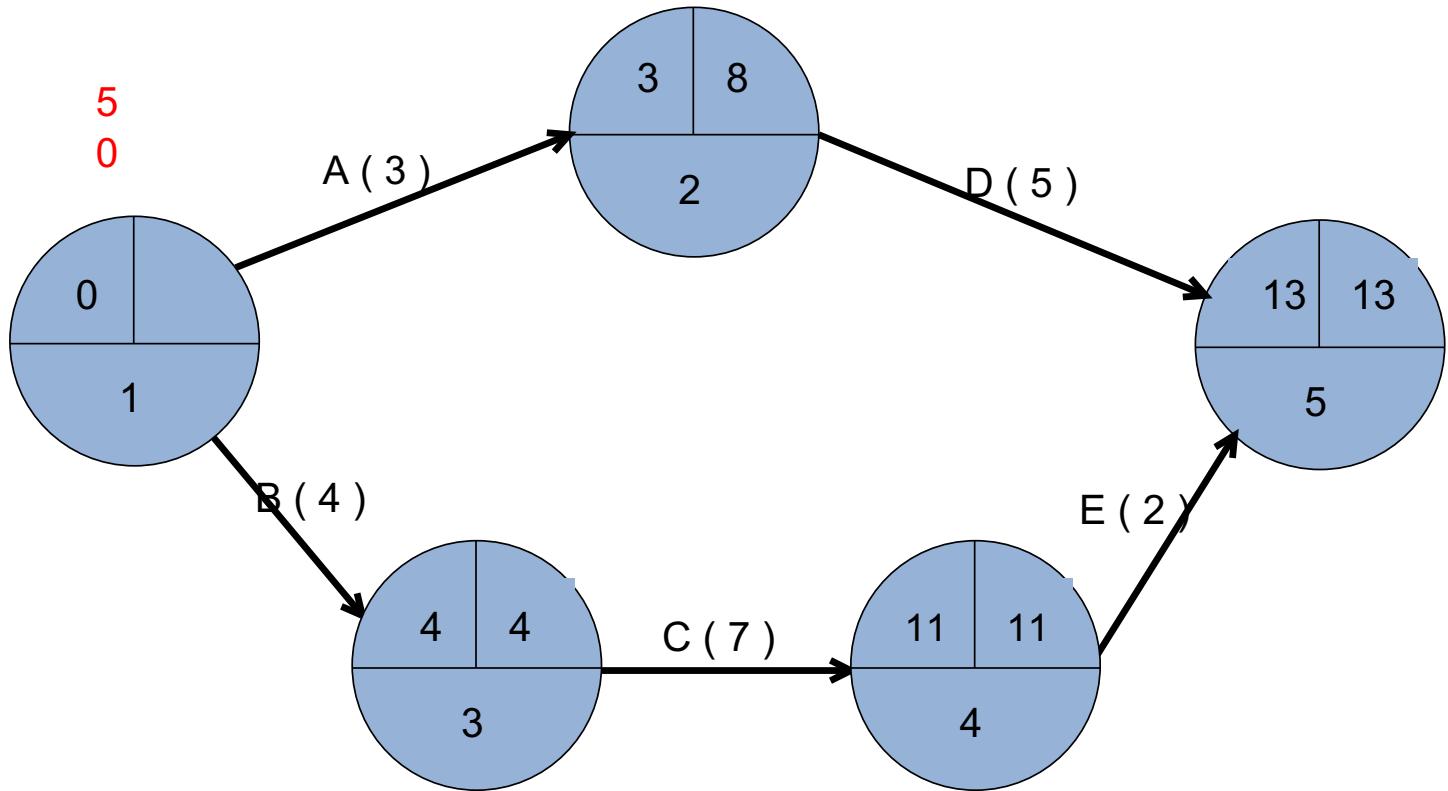
CPM – Critical Path Method



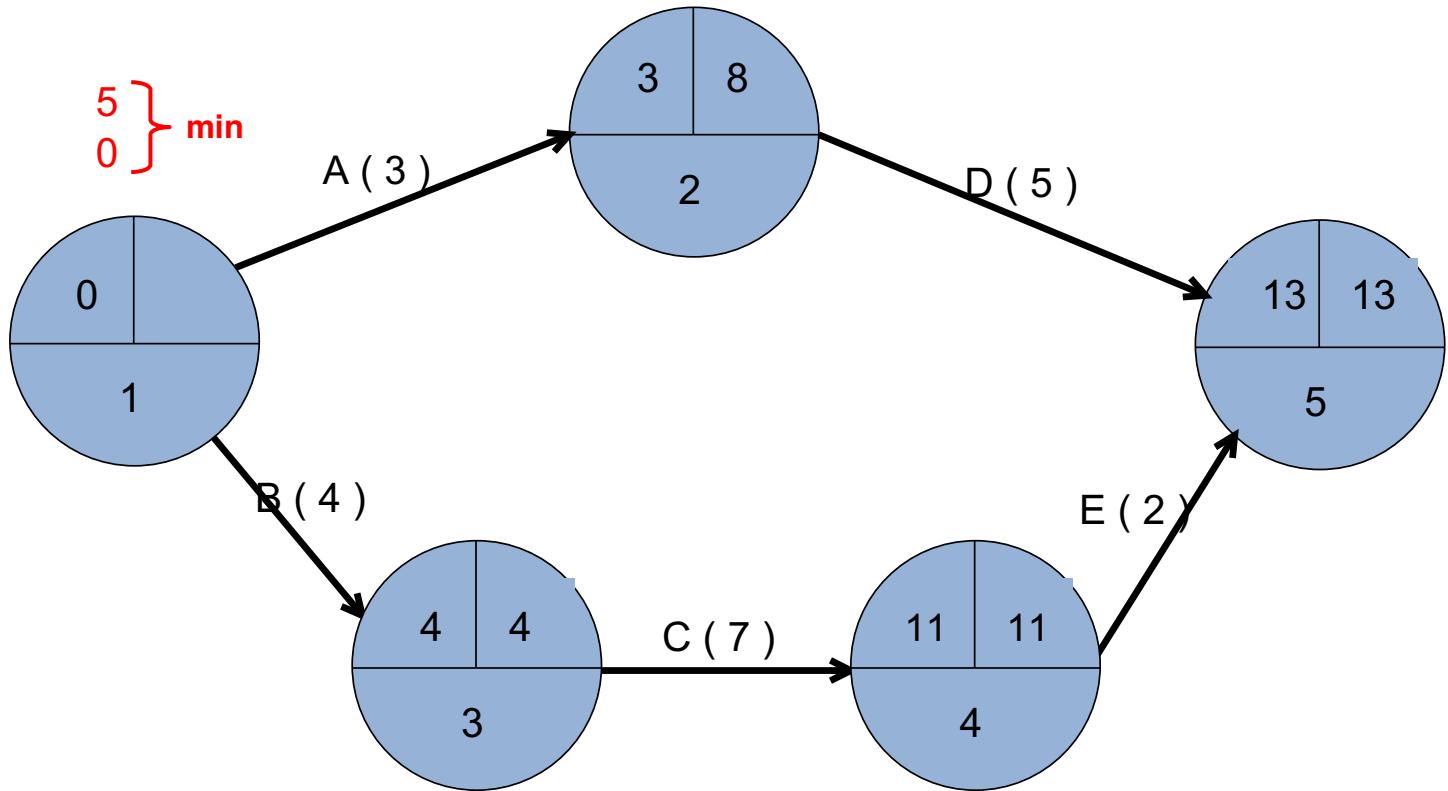
CPM – Critical Path Method



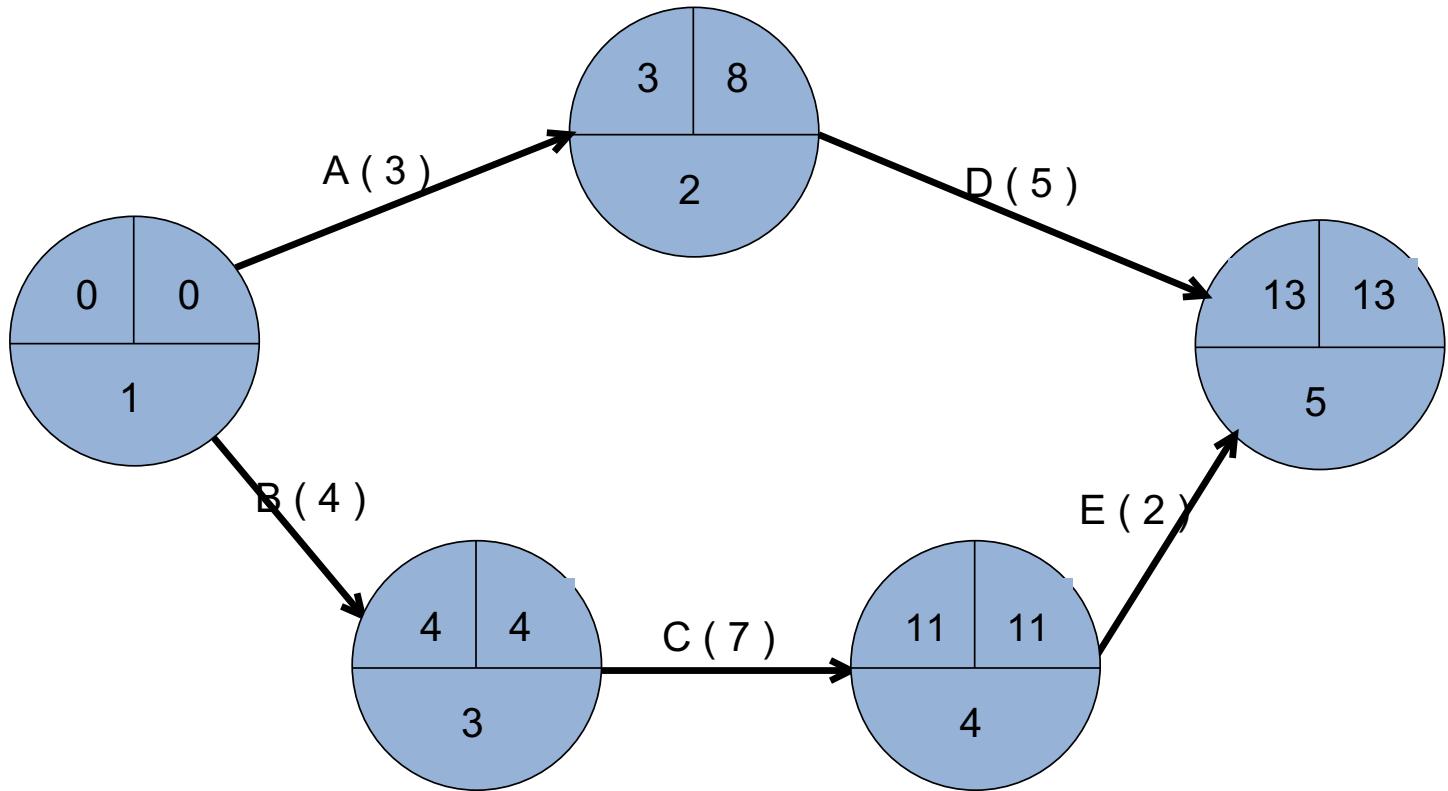
CPM – Critical Path Method



CPM – Critical Path Method



CPM – Critical Path Method

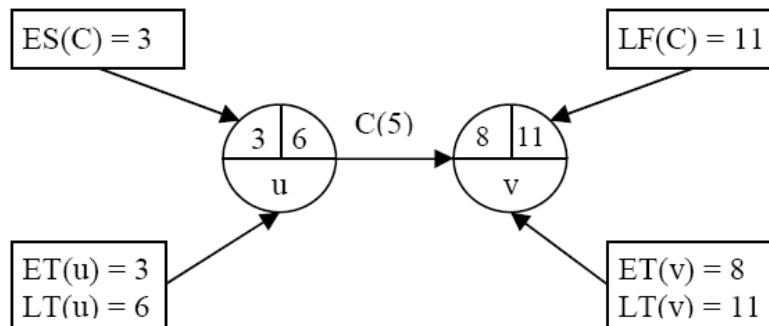


CPM – Critical Path Method

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

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

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



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

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.

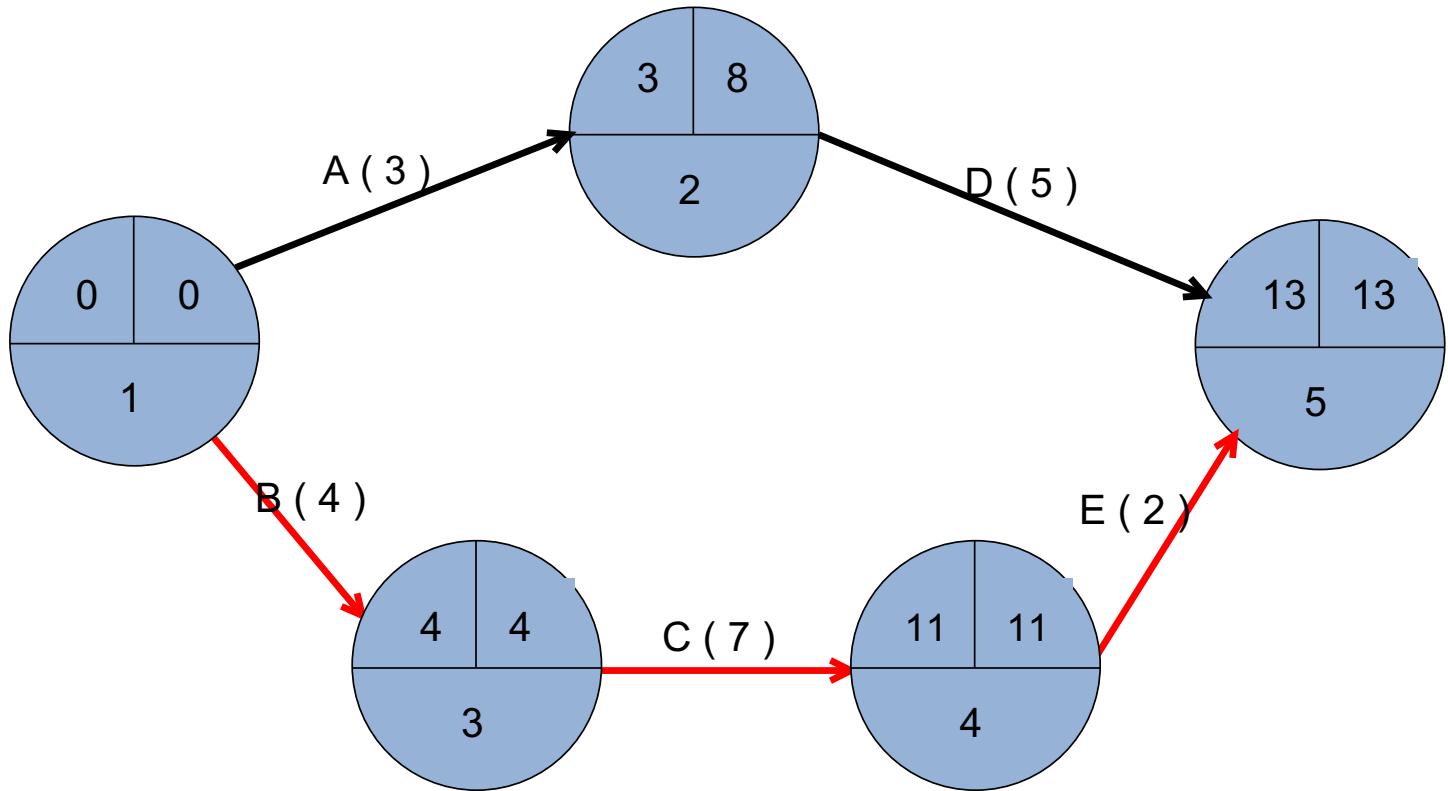
CPM – Critical Path Method

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

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

CPM – Critical Path Method



USE CASE DIAGRAM

Scenario Based

Requirements Specification vs Analysis Model

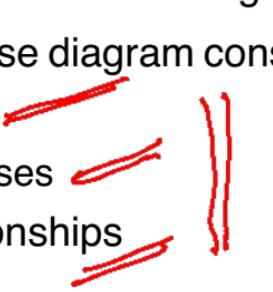
Both focus on the requirements from the user's view of the system

- The **requirements specification** uses natural language (derived from the problem statement)
- The **analysis model** uses a formal or semi-formal notation (we use UML)

USE CASE DIAGRAM

- The use case represents the different ways in which a system can be used by the users
- A **use case** describes interactions of actors with the target system to perform a function.
- The use case model can be documented by drawing a use case diagram and writing an accompanying text elaborating the drawing (use case specification)

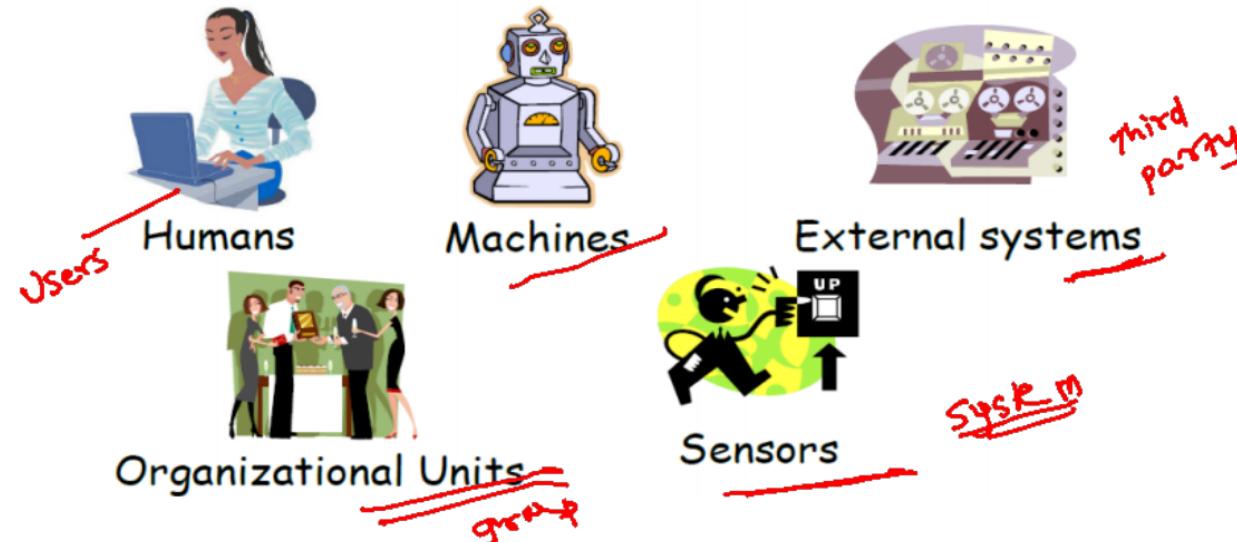
Use Case Diagrams

- Use case diagrams are used to visualize, specify, construct, and document the (intended) behavior of the system, during requirements capture and analysis.
 - Provide a way for developers, domain experts and end-users to Communicate.
 - Serve as basis for testing.
 - The use case diagram consists of
 - Actor
 - Use cases
 - Relationships
- 

Actors

External objects that produce/consume data:

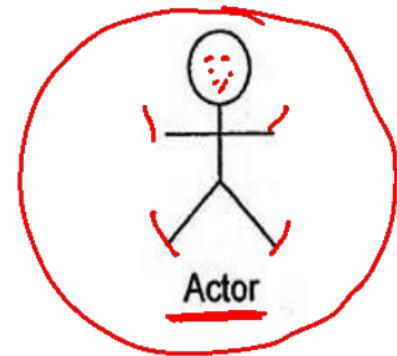
- Must serve as sources and destinations for data
- Must be external to the system



ACTOR

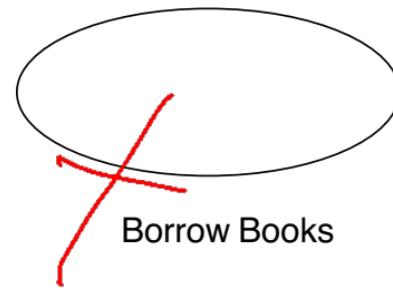
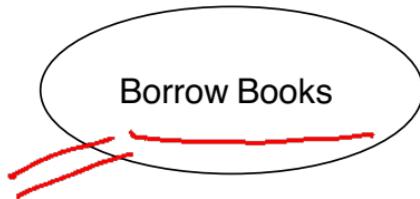


- Each Actor must be linked to a use case
- Represented by stick figure
- Labelled using a descriptive noun or phrase
Eg: Librarian, Doctor, Student, Customer



USE CASE

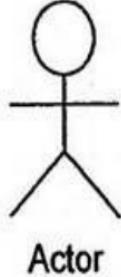
- Labelled using a descriptive verb-noun phrase
- Represented using an ellipse with name of the use case written inside the ellipse or written below the ellipse



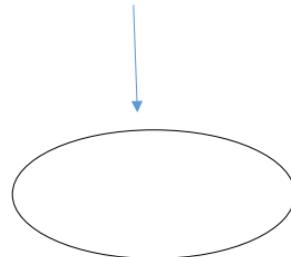
- Consider the following scenario:

"A patient calls the clinic to make an appointment for a yearly checkup"

- A patient calls the clinic to make an appointment for a yearly checkup.



Actor



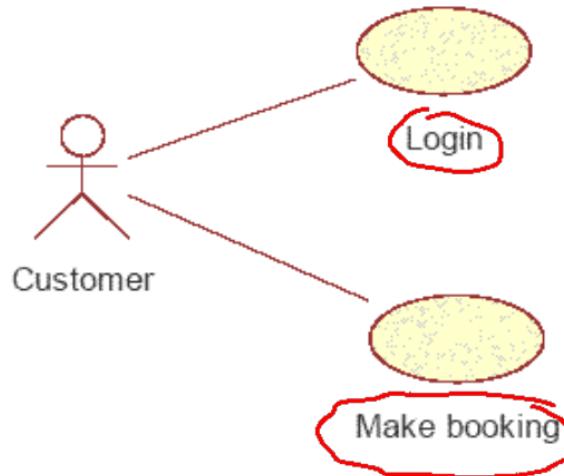
Use case

- The picture below is a **Make Appointment** use case for the medical clinic.
- The actor is a **Patient**. The connection between actor and use case is a **communication association** (or **communication** for short).



RELATIONSHIPS

- Represents communication between actor and use case
- Depicted by a line connecting actor and use case

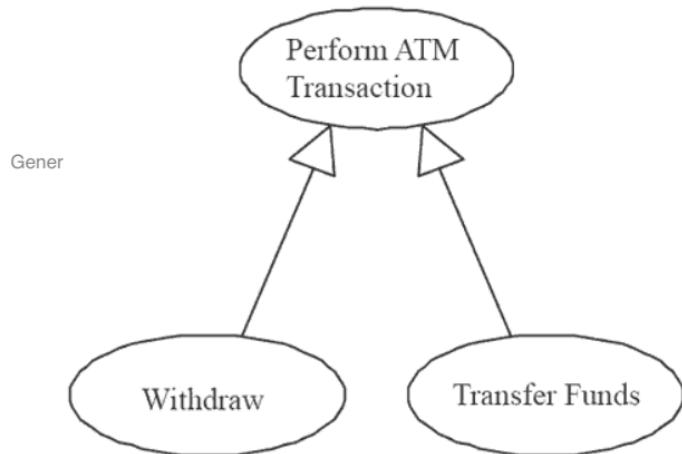


- Other Types of Relationships for Use Cases

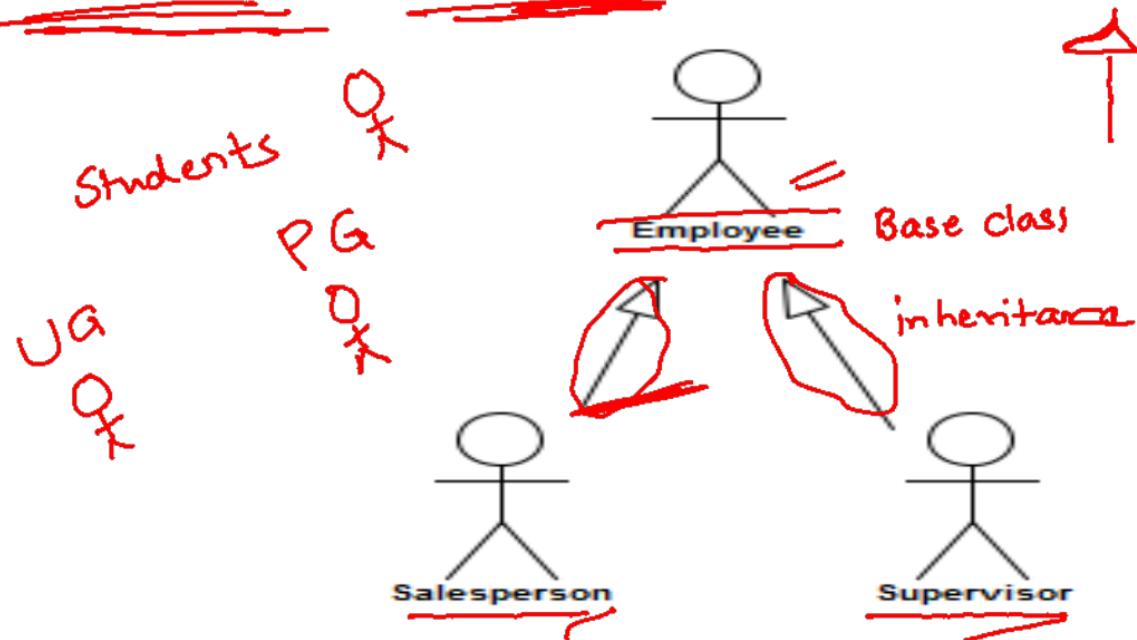
- Generalization (in old version 1.0 but removed from 2.0)
- Include
- Extend

Generalization Relationship

- refers to the relationship which can exist between two use cases and which shows that one use case (child) inherits the structure, behavior, and relationships of another actor (parent).
- The child use case is also referred to the more specialized use case while the parent is also referred to as the more abstract use case of the relationship.
- Generalization is possible within use cases and also between actors
- Represented by a line and a hollow arrow

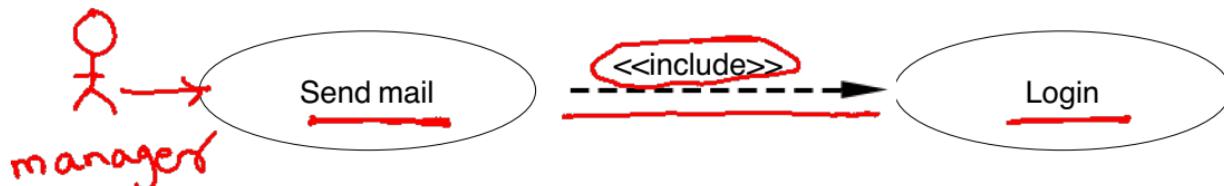


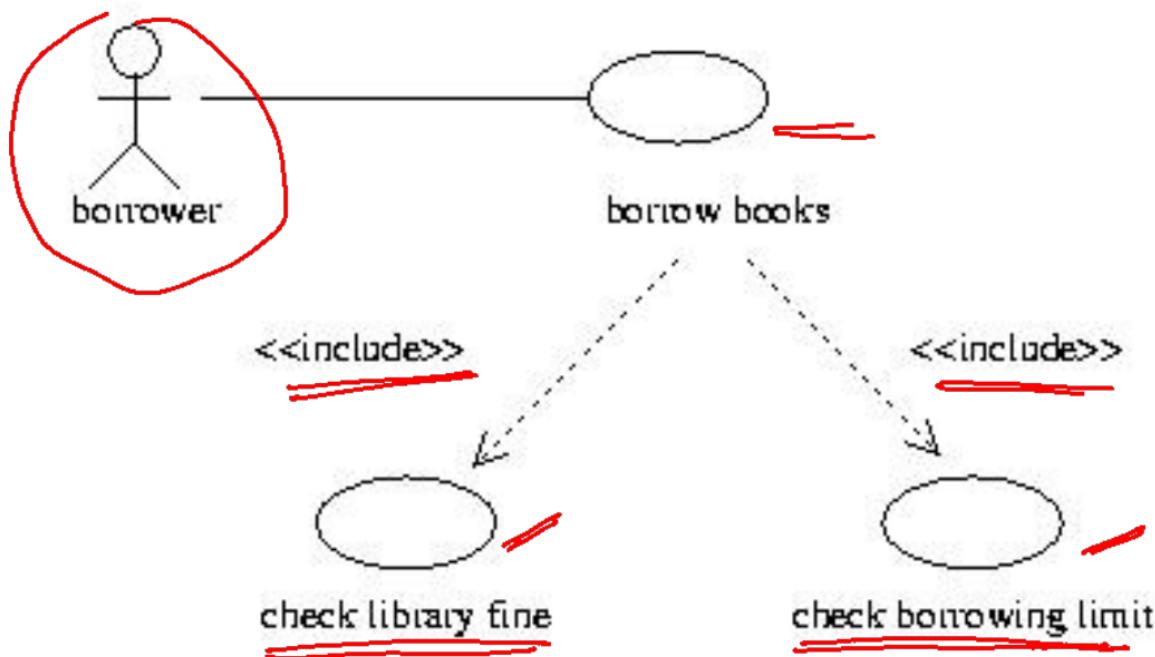
- Generalization between actors



Include Relationship

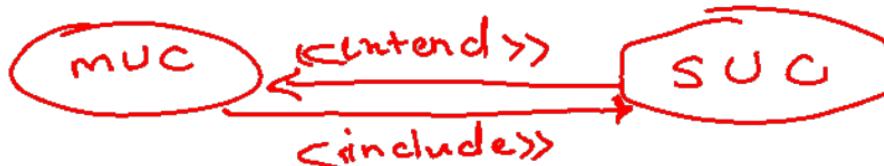
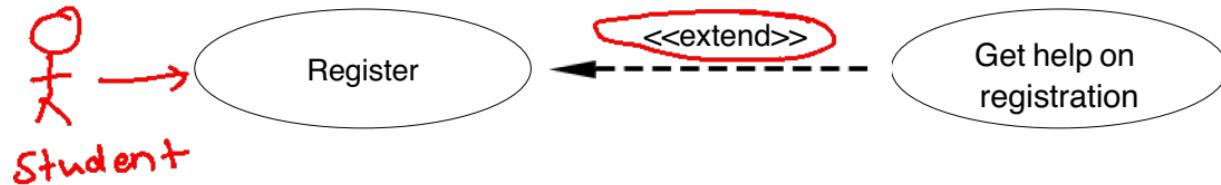
- Represents the inclusion of the functionality of one use case within another
- Arrow is drawn from the base use case to the used use case
- Write <<include>> above arrowhead line





Extend relationship

- Represents the extension of the use case to include optional functionality
- Arrow is drawn from the extension use case to the base use case
- Write <<extend>> above arrowhead line



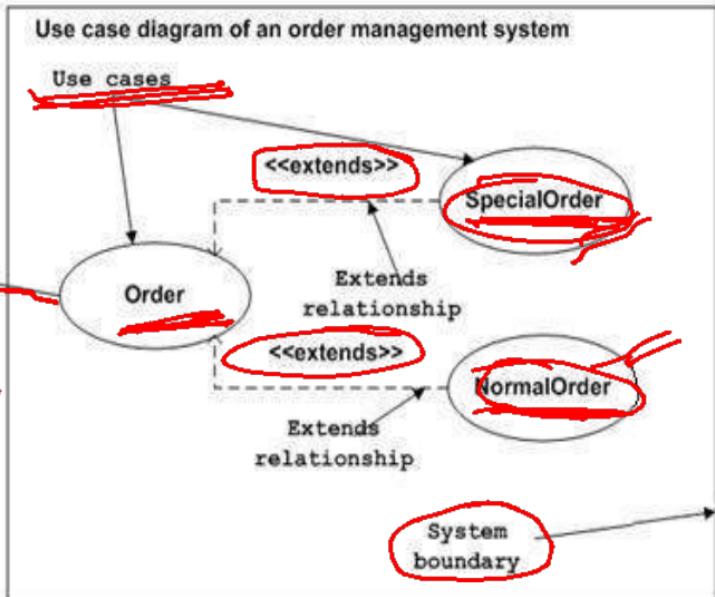
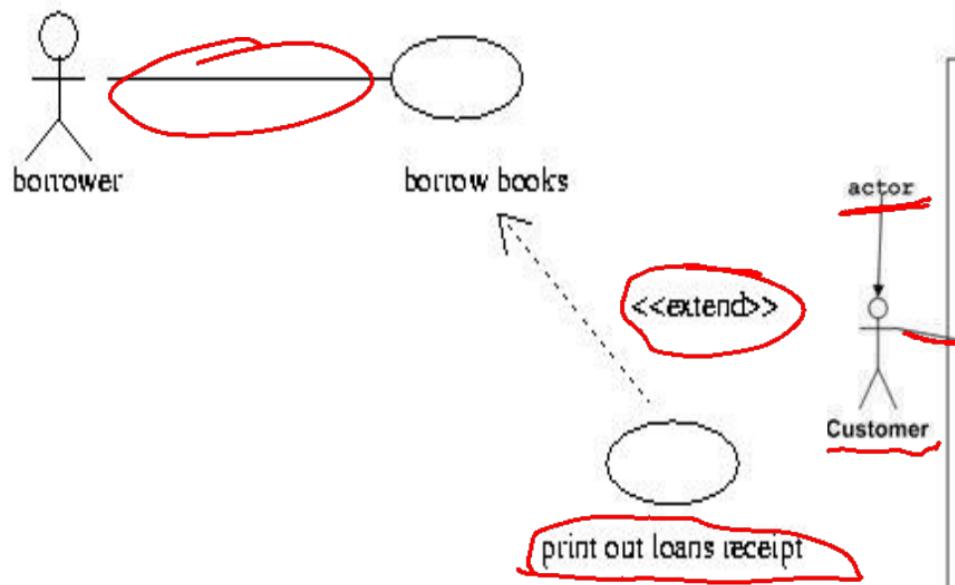
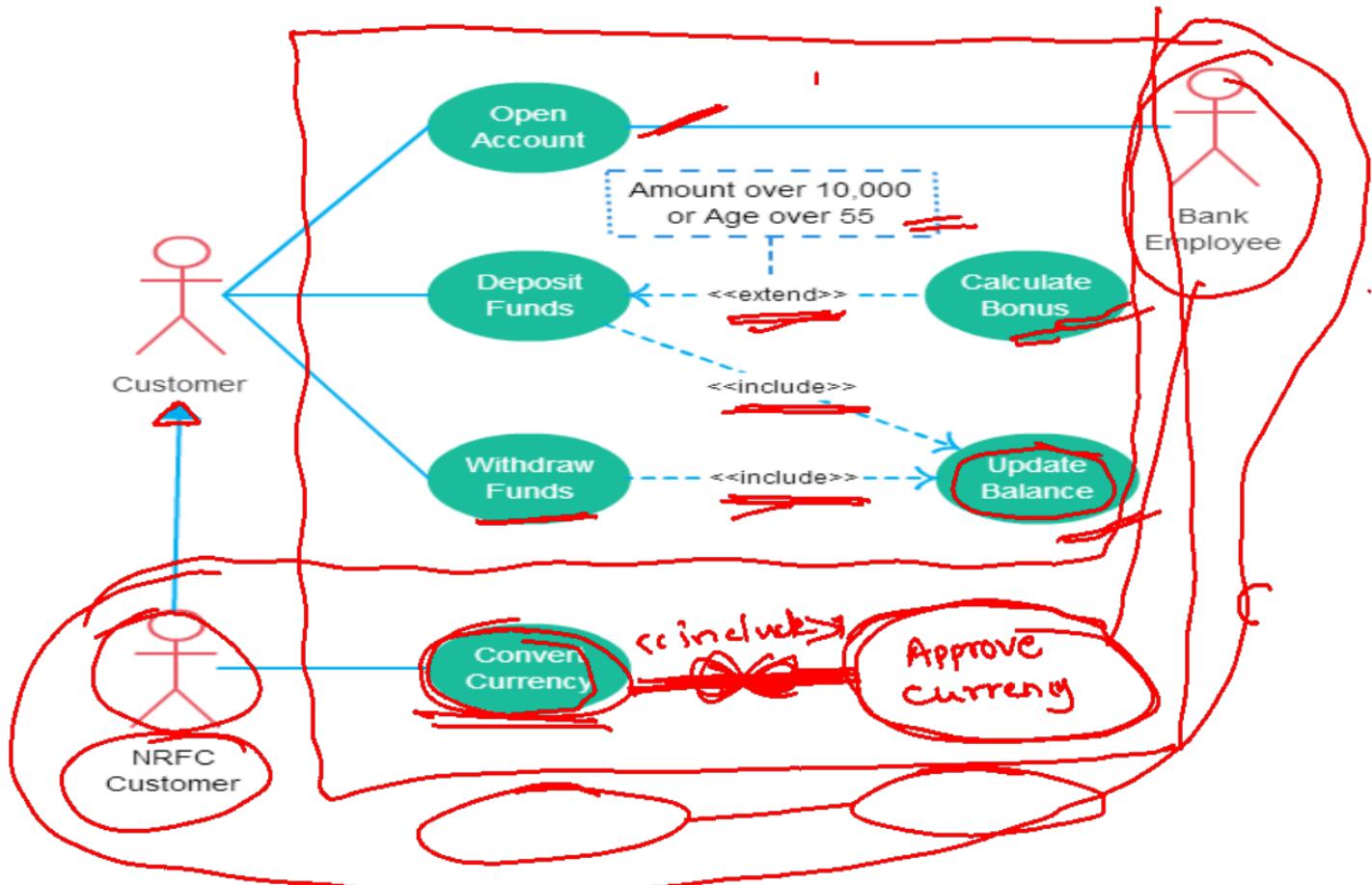
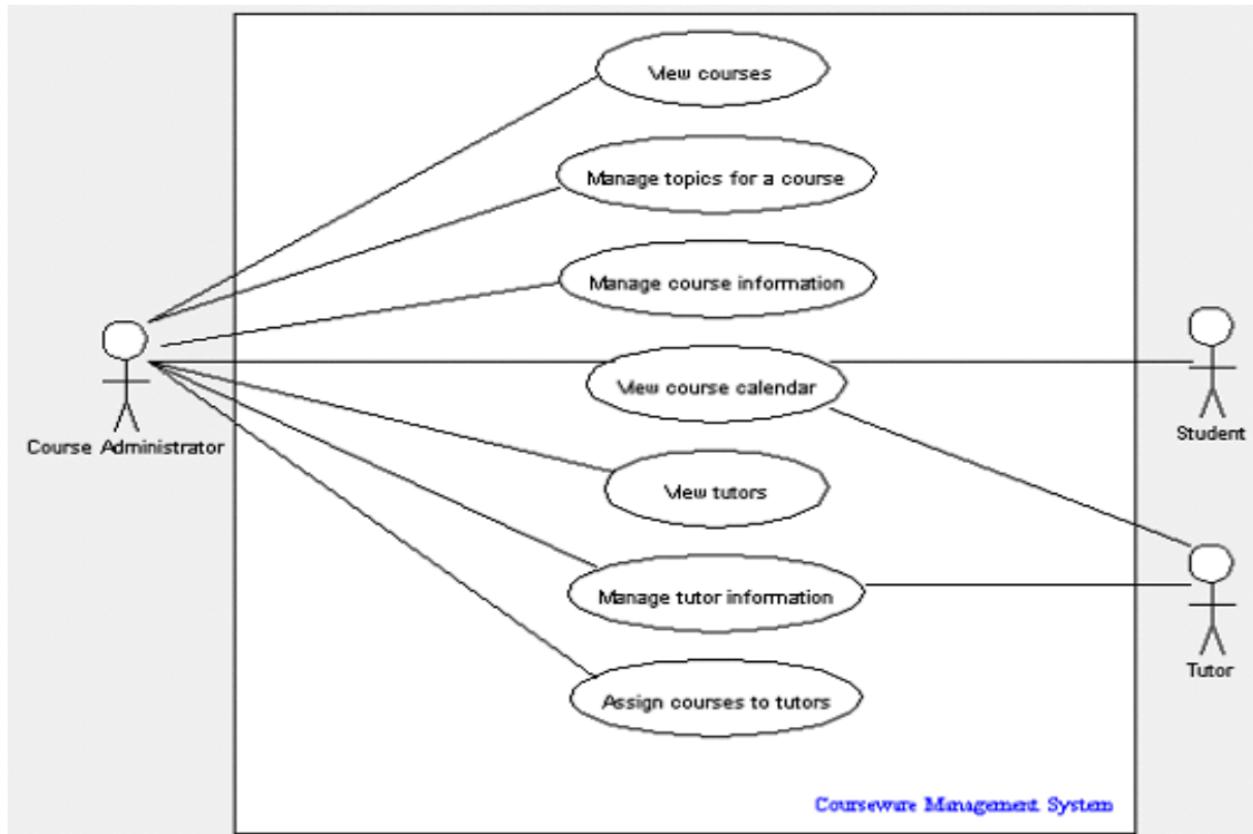


Figure: Sample Use Case diagram



Use case diagram example

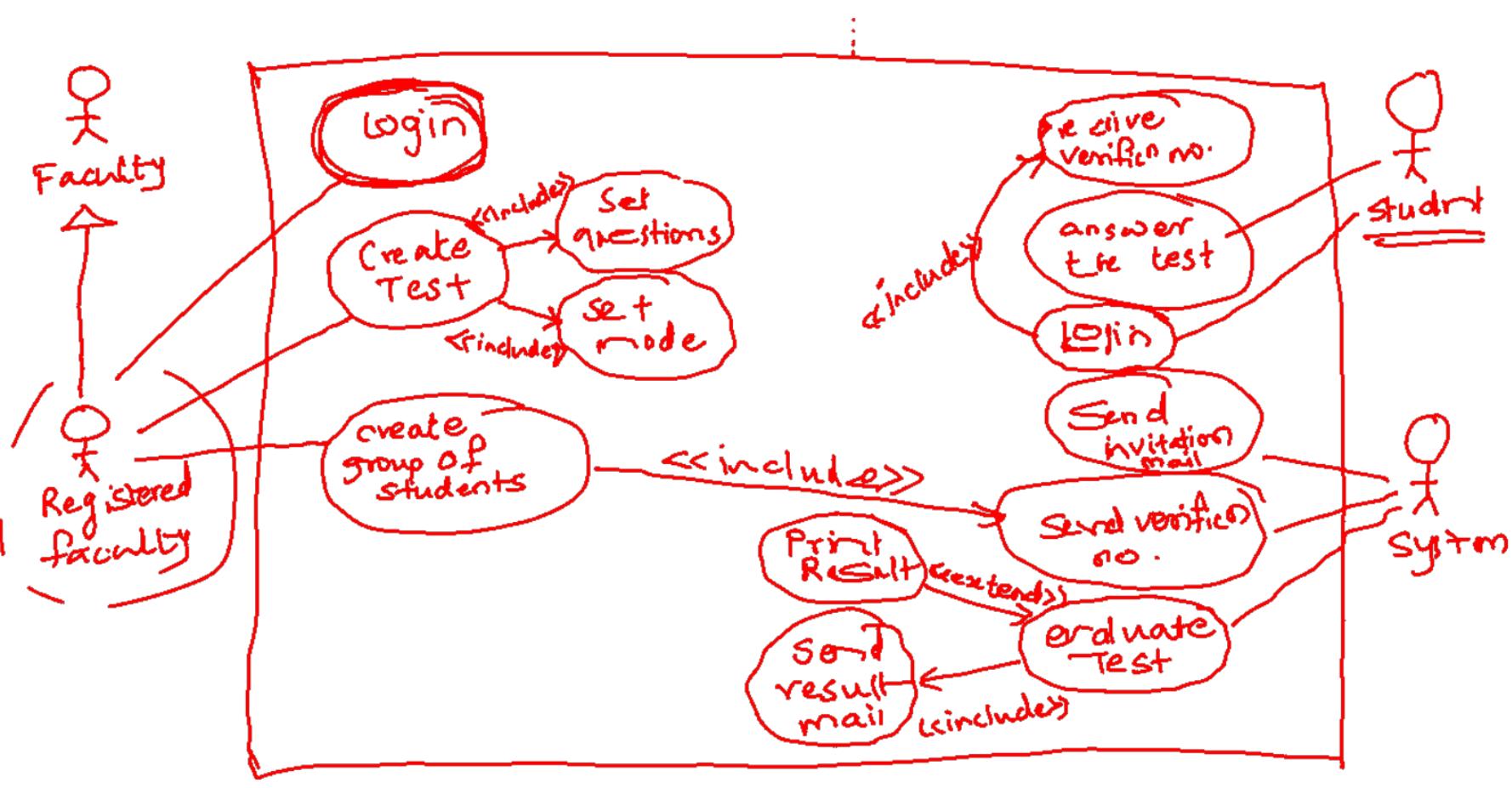


Problem Statement 1

- The System is intended for faculty members to conduct online examination. Every registered faculty member can login to the system, create test, set the questions, set the mode for each test etc. Faculty member also decides which groups of students have to take a particular test and accordingly creates group of students. The system should send mail to the students inviting them to take the test.
- Students can login using the verification number received and then answer the test questions. The System should evaluate student answers and send the result via mail or can print the result.

Problem Statement 1

The System is basically intended for **faculty** members to conduct online examination. Every **registered faculty** member can login to the system, create test, set the questions, set the mode for each test etc. Faculty member also decides which groups of students have to take a particular test and accordingly creates group of students. The system should send mail to the students inviting them to take the test. **Students** can login using the verification number received and then answer the test questions. The **System** should evaluate student answers with that specified by faculty and send the result via mail.



Problem Statement 2

An **auto rental company** wants to develop an automated system that would handle car reservations, customer billing, and car auctions. Usually a customer reserves a car, picks it up, and then returns it after a certain period of time. At the time of pick up, the customer has the option to buy or waive collision insurance on the car. When the car is returned, the customer receives a bill and pays the specified amount. In addition to renting out cars, every six months or so, the auto rental company auctions the cars that have accumulated over 20,000 miles.

Problem Statement 3

Draw the use-case diagram for Hotel Information System. There are two types of customers: Tour-group customers and Individual customers. Both can book, cancel, check-in and check-out of a room by Phone or via the Internet. There are booking process clerk and reception staff who manage it. A customer can pay his bill by credit card or by cash. After frequent booking of hotel more than 5 times, customer can go for booking of special package of stay for 6th time. Clerk also maintains the resources available in the hotel and reception staff takes care of feedback collection from the customer.

Problem Statement 4

A student may register for classes during a specified registration period. To register, a student must see their advisor. The advisor must approve each course that the student has selected. The advisor will use the registration system to determine if the student has met the course prerequisites, is in good academic standings and is eligible to register. If the advisor approves the courses, the advisor enters the student's college id into the course registration system. The course registration number for each course is entered. The course description, course number and section for those courses will automatically display. The system will check for schedule conflicts before saving the registrations. A bill for the courses will print in the financial administrator's office. The student should proceed to pick it up. Faculty can use the registration system to check enrollments in their classes, get a class list, check a student's transcript, look up a student's phone number and other such student information. The registrar can use the registration system to enter new classes for an upcoming semester, cancel a class, and check conflicts in classroom/faculty assignments. Admissions use the registration system to add new students. Enrollment services use the registration system to report on retention, update student information, and check fulfillment of graduation requirements for those students planning to graduate.

Types of Requirements

- **Functional requirements**
 - Describe the interactions between the system and its environment independent from the implementation
“An operator must be able to define a new game.”
- **Nonfunctional requirements**
 - Aspects not directly related to functional behavior.
“The response time must be less than 1 second”
- **Constraints**
 - Imposed by the client or the environment
 - “The implementation language must be Java”
 - Called “**Pseudo requirements**” in the text book.

Functional vs. Nonfunctional Requirements

Functional Requirements

- Describe user tasks that the system needs to support
- Phrased as actions
 - “Advertise a new league”
 - “Schedule tournament”
 - “Notify an interest group”

Nonfunctional Requirements

- Describe properties of the system or the domain
- Phrased as constraints or negative assertions
 - “All user inputs should be acknowledged within 1 second”
 - “A system crash should not result in data loss”.

Types of Nonfunctional Requirements

- Usability
 - Reliability
 - Robustness
 - Safety
 - Performance
 - Response time
 - Scalability
 - Throughput
 - Availability
 - Supportability
 - Adaptability
 - Maintainability
 - Implementation
 - Interface
 - Operation
 - Packaging
 - Legal
 - Licensing (GPL, LGPL)
 - Certification
 - Regulation
- Constraints or
Pseudo requirements

Quality requirements

Nonfunctional Requirements: Examples

- “Spectators must be able to watch a match without prior registration and without prior knowledge of the match.”
? *Usability Requirement*
- “The system must support 10 parallel tournaments”
? *Performance Requirement*
- “The operator must be able to add new games without modifications to the existing system.”
? *Supportability Requirement*

USE CASE SPECIFICATION

(detailed version of a use case)

Use Case Specification

1. **Brief Description** - (about the use case)
2. **Actors** – list out the actors involved
3. **Preconditions** - A textual description that defines any constraints on the system at the time the use case may start.
4. **Basic flow of events** - describes what "normally" happens when the use case is performed.
5. **Alternative flows** - covers behavior of an optional or exceptional character relative to normal behavior, and also variations of the normal behavior. Think of the alternate flows of events as "detours" from the basic flow of events.
6. **Postconditions** - A textual description that defines any constraints on the system at the time the use case will terminate.

Use-Case Specification: Withdraw Cash

1. Brief Description

This use case describes how a Bank Customer uses an ATM to withdraw money from a bank account.

2. Actors

- Customer
- Bank

3. Preconditions:

- The bank Customer must possess a **bank card**.
- The network connection to the **Bank System** must be active.
- The system must have at least some cash that can be dispensed.
- The cash withdrawal **service option** must be available.

4. Basic Flow of Events

4.1 Insert Card

- The use case begins when the actor **Customer** inserts his/her **bank card** into the card reader on the ATM.
- The system allocates an **ATM session identifier** to enable errors to be tracked and synchronized between the ATM and the Bank System.

4.2 Read Card

- The system reads the **bank card information** from the card.

4.3 Authenticate Customer

- Authenticate the use of the **bank card** by the individual using the machine

4.4 Select Withdrawal

- The system displays the **service options** that are currently available on the machine.
- The Customer selects to withdraw cash.

4.5 Select Amount

- The system prompts for the amount to be withdrawn by displaying the list of standard withdrawal amounts.
- The Customer selects an amount to be withdrawn.

4.6 Confirm Withdrawal

- Customer gives conformation to withdraw cash

4.7 Eject Card

- The system ejects the Customer's **bank card**.
- The Customer takes the **bank card** from the machine.

4.8 Dispense Cash

5. Alternative Flows

5.1 Invalid User

5.2 Amount Exceeds Withdrawal limit

5.3 Insufficient cash

5.4 Money not removed

6. Postconditions

- The ATM has returned the card and dispensed the cash to the Customer and the withdrawal is registered on the Customer's account.
- The ATM has returned the card to the Customer and no withdrawal is registered on the Customer's account.
- The ATM has returned the card but has not supplied the amount of cash registered as withdrawn from the Customer's account. The discrepancy is registered in the ATM's log.
- The ATM has kept the card, no withdrawal has registered on the Customer's account and the Customer has been notified where to contact for more information.

Write a use case specification for

- i) **book ticket** use case in a **railway ticket booking system**.
- ii) **Order item** use case in an **online shopping website**.

Millennium Travel Corporation (MTC) travel agency plans to become a market leader by augmenting its human travel agents with an automated travel agent system for processing flight reservations. The automated travel agent will intermediate between travelers and the MTC corporate computing system, which interfaces with commercial airline reservation services. Like a human travel agent, it will assist travelers in booking, changing, and canceling flight reservations. If, for any reason, a traveler making a flight reservation travel request prefers human assistance, she will have the option to interact directly with a human travel agent.

The MTC automated travel agent system will process a wide range of flight reservation service requests. These include but are not limited to: inquiring about flights and airfares, making, changing, and canceling traveler profiles and accounts, booking, changing, confirming, and canceling flight reservations, generating travel itineraries.

A user with a valid system account and a valid travel account logs in to the system, requests to book a flight reservation, selects a flight, selects a payment method, and specifies delivery services for the flight tickets and travel itineraries.

The travel agent system must be capable of providing fast, accurate, and courteous ("user friendly") services for all requests supported. The system must be able to answer inquiries about flights and fares, generate, modify, and cancel traveler profiles and travel accounts, make, change, complete, and cancel reservations, obtain payment method and verify traveler credit line, generate travel itineraries and arrange for delivery of flight tickets and flight itineraries.

Scenario-Based Modeling

Activity Diagram : Control Flow

Analysis Modeling Approaches

- Structured analysis
 - Considers data and the processes that transform the data as separate entities
 - Data is modeled in terms of only attributes and relationships (but no operations)
 - Processes are modeled to show the 1) input data, 2) the transformation that occurs on that data, and 3) the resulting output data
- Object-oriented analysis
 - Focuses on the definition of classes and the manner in which they collaborate with one another to fulfill customer requirements

Elements of the Analysis Model

Object-oriented Analysis

Scenario-based modeling

- Use case text
- Use case diagrams
- Activity diagrams
- Swim lane diagrams

Procedural/Structured Analysis

Flow-oriented modeling

- Data structure diagrams
- Data flow diagrams
- Control-flow diagrams
- Processing narratives

Class-based modeling

- Class diagrams
- Analysis packages
- CRC models
- Collaboration diagrams

Behavioral modeling

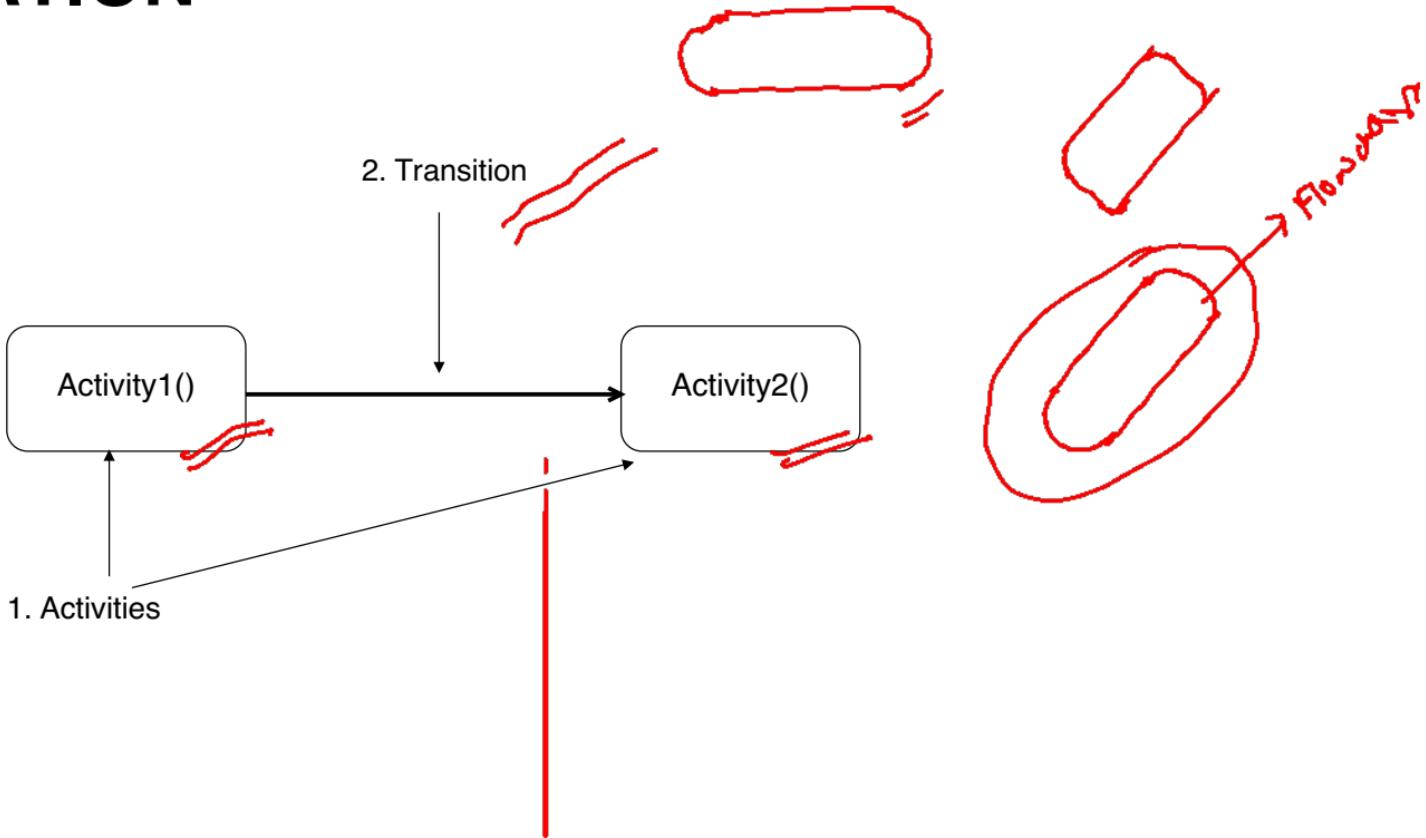
- State diagrams
- Sequence diagrams

Developing an Activity Diagram

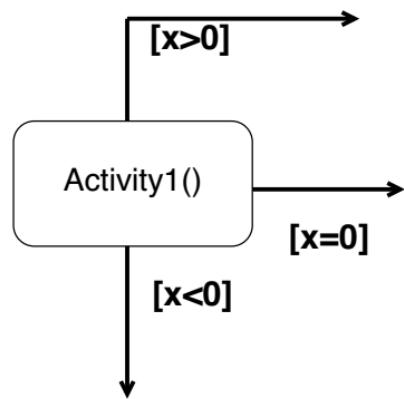
Activities

- Describes how activities are coordinated.
- Is particularly useful when you know that an operation has to achieve a number of different things, and you want to model what the essential dependencies between them are, before you decide in what order to do them.
- Record the dependencies between activities, such as which things can happen in parallel and what must be finished before something else can start.
- Represents the workflow of the process.

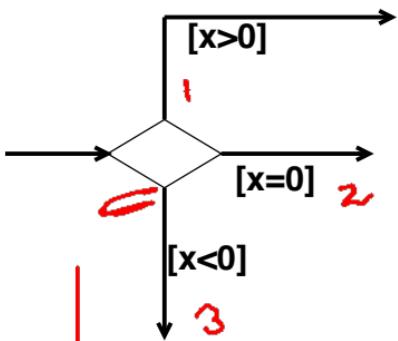
NOTATION



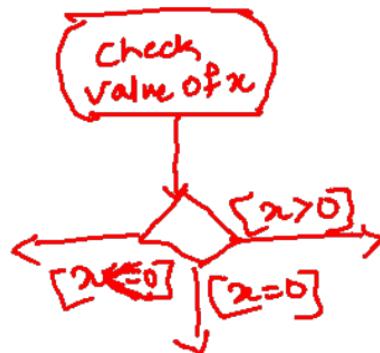
NOTATION - 2



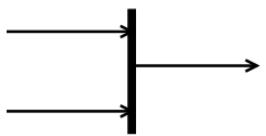
3. Decision Diamond



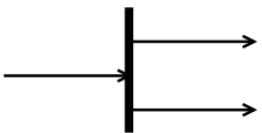
[]
Guard cond'n



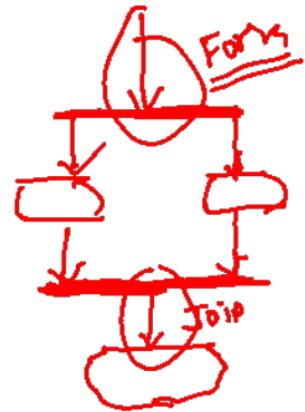
NOTATION - 3



4.1 Synch. Bar (Join)



4.2 Splitting Bar (Fork)



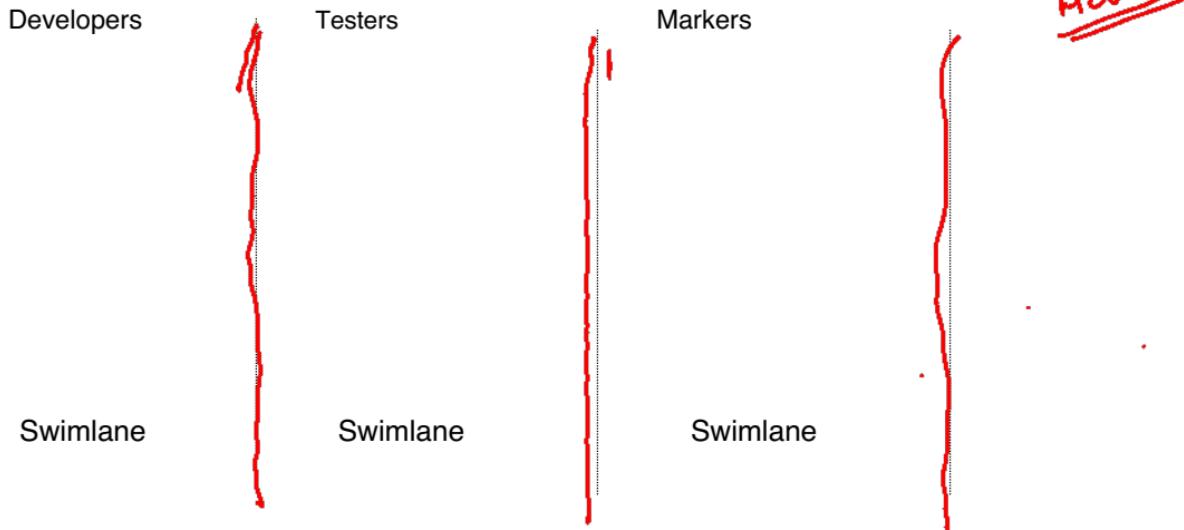
Notation - 3



5. Start & Stop Markers

Notation - 4

- Simple Activity diagram
- swimlane diagram ✓



Application/Department/Group/Role Boundaries

Activity Diagrams

- Activity diagrams commonly contain
 - Activity states and action states
 - Transitions
 - Objects

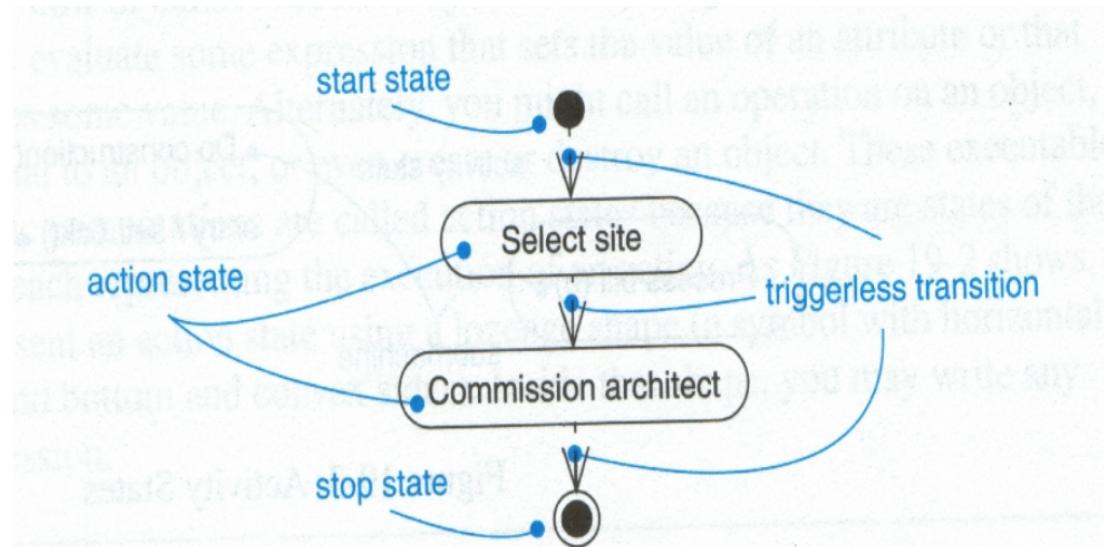
Action States and Activity States

- Action states are atomic and cannot be decomposed
 - Work of the action state is not interrupted
- Activity states can be further decomposed
 - Their activity being represented by other activity diagrams
 - They may be interrupted

Transitions

- When the action or activity of a state completes, flow of control passes immediately to the next action or activity state
- A flow of control has to start and end someplace
 - initial state -- a solid ball  *Start*
 - stop state -- a solid ball inside a circle  *Stop*

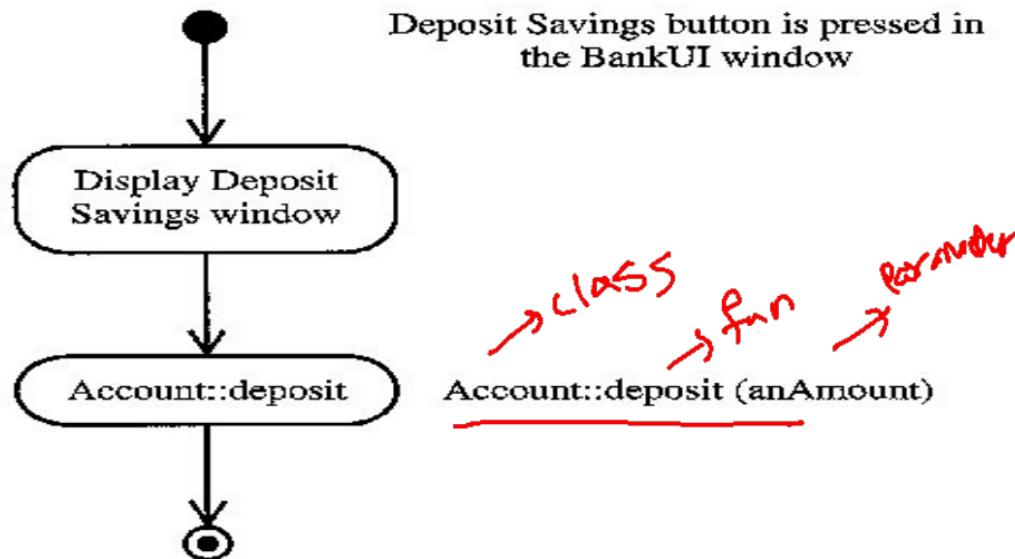
Transitions



Activity Diagram: Example (1)

FIGURE 12-24

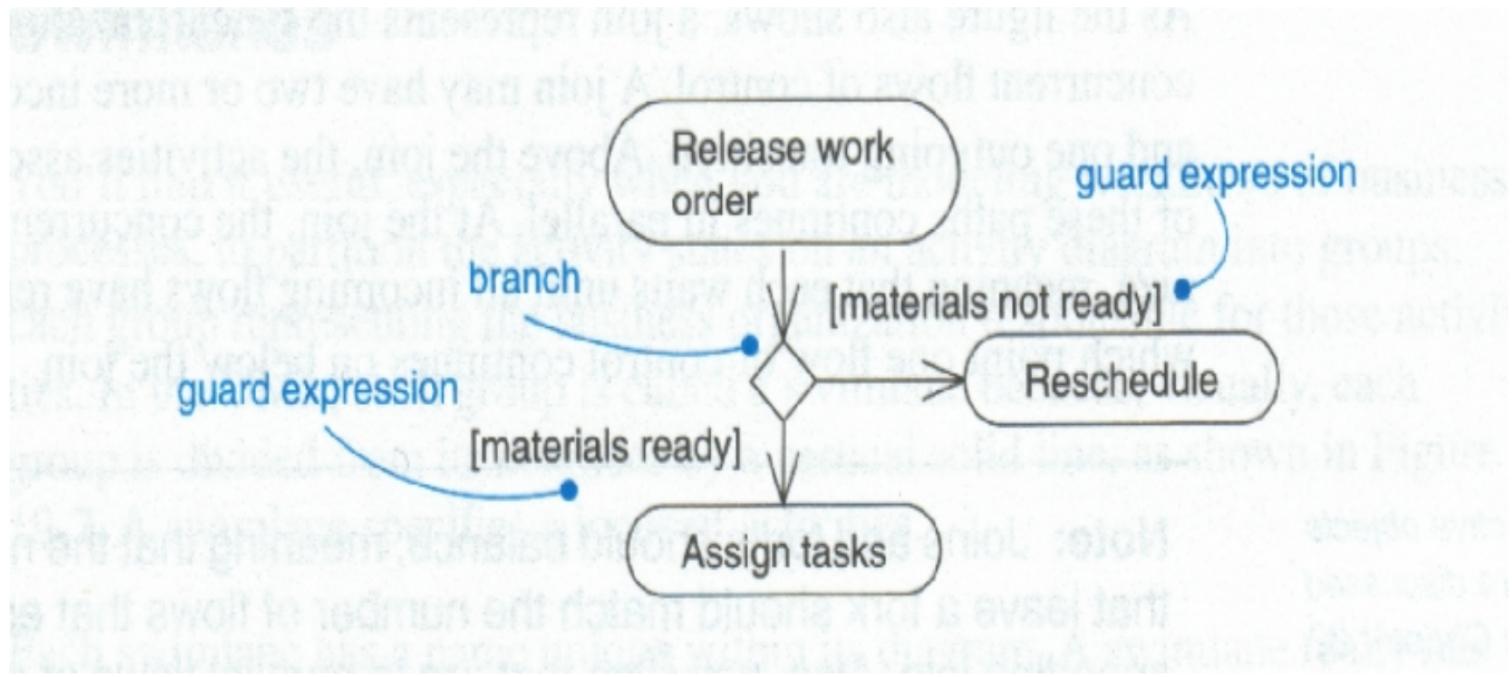
Activity diagram for processing a deposit to a savings account.



Branching

- A branch specifies alternate paths taken based on some Boolean expression
- A branch may have one incoming transition and two or more outgoing ones

Branching



Activity Diagram: Example

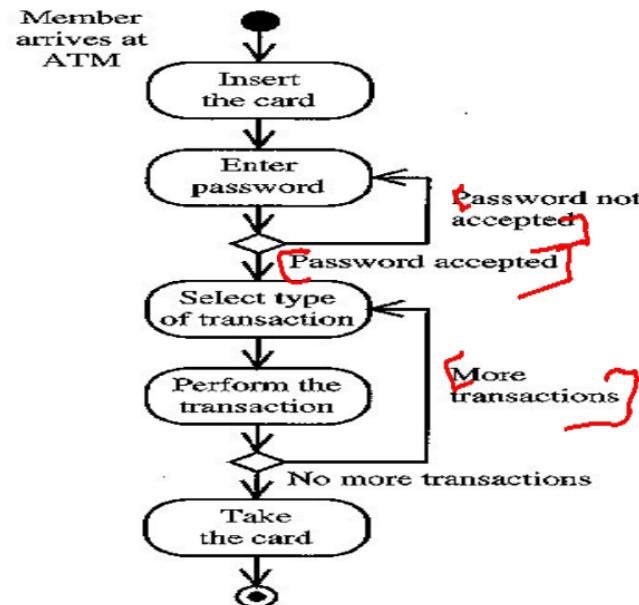


FIGURE 6-8

Activities involved in an ATM transaction.

Forking and Joining

- Use a synchronization bar to specify the forking and joining of parallel flows of control
- A synchronization bar is rendered as a thick horizontal or vertical line

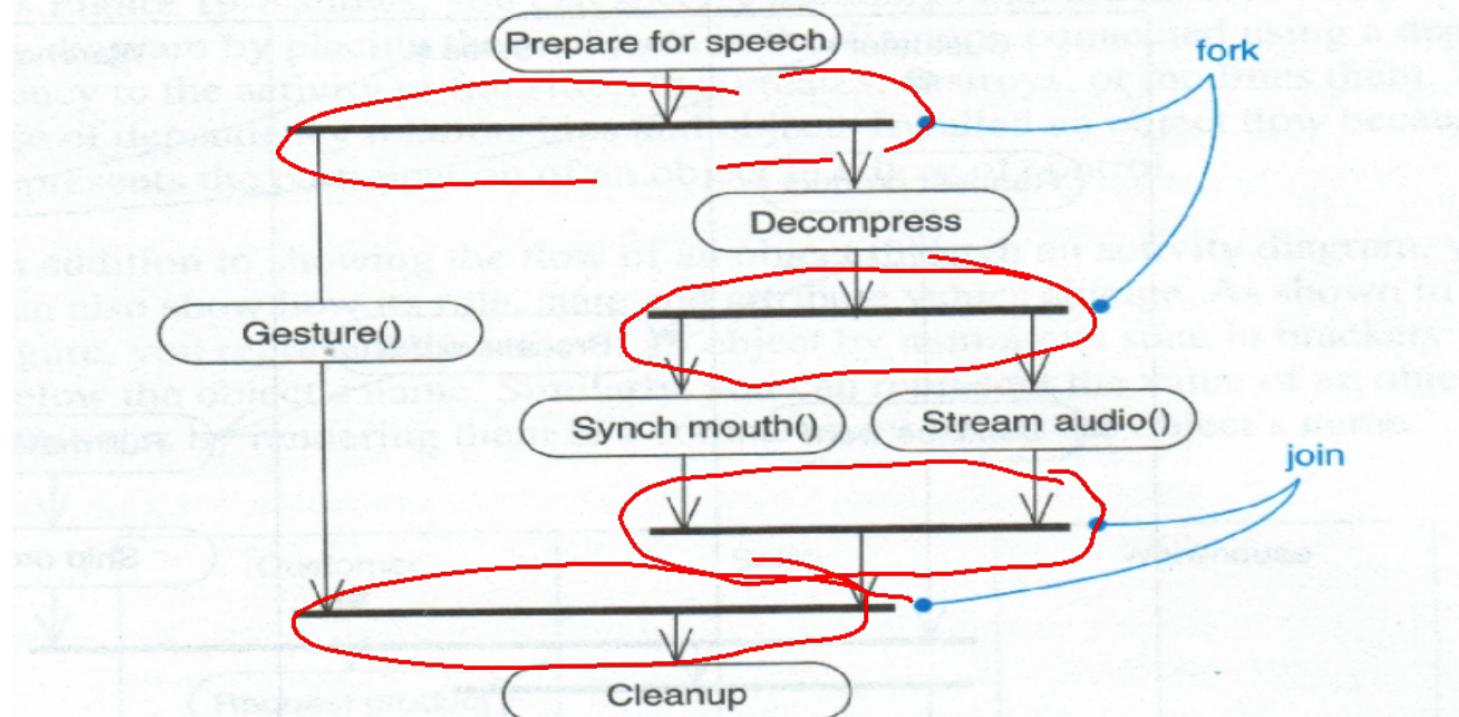
Fork

- A fork may have one incoming transitions and two or more outgoing transitions
 - each transition represents an independent flow of control
 - conceptually, the activities of each of outgoing transitions are concurrent
 - either truly concurrent (multiple nodes)
 - or sequential yet interleaved (one node)

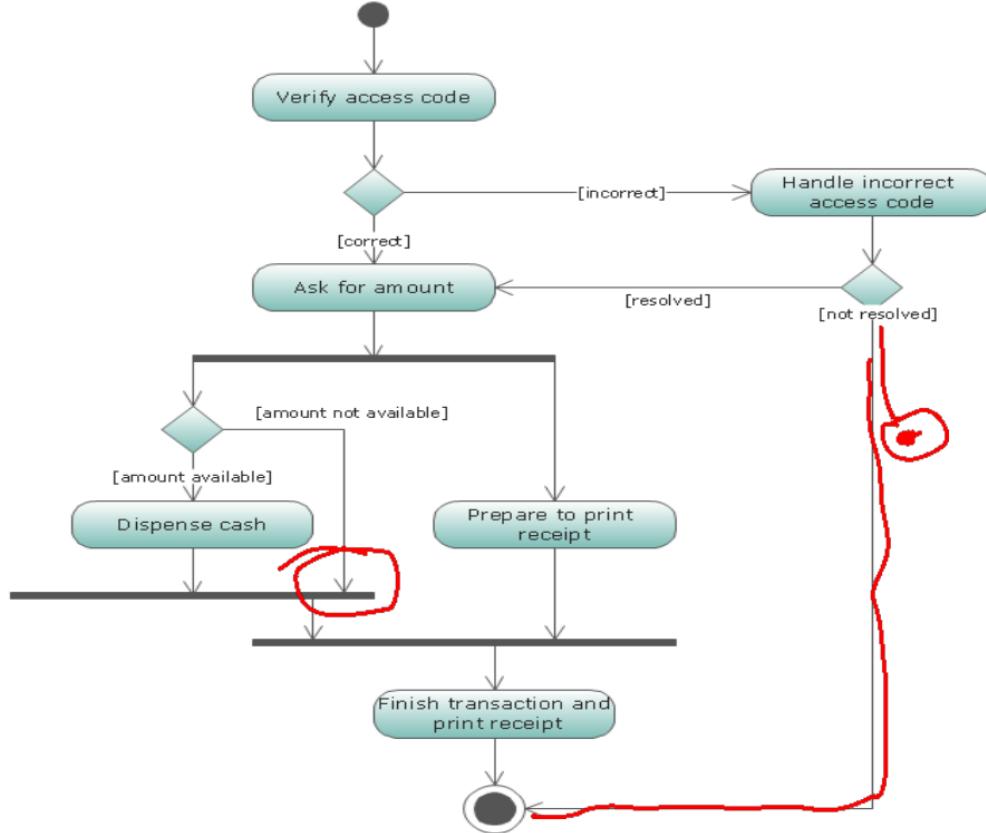
Join

- A join may have two or more incoming transitions and one outgoing transition
 - above the join, the activities associated with each of these paths continues in parallel
 - at the join, the concurrent flows synchronize
 - each waits until all incoming flows have reached the join, at which point one flow of control continues on below the join

Fork and Join Example



UML Activity Diagram



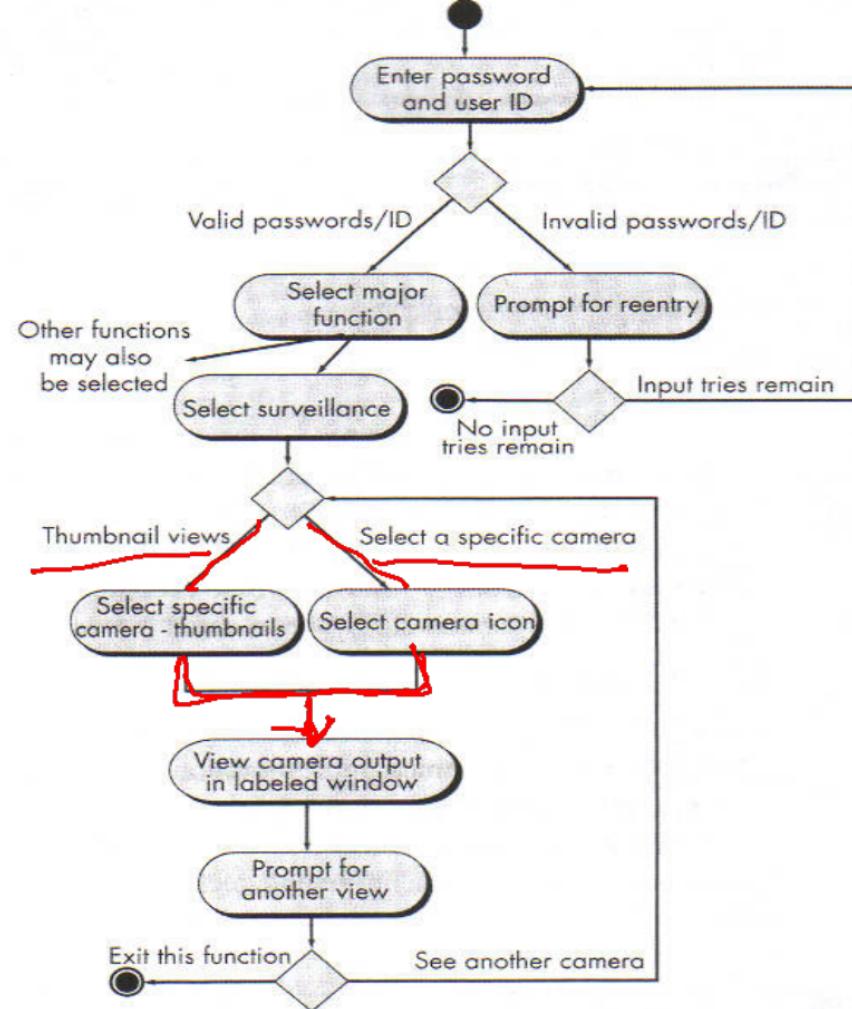
Revision: Activity Diagram

- **Rounded rectangle** [] actions.
- **Diamond** [] decisions/merge
- **Bars** [] The start (fork) or end (join) of concurrent activities.
- **Black circle** [] start (initial state)
- **Encircled black** [] end (final state)
- **Arrow** [] flow of control

Scenario:

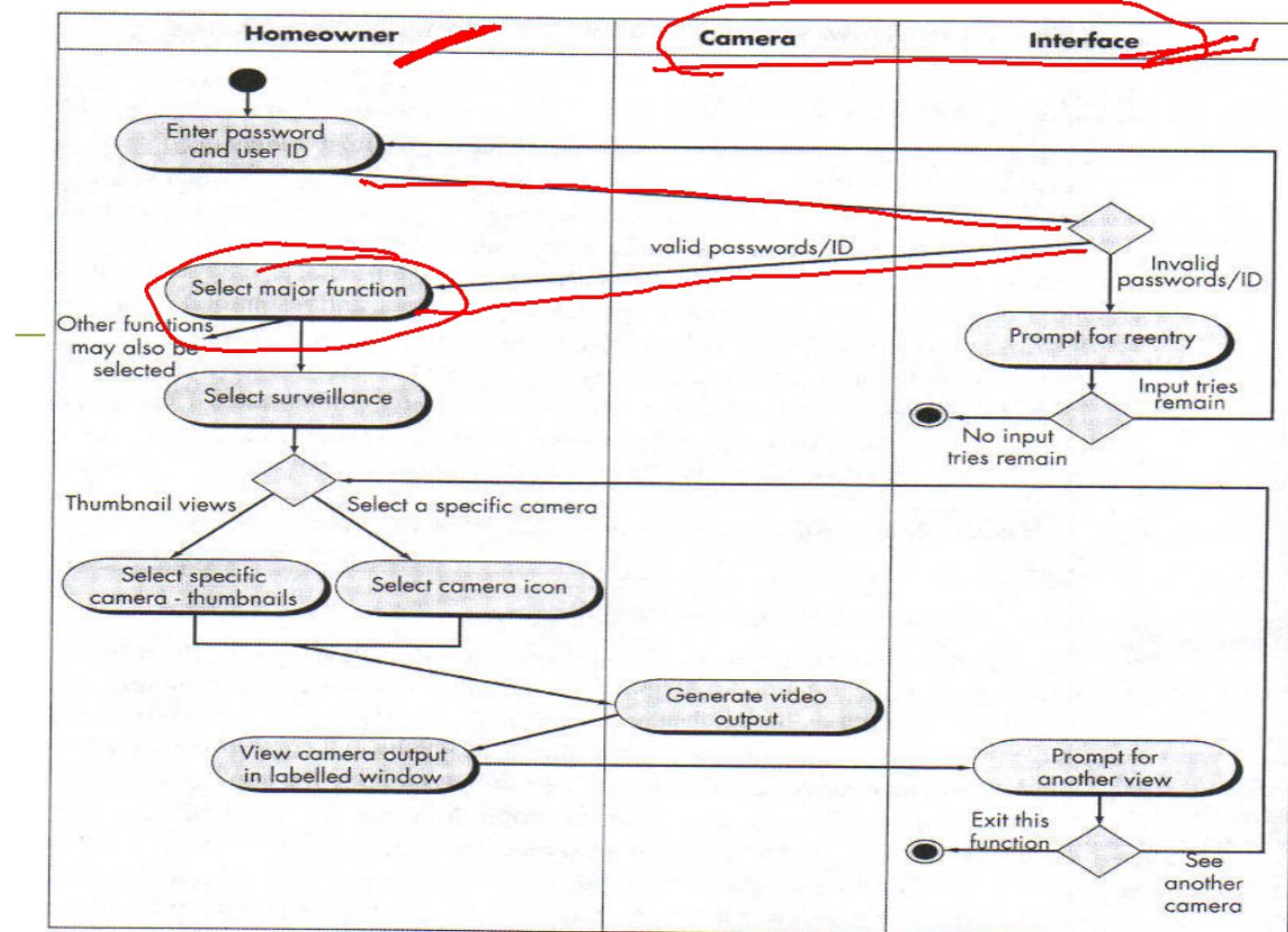
1. The homeowner logs onto the *SafeHome Products* Web site.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects "surveillance" from the major function buttons.
6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the "view" button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

Activity diagram for Access camera surveillance—display camera views function

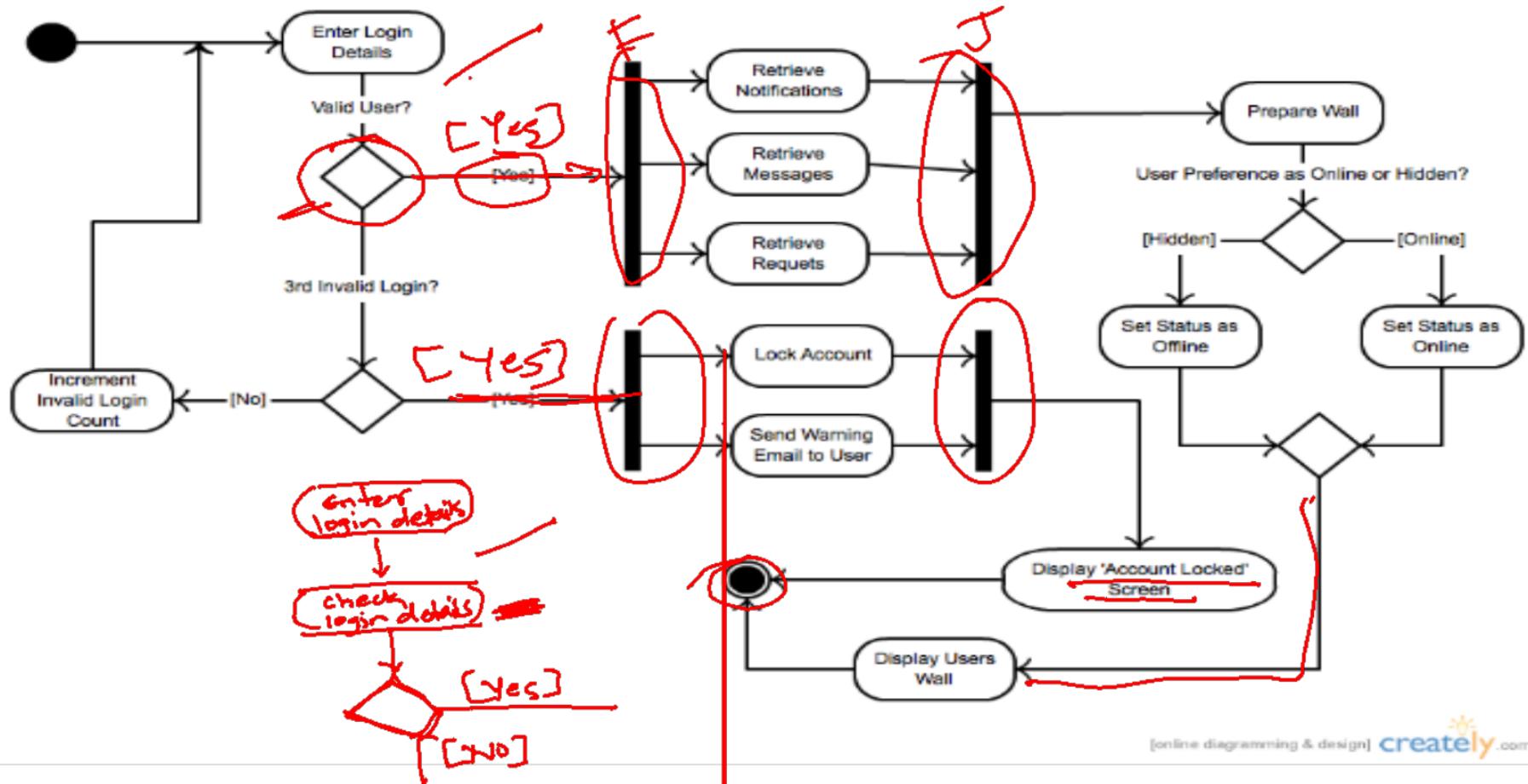


Swimlane Diagrams

- A variation of activity diagram
- Indicate which actor has responsibility for the action described by an activity rectangle
- A UML swimlane diagram represents the flow of actions, decisions and indicates which actors perform each



Facebook is an online social media website, where the users must register before using the it. Once the user is authenticated by site, he can retrieve notifications, messages and requests. Further, in the next step the user can change the user preferences as online or hidden. Finally the status will be displayed in user's wall. User has option to login by entering credentials for three times. Third unsuccessful login will lock the account, send warning email of the user and display 'account locker' message on the screen. Else the system will prompt for login credentials again.

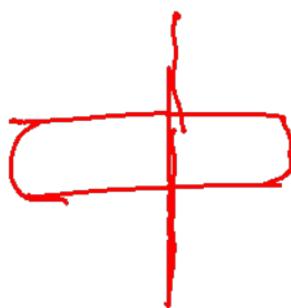


Swimlanes (1)

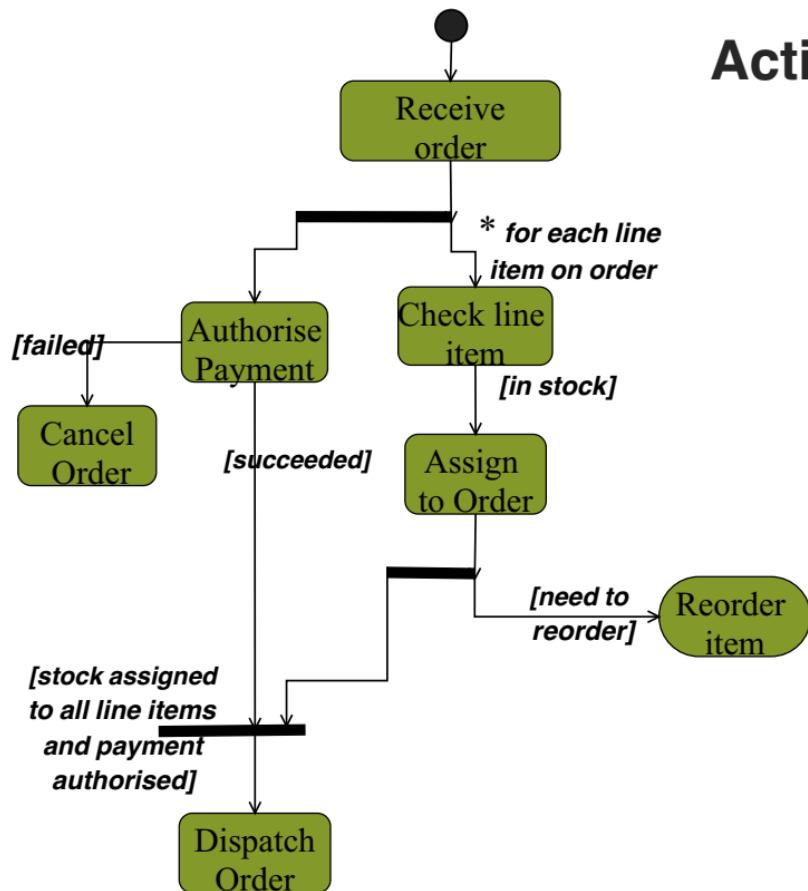
- A swimlane specifies a locus of activities
- To partition the activity states on an activity diagram into groups
 - each group representing the business organization responsible for those activities
 - each group is called a swimlane
- Each swimlane is divided from its neighbor by a vertical solid line

Swimlanes (2)

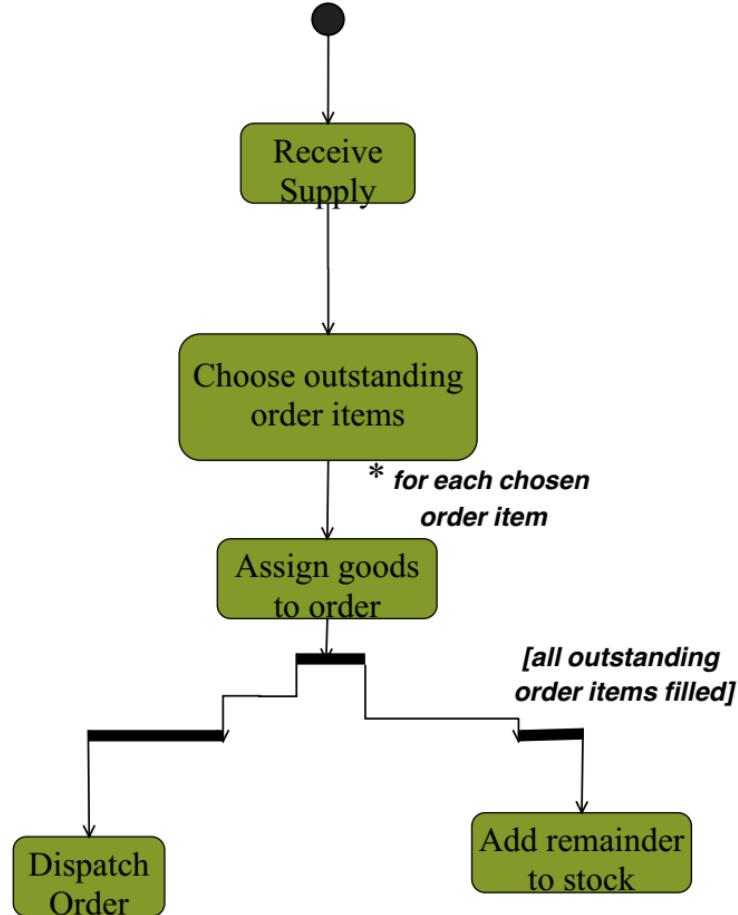
- Each swimlane has a name unique within its diagram
- Each swimlane may represent some real-world entity
- Each swimlane may be implemented by one or more classes
- Every activity belongs to exactly one swimlane, but transitions may cross lanes



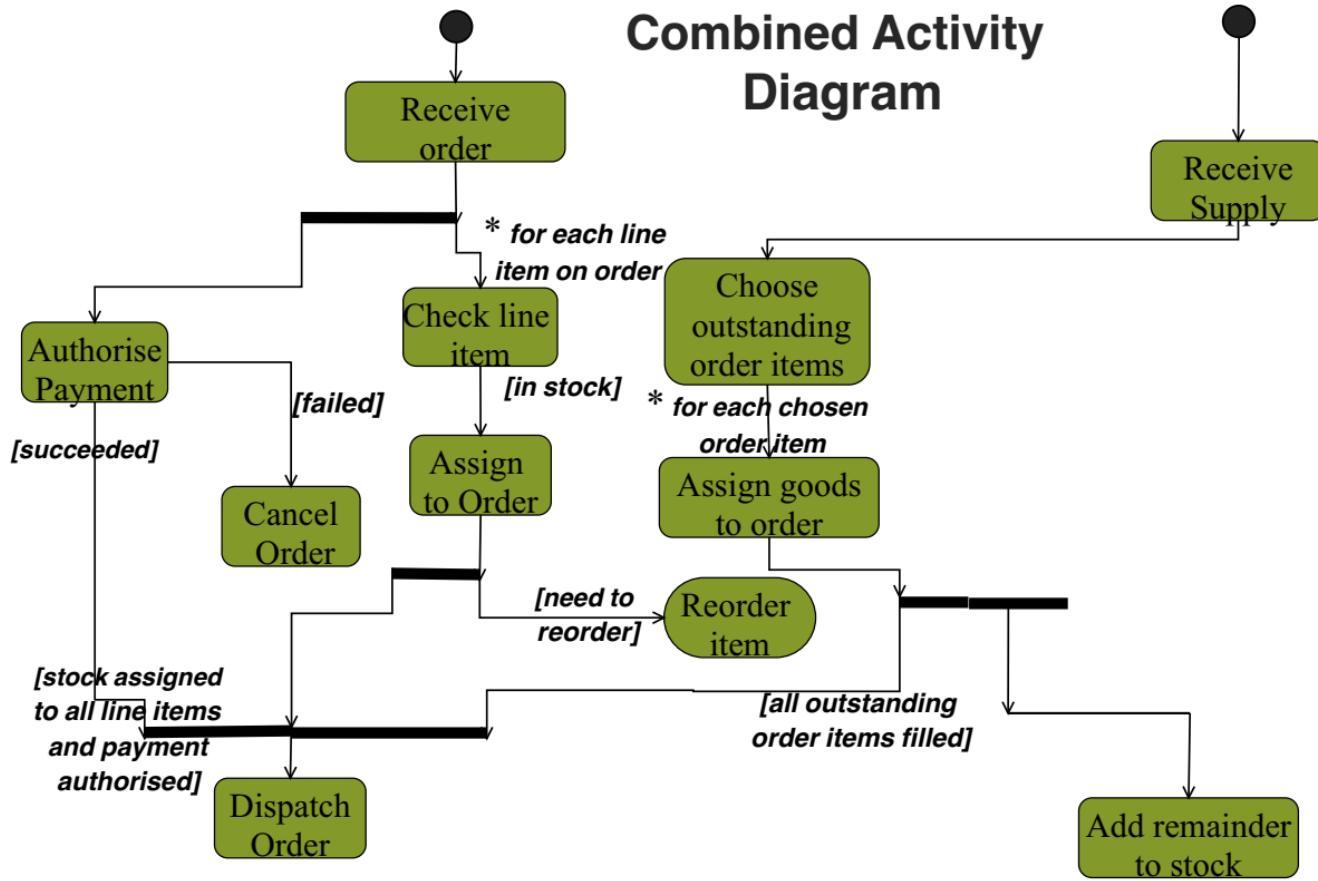
Activity Diagram for Receiving an Order

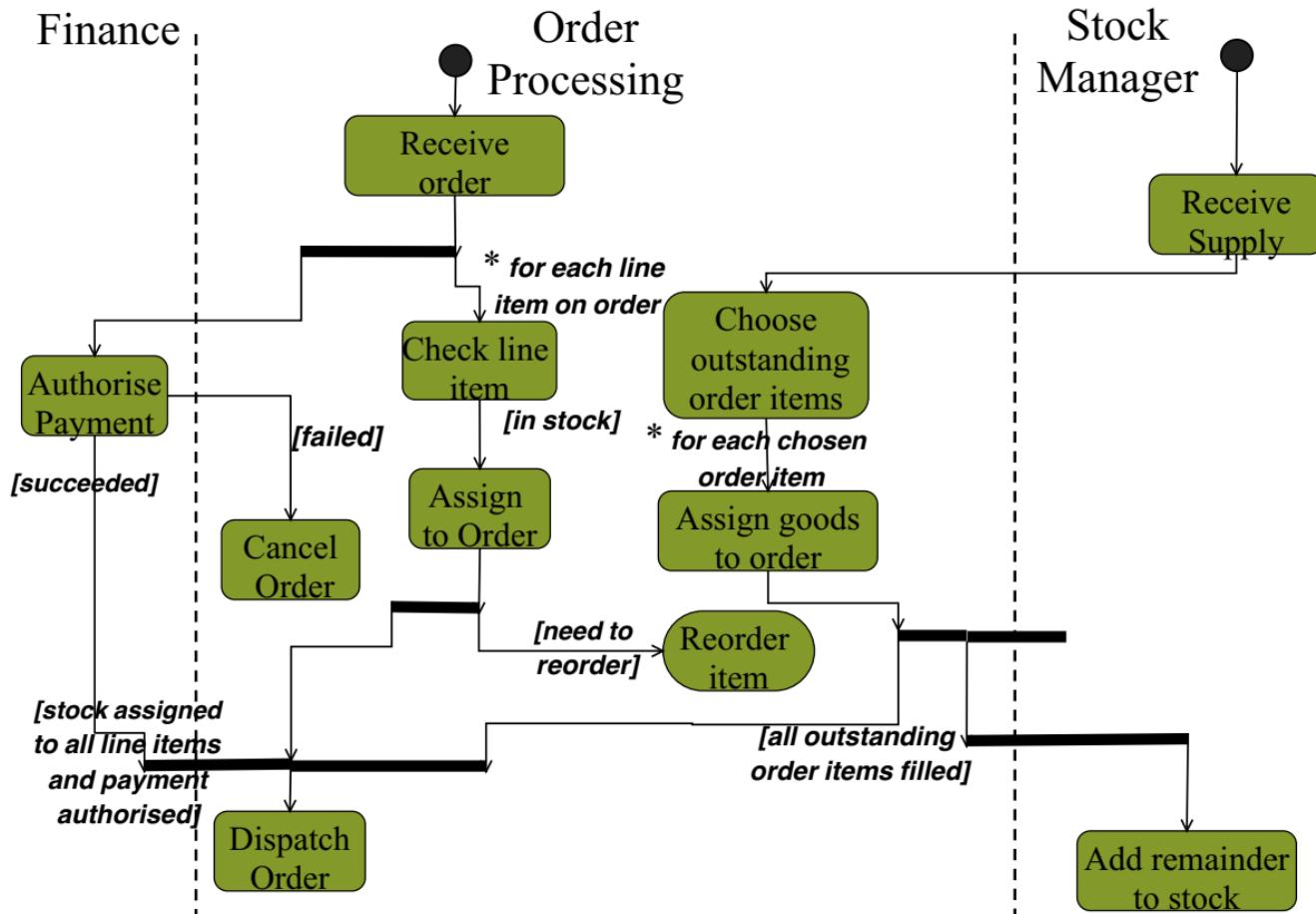


Activity Diagram for receiving Supply

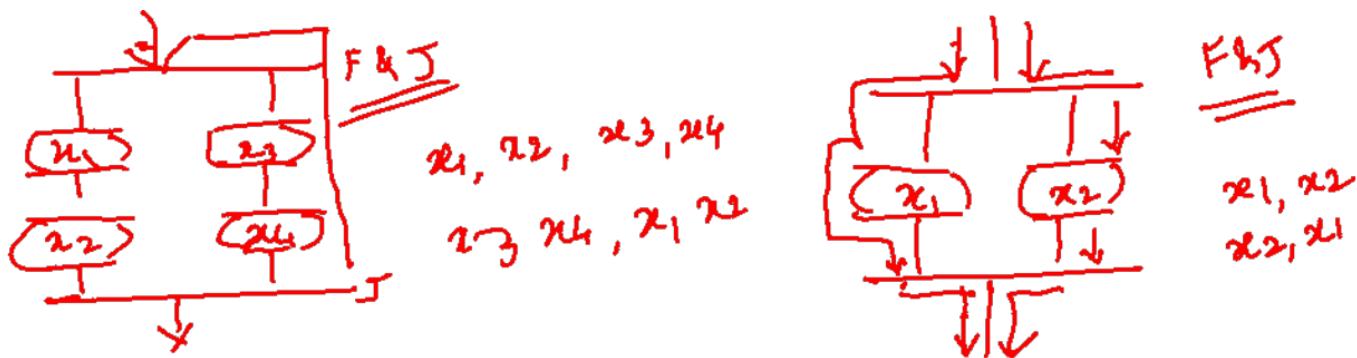


Combined Activity Diagram





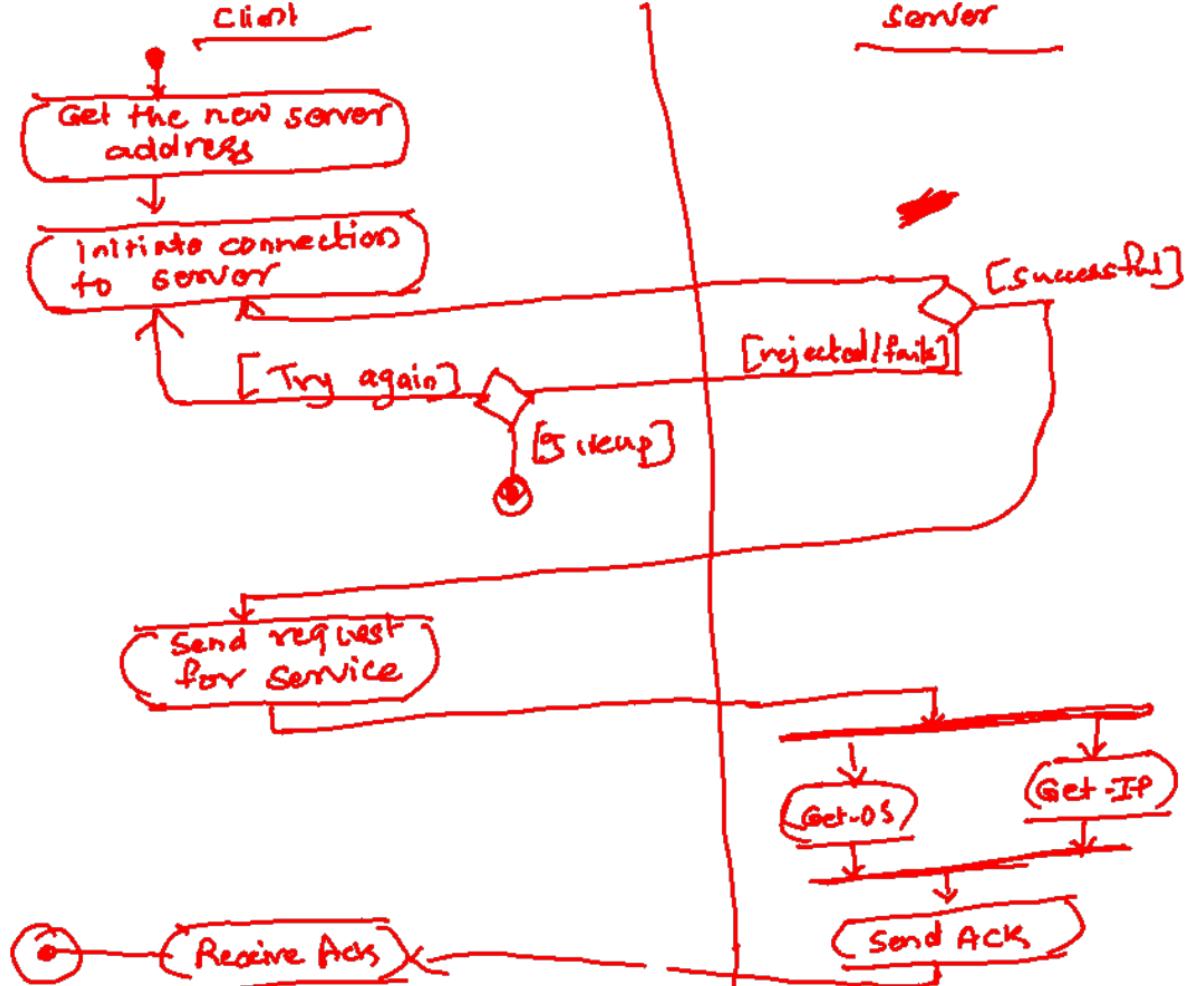
With Swimlanes



- The client needs to know the whole address of a server for the communication. Client performs some work which includes, making a decision to initiate a connection to a server. If connection to the server fails, or the server rejects the connection, the client may try again or may give up. On successful connection, the client reads the request to execute the services. GET_OS and GET_IP are the services which will execute parallel and acknowledgement will be sent to the client by the server.

Draw an activity diagram.
Swimlane





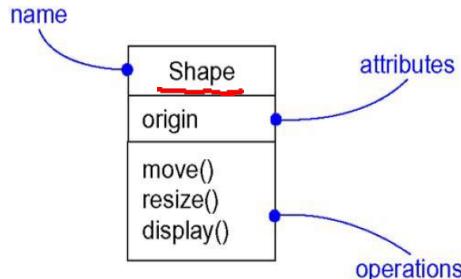
- The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a personal identification number (PIN) - both of which will be sent to the bank for validation as part of each transaction. The customer will then be able to perform one or more transactions.
- The card will be retained in the machine until the customer indicates that he/she desires no further transactions, at which point it will be returned. The ATM will communicate each transaction to the bank and obtain verification that it was allowed by the bank. If the bank determines that the customer's PIN is invalid, the customer will be required to re-enter the PIN before a transaction can proceed.
- If the customer is unable to successfully enter the PIN after three tries, the card will be permanently retained by the machine, and the customer will have to contact the bank to get it back. If a transaction fails for any reason other than an invalid PIN, the ATM will display an explanation of the problem, and will then ask the customer whether he/she wants to do another transaction.

- The ATM will also maintain an internal log of transactions to facilitate resolving ambiguities arising from a hardware failure in the middle of a transaction. Entries will be made in the log when the ATM is started up and shut down, for each message sent to the Bank (along with the response back, if one is expected), for the dispensing of cash, and for the receiving of an envelope.
- Log entries may contain card numbers and amounts, but for security will never contain a PIN. To avail ATM facility, a customer is required to open/have an account in the bank and apply for the ATM card. A customer can have one or more accounts and for each account, only one ATM card will be provided.
- The bank also provides SMS updates for every transaction of customer's account. To obtain SMS updates, customer is required to register his/her mobile number against his account in the bank.

CLASS DIAGRAM

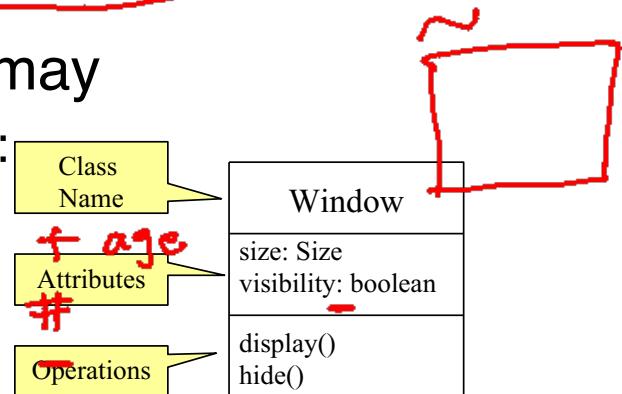
Introduction

- Most important building block
- Description of a set of objects that share the same attributes, operations, relationships and semantics
- Used to capture the vocabulary of the system
 - Include abstractions of the problem domain
 - Classes that make up the implementation
- Form a part of a balanced distribution of responsibilities across the system



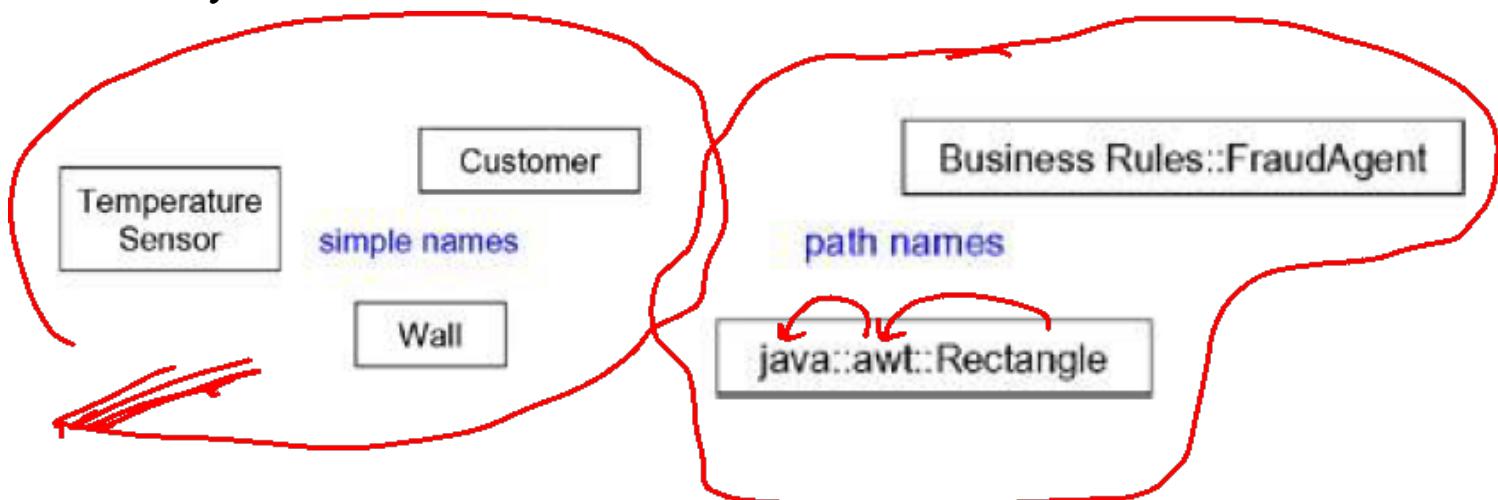
Class

- Describes a set of objects having similar:
 - Attributes (status)
 - Operations (behavior)
 - Relationships with other classes
- Attributes and operations may
 - have their visibility marked:
 - "+" for *public*
 - "#" for *protected*
 - "-" for *private*
 - "~" for *package*



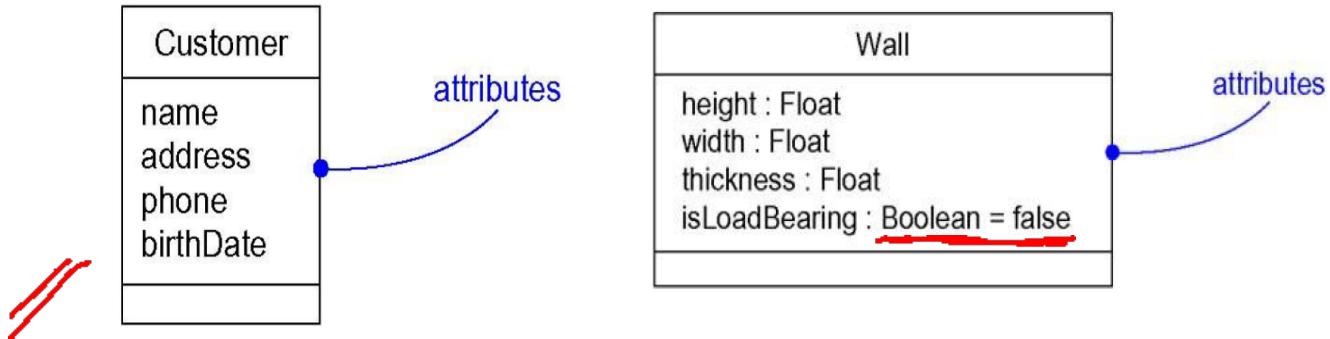
Class Names

- Name must be unique within its enclosing package
- Simple name and Path name
- Class names are noun phrases from the vocabulary of the system



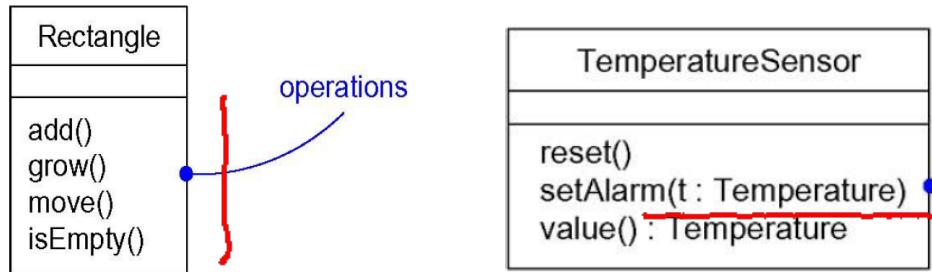
Attributes

- Named property of a class
 - Describes a range of values that instances of the property hold
 - Shared by all objects of that class
- An abstraction of the kind of data or state of an object
 - At a given moment, an object will have specific values for each of its attributes
- Noun phrase that represents some property



Operations

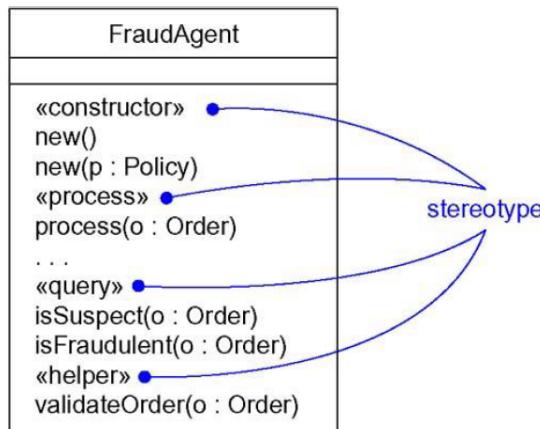
- A service that can be requested from any object of the class to affect behavior
- Invoking an operation changes the object's data or state
- Verb phrase representing some behavior of the class



obj : class

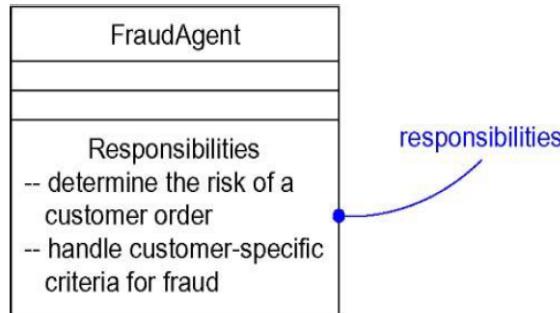
Organizing Attributes and Operations

- Don't have to show every attribute and operation at once
- End each list with ellipsis (...)
- Can also prefix each group with a descriptive category using stereotypes



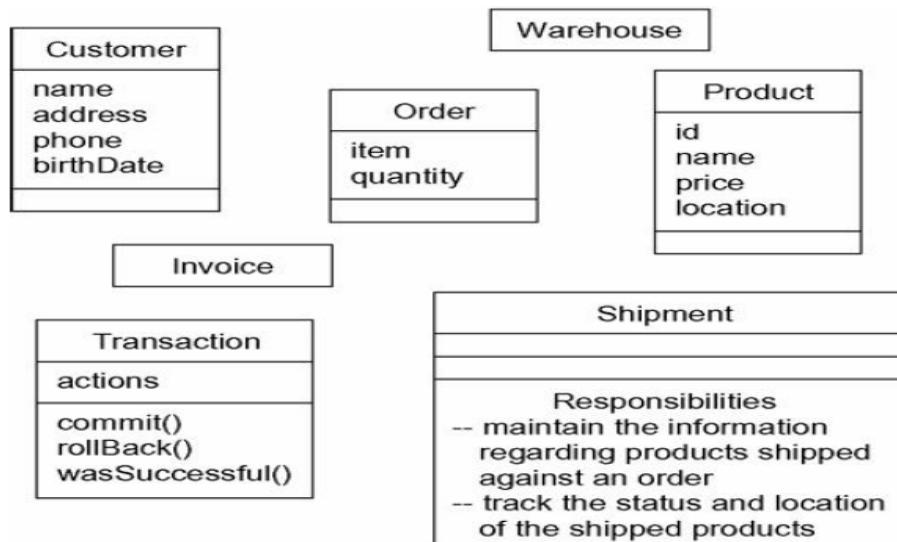
Responsibilities

- Is a contract or an obligation of a class
 - All objects of the class have the same kind of state and behavior
- Attributes and operations are the features that carry out the class's responsibility
 - TemperatureSensor class responsible for measuring temperature and raising an alarm
- Techniques used - CRC cards and use case based-analysis



Modeling the Vocabulary of a System

- Identify the things that describe the problem
- For each abstraction, identify a set of responsibilities
- Provide the attributes and operations needed to carry out those responsibilities



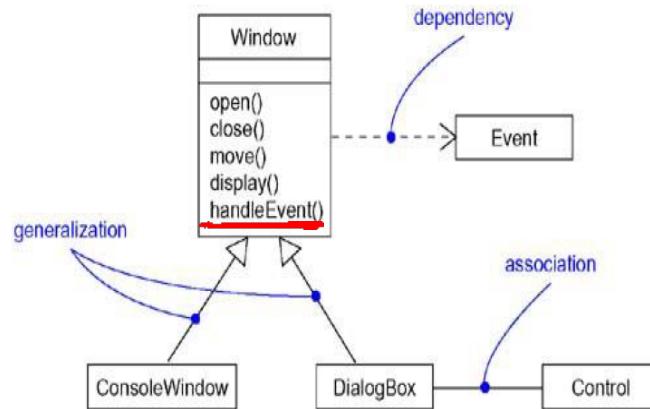
CLASS RELATIONSHIPS

Introduction

- Classes collaborate with other classes in a number of ways
- Three kinds of relationships in OO modeling
 - Dependencies
 - Generalizations
 - Associations
- Provide a way for combining your abstractions
- Building relationships is not like creating a balanced set of responsibilities

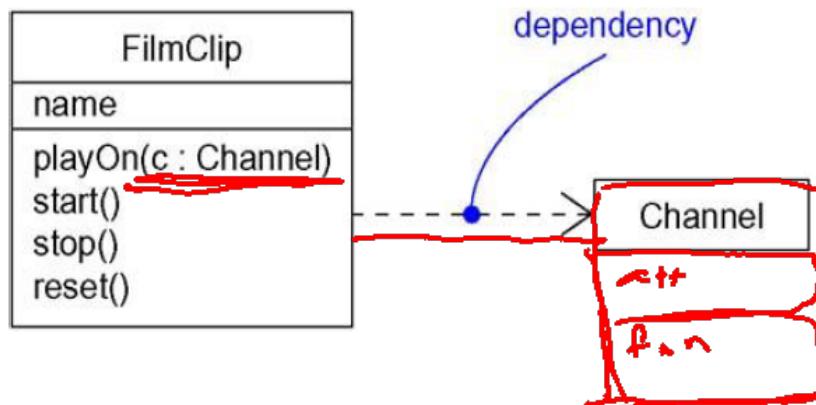
Introduction

- Relationship is a connection among things
- UML graphical notation
 - Permits to visualize relationships apart from any specific programming language
 - Emphasize the most important parts of a relationship
 - Name
 - Things it connects
 - Properties



Dependency

- *Using* relationship
 - States that a change in specification of one thing may affect another thing that uses it
- Used in the context of classes
 - Uses operations or variables/arguments typed by the other class
 - Shown as an argument in the signature of an operation
 - If used class changes, the operation of the other class may be affected



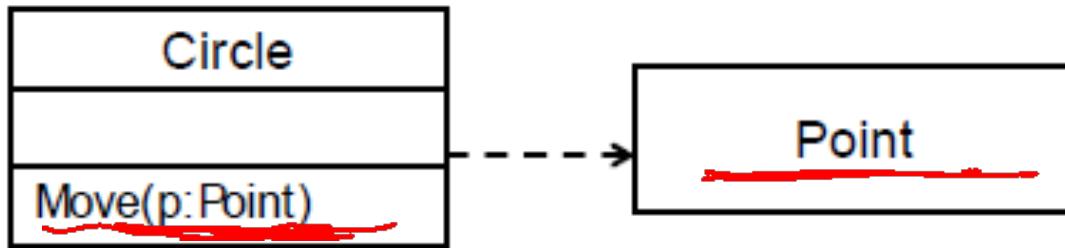
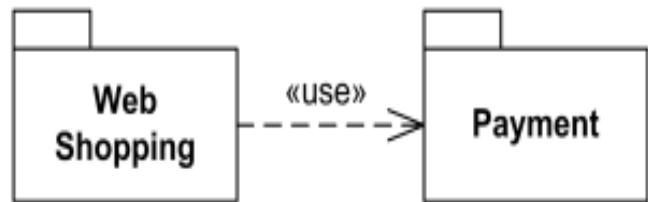
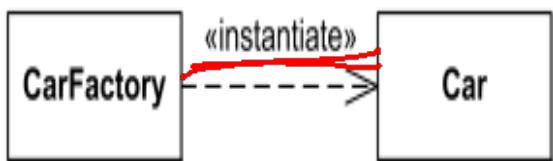
Dependency

- Dependency is a weaker form of relationship which indicates that one class depends on another because it uses it at some point in time.
- One class depends on another if the independent class is a parameter variable or local variable of a method of the dependent class.
- This is different from an association, where an attribute of the dependent class is an instance of the independent class.



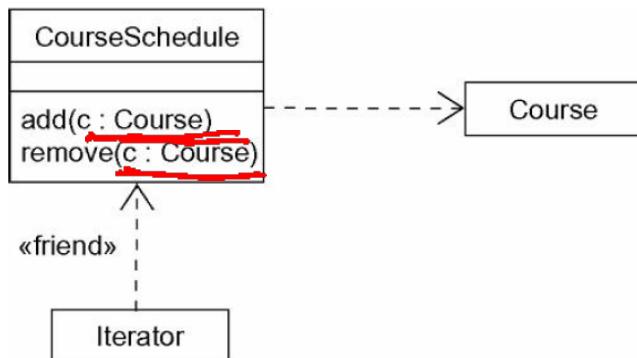
Dependency

*~ car class
car package*



Modeling Simple Dependencies

- To model the using dependency
 - Create a dependency pointing from the class with the operation to the class used as a parameter in the operation

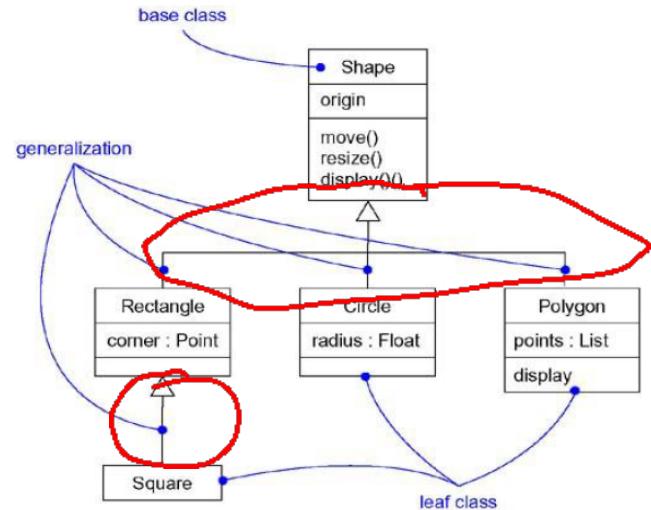


Generalization

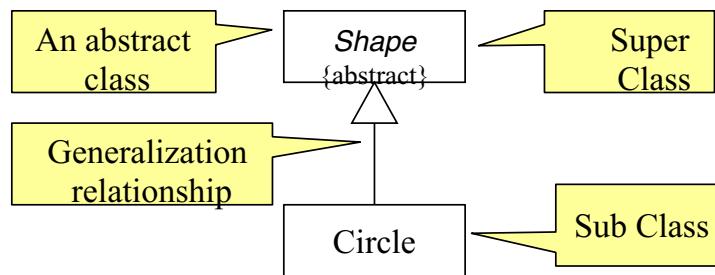
- Relationship between a general thing (superclass) and a more specific kind of that thing (subclass)
- “Is-a-kind-of” relationship
- Child is substitutable for the parent
- Polymorphism
 - Operation of child has the same signature as an operation in the parent but overrides it



Generalization



{abstract} is a tagged value that indicates that the class is abstract.
The name of an abstract class should be italicized



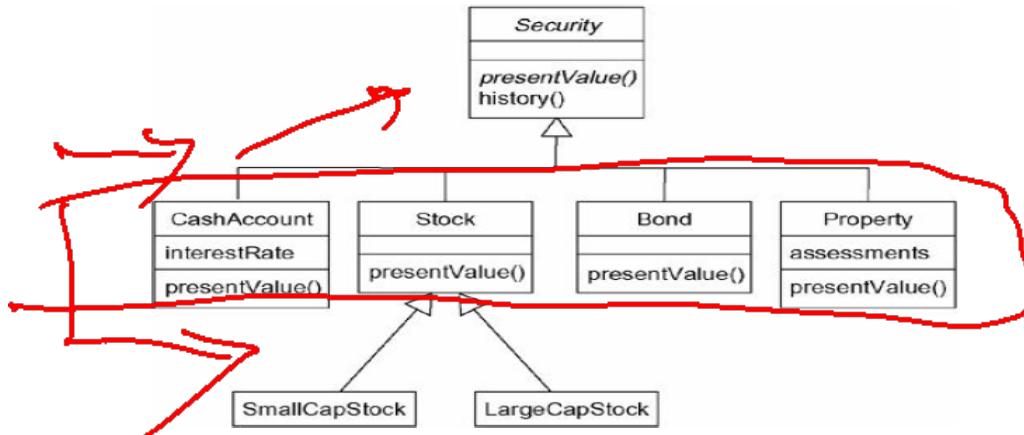
Generalization

- A sub-class inherits from its super-class
 - Attributes
 - Operations
 - Relationships
- A sub-class may
 - Add attributes and operations
 - Add relationships
 - Refine (override) inherited operations
- A generalization relationship **may not** be used to model interface implementation.



Modeling Single Inheritance

- Find classes that are structurally or behaviorally similar while modeling the vocabulary
- To model inheritance
 - Look for common responsibilities
 - Elevate these to a more general class
 - Specify that the more specific class inherits from the more general class



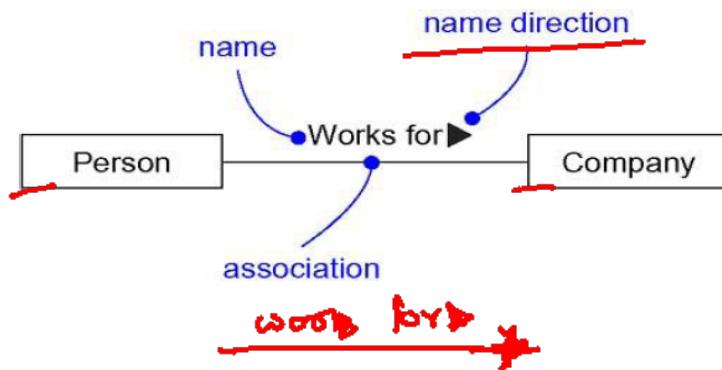
Association

- Structural relationship
 - Specifies that objects of one thing are connected to objects of another
- Can navigate from objects of one class to the other
- Self Association
 - Connects a class to itself
- Binary Association
 - Connects exactly two classes



Association : Adornments

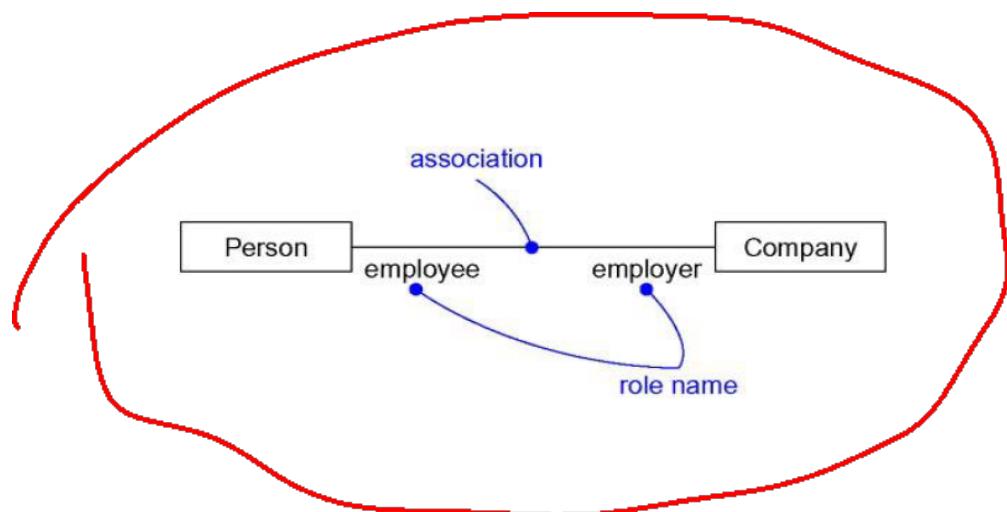
- Name
 - Use a name to describe the nature of the relationship
 - Can give a direction to the name



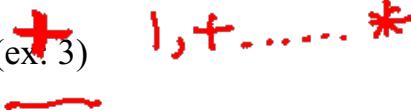
Association : Adornments

- Role

- The face that the class at the far end of the association presents to the class at the near end
- Explicitly name the role a class plays (*End name* or *Role name*)
- Same class can play the same or different roles in other associations



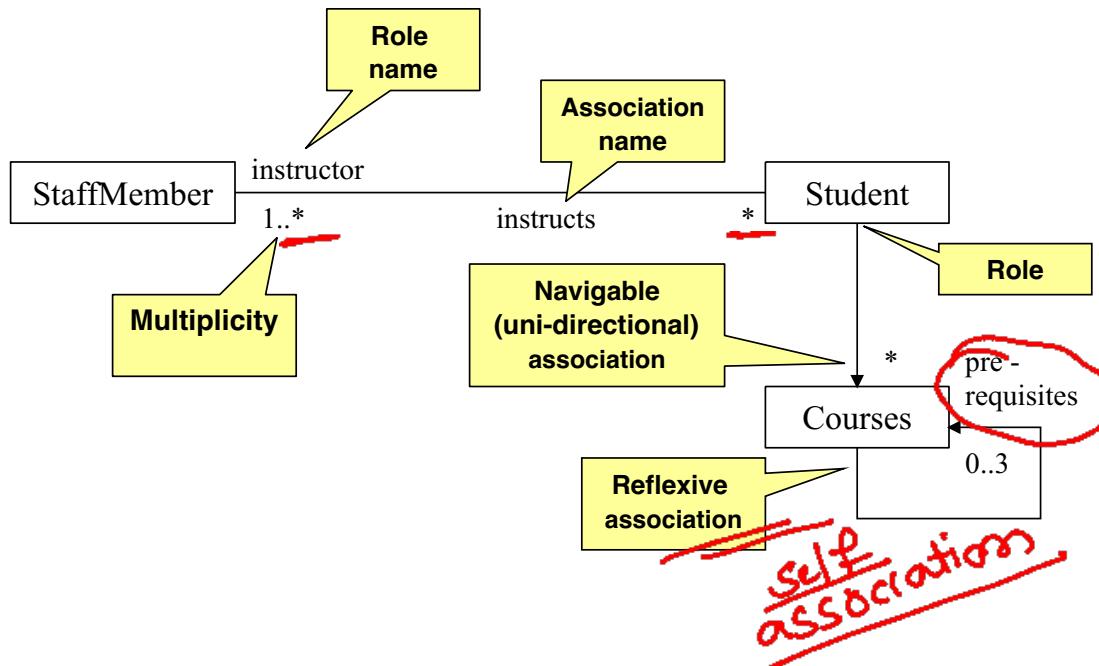
Association : Adornments

- Multiplicity
 - States how many objects may be connected across an instance of an association
 - An expression that evaluates to a range of values or an explicit value
 - Multiplicity at one end of an association means
 - For each object of the class at the opposite end, there must be that many objects at the near end
 - Show a multiplicity of
 - Exactly one (1)
 - Zero or one (0 .. 1)
 - Many (0 .. *)
 - One or more (1 .. *)

 - State an exact number (ex: 3)


Multiplicity

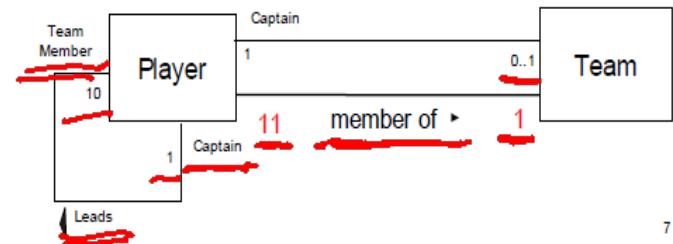
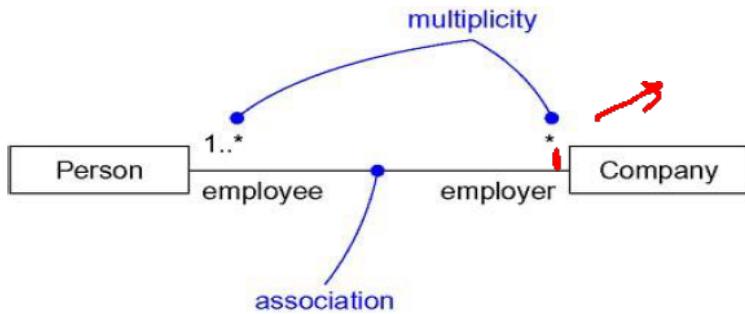
- 1:1
- 1:N
- N:1
- N:M
- Meta character – “*” and “+”:

Associations (cont.)

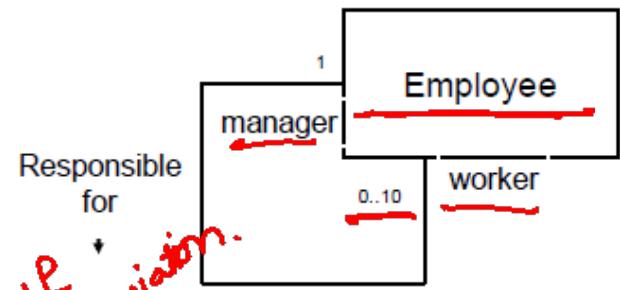
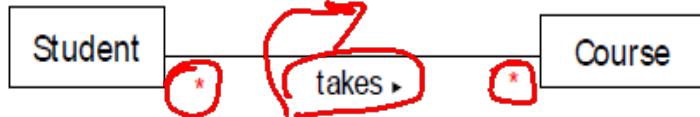


Association : Adornments

- Multiplicity
 - Specify complex multiplicities by using a list, 0..1, 3..4, 6..* (any number of objects other than 2 or 5)

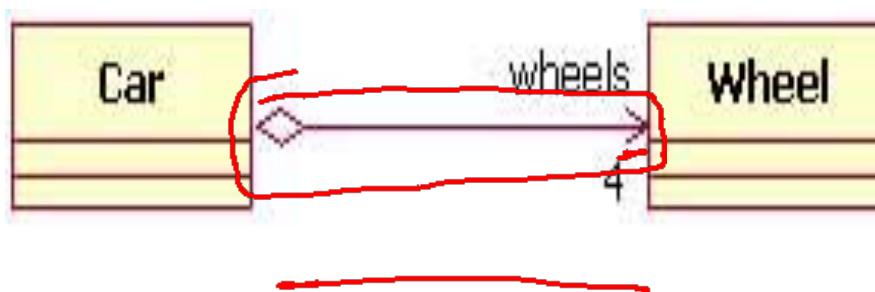


7



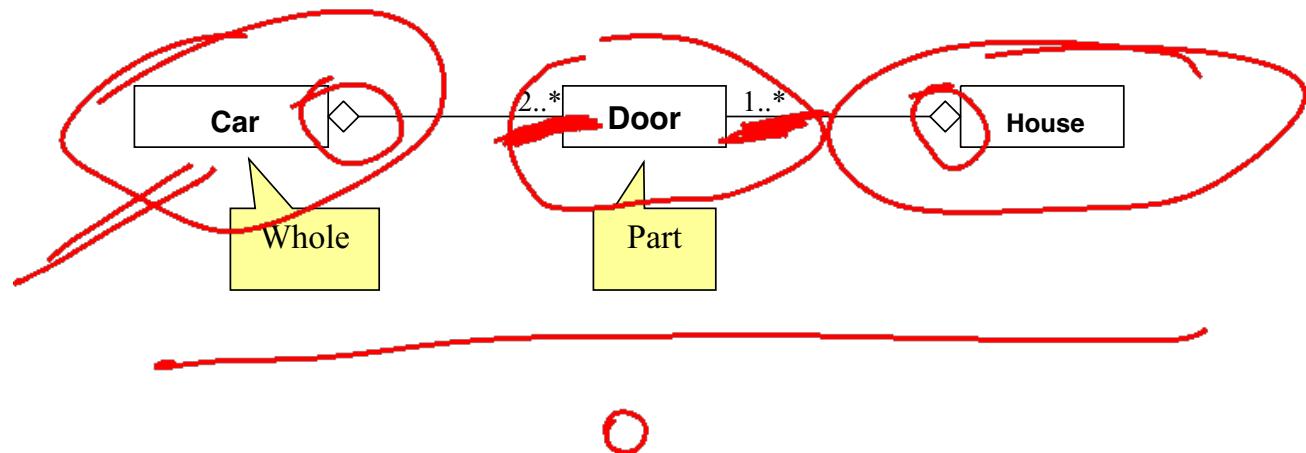
Association : Adornments

- Aggregation
 - Model a whole/part relationship
 - Represents a "has-a" relationship



Aggregation

- A special form of association that models a whole-part relationship between an aggregate (the whole) and its parts.
 - Models a “is a part-part of” relationship.

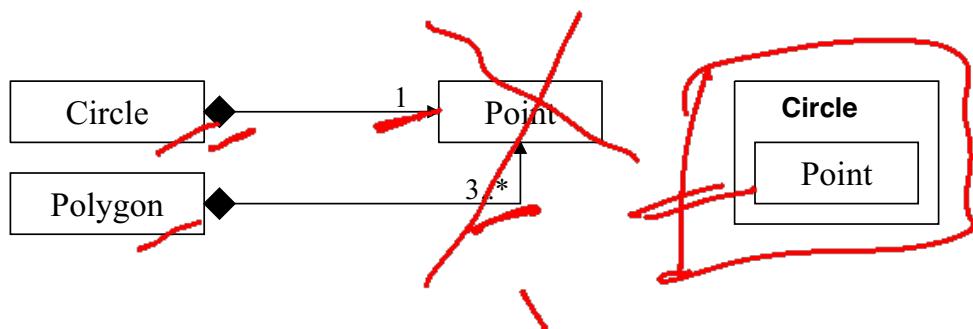


Aggregation (cont.)

- Aggregation tests:
 - Is the phrase “part of” used to describe the relationship?
 - A door is “part of” a car.
 - Are some operations on the whole automatically applied to its parts?
 - Move the car, move the door.
 - Are some attribute values propagated from the whole to all or some of its parts?
 - The car is blue, therefore the door is blue.
 - Is there an intrinsic asymmetry to the relationship where one class is subordinate to the other?
 - A door is part of a car. A car is not part of a door.

Composition

- A strong form of aggregation
 - The whole is the sole owner of its part
 - The part object may belong to only one whole
 - Multiplicity on the whole side must be zero or one.
 - The life time of the part is dependent upon the whole.
 - The composite must manage the creation and destruction of its parts.



Realization

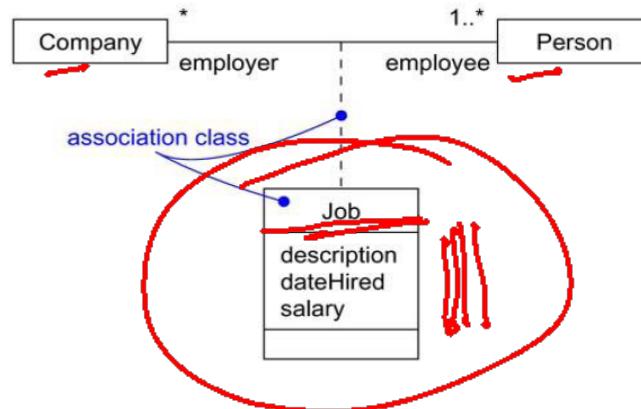
- A realization relationship indicates that one class implements a behavior specified by another class (an interface or protocol).
- An interface can be realized by many classes.
- A class may realize many interfaces.



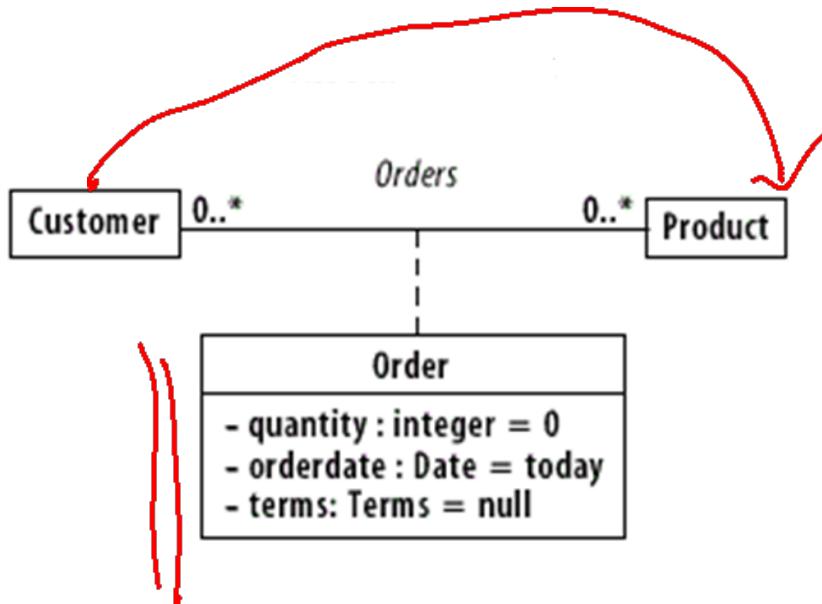
Association

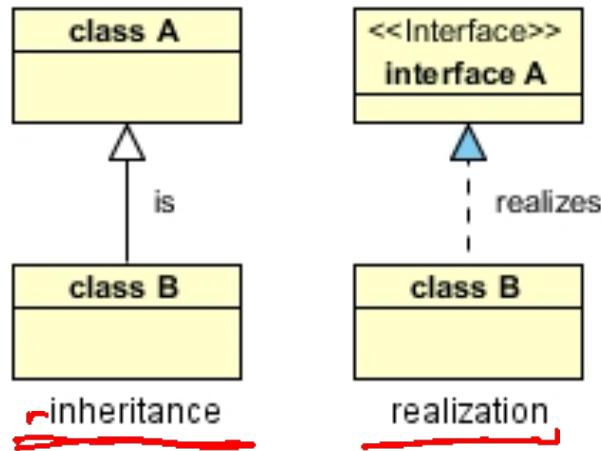
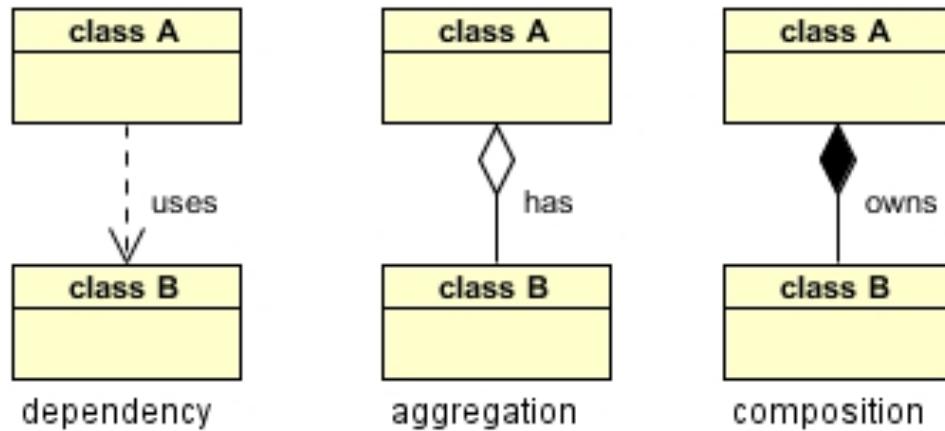
- **Association Classes**

- Association itself may have properties
- Ex: In the relationship between a Company and a Person, there is Job that represent the properties of that relationship
 - Represents exactly one pairing of Person and Company
- Modeling element that has both association and class properties
- Can't attach an association class to more than one association



Association Class

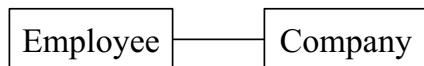




Generaliz

Associations

- An association between two classes indicates that objects at one end of an association “recognize” objects at the other end and may send messages to them.
- Example: “An Employee works for a Company”



Associations (cont.)

- To clarify its meaning, an association may be named.
 - The name is represented as a label placed midway along the association line.
 - Usually a verb or a verb phrase.
- A **role** is an end of an association where it connects to a class.
 - May be named to indicate the role played by the class attached to the end of the association path.
 - Usually a noun or noun phrase
 - Mandatory for reflexive associations

Associations (cont.)

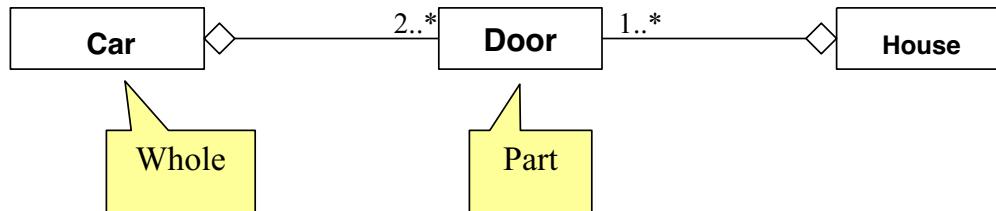
- **Multiplicity**
 - the number of objects that participate in the association.
 - Indicates whether or not an association is mandatory.

Multiplicity Indicators

Exactly one	1
Zero or more (unlimited)	* (0..*)
One or more	1..*
Zero or one (optional association)	0..1
Specified range	2..4
Multiple, disjoint ranges	2, 4..6, 8

Aggregation

- A special form of association that models a whole-part relationship between an aggregate (the whole) and its parts.
 - Models a “is a part-of” relationship.

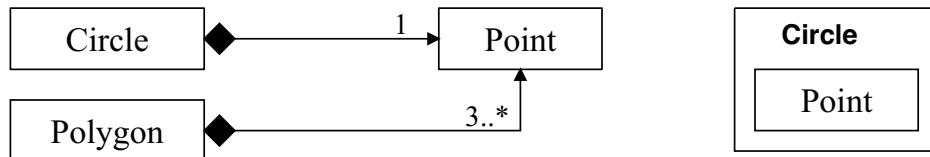


Aggregation (cont.)

- Aggregation tests:
 - Is the phrase “part of” used to describe the relationship?
 - A door is “part of” a car
 - Are some operations on the whole automatically applied to its parts?
 - Move the car, move the door.
 - Are some attribute values propagated from the whole to all or some of its parts?
 - The car is blue, therefore the door is blue.
 - Is there an intrinsic asymmetry to the relationship where one class is subordinate to the other?
 - A door **is** part of a car. A car **is not** part of a door.

Composition

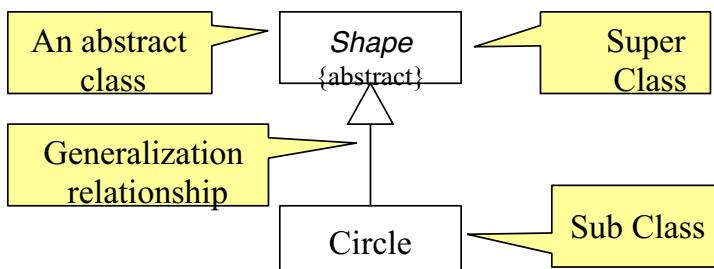
- A strong form of aggregation
 - The whole is the sole owner of its part.
 - The part object may belong to only one whole
 - Multiplicity on the whole side must be zero or one.
 - The life time of the part is dependent upon the whole.
 - The composite must manage the creation and destruction of its parts.



Generalization

- Indicates that objects of the specialized class (subclass) are substitutable for objects of the generalized class (super-class).
 - “is kind of” relationship.

{**abstract**} is a tagged value that indicates that the class is abstract.
The name of an abstract class should be italicized



Guidelines for Identifying Association

Class A and B are associated if

- An object of class A sends a message to an object of class B
- An object of class A creates an object of class B
- An object of class A has an attribute whose values are objects of class B
- An object of class A receives a message with an object of class B as an argument

Guidelines for Identifying a Super-sub Relationship

- Top-down

 Look for noun phrases composed of adjectives in a class name.

- Bottom up

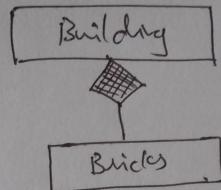
 Look for classes with similar attributes or methods

Identifying the Composition & Aggregation/a-part-of Relationship

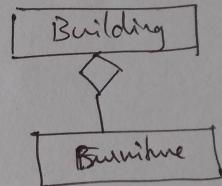
- **Composition** - a physical whole is constructed from physical parts (Assembly)
 - ? Eg1: Building constructed by bricks, stones
 - ? Eg2: ATM with Card Reader, Console, Printer, Key Pad
- **Aggregation** - a physical whole encompasses but is not constructed from physical parts (Container)
 - ? Eg1: Building with Furniture, Appliances
 - ? Eg2: Car with AC and Radio
- **Collection-member** – a ~~conceptual whole~~ encompasses parts that may be ~~physical or conceptual~~
 - ? Eg: Employer, employees

Examples on Relationships

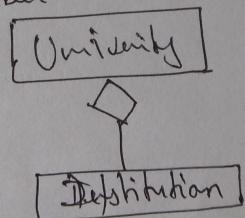
③ Composition (Assembly)



④ Aggregation (Container)



⑤ Collection Member



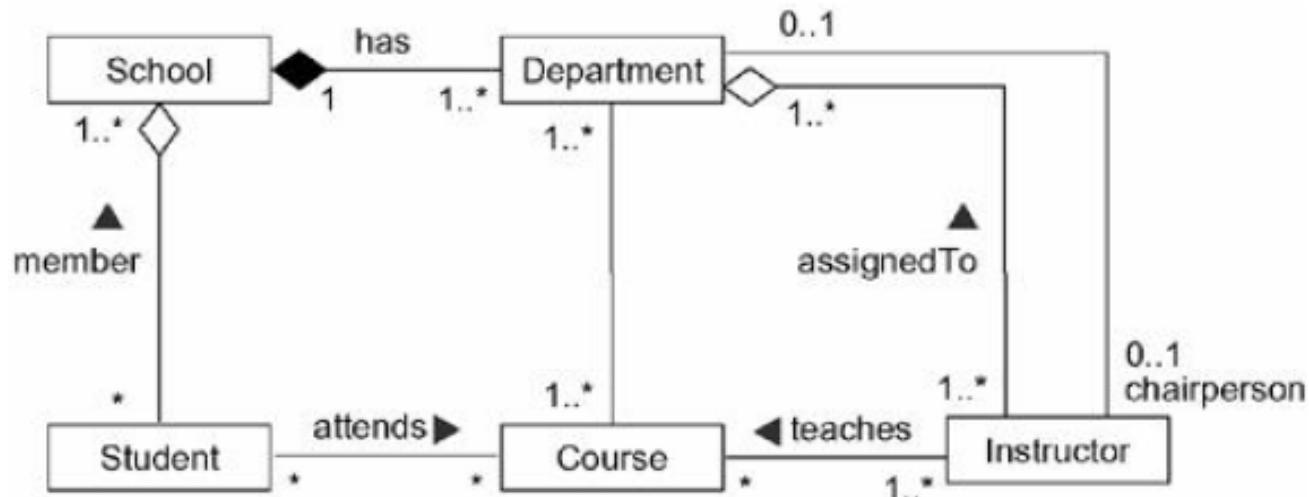
Modeling Structural Relationships

- Given an association between two classes
 - Both rely on each other in some way
 - Can navigate in either direction
 - Specifies a structural path across which objects can interact
- To model structural relationships
 - Navigation from object of one class to object of other – data-driven
 - Specify multiplicity and role names (helps to explain the model)
 - Mark as an aggregation if one class is structurally a whole compared with classes at the other end that look like parts

Modeling Structural Relationships

- Identify the association relationships among classes drawn from an information system for a school
 - School has one or more departments
 - Department offers one or more Courses
 - A particular Course will be offered by only one Department
 - Department has Instructors and Instructors can work for one or more Departments
 - Student can attend any number of Courses in a School
 - Every Course may have any number of Students
 - Instructors can teach zero or more Courses
 - Same Course can be taught by different Instructors
 - Students can be enrolled in more than one School
 - For every Department, there is exactly one Instructor who is the chairperson

Modeling Structural Relationships



Object Relationship in Code

- **Association**

```
public class A {  
    public void doSomething(B b) {}  
}
```

- **Aggregation**

```
public class A {  
    private B b1;  
    public void setB(B b) { b1 = b; } }
```

- **Composition**

```
public class A {  
    private B b1;  
    public A() {  
        b1 = new B();  
    }}
```

Object Relationship in Code

- **Generalization**

```
public class A {  
    ...  
}  
// class A  
  
public class B extends A {  
    ...  
}  
// class B
```

- **Realization**

```
public interface A {  
    ...  
}  
// interface A  
  
public class B implements A {  
    ...  
}  
// class B
```

Aggregation vs Composition

1. **Dependency:** Aggregation implies a relationship where the child **can exist independently** of the parent.

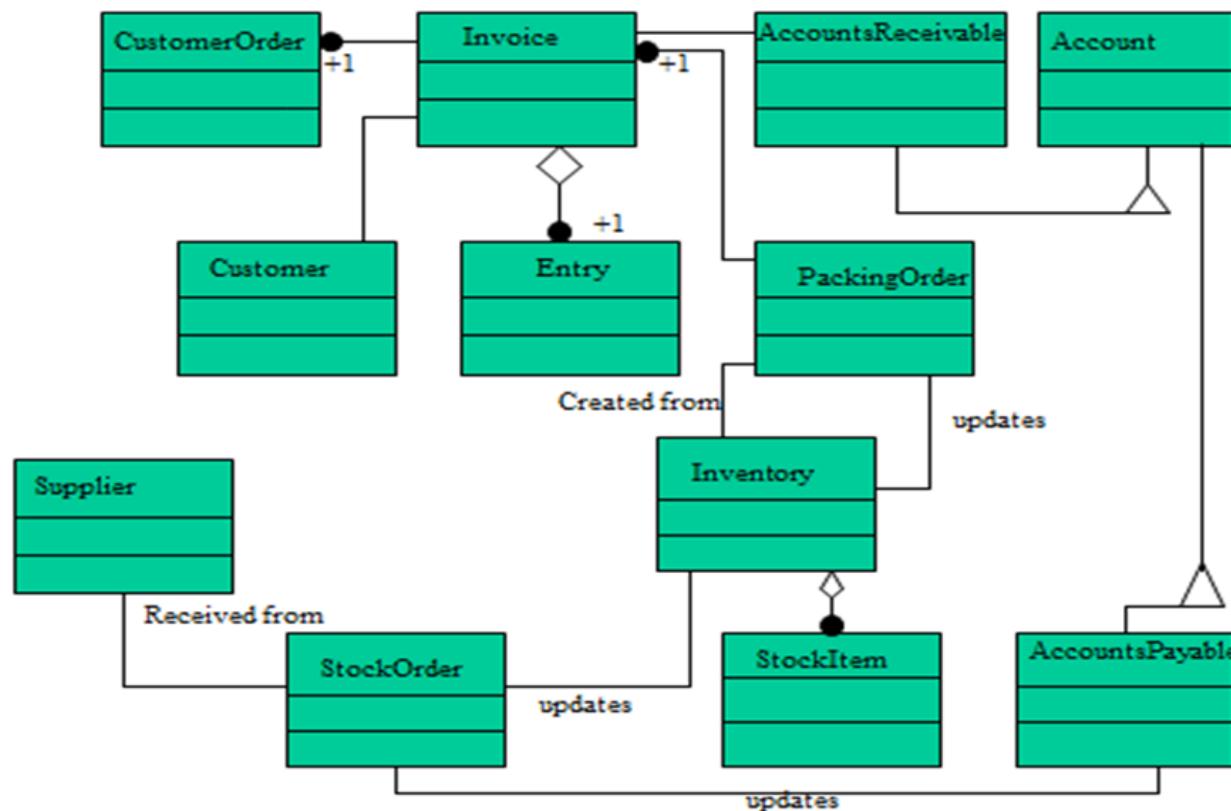
For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child **cannot exist independent** of the parent.

Example: Human and heart, heart don't exist separate to a Human

2.Type of Relationship: Aggregation relation is “**has-a**”, and composition is “**part-of**” relation.

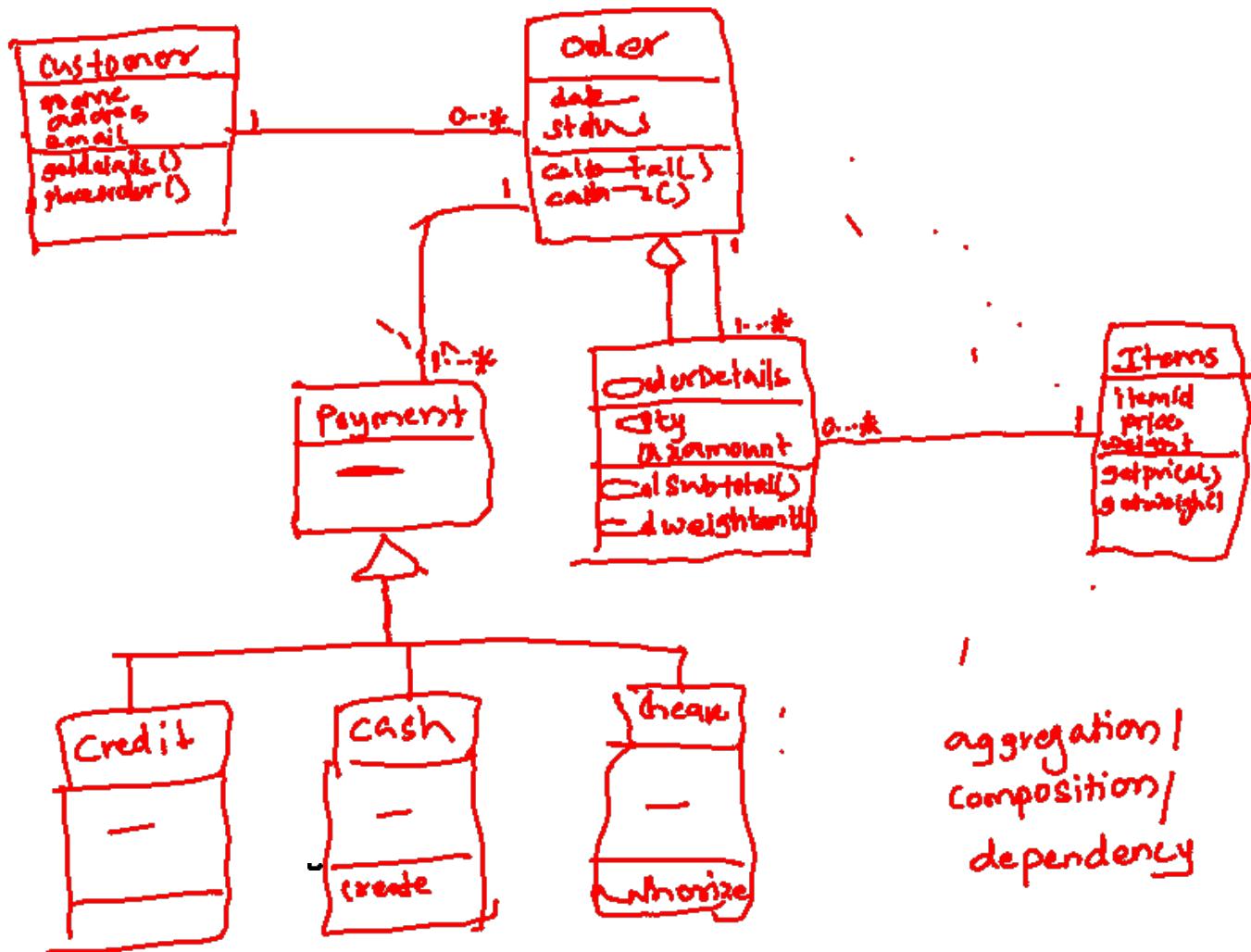
3. Type of association: Composition is a **strong** Association whereas Aggregation is a **weak** Association.

- An Invoice is composed of one or more Entries
- An Invoice is created for exactly one Customer
- An Invoice is created from one or more CustomerOrders
- An Invoice is used to update the AccountsReceivable account
- An Inventory is composed of zero or more StockItems
- A PackingOrder is initiated from one or more Invoices
- A PackingOrder is created from Inventory items
- A PackingOrder is used to update the Inventory
- A StockOrder is received from a Supplier
- A StockOrder is used to update the Inventory
- A StockOrder is used to update the AccountsPayable account

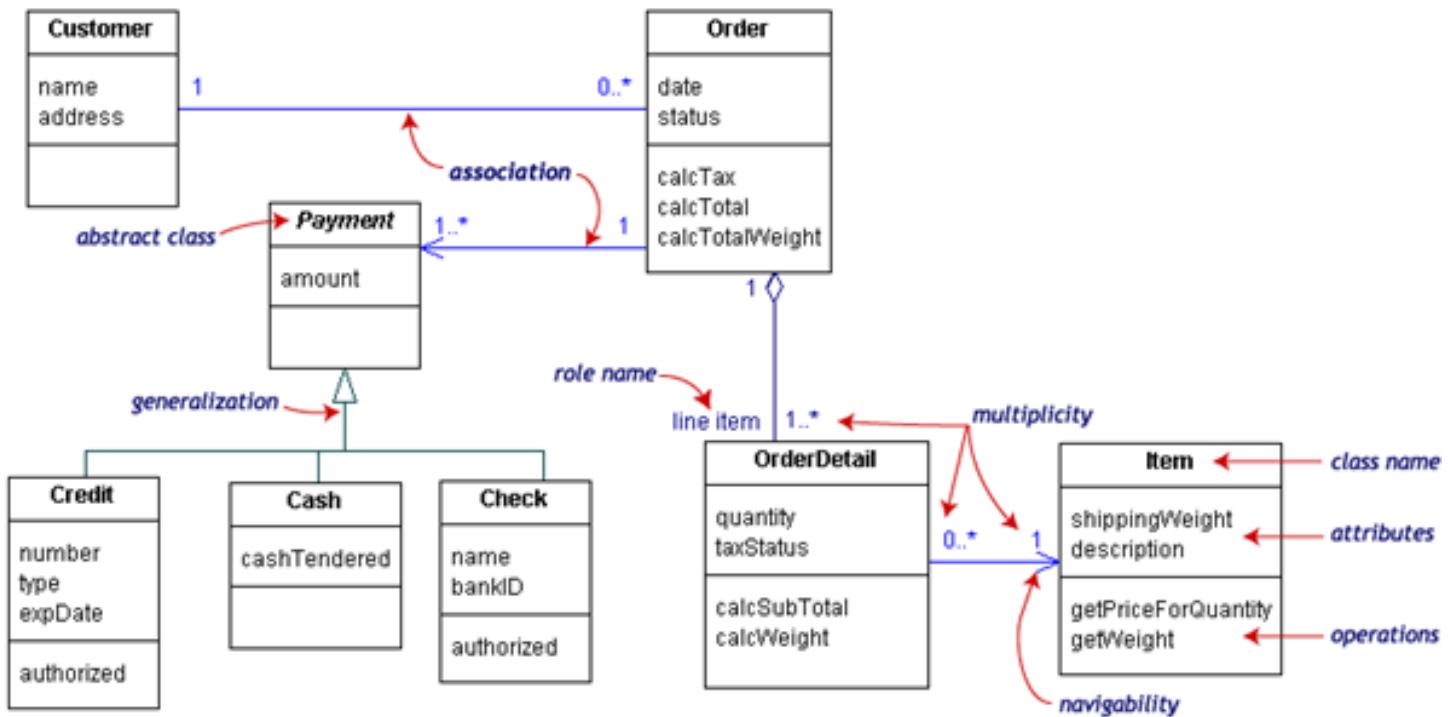


Models a customer order from a retail catalog.

A customer order from a retail catalog. The central class is the Order. Associated with it are the Customer making the purchase and the Payment. A Payment is one of three kinds: Cash, Check, or Credit. The order contains OrderDetails (line items), each with its associated Item.



aggregation /
composition /
dependency



Identifying a list of candidate classes: Example

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 members for 3 weeks. Member of the library can normally borrow up to 6 items at a time, but members of staff may borrow up to 12 items at one time. Only members of staff may borrow journals.

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

Identified Classes for Library System

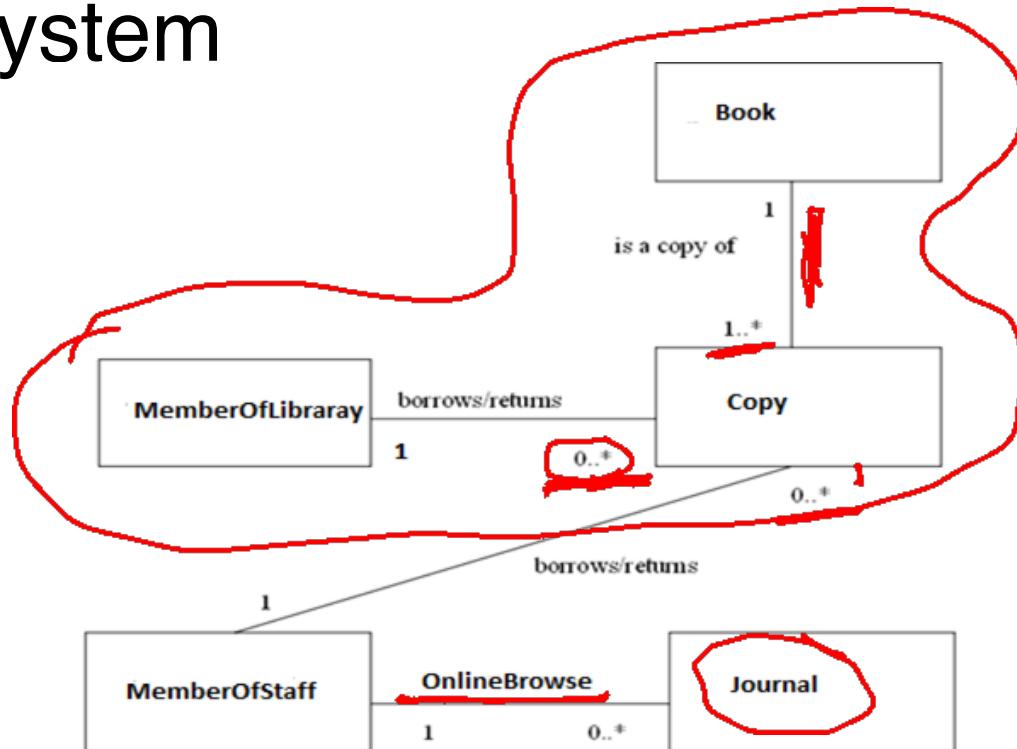
Book

MemberOfLibrary

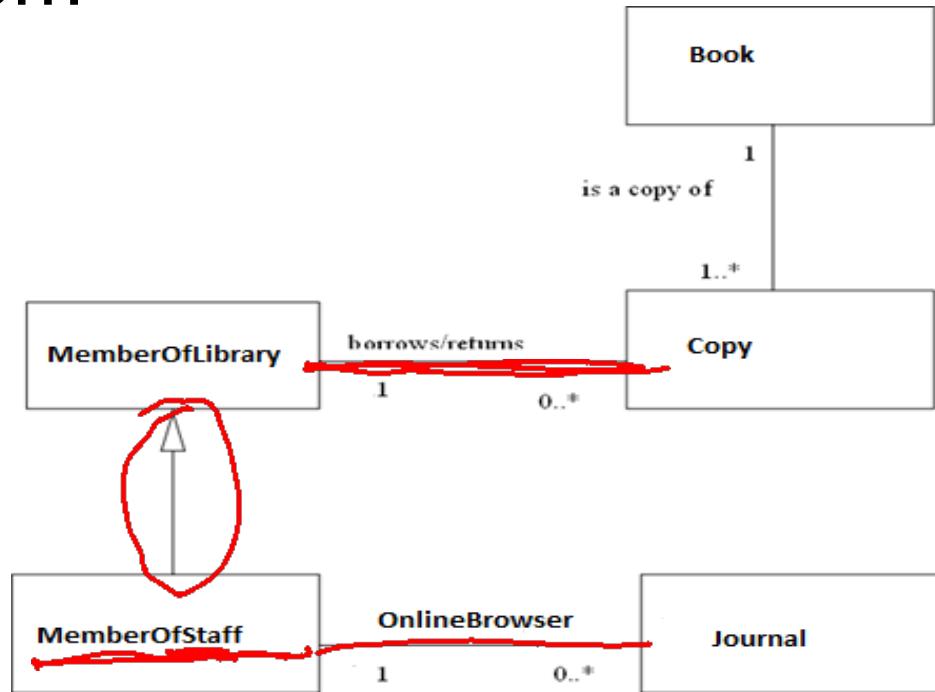
MemberOfStaff

Journal

Initial Class model for Library System

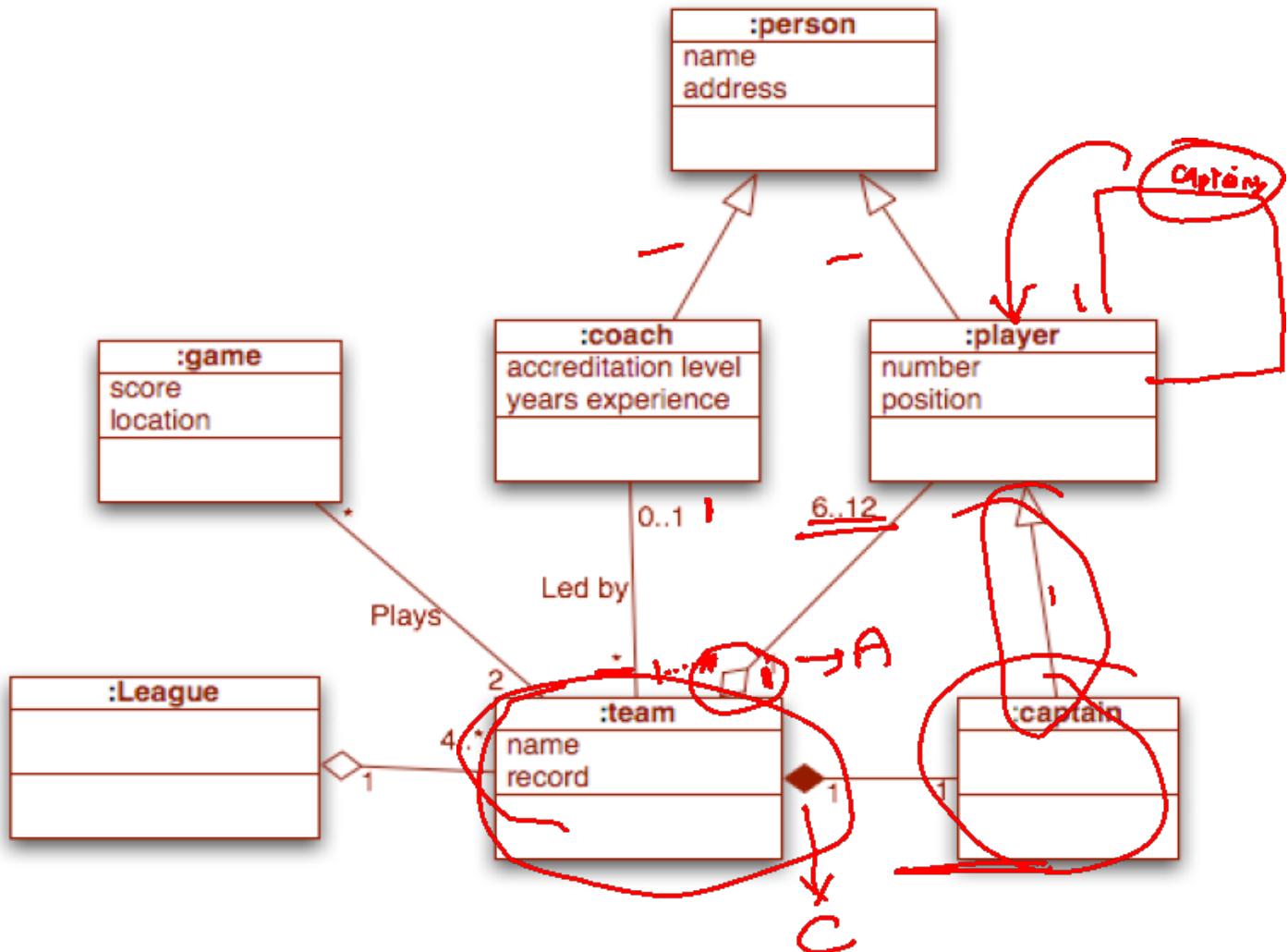


Revised class model for Library System



Class Diagram Example

A hockey league is made up of at least four hockey teams. Each hockey team is composed of six to twelve players, and one player captains the team. A team has a name and a record. Players have a number and a position. Hockey teams play games against each other. Each game has a score and a location. Teams are sometimes lead by a coach. A coach has a level of accreditation and a number of years of experience, and can coach multiple teams. Coaches and players are people, and people have names and addresses.



Design concepts

Functional Independence

- ❑ Cohesion – is an indication of the relative functional strength of a module
- ❑ Coupling – is an indication of the relative interdependence among modules

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

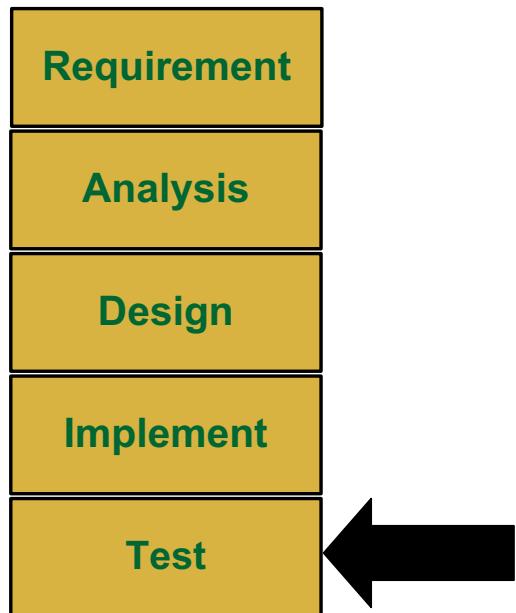
Uses the control structure part of component-level design to derive the test cases

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?



Evaluating the System

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.

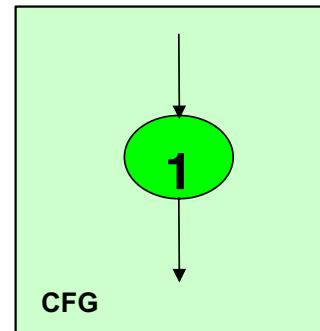
Path:

- A collection of *Nodes* linked with *Directed Edges*.

Simple Examples

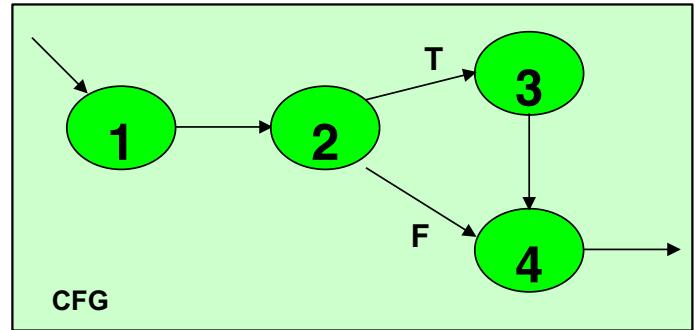
```
Statement1;  
Statement2;  
Statement3;  
Statement4;
```

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



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

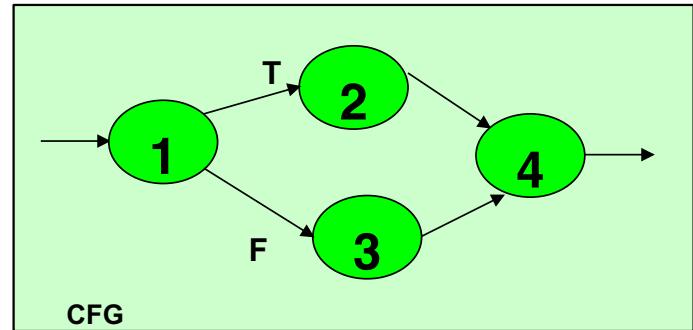
1
2
3
4



More Examples

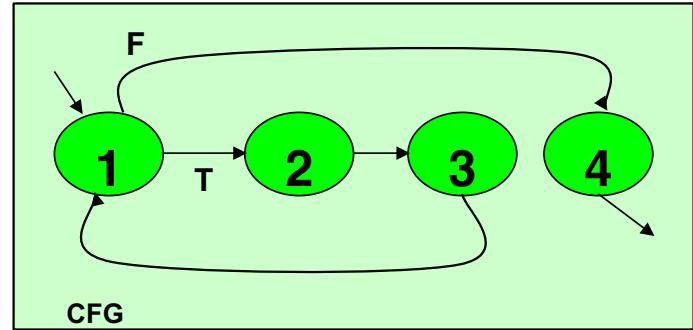
```
if X > 0 then  
    Statement1;  
else  
    Statement2;
```

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:

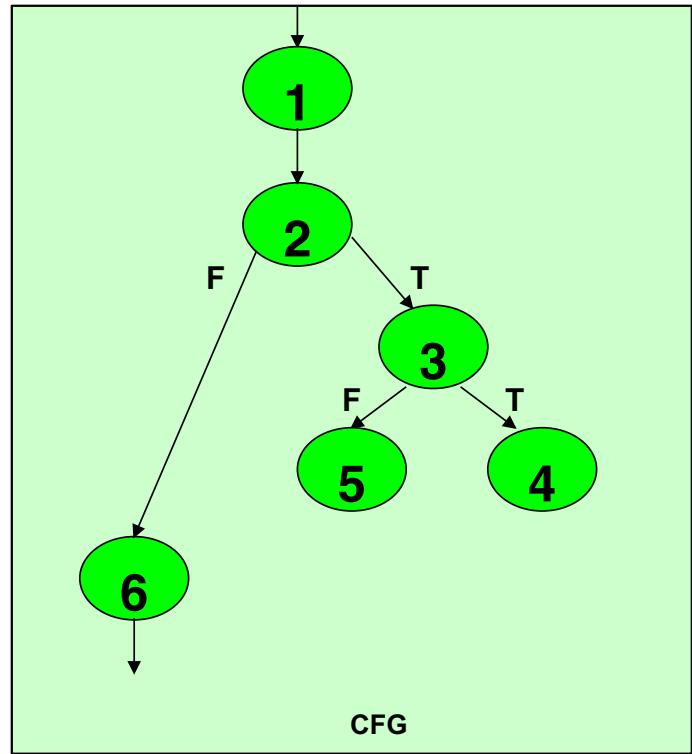
Node 4 in the `if-then-else` example from previous slide.

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?

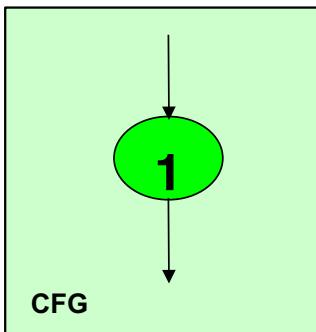
Translating the program into a CFG, an equivalent question is:

- 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:

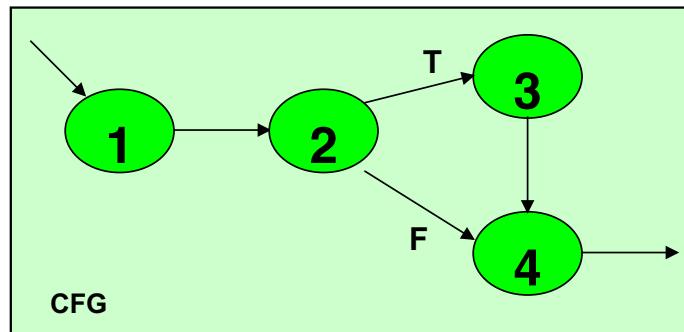
- Given a CFG, what is the set of paths that can cover all arcs and nodes?

Example



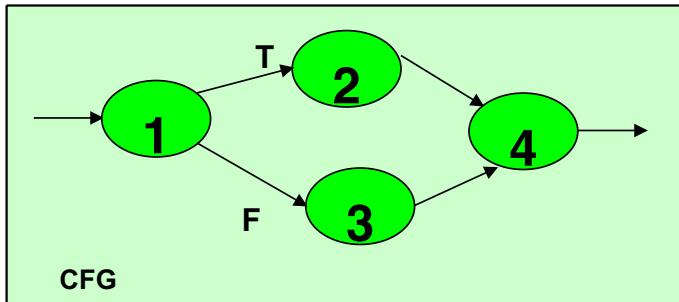
Only **one** path is needed:

- [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.

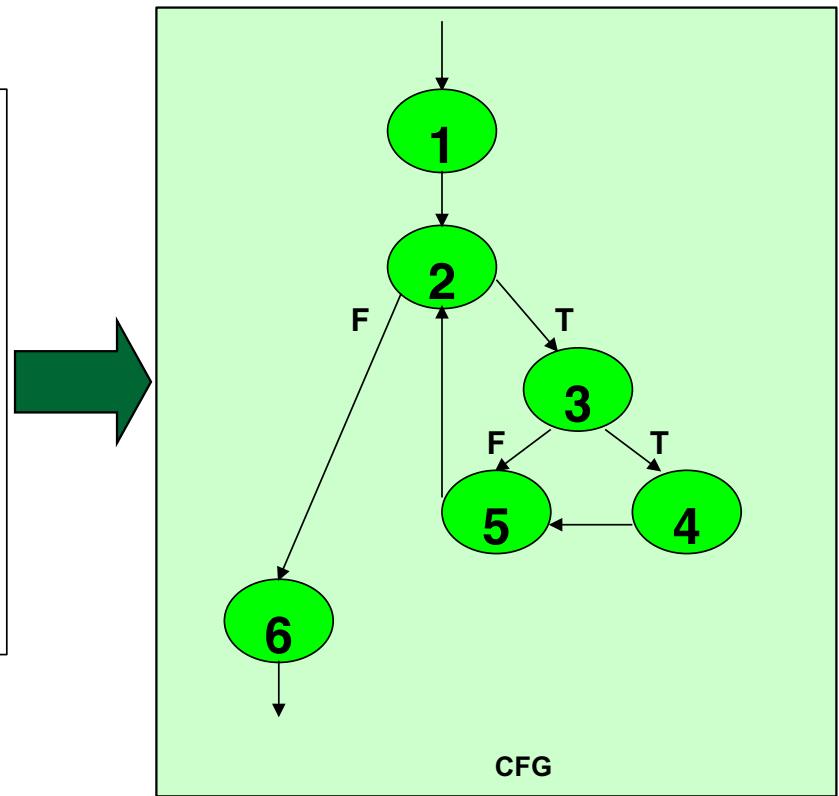
Steps:

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**.

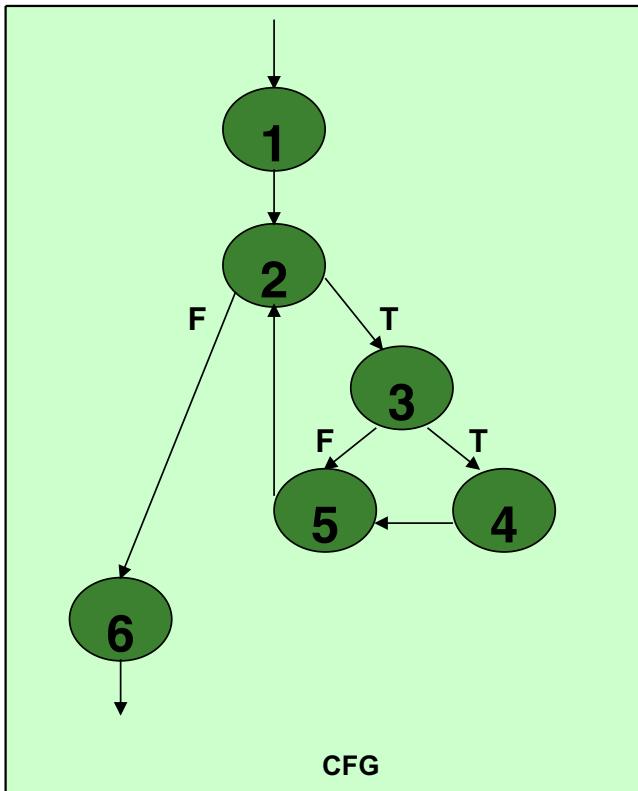
Path Based Testing: Step 1

```
min = A[0];
I = 1;

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



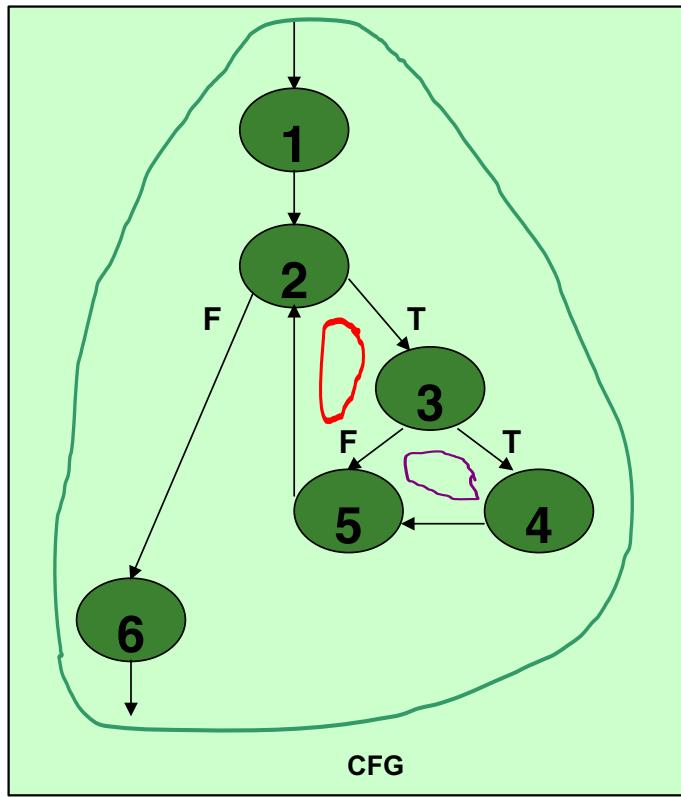
Path Base Testing: Step 2



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



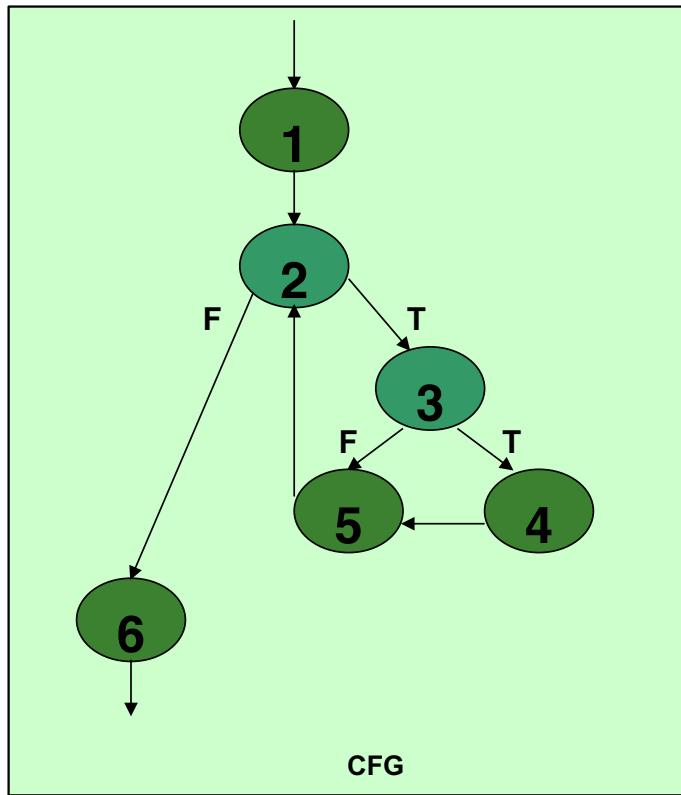
Region: Enclosed area in the CFG.

- 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:

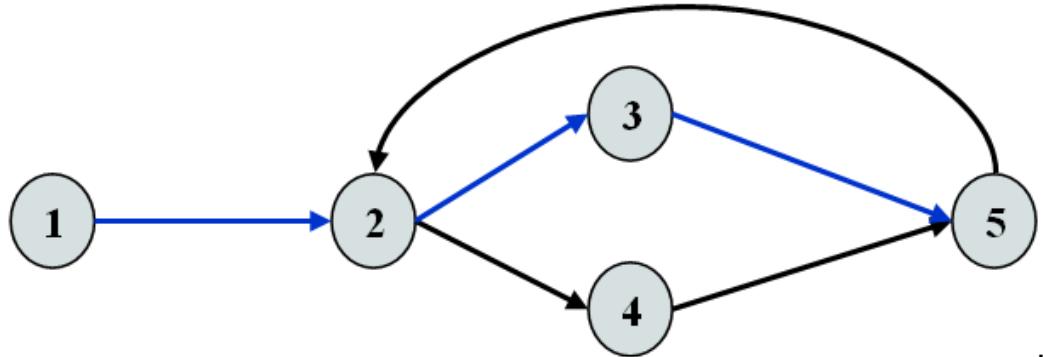
- An **executable** or **realizable path** through the graph from the start node to the end node that has not been traversed before.
- Must** move along **at least one arc** that has not been yet traversed (an unvisited arc).
- The objective is to cover all statements in a program by independent paths.

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



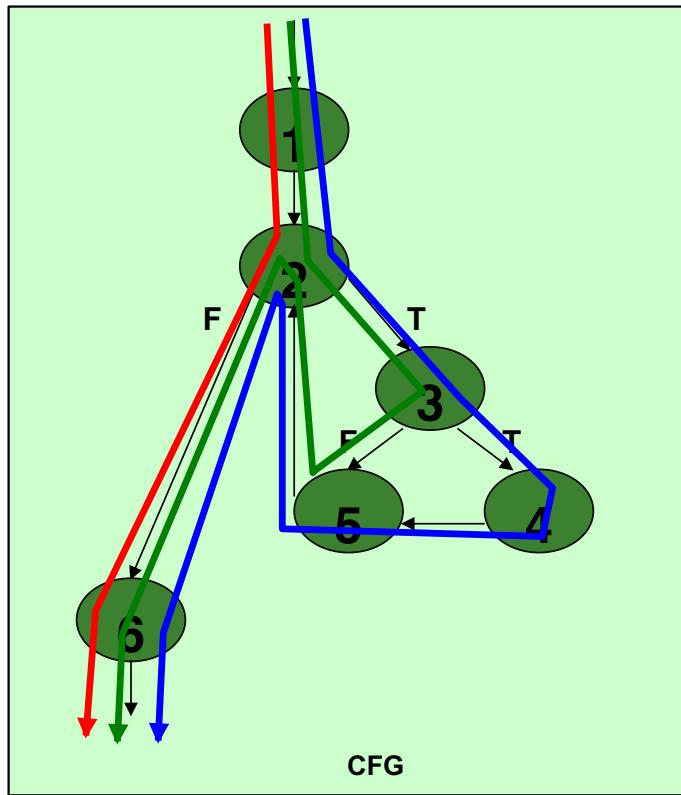
1-2-3-5 can be the first independent path; 1-2-4-5 is another; 1-2-3-5-2-4-5 is one more.

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.

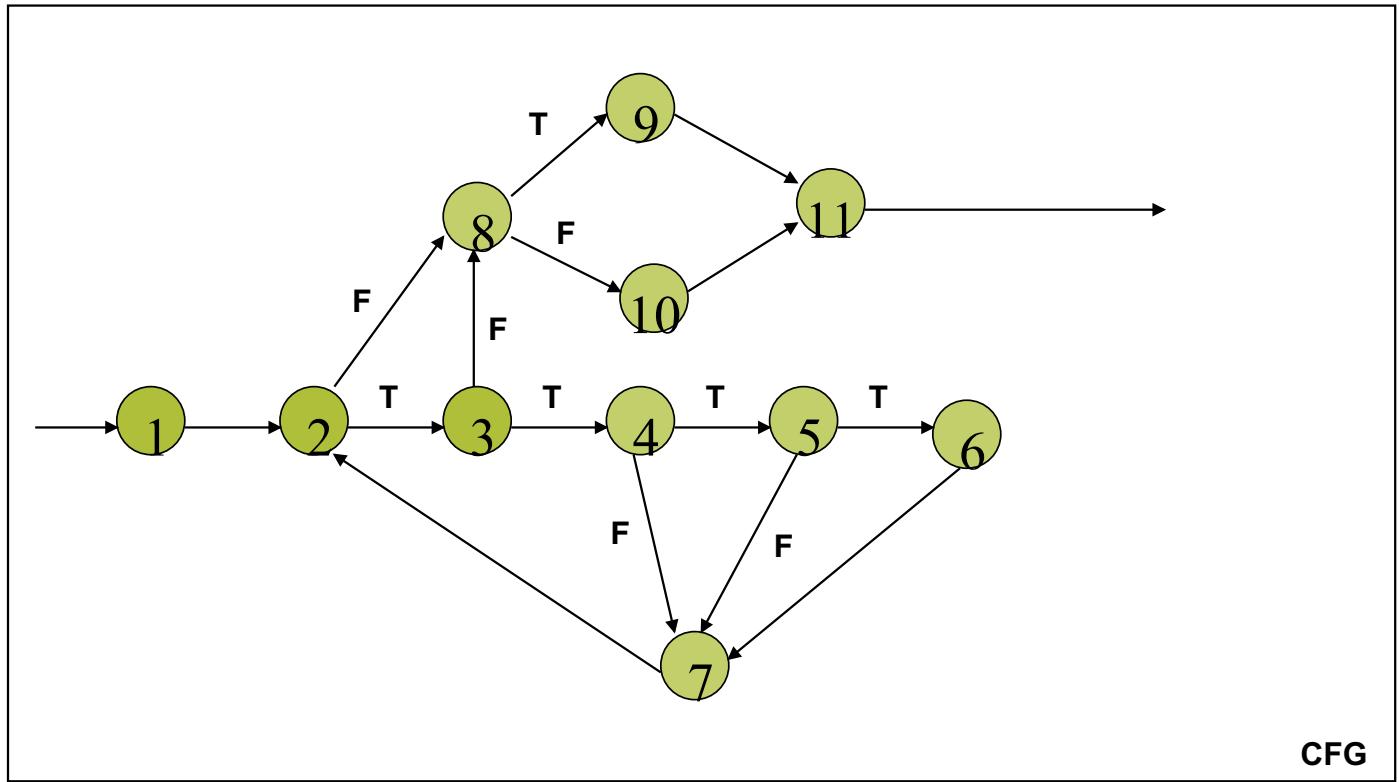
Another Example

```
int average (int[ ] value, int min, int max, int N) {  
    int i, totalValid, sum, mean;  
    i = totalValid = sum = 0; ①  
    while ( i < N && value[i] != -999 ) { ②  
        if (value[i] >= min && value[i] <= max){ ③  
            totalValid += 1; sum += value[i]; ④ ⑤ ⑥  
        } ⑦  
        i += 1; ⑧  
    } ⑨  
    if (totalValid > 0) ⑩  
        mean = sum / totalValid; ⑪  
    else  
        mean = -999; ⑫  
    return mean;  
}
```

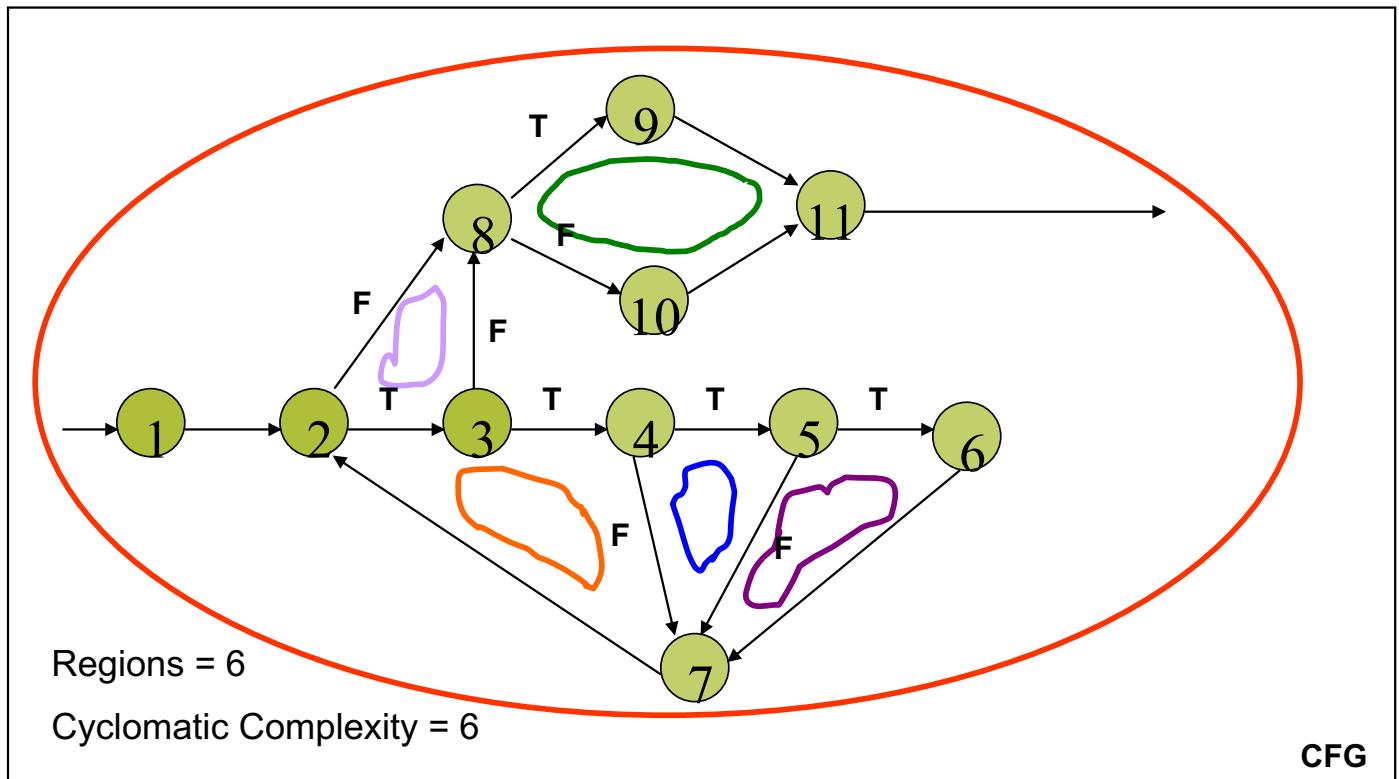
Step 1: Draw CFG

```
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] > 4 min && value[i] < 5 max){  
            totalValid += 1; sum += value[i]; } 6  
            i += 1; } 7  
        if (totalValid > 0) { 8  
            mean = sum / totalValid; } 9  
        else  
            mean = -999; } 11  
        return mean;
```

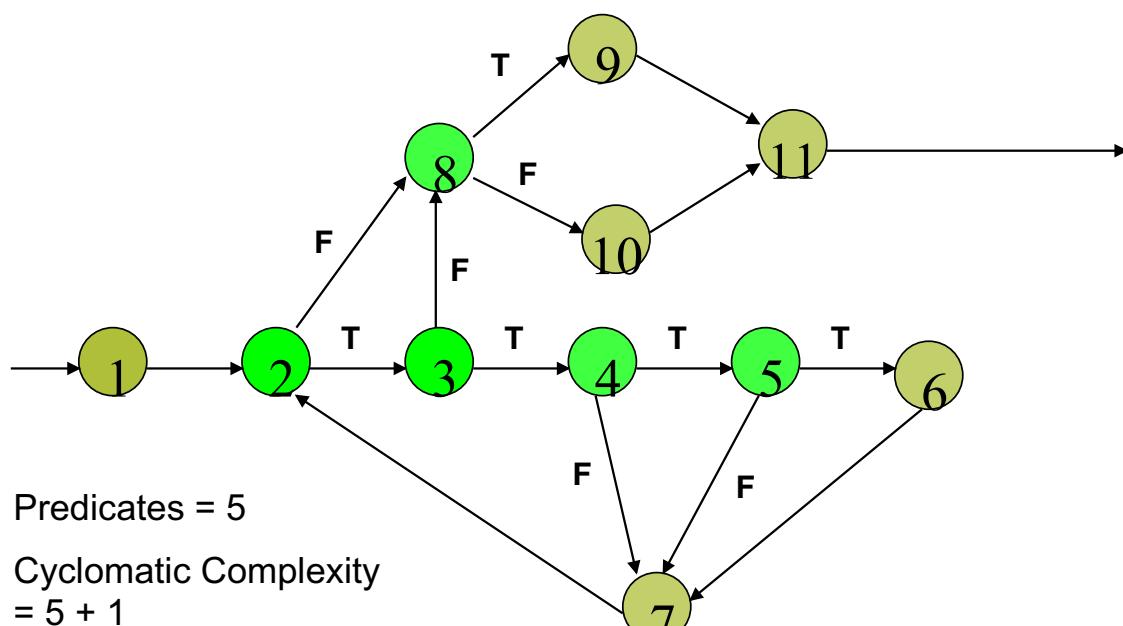
Step 1: Draw CFG



Step 2: Find Cyclomatic Complexity



Step 2: Find Cyclomatic Complexity



CFG

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 &&      2
      value[i] != 2999) {
      .....
}
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

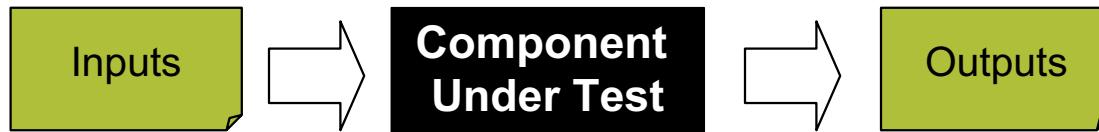
Test Engineers have no access to the source code or documentation of internal working.

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.

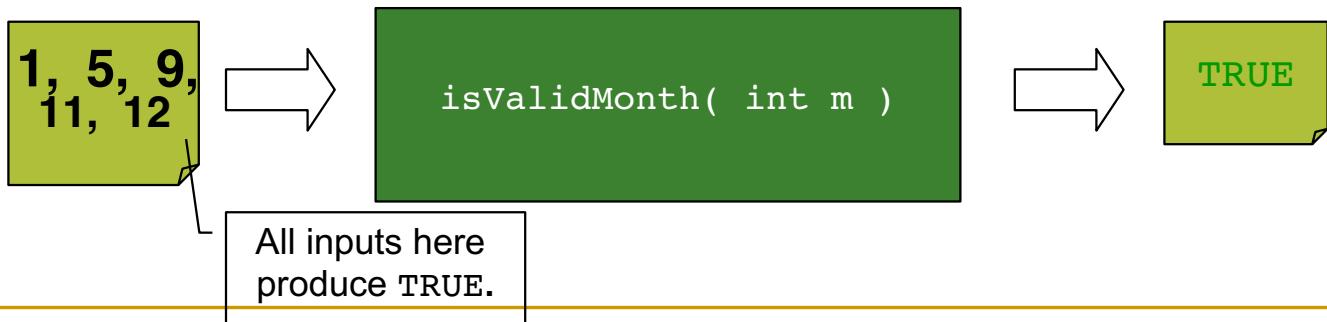
E.g., if value 240 fails the testing, it is *likely* that 241 is likely to fail the test too.

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.



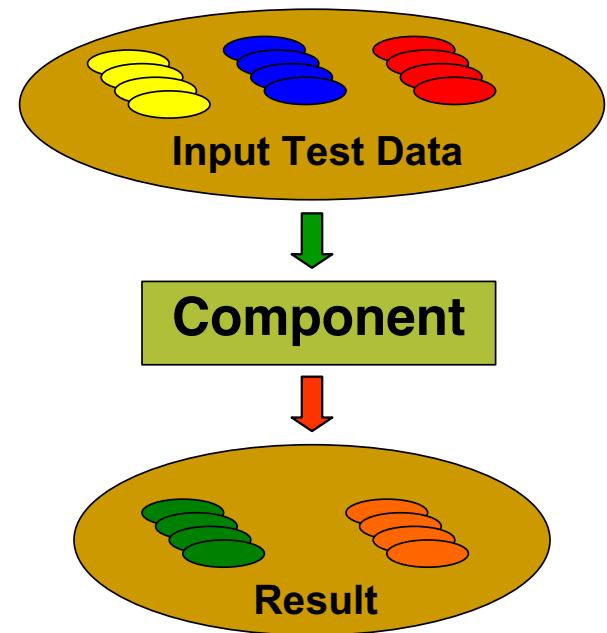
Equivalence Partition: Definition

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

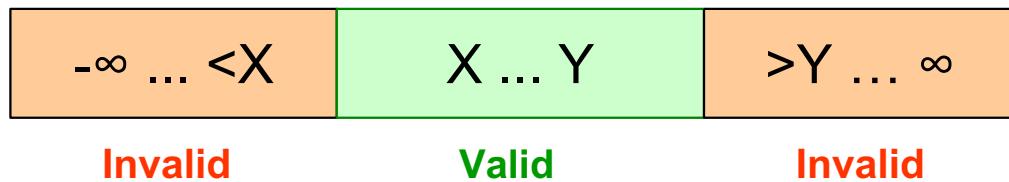
Pick one value from each partition as test case:

- E.g., { -12, 5, 15 }
- Reduce the number of test cases significantly.

Common Partitions

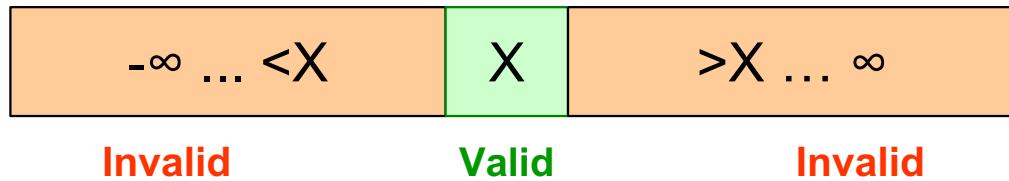
If the component specifies an input range, [X ... Y].

- Three partitions:



If the component specifies a single value, [X].

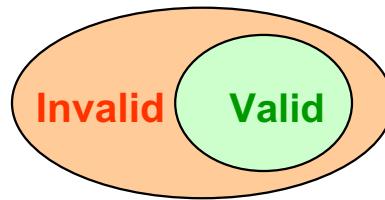
- Three partitions:



Common Partitions

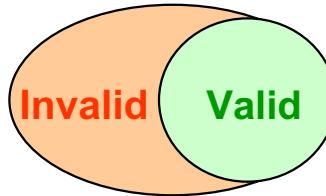
If the component specifies member(s) of a set:

- 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

Area Code (optional)	Prefix	Suffix
----------------------------	--------	--------

- 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 ... 910]	[200 ... 999]	[0000 ... 9999]
3	True	[911]	[1000 ... ∞]	[10000 ... ∞]
4	True	[912 ... 999]	[200 ... 999]	[0000 ... 9999]
5	True	[1000 ... ∞]	[1000 ... ∞]	[10000 ... ∞]

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 ... 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 ... 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

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

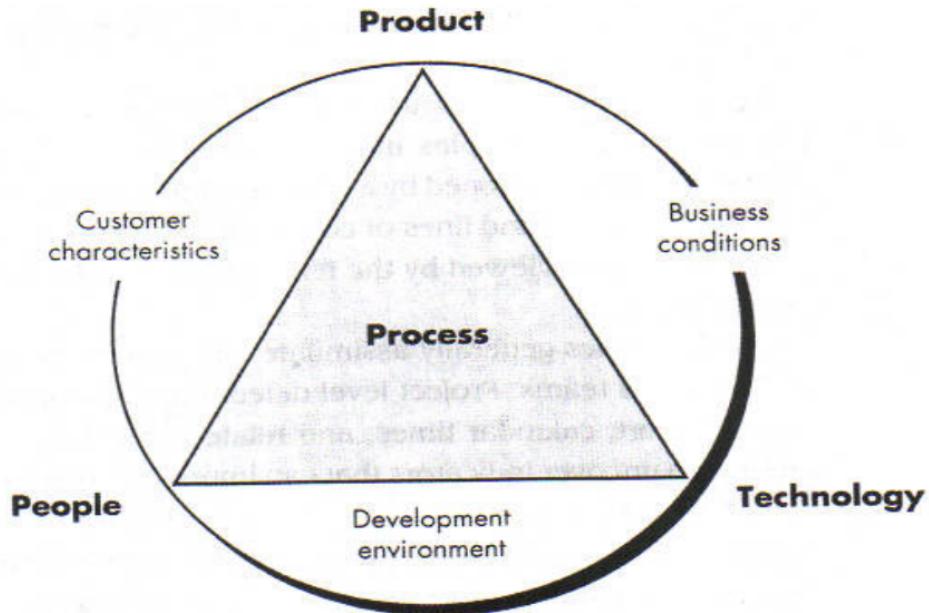
- Evaluate the project team's ability to control quality of software engineering work products.
- Estimate effort and time duration
- Every project should measure input, output and result

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>	count	<u>weighting factor</u>			=	
		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						<input type="text"/>
function points	<hr/>					<input type="text"/>

Adjusted Function Points Count Complexity: 14 Factors F_i

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 x 15,000
	= 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

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

www.qsm.com/?q=resources/function-point-languages-table/index.html

Metrics for Software Quality

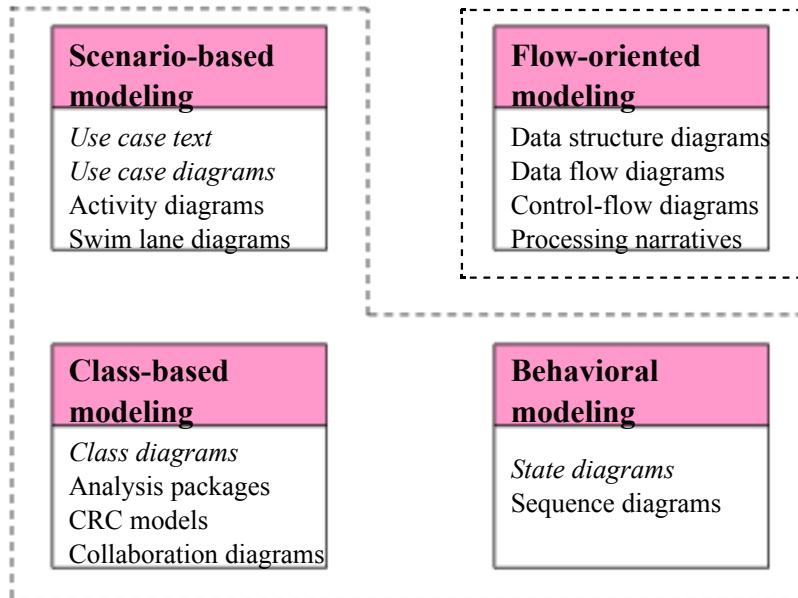
- Correctness
 - This is the number of defects per KLOC, where a defect is a verified lack of conformance to requirements
 - Defects are those problems reported by a program user after the program is released for general use
- Maintainability
 - This describes the ease with which a program can be corrected 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

Elements of the Analysis Model

Object-oriented Analysis



Procedural/Structured Analysis

Sequence Diagrams

Interaction Diagrams

- A series of diagrams describing the *dynamic behavior* of an object-oriented system.
- A set of messages exchanged among a set of objects within a context to accomplish a purpose.
- Often used to model the way a use case is realized through a sequence of messages between objects.

Interaction Diagrams (Cont.)

- The purpose of Interaction diagrams is to:
- Model interactions between objects
- Assist in understanding how a system (a use case) actually works
- Verify that a use case description can be supported by the existing classes
- Identify responsibilities/operations and assign them to classes

Interaction Diagrams (Cont.)

- UML
 - Collaboration Diagrams
 - Emphasizes structural relations between objects
 - Sequence Diagram

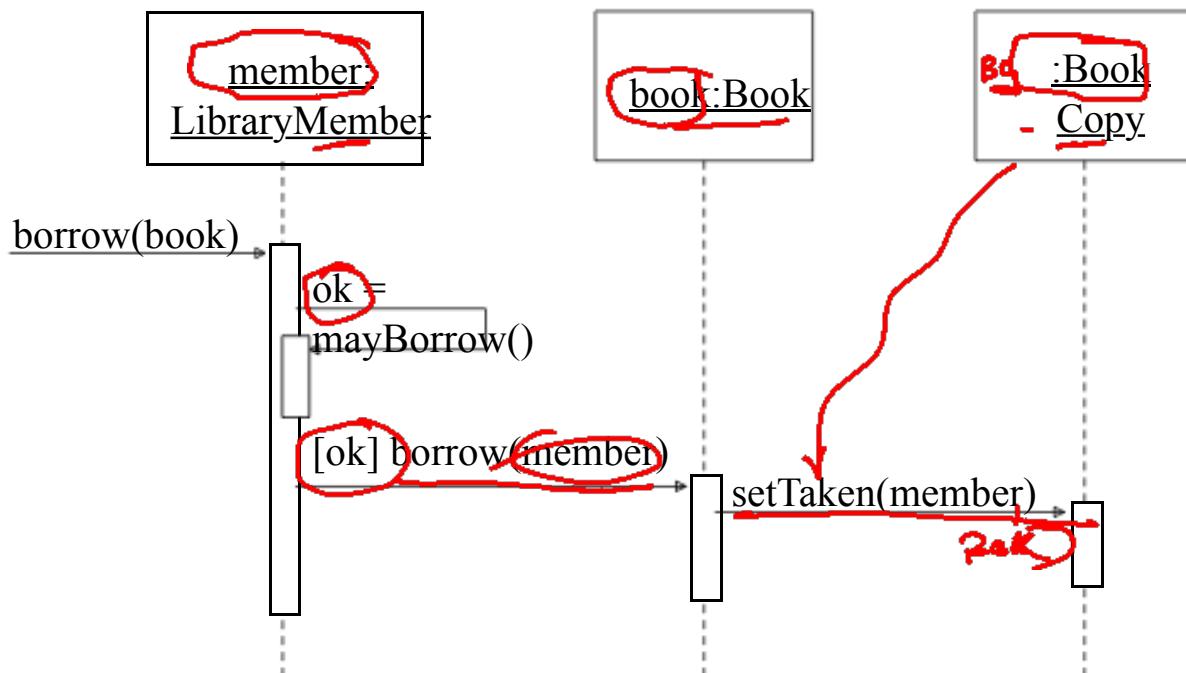
Importance of Sequence Diagrams

- Depict object interactions in a given scenario identified for a given Use Case
- Specify the messages passed between objects using horizontal arrows including messages to/from external actors
- Time increases from Top to bottom

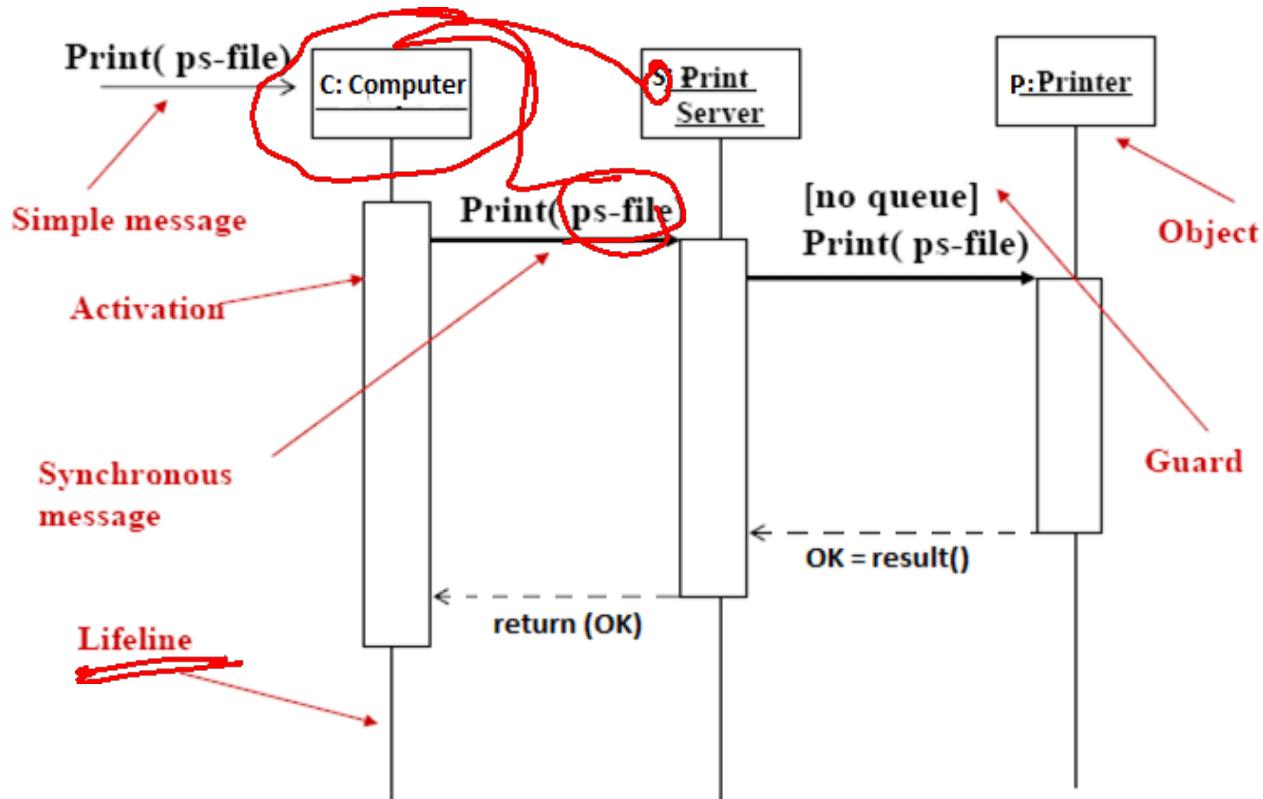
Sequence Diagrams

- Sequence diagrams illustrate how objects interact with each other.
- They focus on message sequences, that is, how messages are sent and received between a number of objects.
- The Branches, Conditions, and Loops may be included.
- Emphasizes time ordering of messages.
- Can model simple sequential flow, branching, iteration, recursion and concurrency.

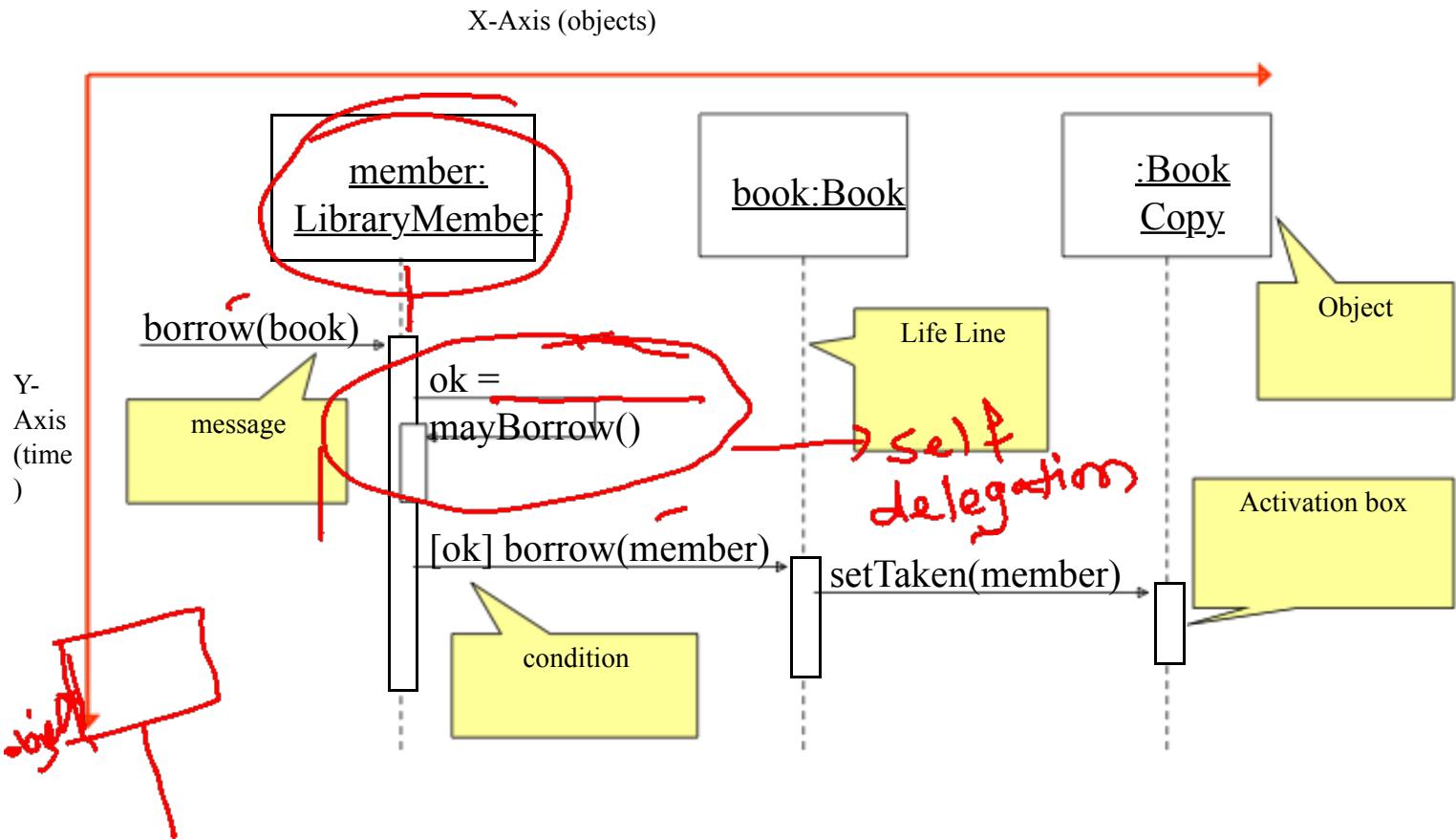
A Sequence Diagram



Sequence Diagram - Syntax



A Sequence Diagram



Object

- Object naming:
- syntax: *[instanceName][:className]*
- Name classes consistently with your class diagram (same classes).
- Include instance names when objects are referred to in messages or when several objects of the same type exist in the diagram.
- The *Life-Line* represents the object's life during the interaction



Messages

- An interaction between two objects is performed as a message sent from one object to another (simple operation call, Signaling, RPC)
- If object obj1 sends a message to another object obj2 some link must exist between those two objects (dependency, same objects)

Message types

- Synchronous messages
- Asynchronous messages

Messages (Cont.)

- A message is represented by an arrow between the lifelines of two objects.
- Self calls are also allowed
- The time required by the receiver object to process the message is denoted by an *activation box*.
- A message is labeled at minimum with the message name.
- Arguments and control information (conditions, iteration) may be included.

Return Values



- Optionally indicated using a dashed arrow with a label indicating the return value.
- Don't model a return value when it is obvious what is being returned, e.g. `getTotal()`
- Model a return value only when you need to refer to it elsewhere, e.g. as a parameter passed in another message.
- Prefer modeling return values as part of a method invocation, e.g. `ok = isValid()`

Asynchronous messages



Asynchronous calls, which are associated with operations, typically have only a send message, but can also have a reply message.

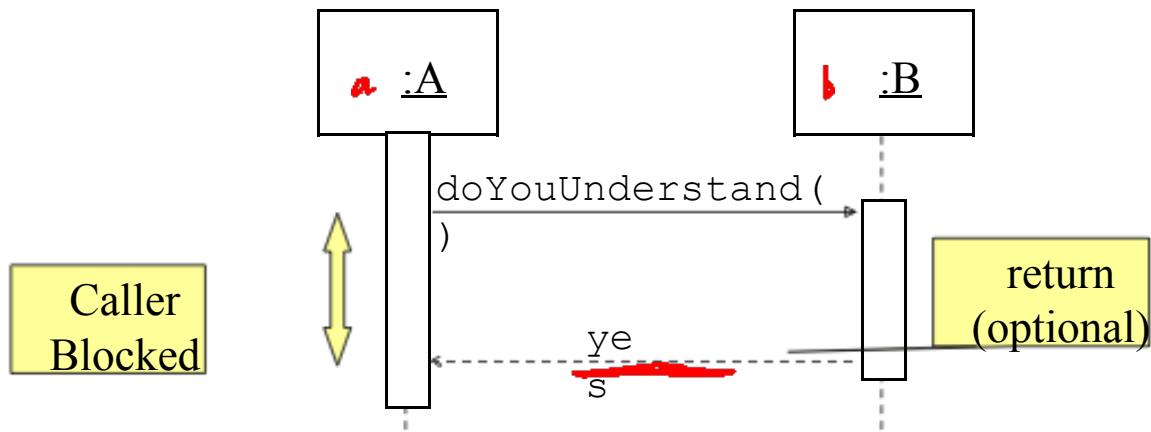
In contrast to a synchronous message, the source lifeline is not blocked from receiving or sending other messages.

You can also move the send and receive points individually to delay the time between the send and receive events.

You might choose to do this if a response is not time sensitive or order sensitive.

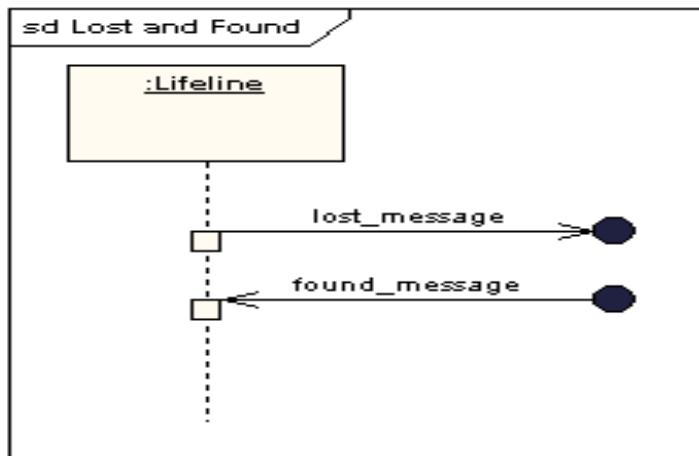
Synchronous Messages

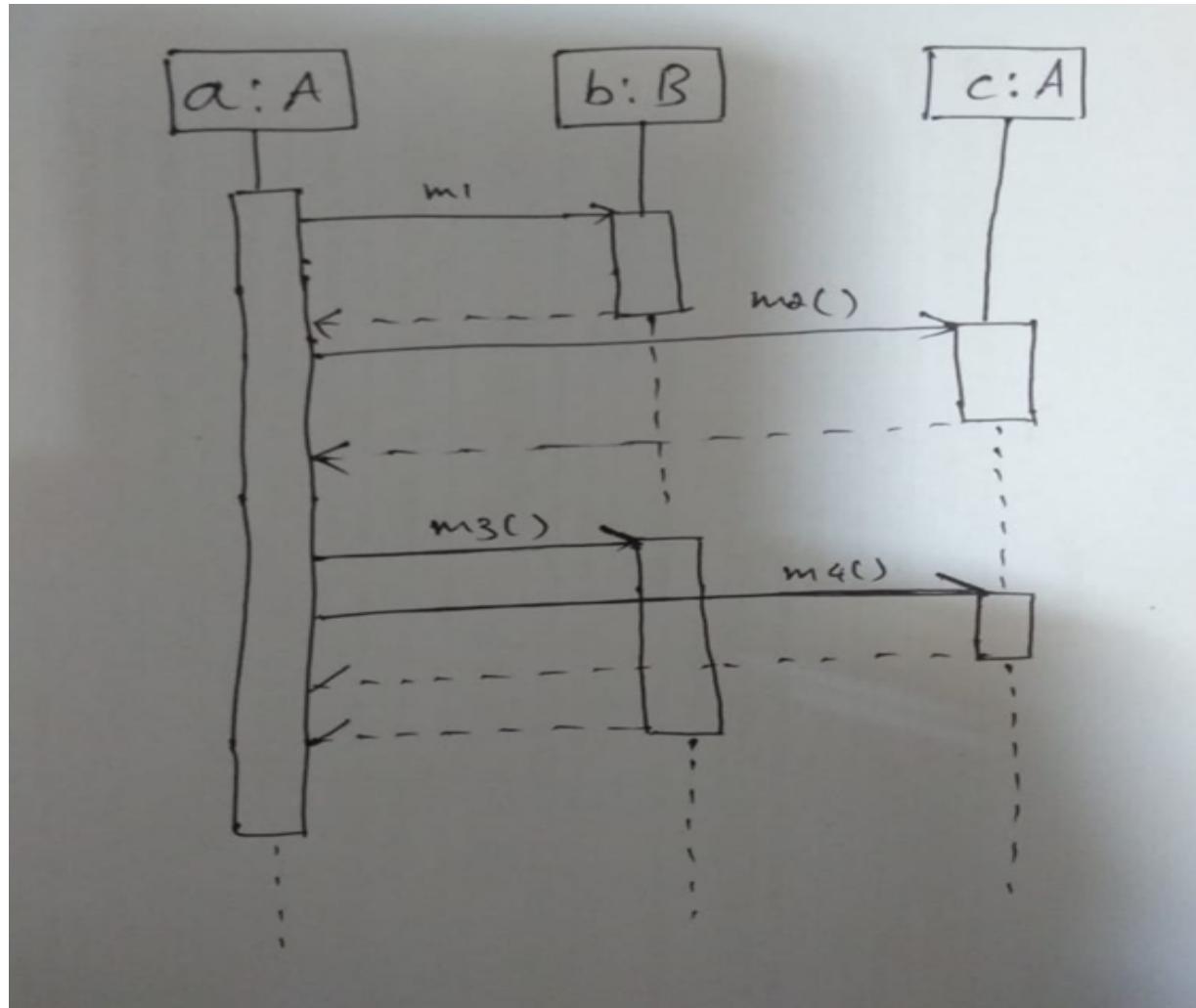
- Nested flow of control, typically implemented as an operation call.
- The routine that handles the message is completed before the caller resumes execution.



Lost and Found Messages

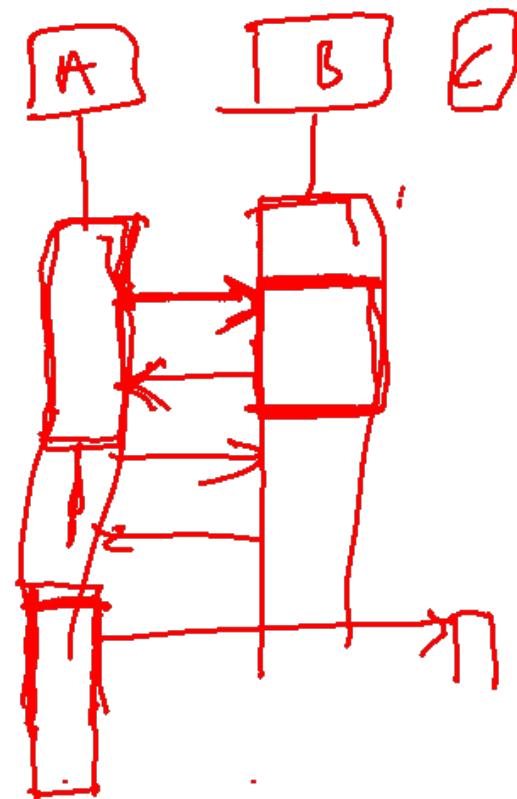
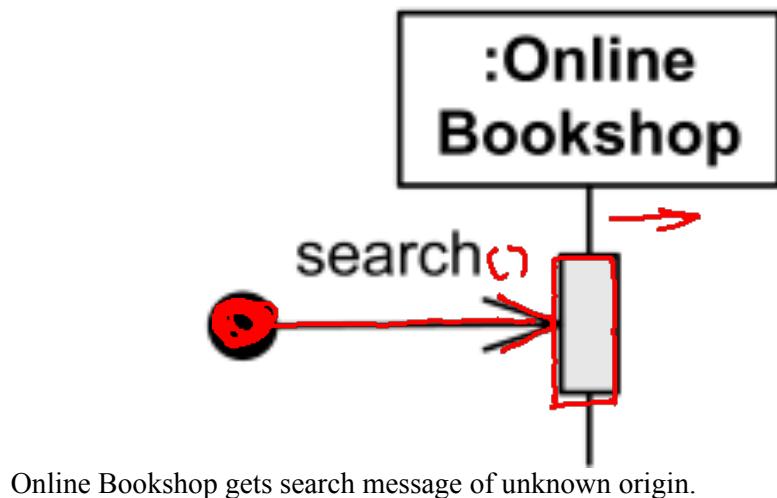
- Lost messages are those that are either sent but do not arrive at the intended recipient, or which go to a recipient not shown on the current diagram.
- Found messages are those that arrive from an unknown sender, or from a sender not shown on the current diagram. They are denoted going to or coming from an endpoint element.





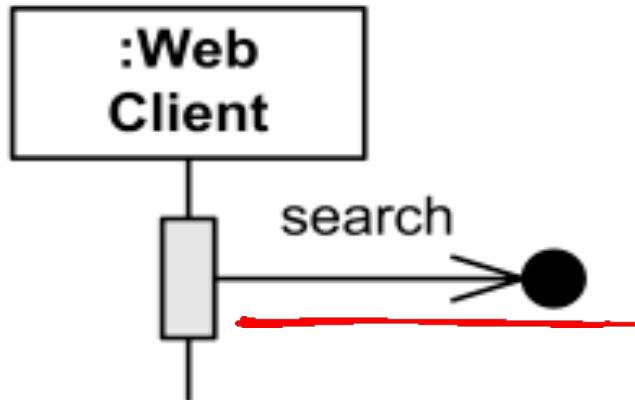
Found & Lost Message

- **Found Message** is a message where the receiving event is known, but there is no (known) sending event



Found & Lost Message

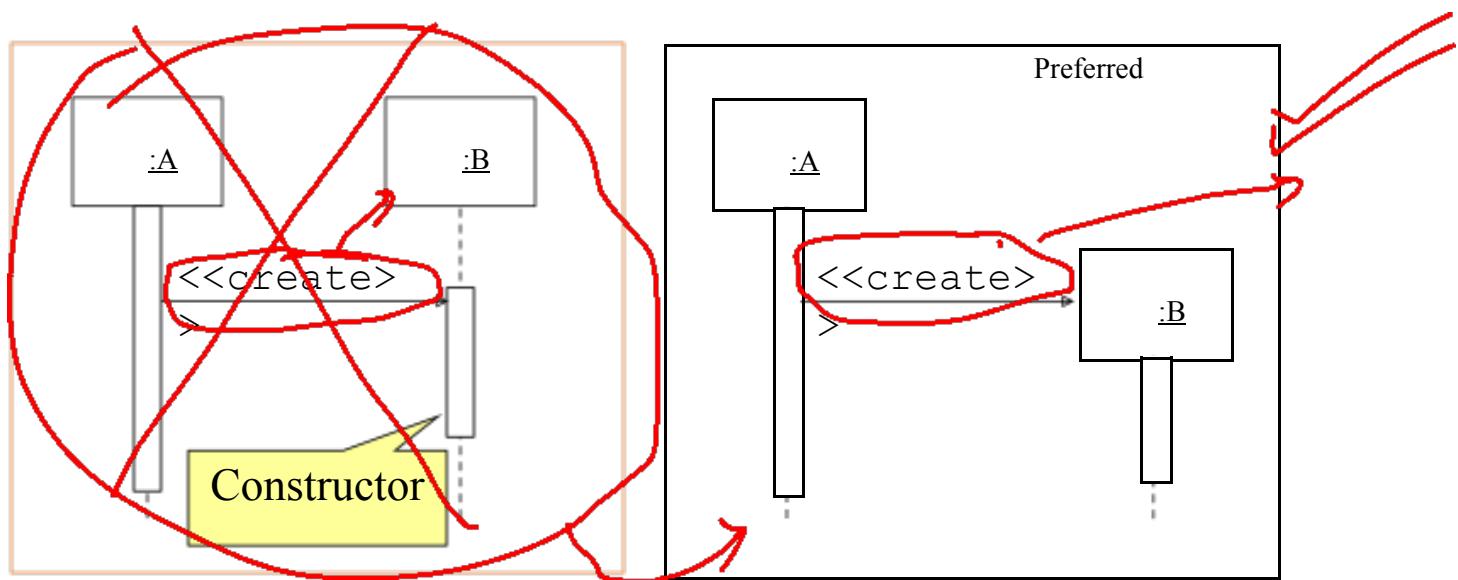
- **Lost Message** is a message where the sending event is known, but there is no receiving event



Web Client sent search message which was lost.

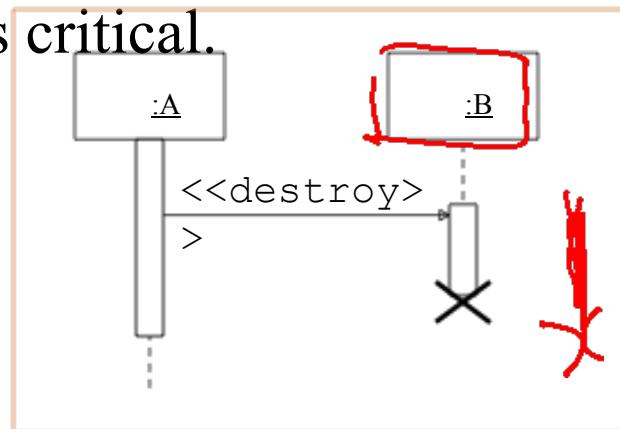
Object Creation

- An object may create another object via a <<create>> message.

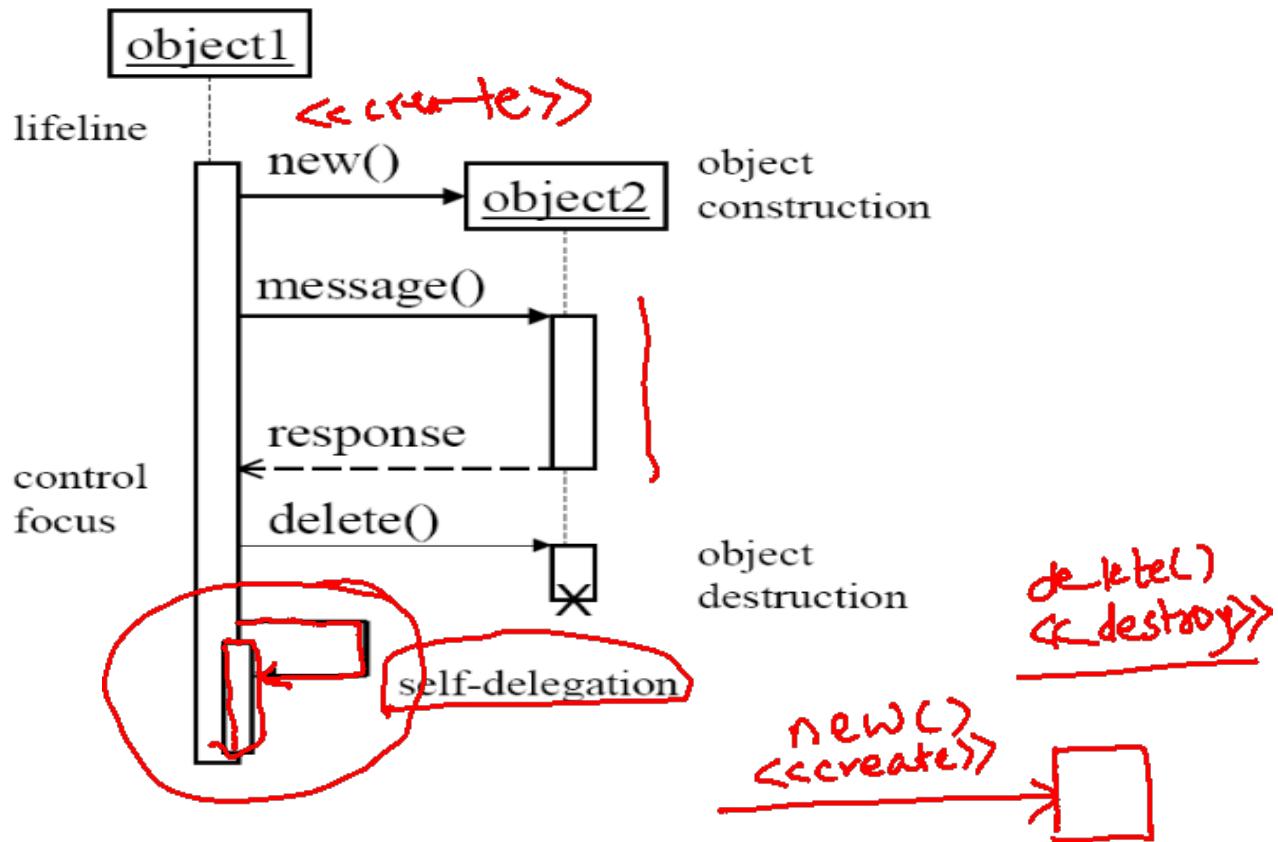


Object Destruction

- An object may destroy another object via a <<destroy>> message.
- An object may destroy itself.
- Avoid modeling object destruction unless memory management is critical.



Sequence Diagram - 2



Control information

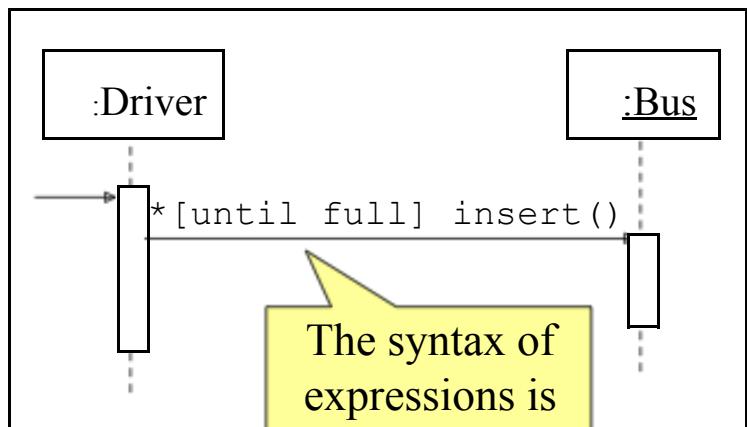
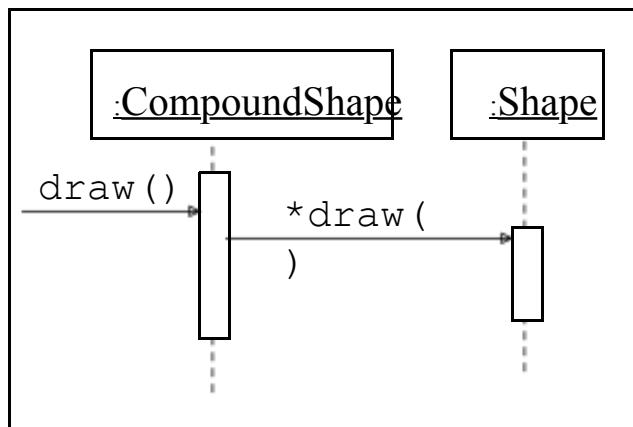
- Condition
 - syntax: '[' ' expression ']' message-label
 - The message is sent only if the condition is true
 - example:
- Iteration
 - syntax: * ['[' ' expression ']'] message-label
 - The message is sent many times to possibly multiple receiver objects.

[ok] borrow(member)



Control Information (Cont.)

- Iteration examples:

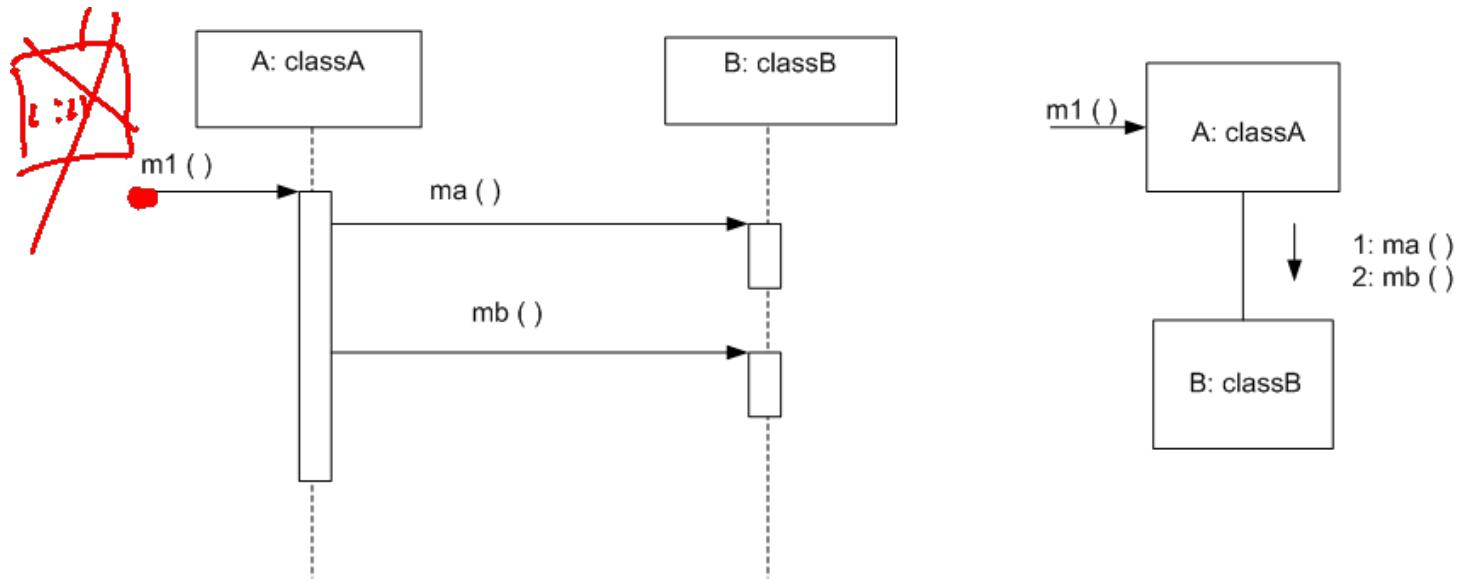


The syntax of
expressions is
not a standard

Control Information (Cont.)

- The control mechanisms of sequence diagrams suffice only for modeling simple alternatives.
- Consider drawing several diagrams for modeling complex scenarios.
- Don't use sequence diagrams for detailed modeling of algorithms (this is better done using *activity diagrams*, *pseudo-code* or *state-charts*).

Sequence Diagram basic example

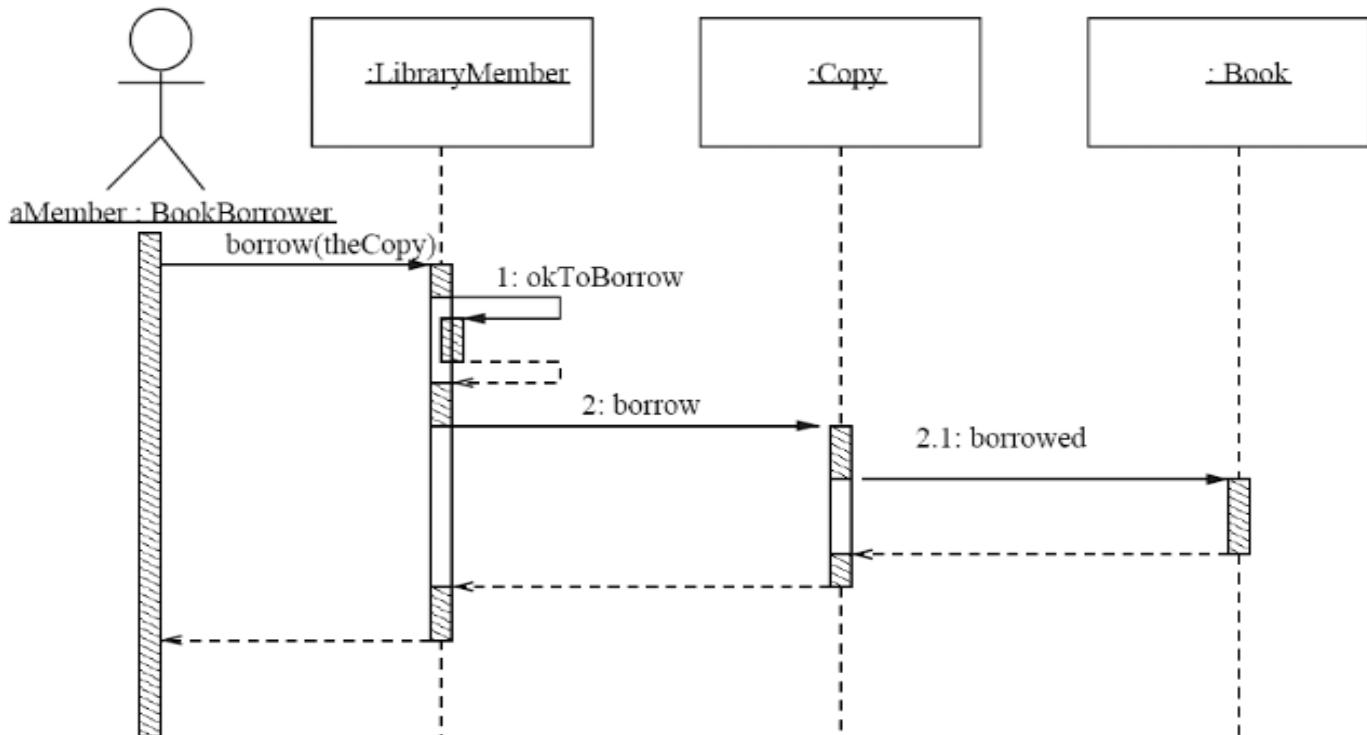


```
public class classA  
private classB B = new ClassB();  
public void m1(){  
    B.ma();  
    B.mb();  
}
```

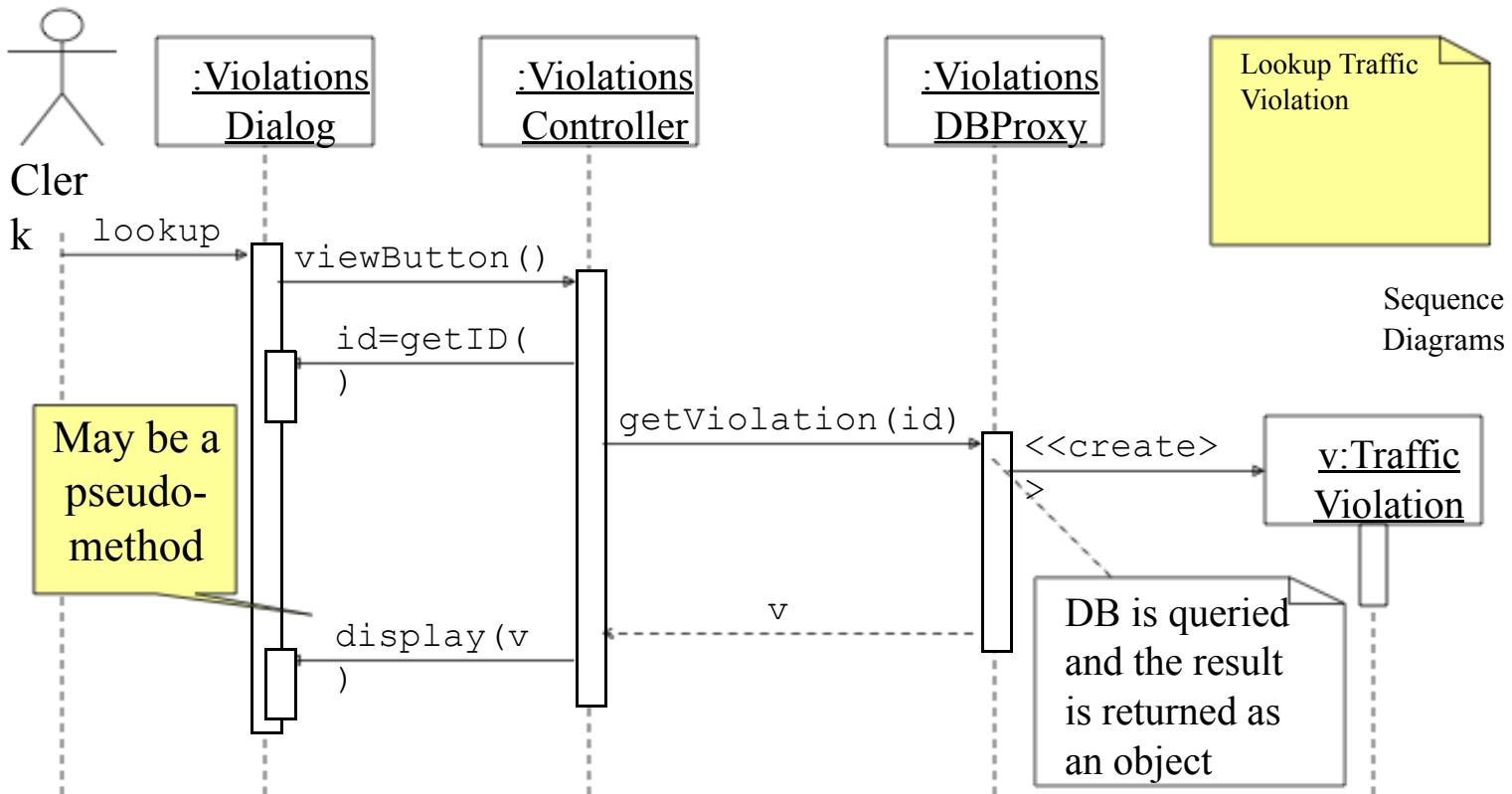
status=m1()

OK

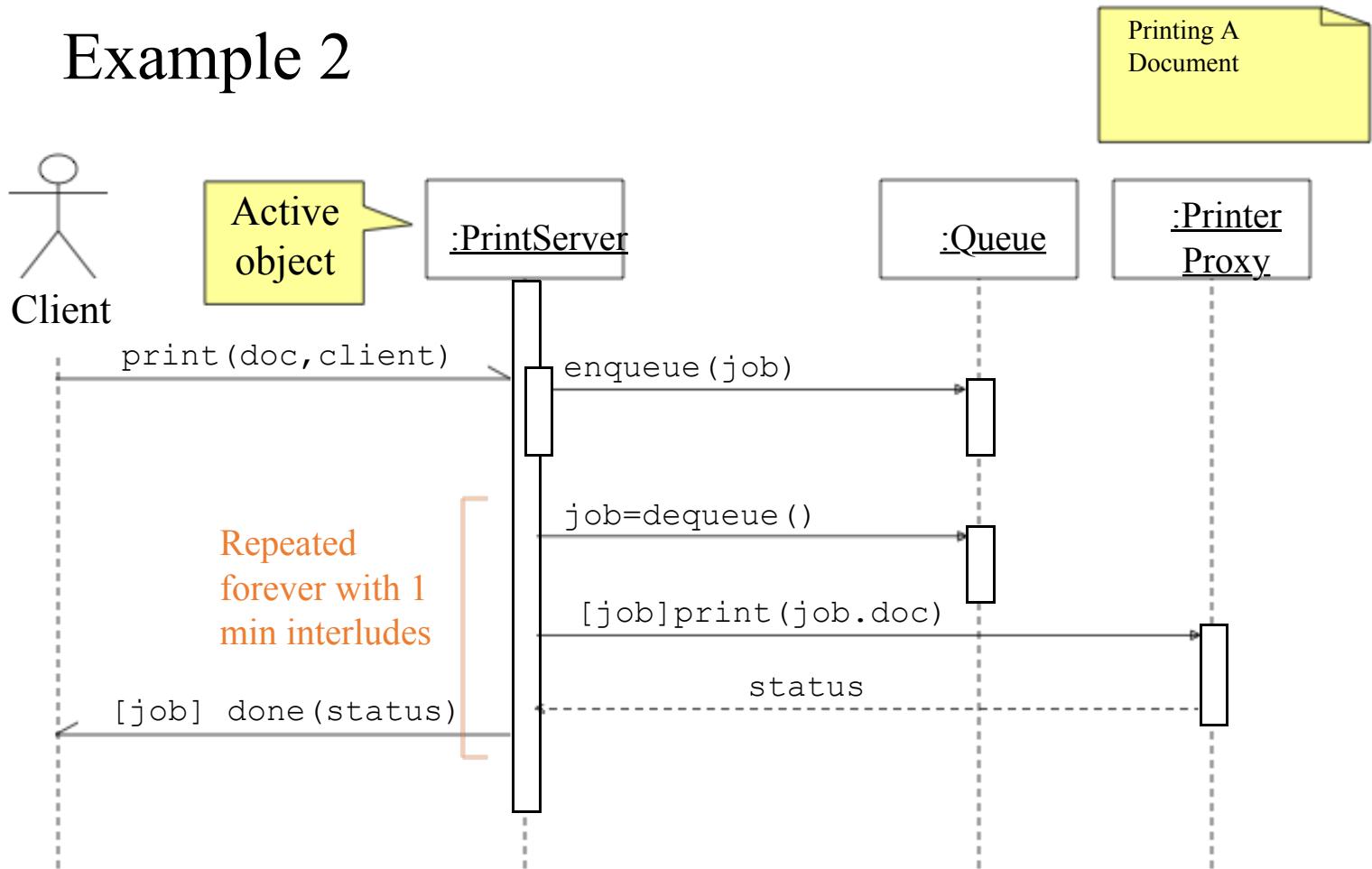
Nested activation

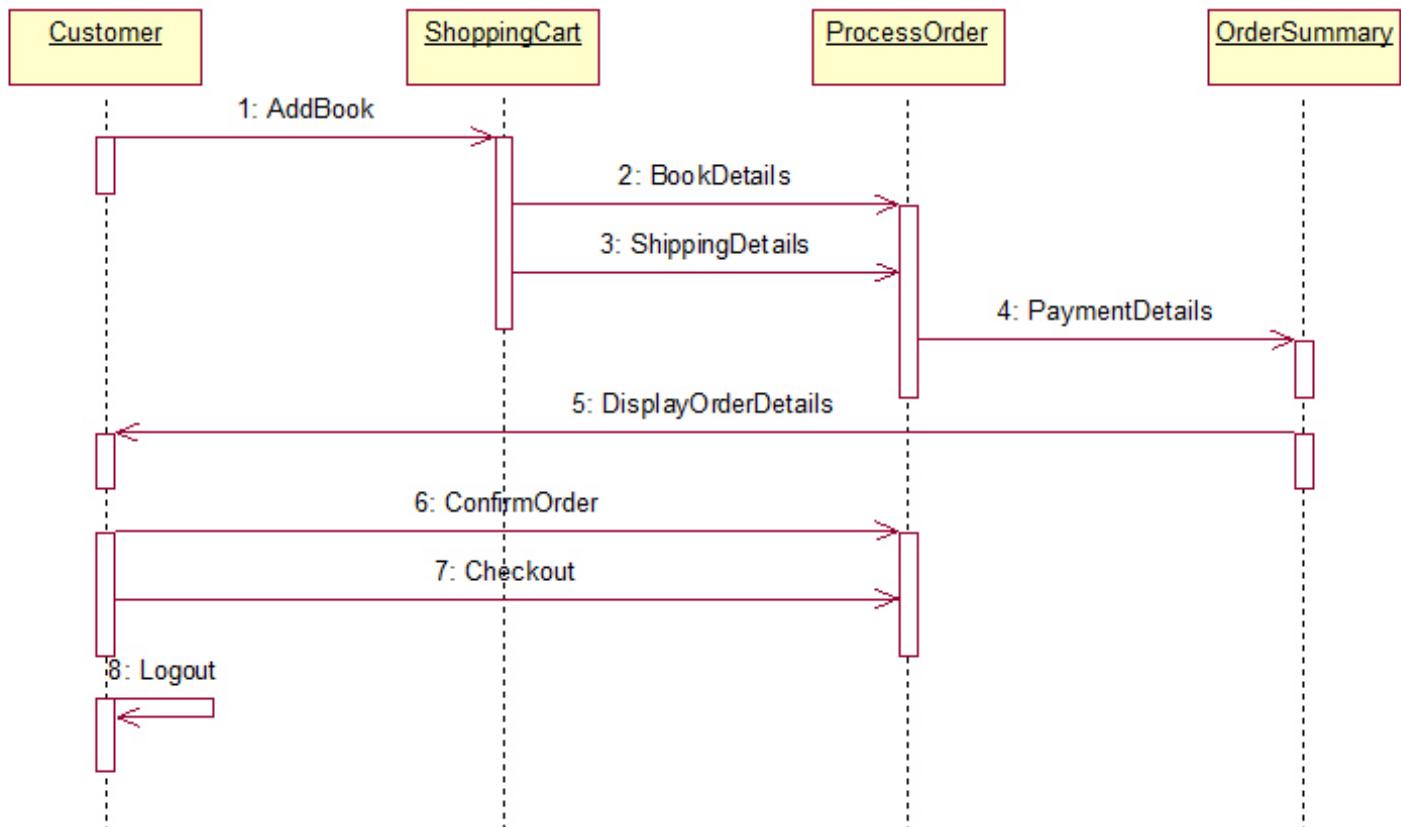


Example 1

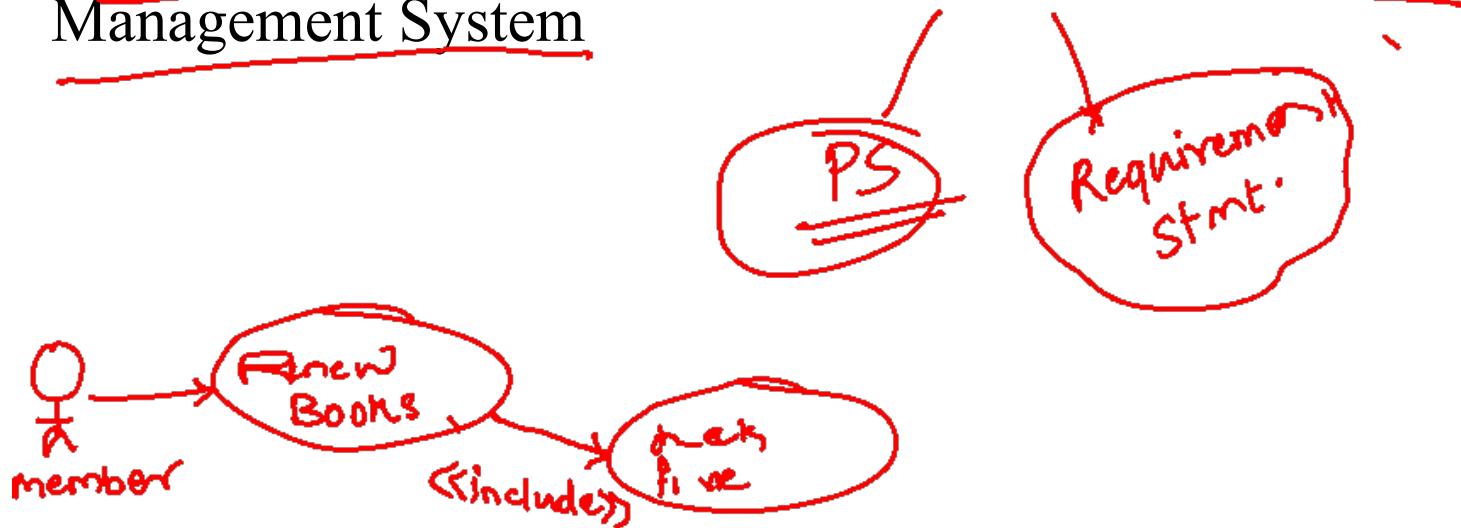


Example 2

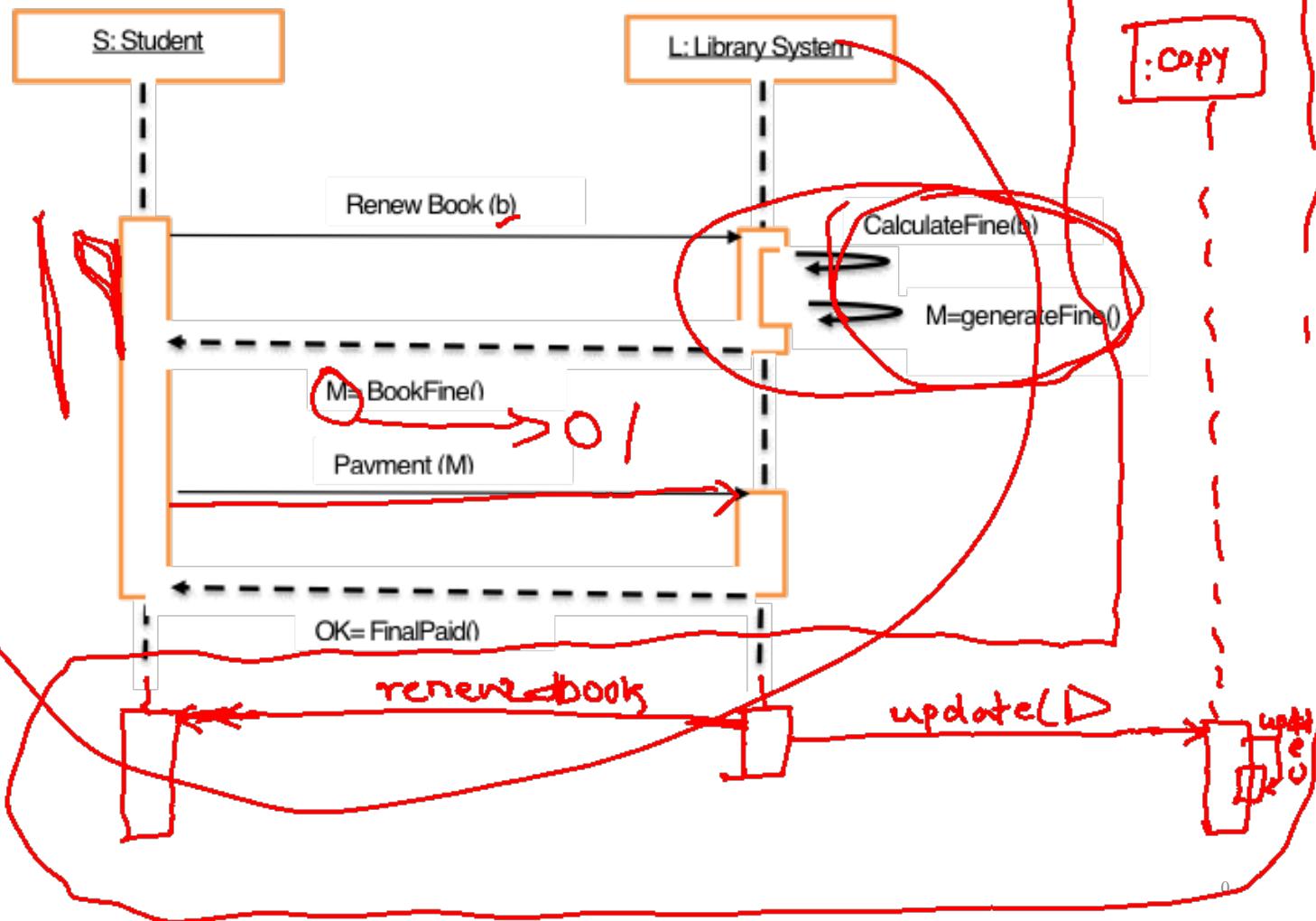




Draw sequence diagram for “Renew books with fine generation for late renewal” scenario in Library Management System



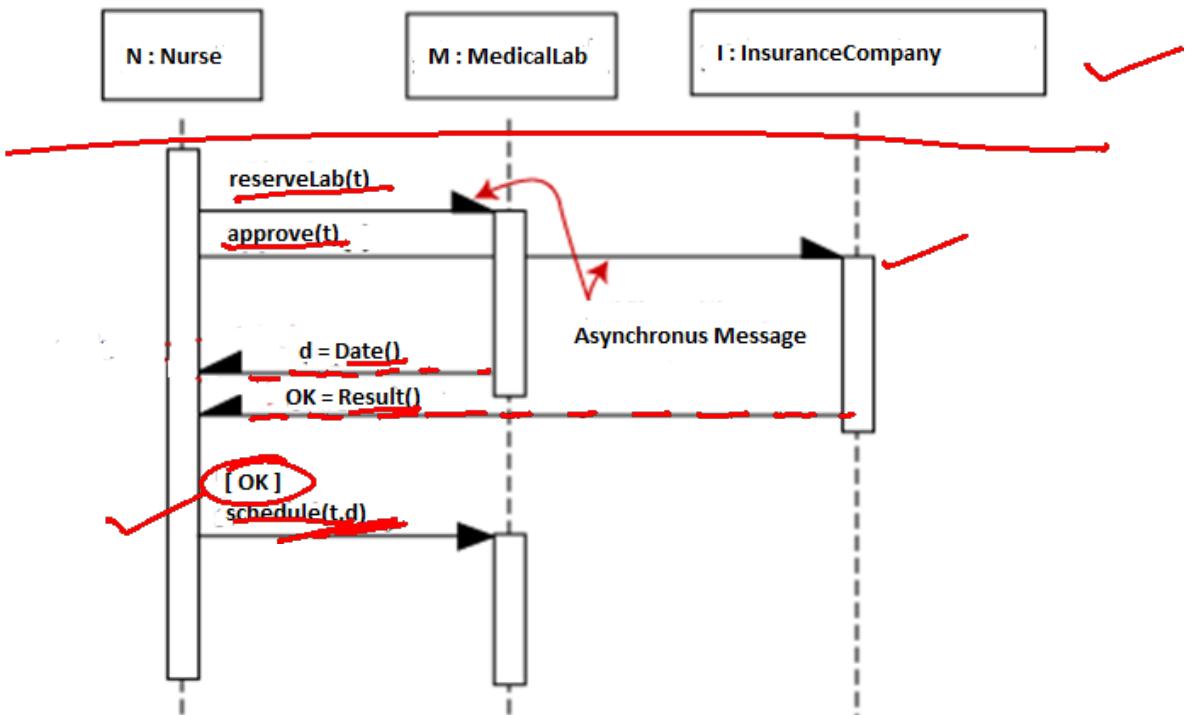
Sequence Diagram:



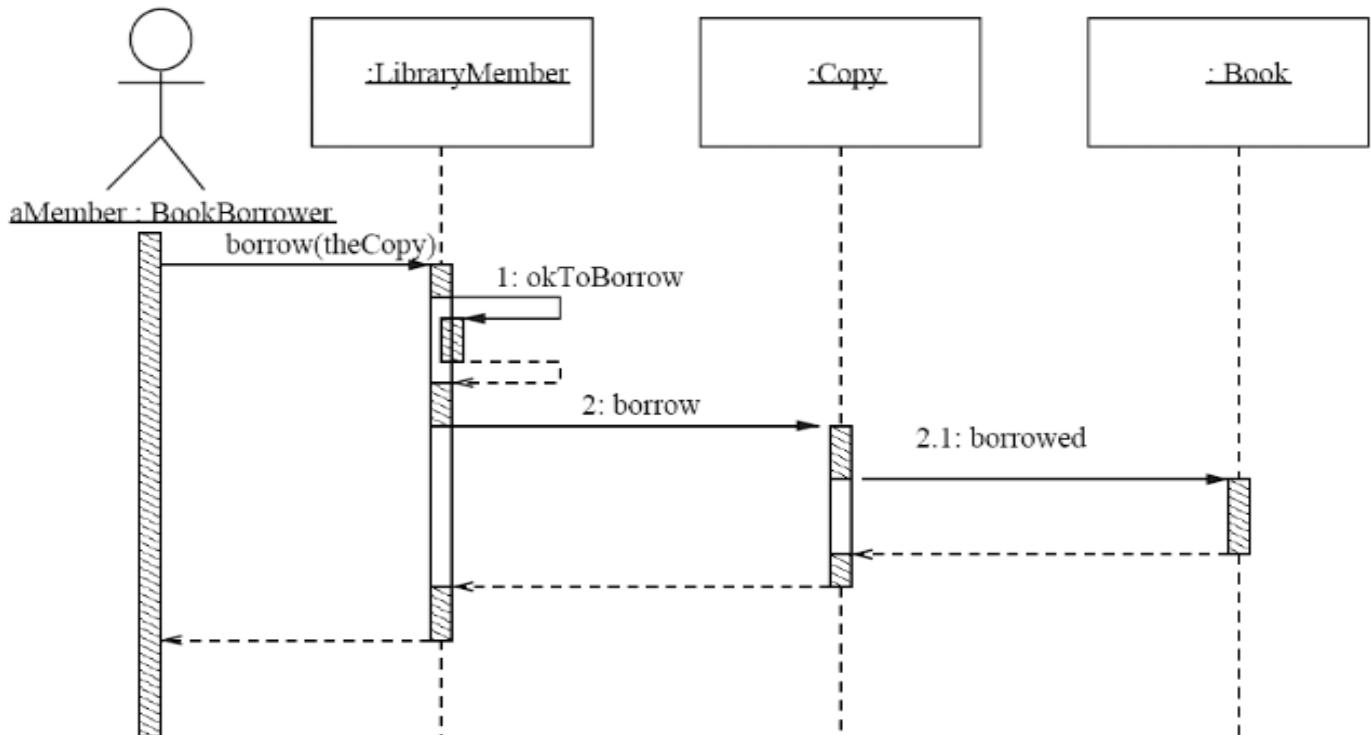
Example

In a hospital, nurse simultaneously requests the medical lab to reserve a date for the patient's diagnostic test (t) and the insurance company to approve the test. If the Insurance Company approves the test, then the Nurse will schedule the test on the date supplied by the Medical lab.

Asynchronous messages



Nested activation



Control information

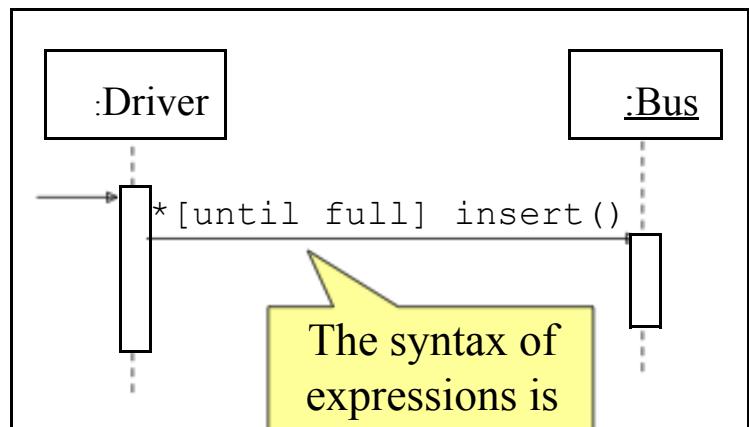
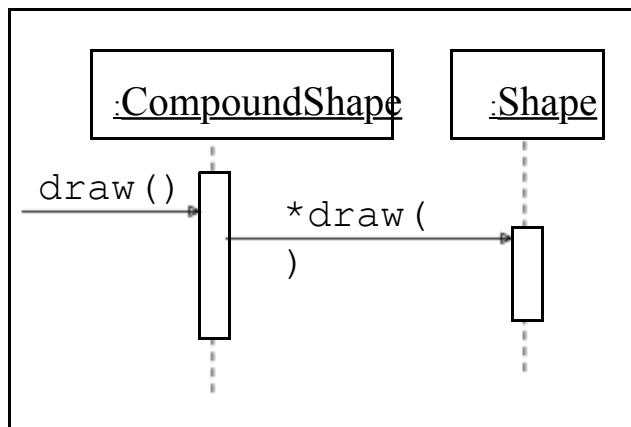
- Condition
 - syntax: '[' ' expression ']' message-label
 - The message is sent only if the condition is true
 - example:
- Iteration
 - syntax: * ['[' ' expression ']'] message-label
 - The message is sent many times to possibly multiple receiver objects.

[ok] borrow(member)



Control Information (Cont.)

- Iteration examples:



The syntax of
expressions is
not a standard

Control Information (Cont.)

- The control mechanisms of sequence diagrams suffice only for modeling simple alternatives.
- Consider drawing several diagrams for modeling complex scenarios.
- Don't use sequence diagrams for detailed modeling of algorithms (this is better done using *activity diagrams*, *pseudo-code* or *state-charts*).

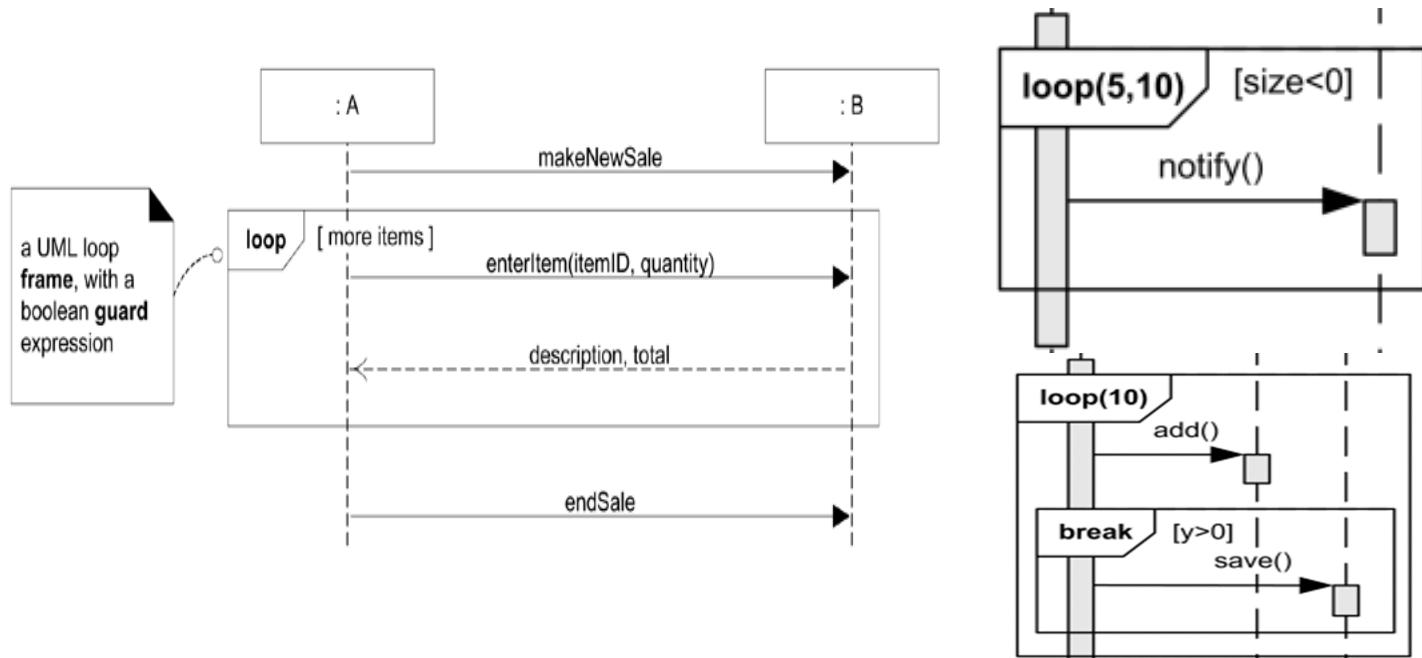
Interaction Fragment in Sequence Diagram

- An Interaction fragment which defines an expression of interaction fragments.
- An interaction fragment is defined by an interaction operator and corresponding interaction operands. Through the use of interaction fragments the user will be able to describe a number of traces in a compact and concise manner.
- interaction fragment may have constraints also called guards in UML

- Typical frame operators are:

Frame Operator	Semantics
alt	Alternative fragment for mutual exclusion conditional logic expressed in the guards.
loop	Loop fragment while guard is true. Can also write loop(n) to indicate looping n times.
opt	Optional fragment that executes if guard is true.
par	Parallel fragments that execute in parallel.
region	Critical region within which only one thread can run.

- Frames in UML Sequence Diagrams :
- To allow the visualization of complex algorithms, sequence diagrams support the notion of frames;
- Frames are regions of the diagrams that have an operator and a guard., Figure.



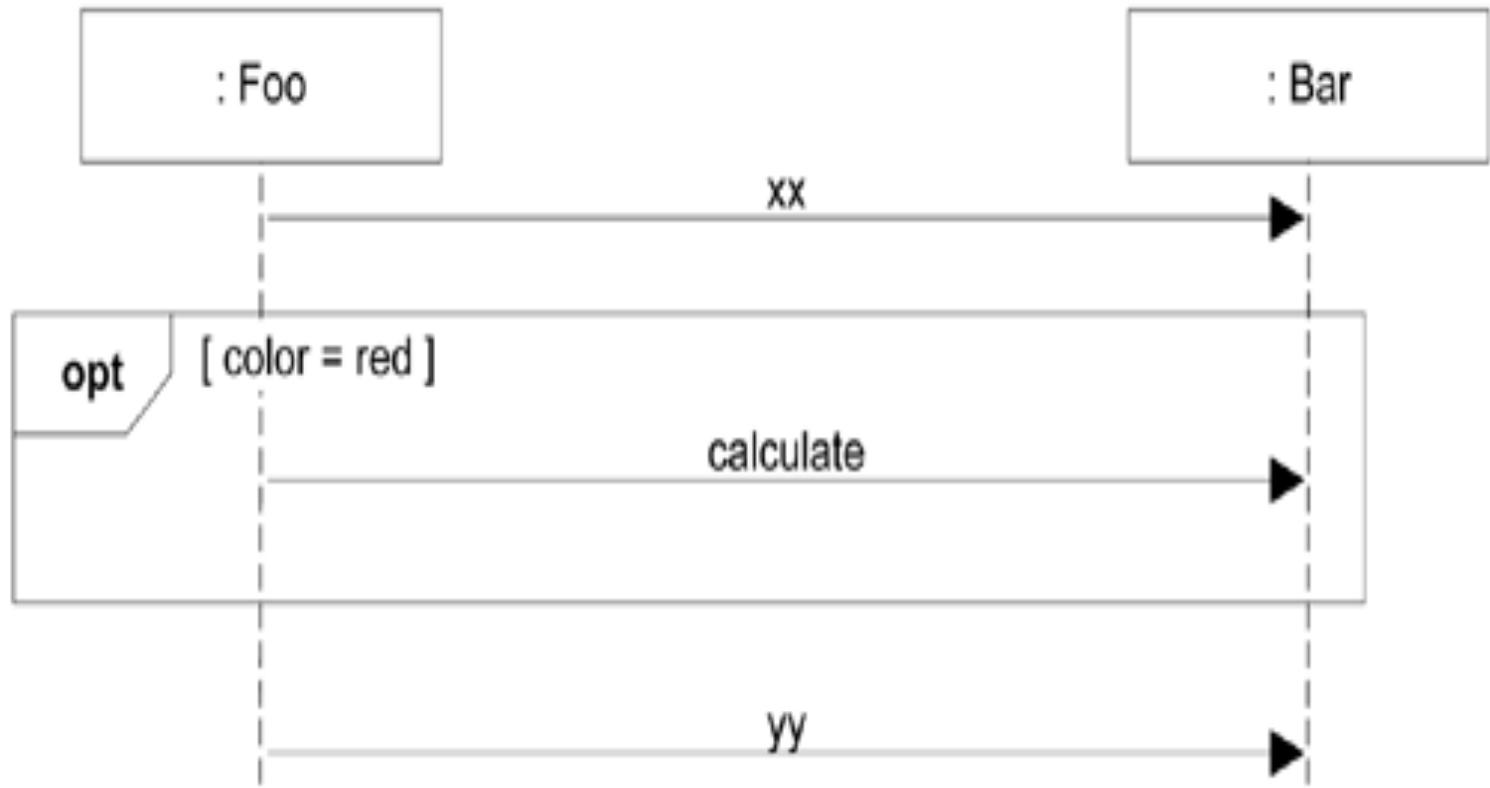


Figure Using the opt Frame

- Next Figure illustrate the use of the alt frame for mutually exclusive alternatives

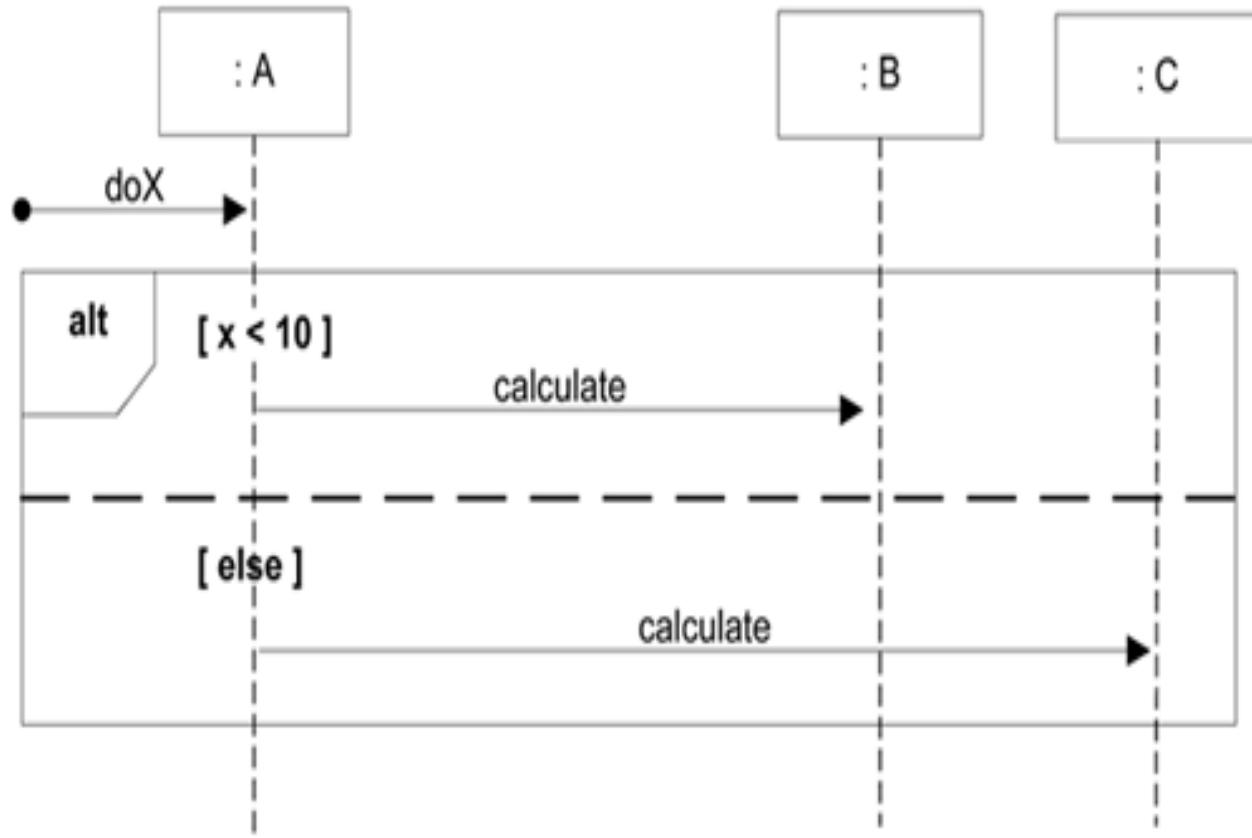
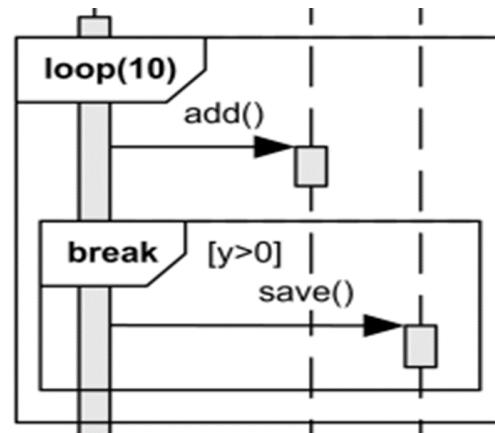
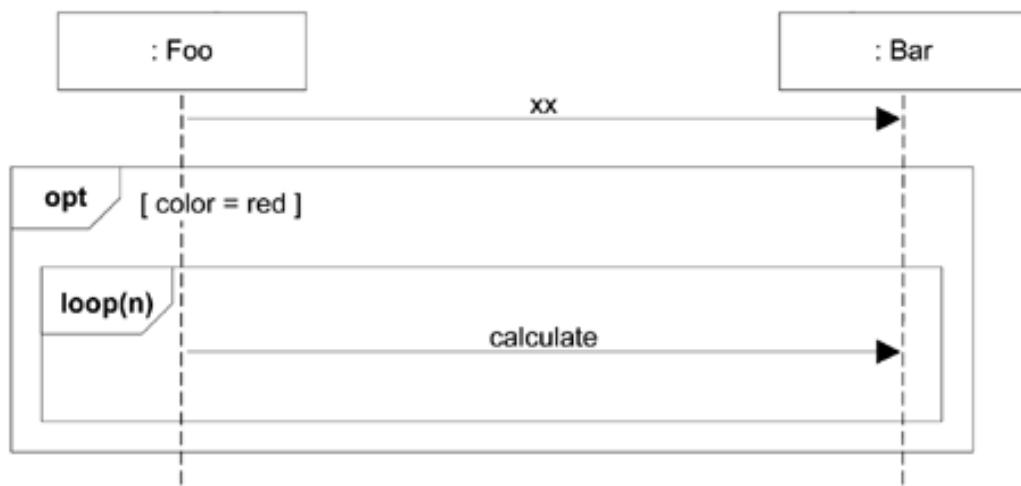


Figure Using the **alt** Frame

- Nesting of Frames :
- Frames can be nested:



- MyManipal is an online social networking service only for MIT students and staff. The users must register before using the site. Once the user is authenticated he can retrieve notification, updates and messages. Further, in the next step the user can change the user preferences as online/ hidden. Finally, the status will be displayed in users' wall. Unsuccessful login will lock the account for 1 minute to prevent from brute force attack.

- The user wants to boot a server with Linux operating system (OS). So whenever the user sends a signal *start* the operating system will load RAM and the booting process will be initiated. After a time interval, the OS will get its current date and time. Moreover, the user will be acknowledged with current date *d* and time *t* by the OS within a stipulated time period. Based on the user request the OS will spawn two processes simultaneously namely Pi and Pj. The order of creation of above two processes is random. Each process will return its *pid* to OS. Due to the scarcity of resources the OS is required to kill any one of the processes. The built-in functionality in OS called *Preemptive Algorithm* chooses Pi as victim process. OS sends a signal *kill* to kill the process Pi. If the OS is unsuccessful in killing the process Pi, then the OS will send *kill* message to kill the process Pj.

State Diagram

State Chart Diagram
State Transition Diagram

Introduction

- Two interaction diagrams with objects of the same class receiving the same messages may respond differently
- This is because an object's behavior is affected by the values of its attributes
- UML State Machine Diagram records these dependencies

State Transition Diagram

- A state transition diagram is a technique to depict:
 1. The states of an entity
 2. The transitions of states of the entity
 3. The trigger or the event that caused the transition of state of the entity
- The **entity** may be a physical device such as a light switch or a vending machine; it may be a software system or component such as a word processor or an operating system; it may be a biological system such as a cell or a human; or ----

This modeling technique came from a more formal area called **automata theory**. State transition diagram depicted a **Finite State Machine**.

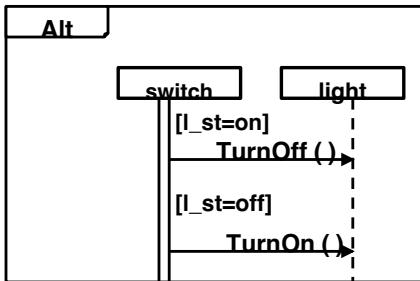
Software Program View

- The end product of a software is a program which executes. In depicting the program (or an object) we can consider:
 - Variables which take on different values
 - Control structure and assignment statements (events) in the program that change the values of the variables

1. *Combination of values of the data (variables & constants) at any point of the program represent the program state at that point.*
2. *The change made to the values of the variables through assignment statements represent a transition of state*

A very simple example

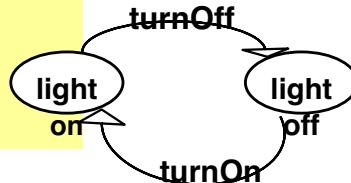
light switch



From State (light)	Event (switch)	To State (light)
on	turnOff	off
off	turnOn	on

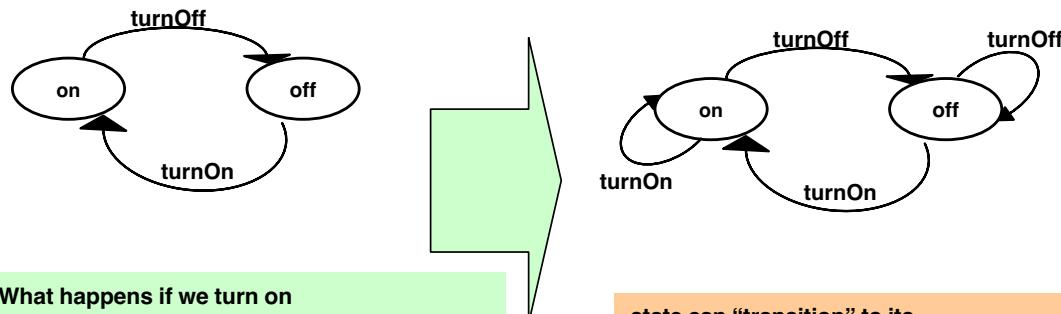
1. “**Sequence diagram**”
(alternative fragment)
for switch and light
interaction

2. “**State transition table**”
for light with switch events



3. “**State transition diagram**”
for light with switch events

A little “more” on the light switch



What happens if we turn on
a light that is already on?

state can “transition” to its
current state

Using State Transition Diagram

- Model the entity at the “abstraction” level where the *number of states is* “manageable.”
 1. List (design) the states (should not be large)
 2. List events that will trigger the state transition (should not be big)
 3. There must be a starting state
 4. There must be a terminating state or states
 5. Design the transition rules (the bulk of your design work is thinking through the transition rules)

1.The above is not necessarily performed in sequence; iterate through these.

2. Even with a modest number of states and events, the state transition diagram, which really depicts the transition rules, can be enormous.

State Diagrams

- State diagrams are used to show possible states a single object can get into
 - shows states of an object
- How object changes state in response to events
 - shows transitions between states

Elements of State Diagram

- Start marker
- Stop marker
- States – box with rounded corners
- Transitions – shown as arrows between states
- Events – that cause transition between states
- Action – an object's reaction to an event
- Guard

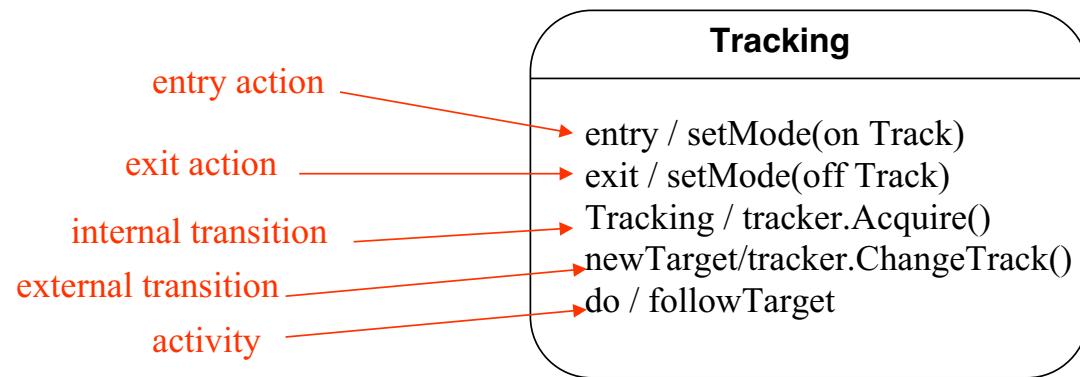
Naming Conventions for a state

- Each unique state has a unique name
- State Names are **verb** phrases
- A noun to name a state – **incorrect**

State Diagrams: States

- States are represented as rounded boxes which contain:
 - the **state name**
 - and the following optional fields
 - **entry and exit actions:** entry and exit actions are executed whenever the state is entered or exited, respectively
 - **Internal transitions:** A response to an event that causes the execution of an action but does not cause a change of state or execution of exit or entry actions.
 - **External transition:** A response to an event that causes a change of state or a self-transition, together with a specified action.
 - **Activities:** Typically, once the system enters a state it sits idle until an event triggers a transition. Activities help you to model situations where while in a state, the object does some work that will continue until it is interrupted by an event

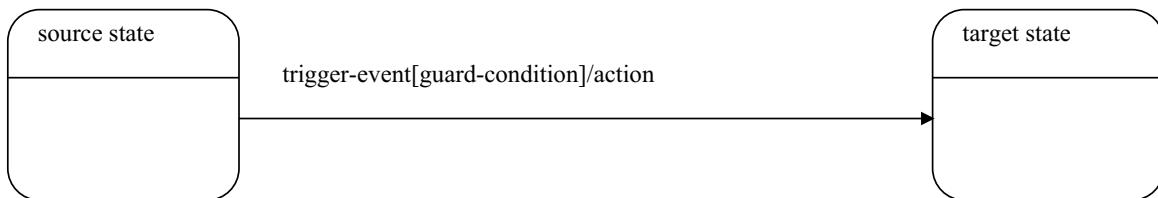
State Diagrams: States



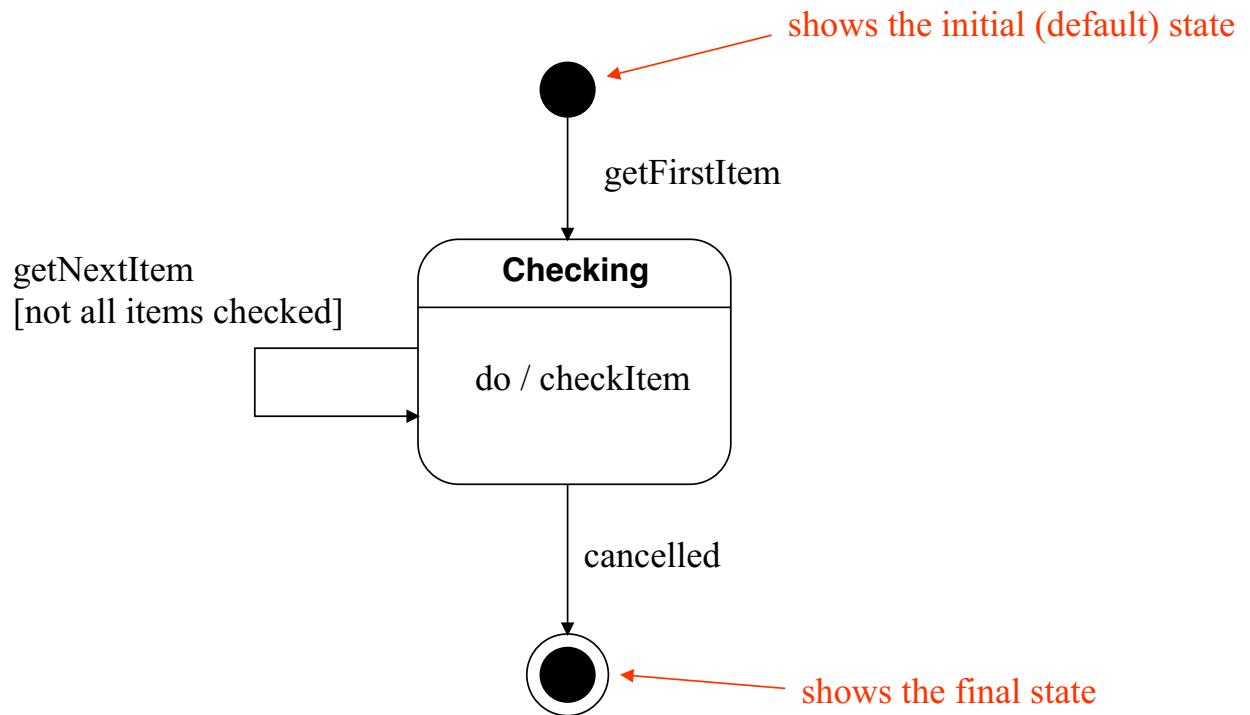
Note that, “entry”, “exit”, “do” are keywords

State Diagrams: Transitions

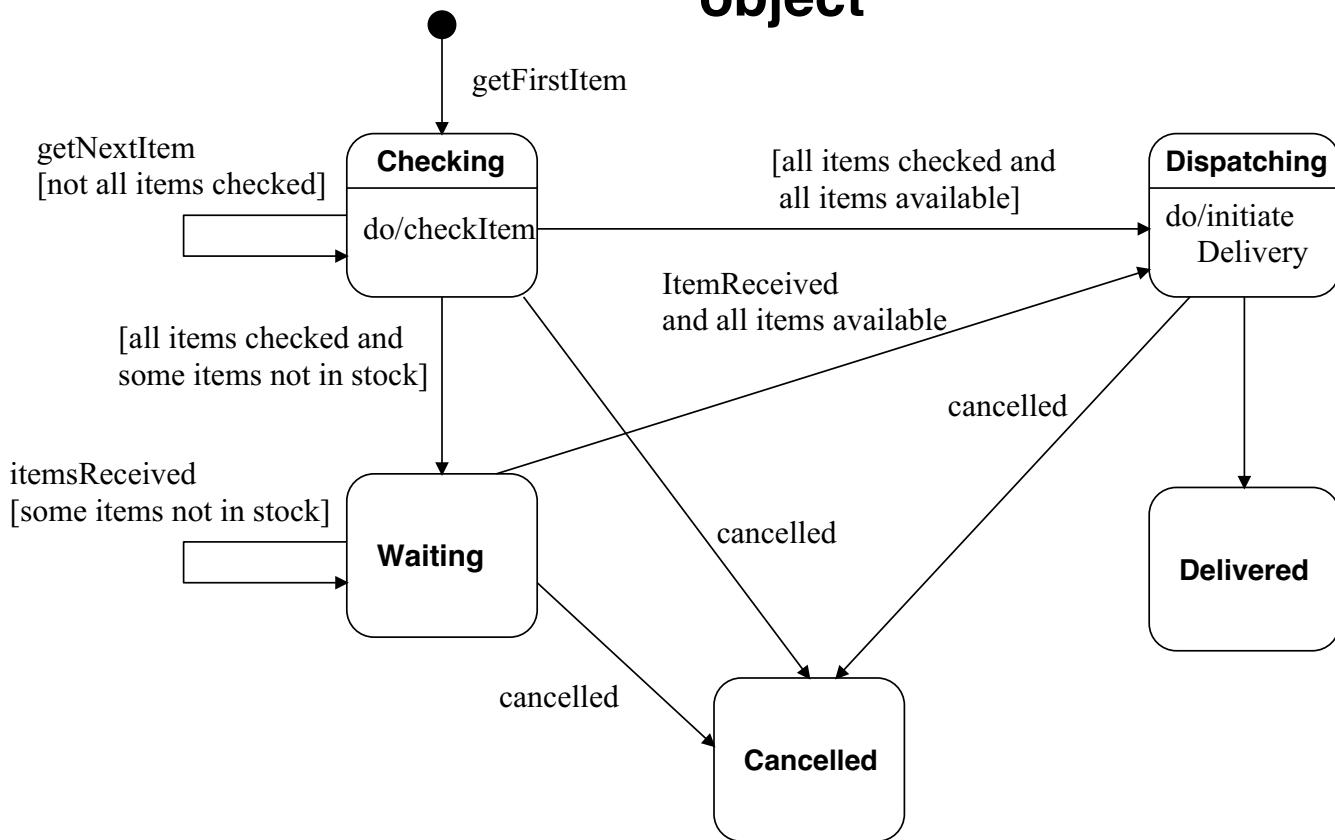
- Transitions
 - **source state** and **target state**: shown by the arrow representing the transition
 - **trigger event**: the event that makes the transition fire
 - **guard condition**: a Boolean expression that is evaluated when the trigger event occurs, the transition can fire only if the guard condition evaluates to true
 - **action**: an executable atomic computation that can directly act on the object that owns the state machine or indirectly on other objects that are visible to the object
 - **initial and final states**: shown as filled black circle and a filled black circle surrounded by an unfilled circle, respectively



State Diagrams

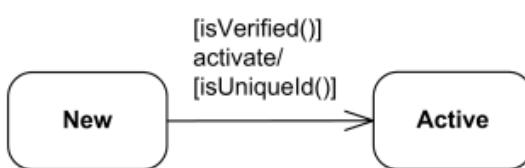


State Diagram Example: States of an Order object



Types of State Machine Diagram

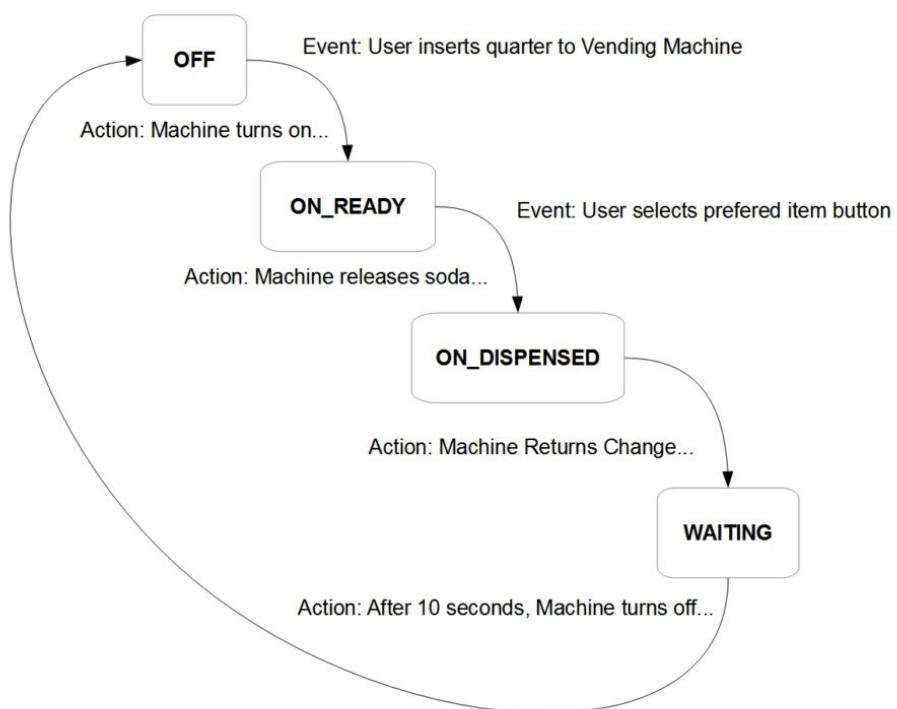
- Protocol State Machines: These are used to express a **usage protocol** or a **lifecycle** of some **classifier**. It shows which operations of the classifier may be called in each state of the classifier, under which specific conditions, and satisfying some optional postconditions after the classifier transitions to a target state.
- Behavioral State Machines: These are specialization of behavior and is used to specify discrete behavior of a part of designed system through finite state transitions.



Example of Protocol State Machines



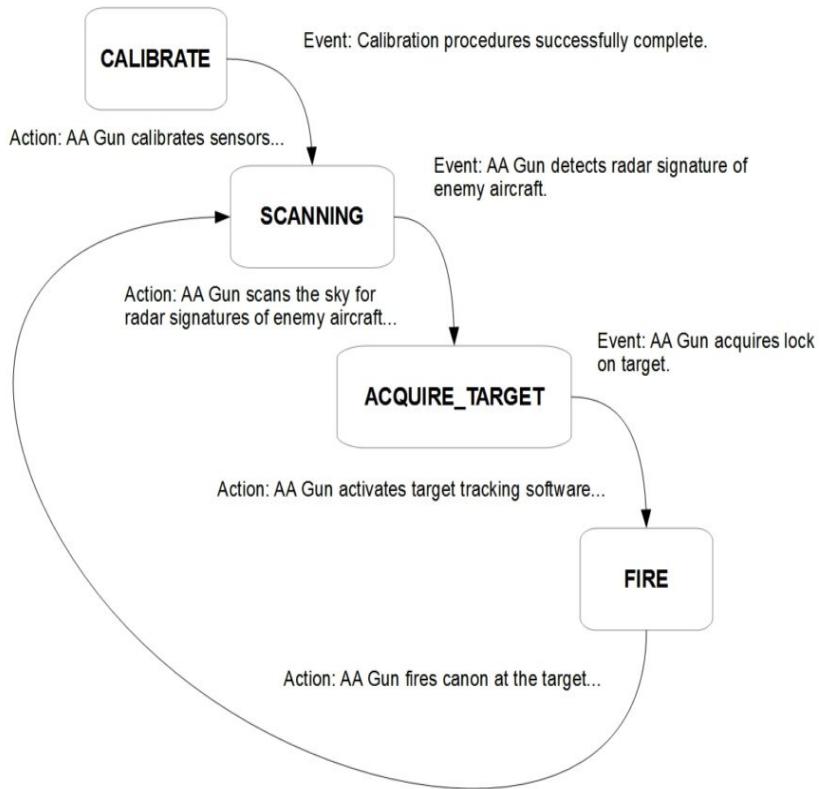
Soda Vending Machine



Example of Behavioral State Machines



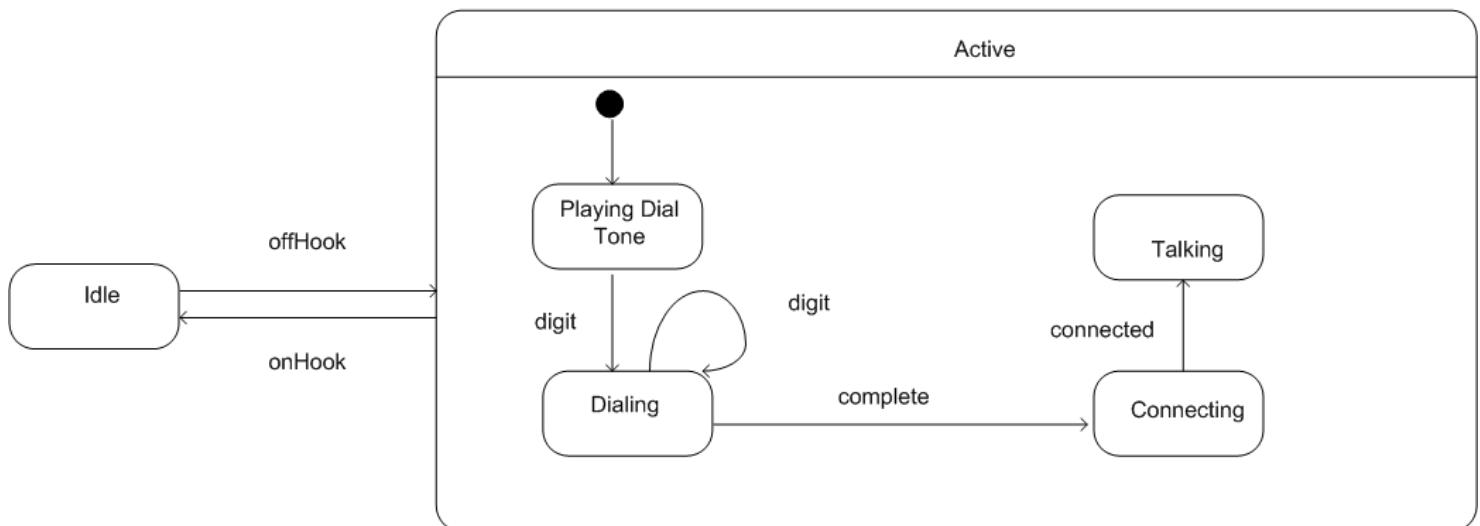
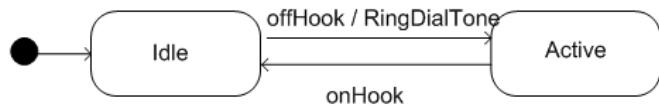
Anti-Aircraft (AA) Gun



Advanced State Machine Modeling

- Nested states
- Concurrent states

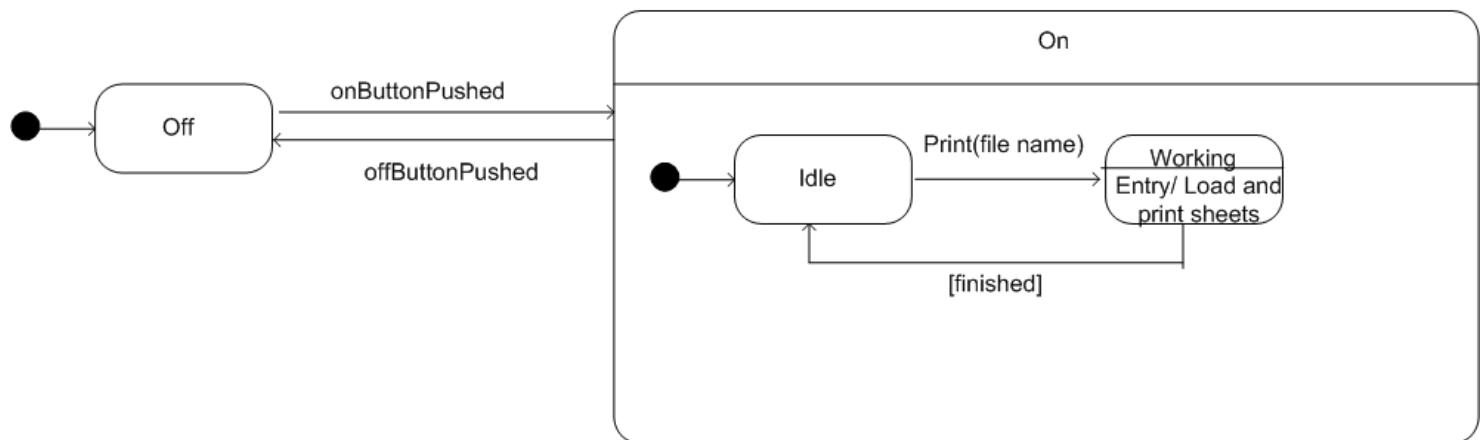
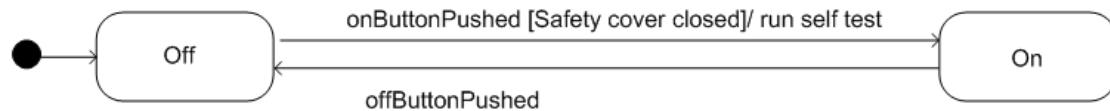
Nested State Machine- Telephone



Nested State Machine- Printer

- Example: When the printer is in On state, it may also be in Idle or Working
- To show these two states, draw lower level state diagrams within On state
- When the printer is on, it begins at Idle state, so printer is in both On and Idle state
- When print message is received, it moves to working state and also remains in On state

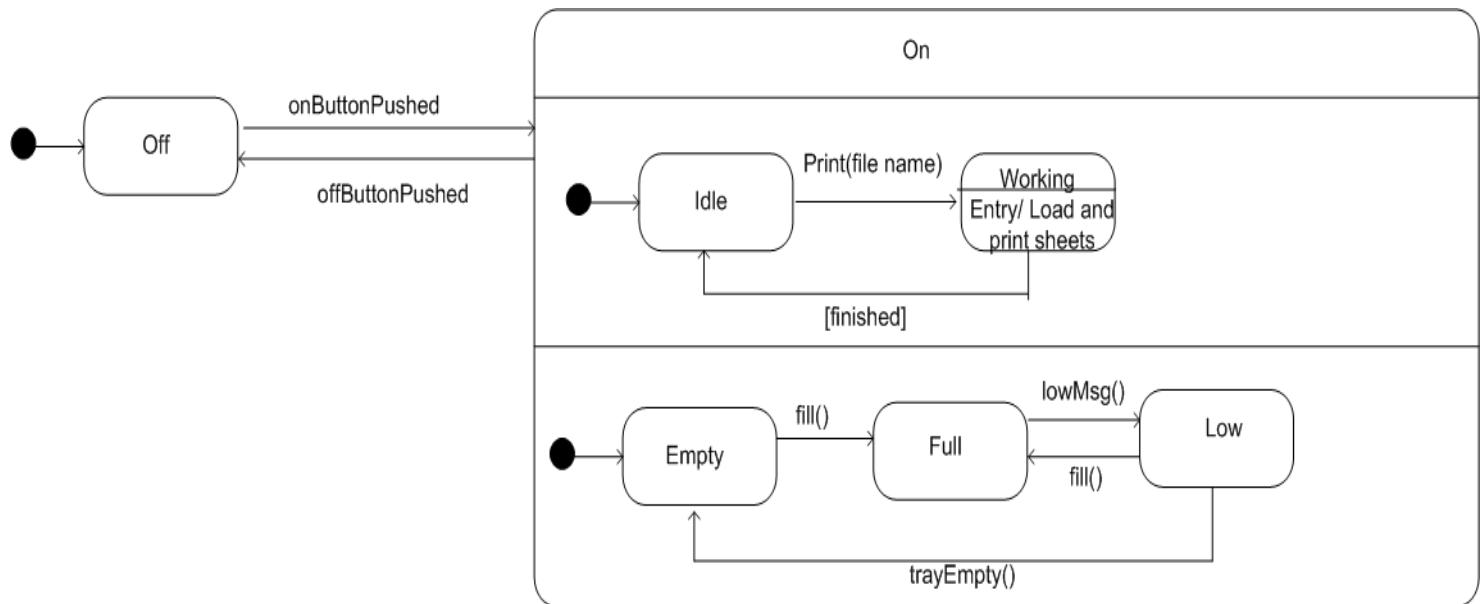
Nested State Machine- Printer



Concurrent states

- A Printer object cycles between two separate paths. The two independent paths are;
 - ❑ Representing states of the work cycle.
 - ❑ Representing states of the input paper tray

Concurrent State Machine- Printer



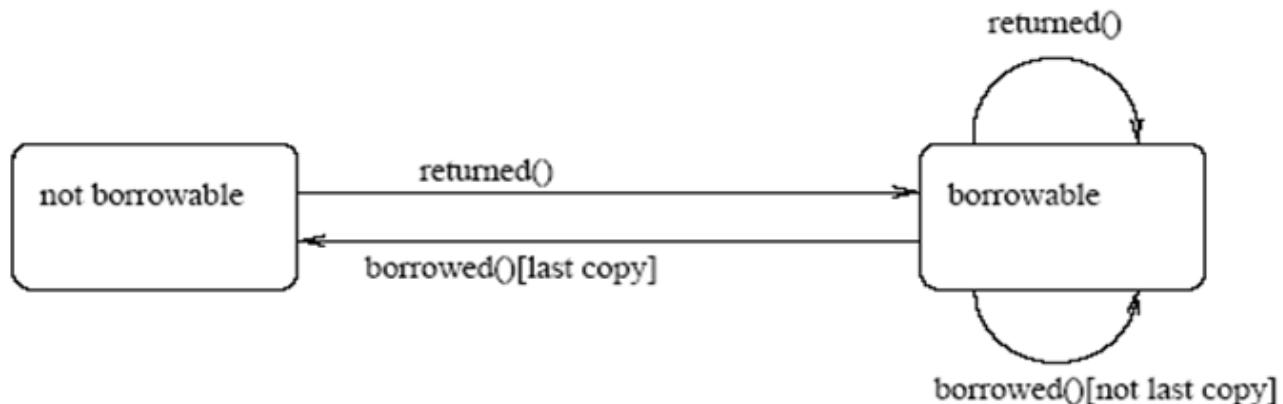
State Diagrams Importants

- Use them to show the behavior of a single object
not many objects
 - for many objects use interaction diagrams
- Do not try to draw state diagrams for every class
in the system, use them to show interesting
behavior and increase understanding

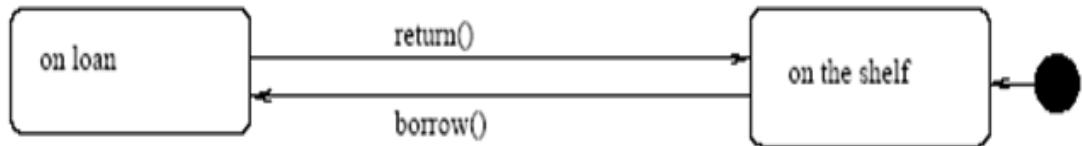
CRC for Copy & Book

Copy	Book
Responsibilities	Collaborators
Maintain data about a particular copy of a book Inform corresponding Book when borrowed and returned	Book
Book	
Responsibilities	Collaborators
Maintain data about one book Know whether there are borrowable copies	

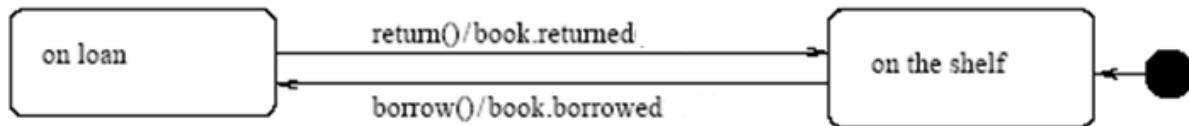
State Machine Diagram for class *Book*



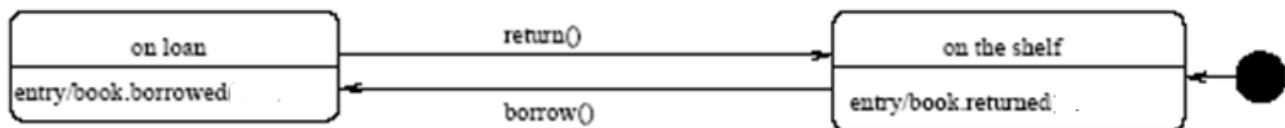
State Machine Diagram for class *Copy*



State Machine Diagram for class *Copy* with actions



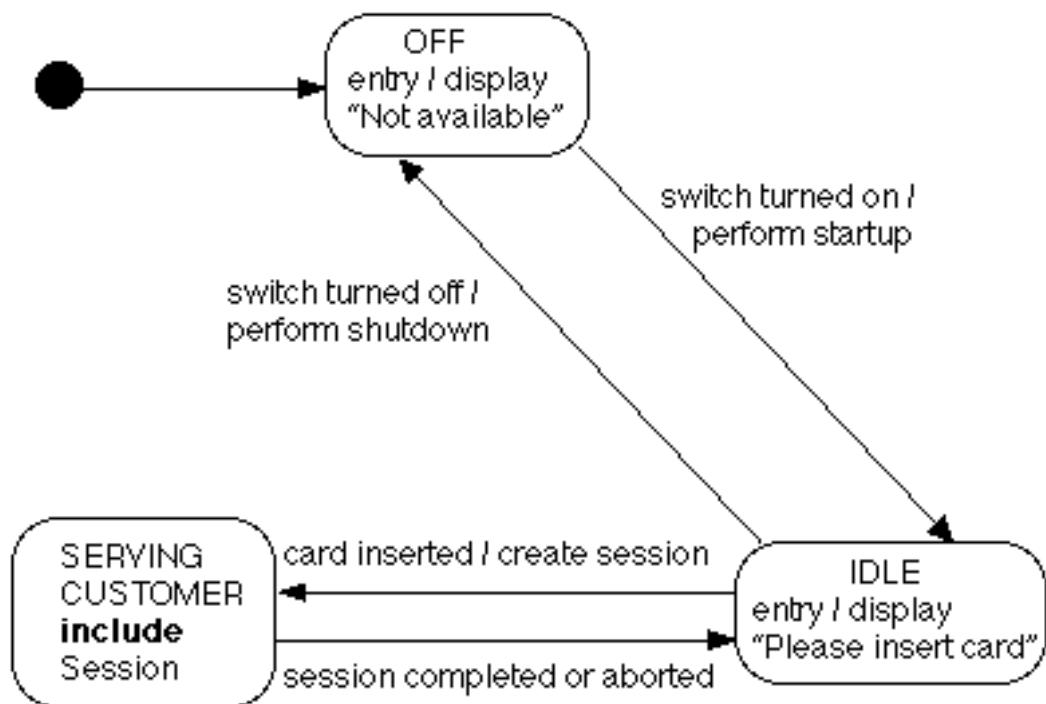
State Machine Diagram for class *Copy* with entry actions



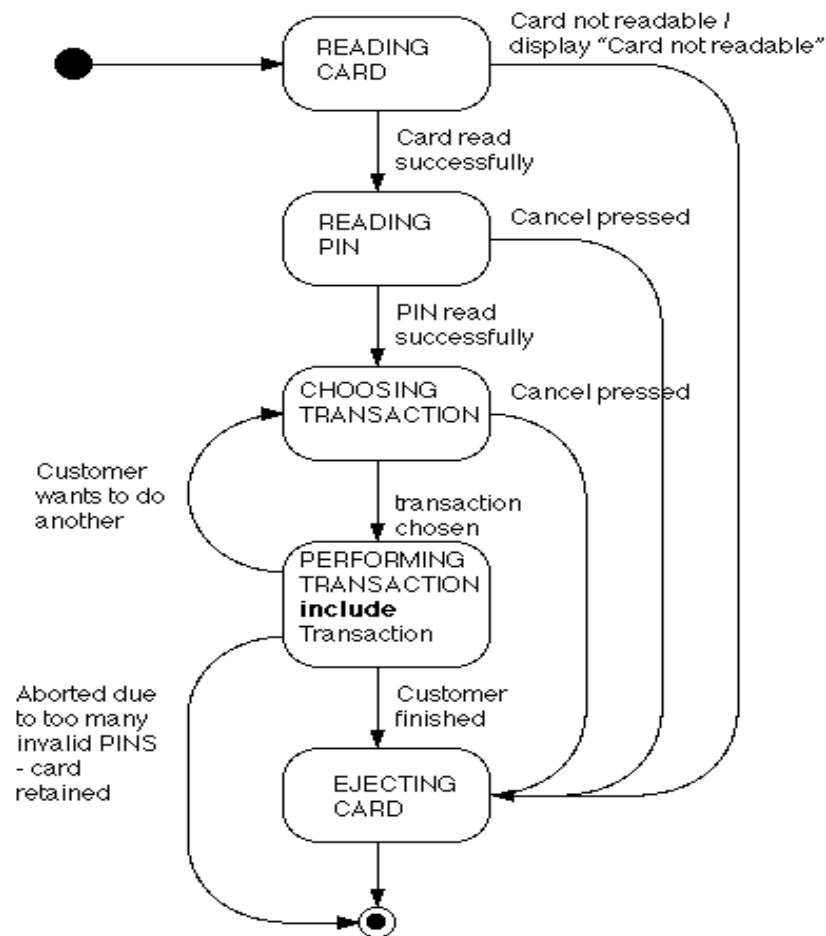
State Machine Diagram for class *Copy* with exit actions



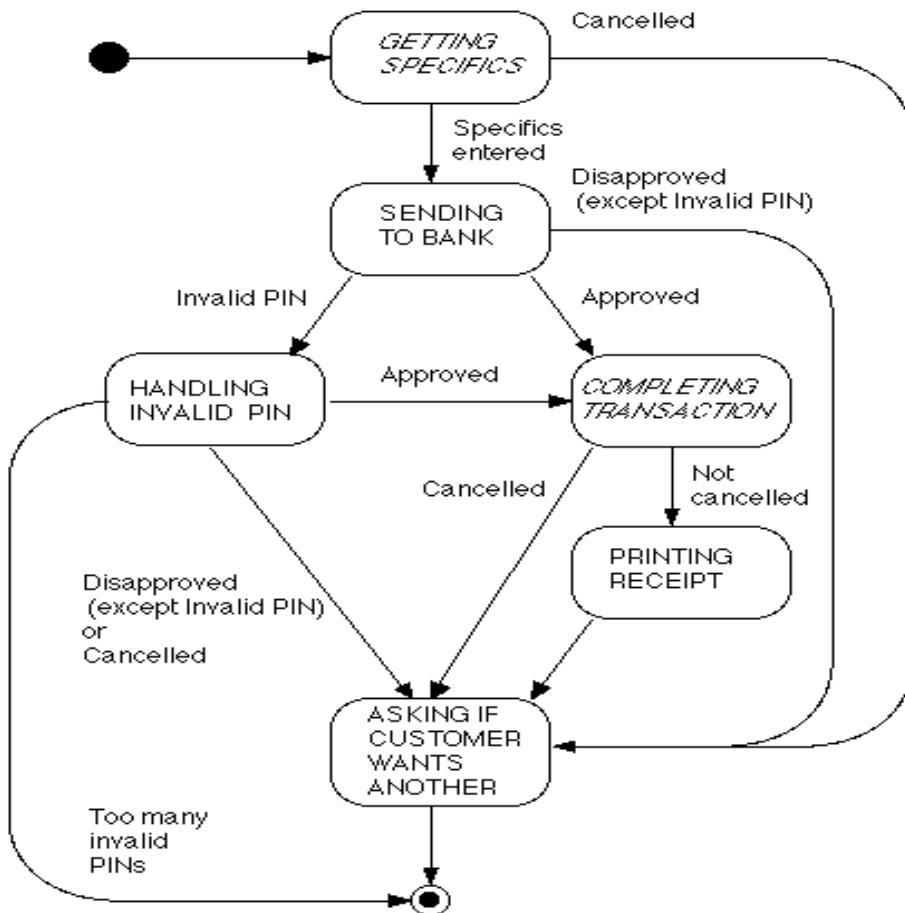
State-Chart for Overall ATM (includes System Startup and System Shutdown Use Cases)



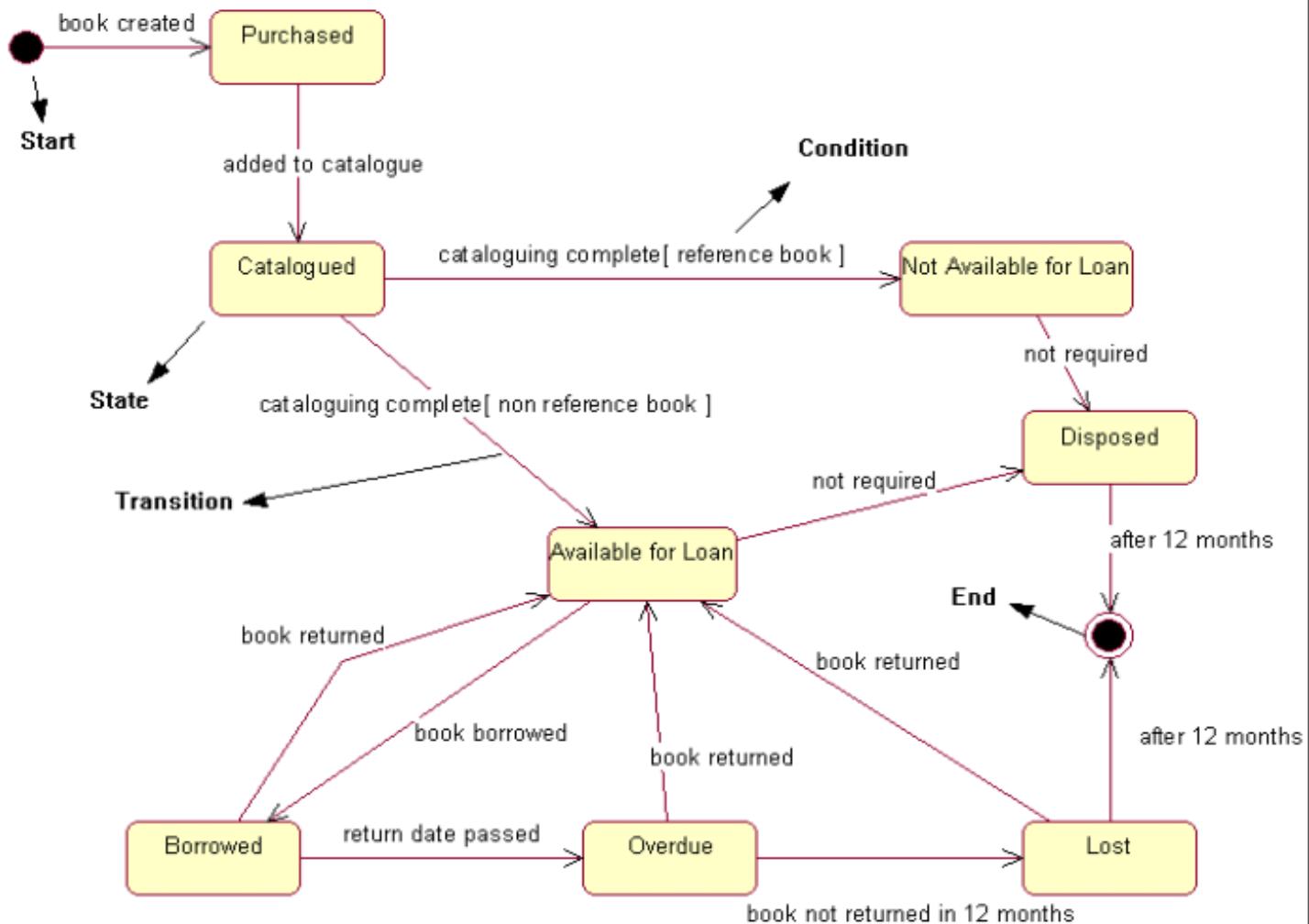
State-Chart for One Session



State-Chart for One Transaction
(italicized operations are unique to each particular type of transaction)



Librarians categorize the library books into loanable and non-loanable books. The non-loanable books are the reference books. However, the loanable books are the non-reference books. After cataloguing the books, the books are available for loan. Students who borrow the library books should return them back before the due date. Books that are 12 months over the due date would be considered as a lost state. However, if those books are found in the future, they must be returned back to the library. When the books are found not required in the library or have been damaged, the book would be disposed.



Difference between Activity and State Chart Diagram

State diagram shows the object undergoing a process. It gives a clear picture of the changes in the object's state in this process.

e.g: ATM withdraw

Card object state: Checking, Approving, Rejecting

Activity diagram is a fancy flow chart which shows the flow of activity of a process.

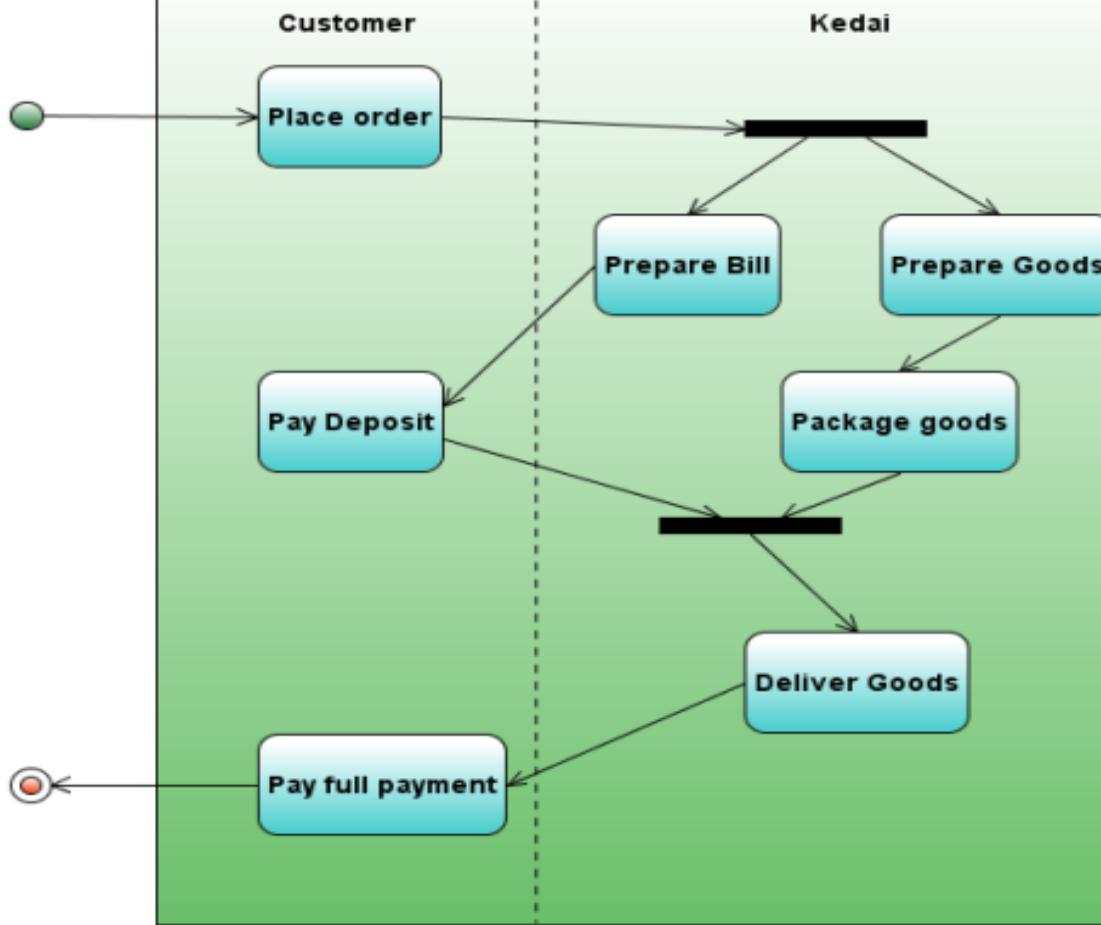
e.g: ATM withdraw

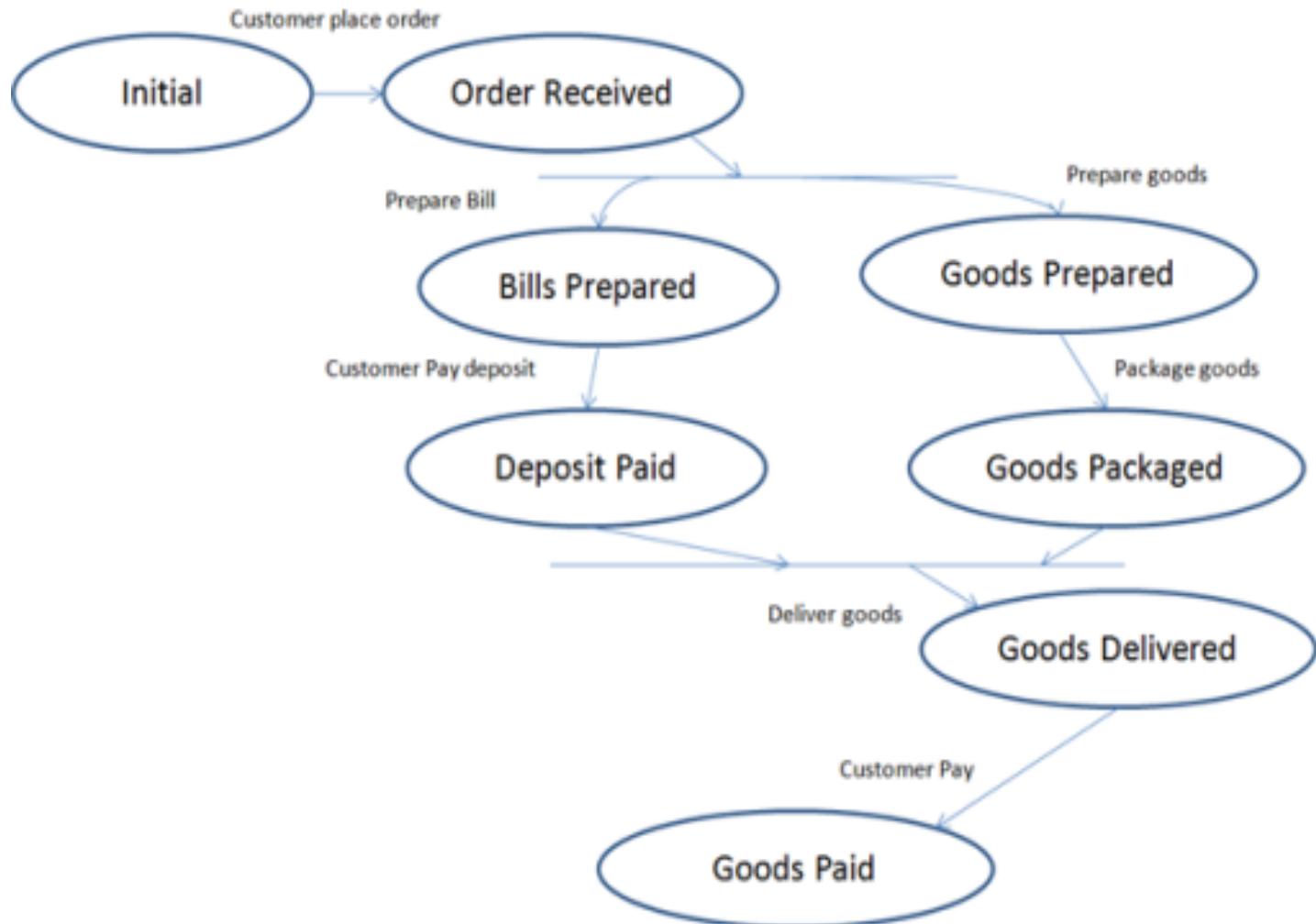
Withdraw activity: Insert Card, Enter PIN, Check balance, with draw money, get card

State chart shows the dynamic behavior of an object.

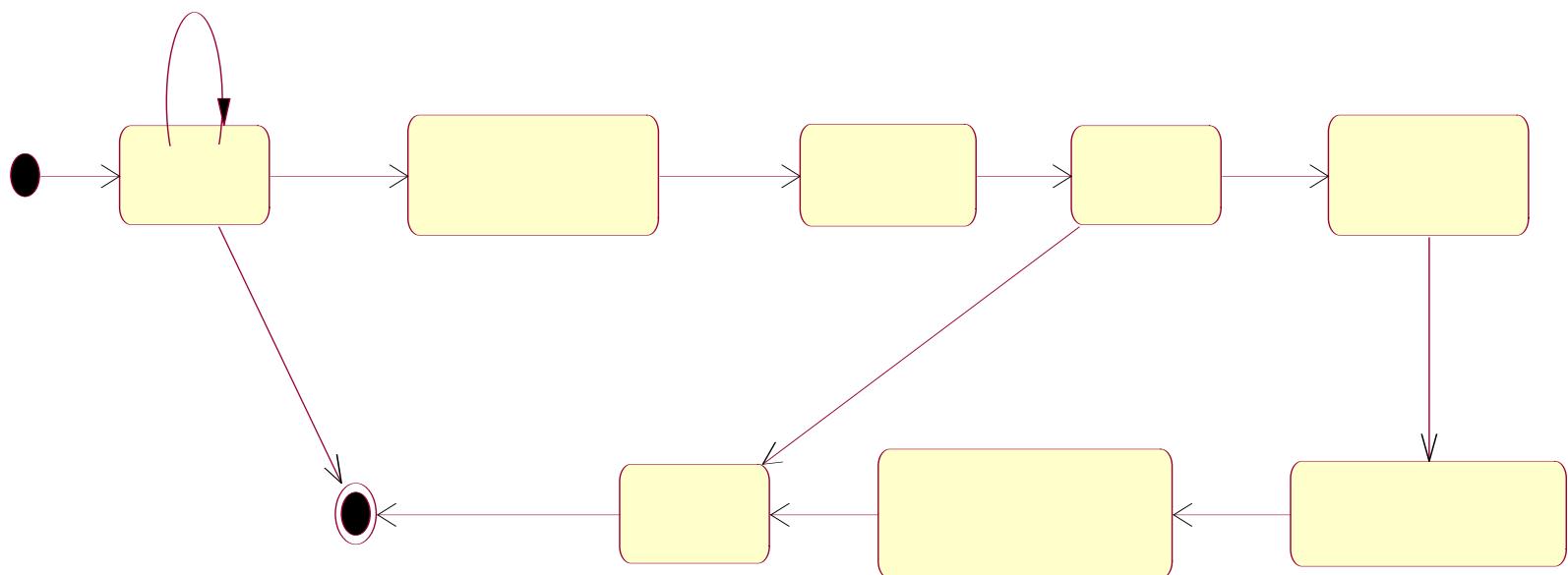
Activity diagram shows the workflow behavior of an operation as set of actions

Slide 6 (arham)

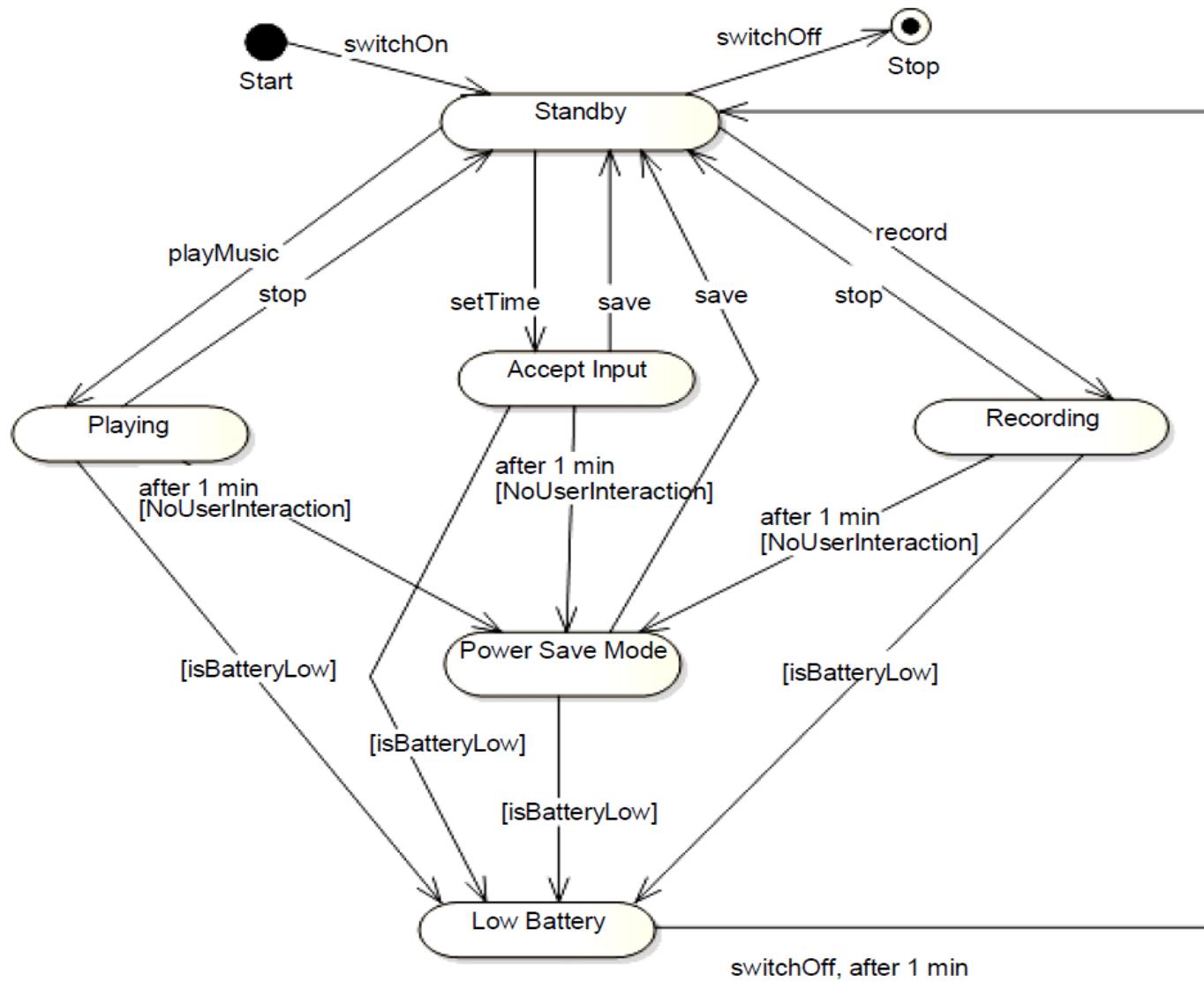




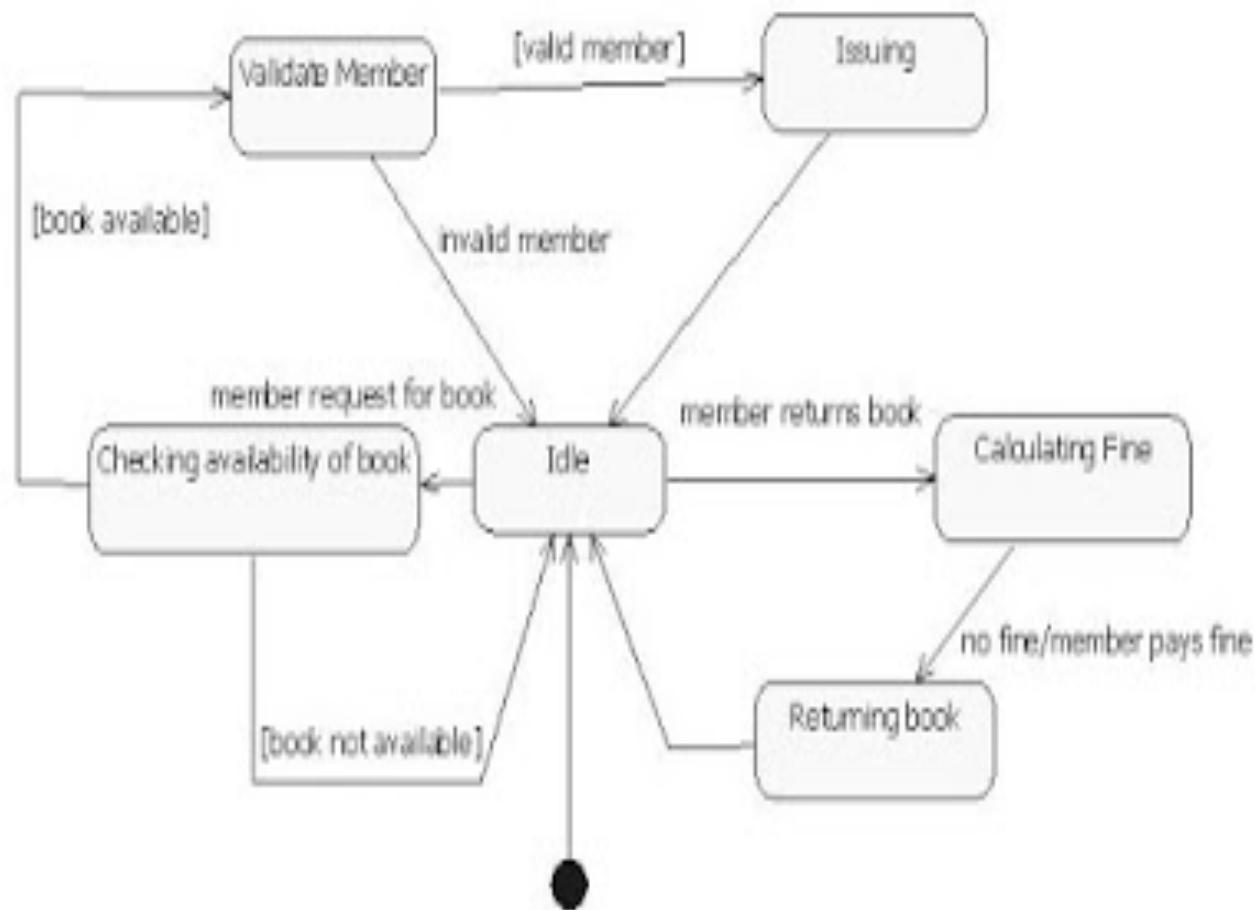
Online Shopping System



Music Player System



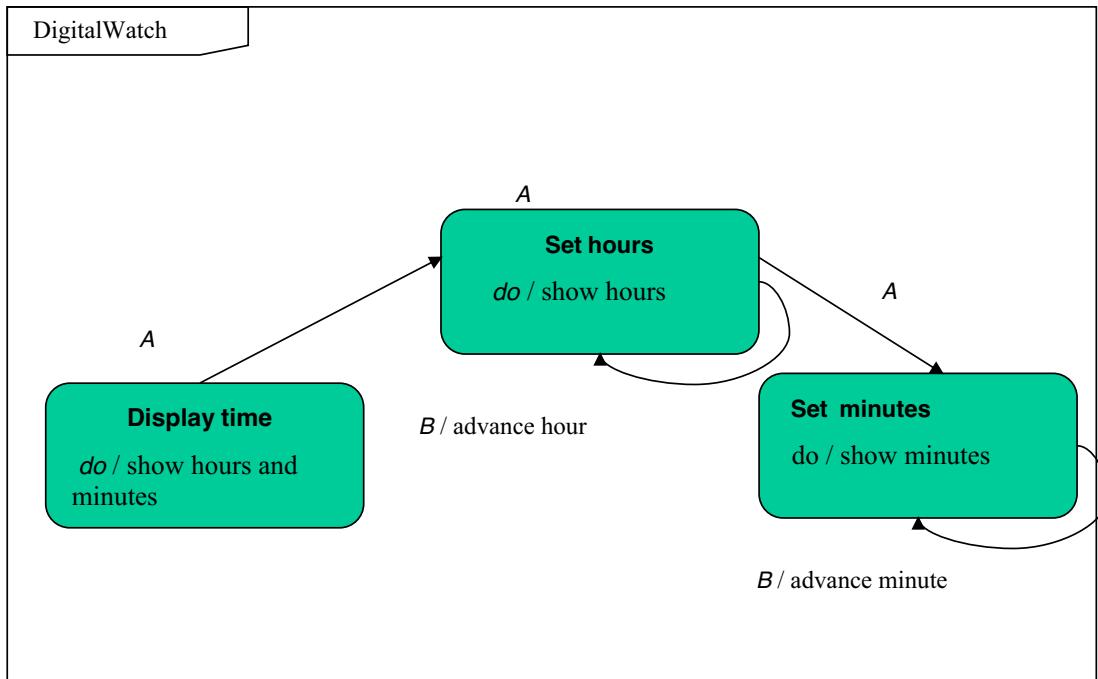
Library Management System



Problem # 1

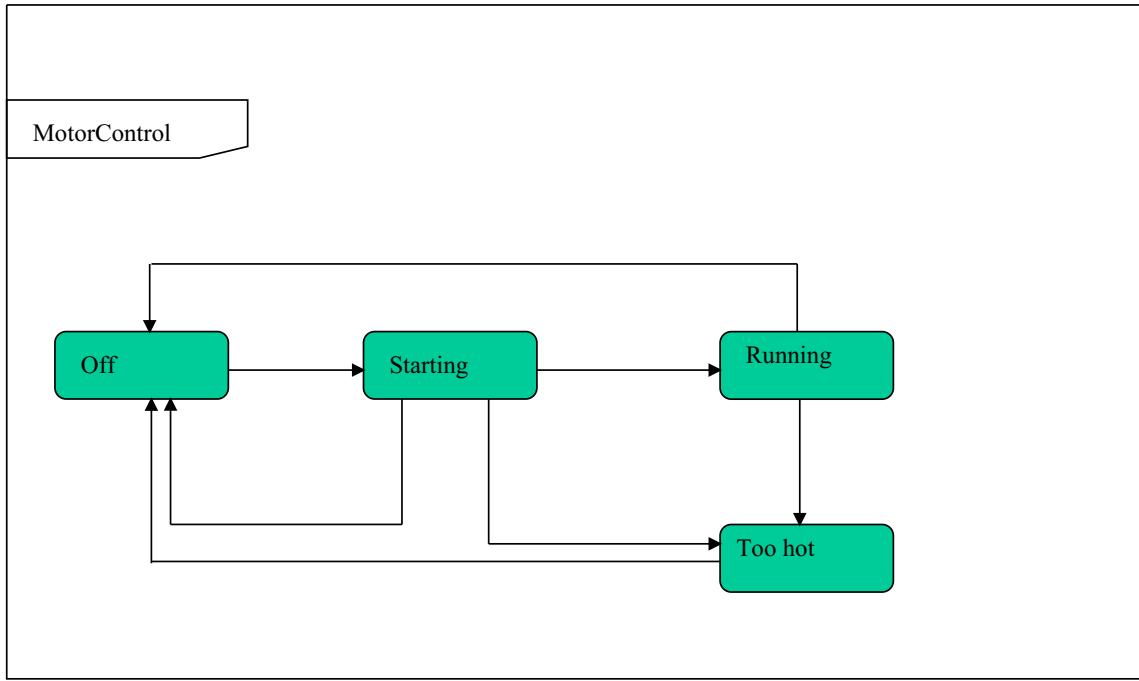
- A simple digital watch has a display and two buttons to set it, the A button and the B button. The watch has two modes of operation, display time and set time.
- In the display time mode, the watch displays hours and minutes, separated by a flashing colon.
- The set time mode has two submodes, set hours and set minutes. The A button selects modes. Each time it is pressed, the mode advances in the sequence: display, set hours, set minutes, display, etc.
- Within the submodes, the B button advances the hours or minutes once each time it is pressed. Buttons must be released before they can generate another event.
- Prepare a state diagram of the watch.

Solution of Problem # 1



Problem # 2

- A separate *appliance control* determines when the motor should be on and continuously asserts on as an input to the motor control when the motor should be running.
- When on is asserted, the motor control should start and run the motor.
- The motor starts by applying power to both the start and the run windings. A sensor, called a starting relay, determines when the motor has started, at which point the start winding is turned off, leaving only the run winding powered. Both winding are shut off when on is not asserted.
- Appliance motors could be damaged by overheating if they are overloaded or fail to start. To protect against thermal damage, the motor control often includes an over-temperature sensor. If the motor becomes too hot, the motor control removes power from both windings and ignores any on assertion until a reset button is pressed and the motor has cooled off.
- **Add the following to the diagram.**
 - *Activities:* apply power to run winding, apply power to start winding.
 - *Events:* motor is overheated, on is asserted, on is no longer asserted, motor is running, reset.
 - *Condition:* motor is not overheated.



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

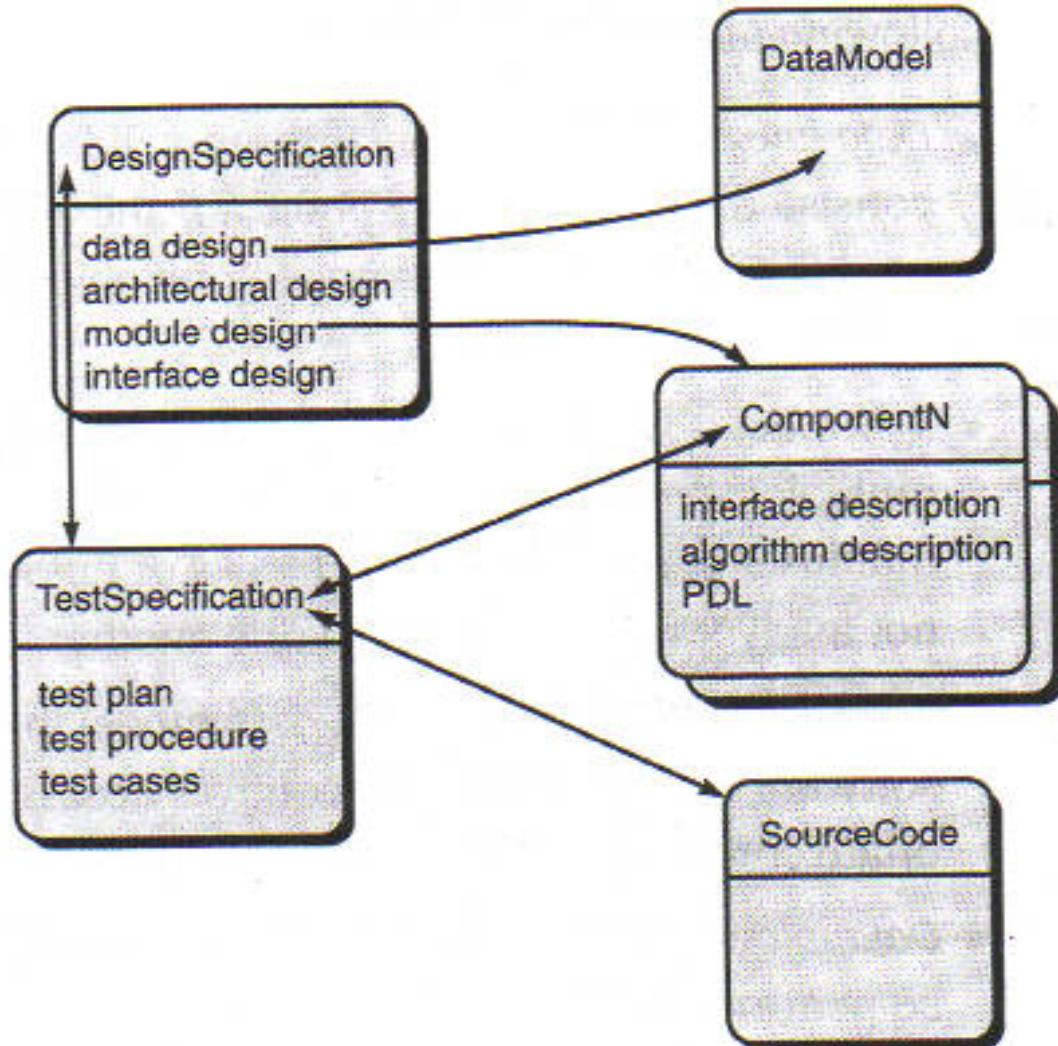
- Errors detected in the software need to be corrected
- New business or market conditions dictate changes in product requirements or business rules
- New customer needs demand modifications of data produced by information systems, functionality delivered by products, or services delivered by a computer-based 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

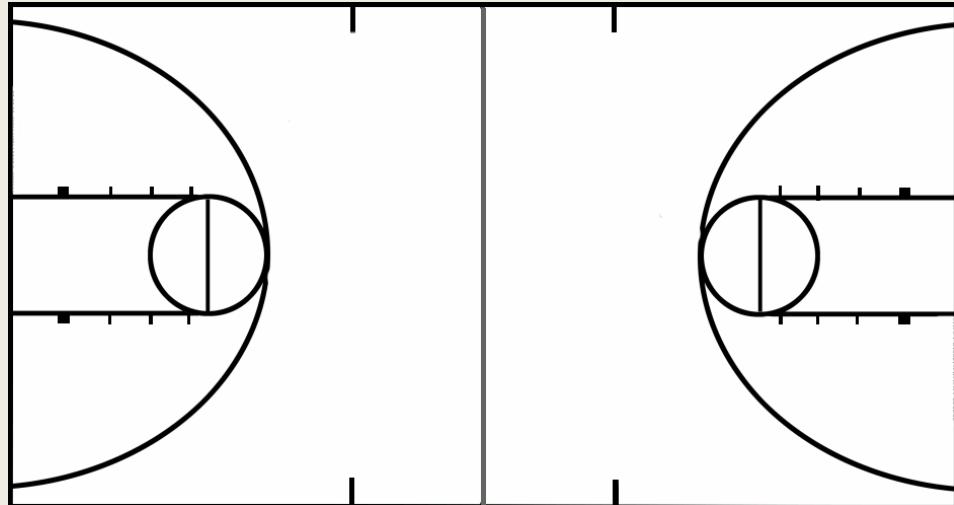
- Configuration elements
 - A set of tools coupled with a file management (e.g., database) system that enables access to and management of each software configuration item
- Process elements
 - A collection of procedures and tasks that define an effective approach to change management for all participants
- 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?

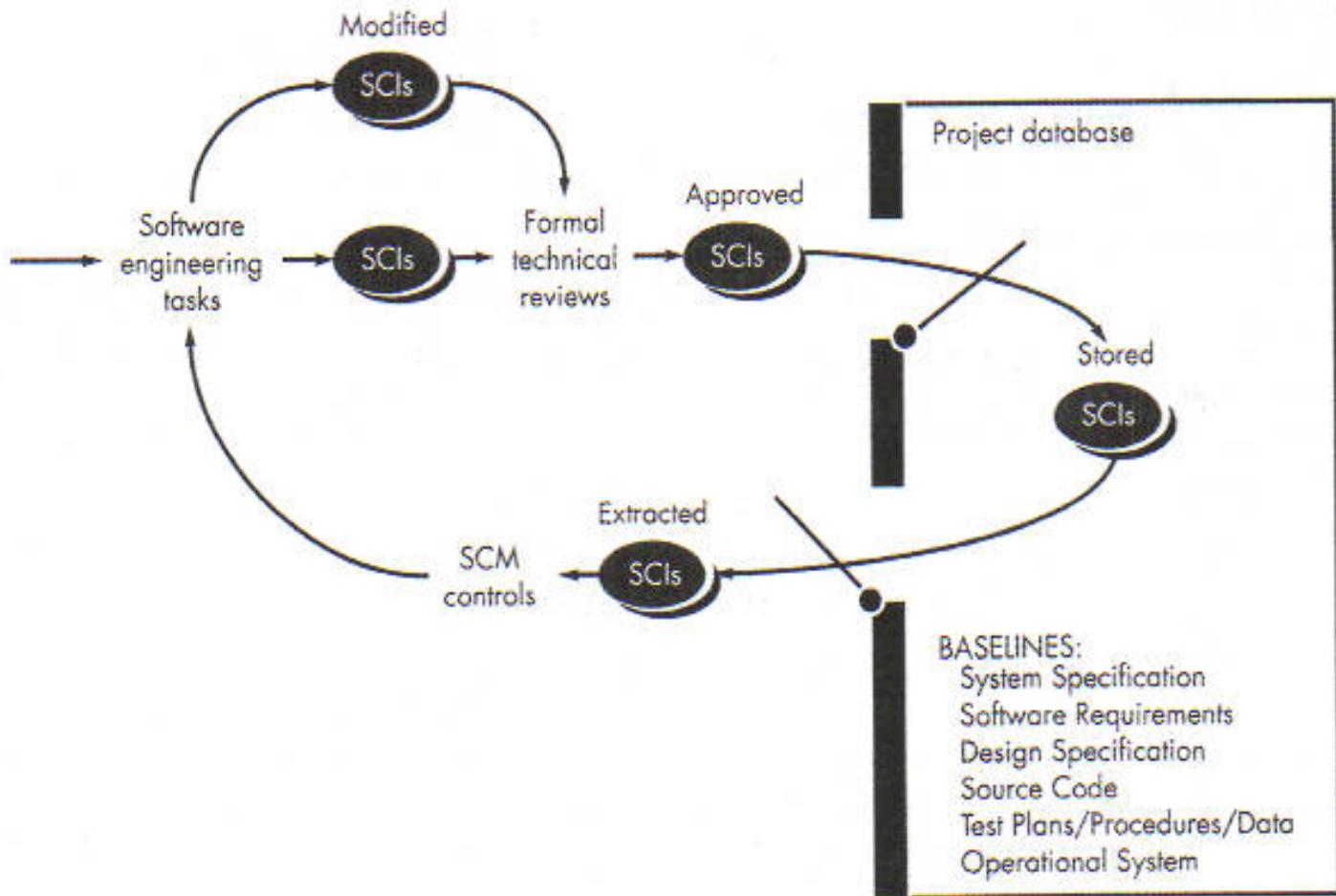


Baseline

- 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

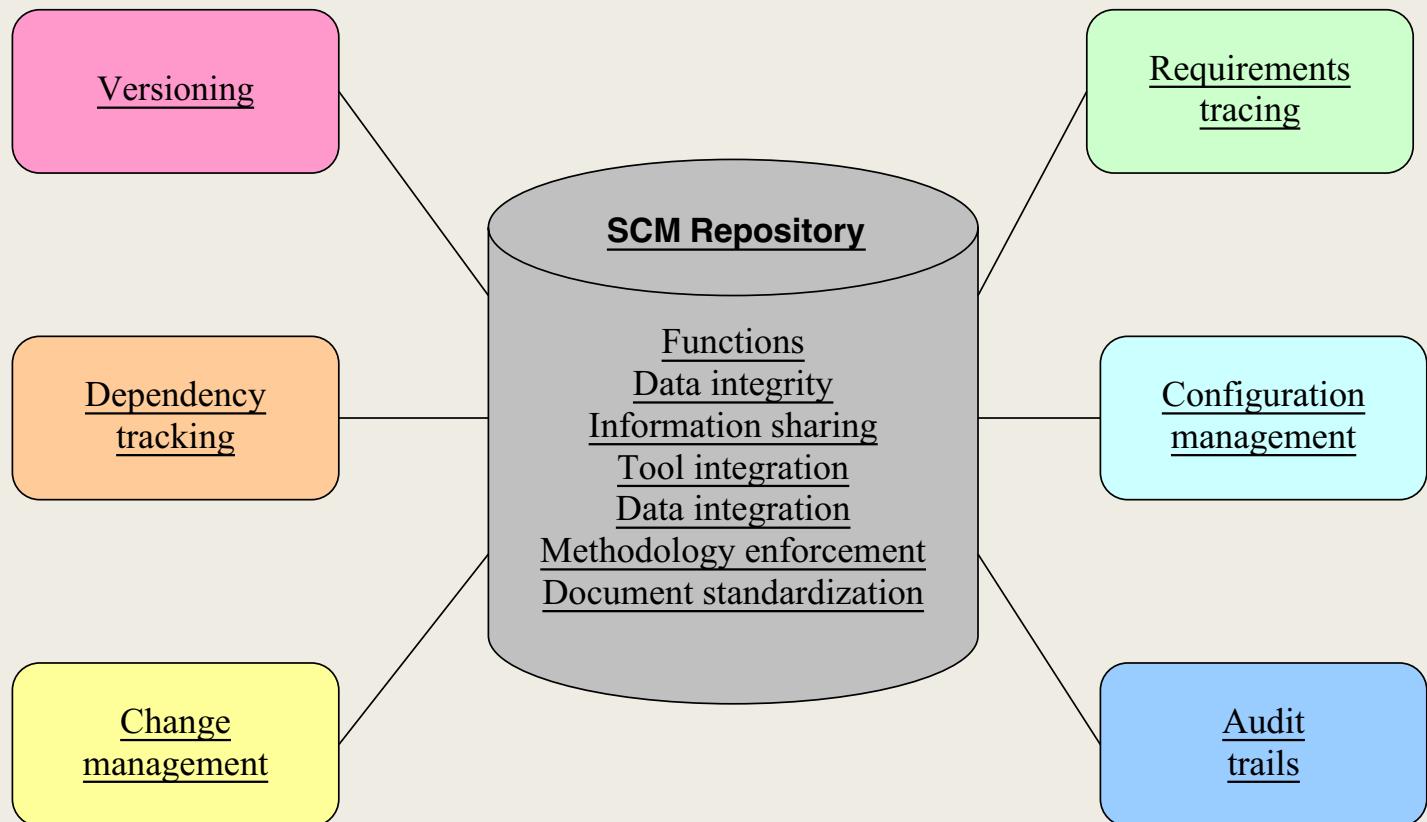
Problems with paper-based repositories (i.e., file cabinet containing folders)

- 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?

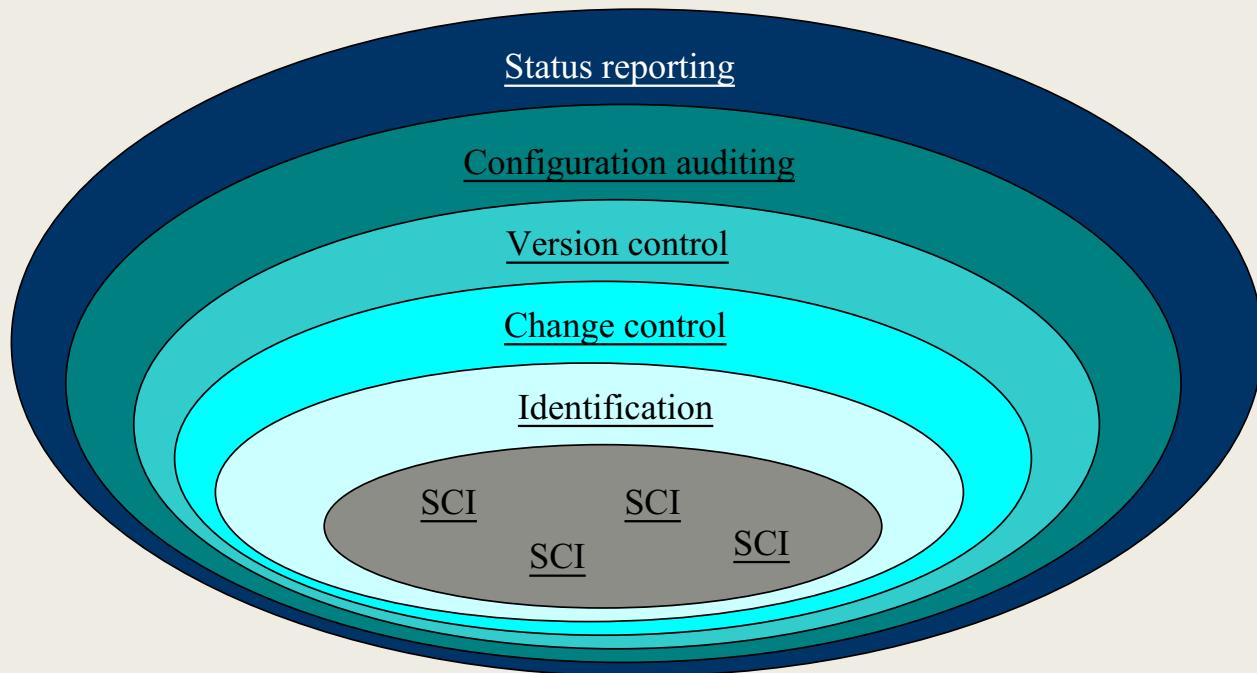
How does an organization control changes before and after software is released to a customer?

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

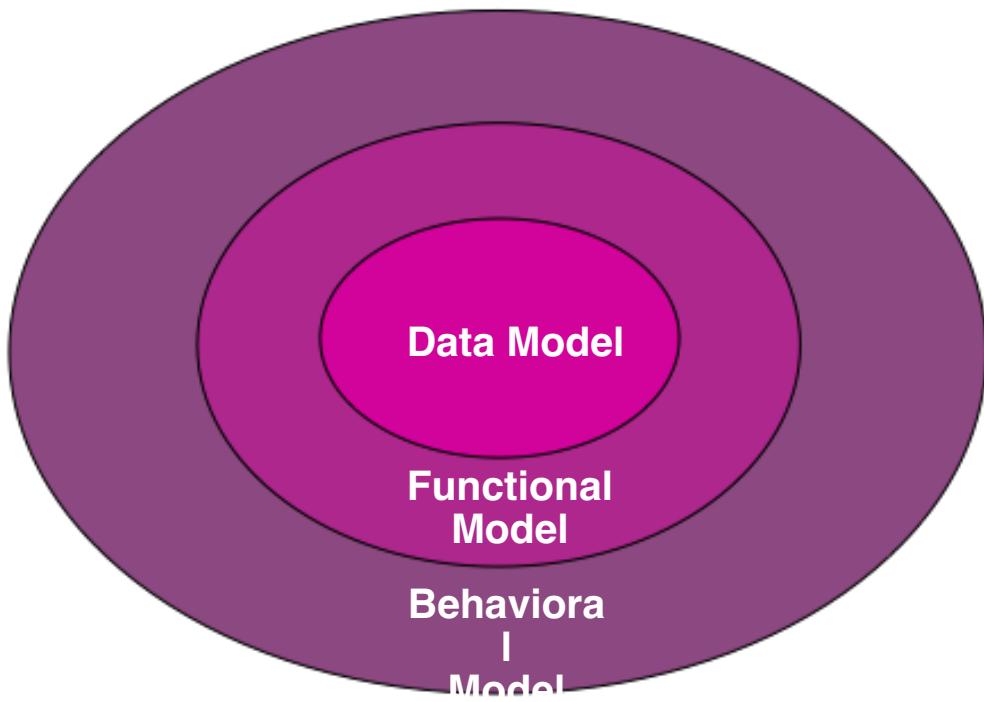
E Overview

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

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

Systems Requirements Specification

I. Introduction

B. Scope

High-level functionality

- I. defined for the system
- II. 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

Software Testing Strategies

Software Testing

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

Introduction

A strategy for software testing integrates the design of software test cases into a well-planned series of steps that result in successful development of the software

The strategy provides a road map that describes the steps to be taken, when, and how much effort, time, and resources will be required

The strategy incorporates test planning, test case design, test execution, and test result collection and evaluation

The strategy provides guidance for the practitioner and a set of milestones for the manager

Because of time pressures, progress must be measurable and problems must surface as early as possible

Strategic Approach

To perform effective testing, conduct effective technical reviews. By doing this, many errors will be eliminated before testing commences.

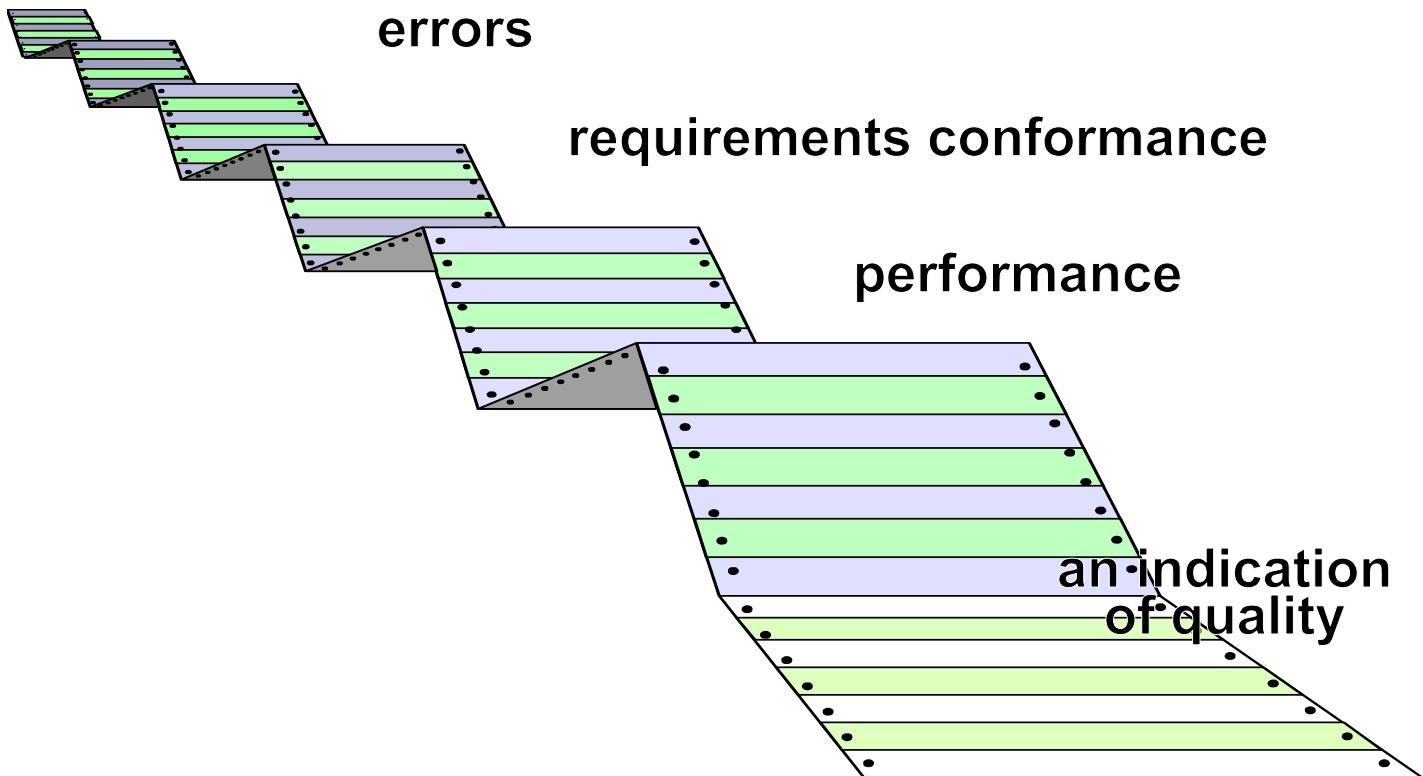
Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.

Different testing techniques are appropriate for different software engineering approaches and at different points in time.

Testing is conducted by the developer of the software and (for large projects) an independent test group.

Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

What Testing Shows



V & V

Software testing is part of a broader group of activities called **verification and validation** that are involved in software quality assurance

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

Validation refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm [Boe81] states this another way:

Verification: "Are we building the product, right?"

Validation: "Are we building the right product?"

Who Tests the Software?



developer

Understands the system
but, will test "gently"
and, is driven by "delivery"



independent tester

Must learn about the system,
but, will attempt to break it
and, is driven by quality

Organizing for Software Testing

Testing should aim at "breaking" the software

Common misconceptions

The developer of software should do no testing at all

The software should be given to a secret team of testers who will test it unmercifully

The testers get involved with the project only when the testing steps are about to begin

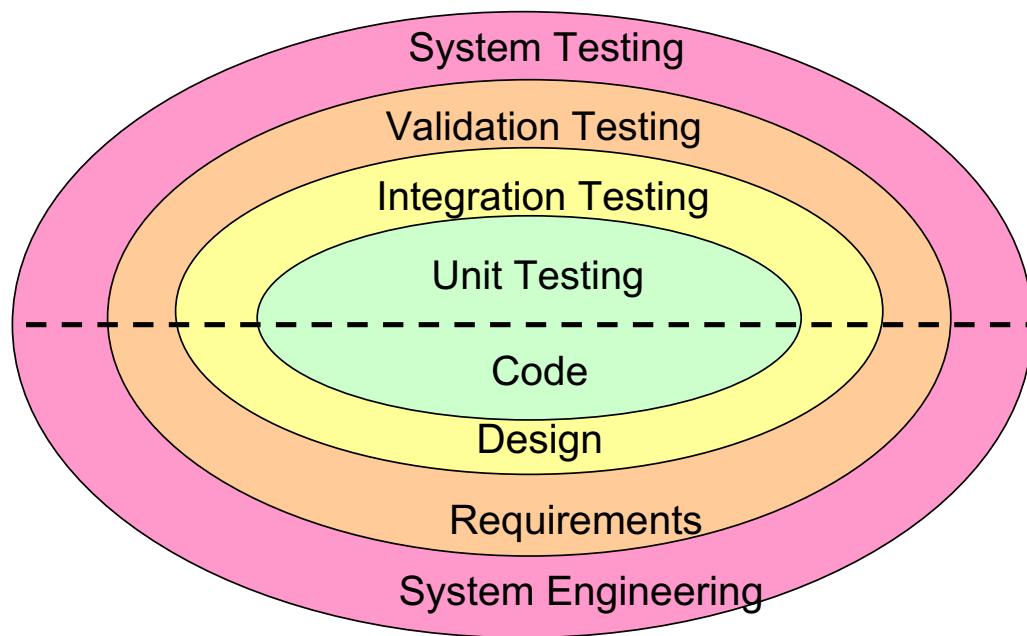
Reality: Independent test group

Removes the inherent problems associated with letting the builder test the software that has been built

Removes the conflict of interest that may otherwise be present

Works closely with the software developer during analysis and design to ensure that thorough testing occurs

Testing Strategy



Criteria for Completion of Testing

When are we finished testing? : No definite answer for this question.

But few responses:

You are never done testing; the burden simply shifts from developer/tester to end user.

You are done testing when you run out of time or money.

Rigorous criteria: *Cleanroom software engineering approach.*

Statistical testing method, executes the series of tests derived from a statistical sample of all possible program executions by all users

Testing Strategy

We begin by ‘testing-in-the-small’ and move toward
‘testing-in-the-large’

For conventional software

The module (component) is our initial focus

Integration of modules follows

For OO software

our focus when “testing in the small” changes from an individual module (the conventional view) to an OO class that encompasses attributes and operations and implies communication and collaboration

Strategic Issues

- Specify product requirements in a quantifiable manner long before testing commences.
- State testing objectives explicitly.
- Understand the users of the software and develop a profile for each user category.
- Develop a testing plan that emphasizes “rapid cycle testing.”
- Build “robust” software that is designed to test itself
- Use effective technical reviews as a filter prior to testing
- Conduct technical reviews to assess the test strategy and test cases themselves.
- Develop a continuous improvement approach for the testing process.

Test Strategies for Conventional Software

Unit Testing

Focuses testing on the function or software module

Concentrates on the internal processing logic and data structures

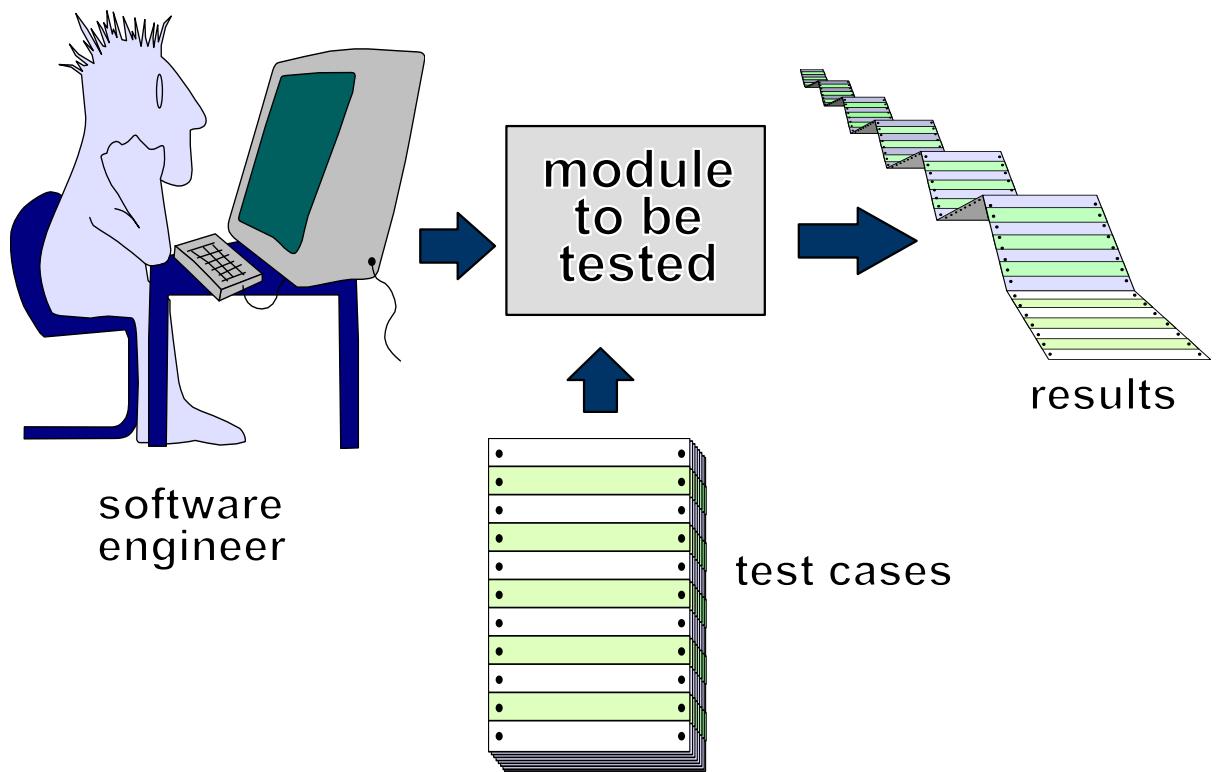
Is simplified when a module is designed with high cohesion

- Reduces the number of test cases

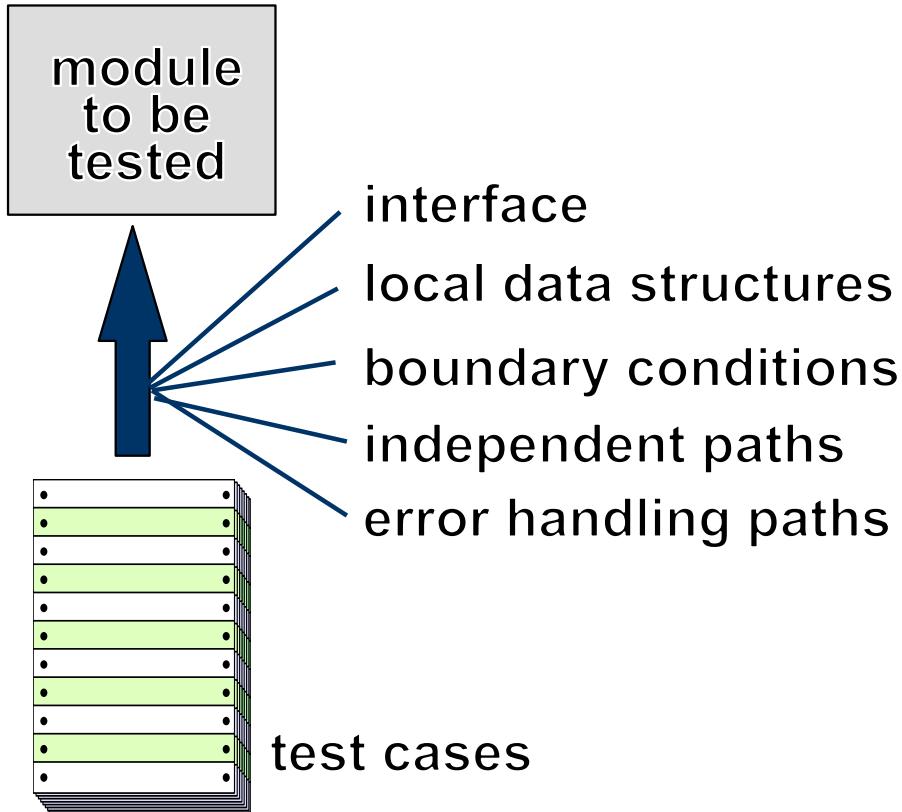
- Allows errors to be more easily predicted and uncovered

Concentrates on critical modules and those with **high cyclomatic complexity** when testing resources are limited

Unit Testing



Unit Testing



Targets for Unit Test Cases

Module interface

Ensure that information flows properly into and out of the module

Local data structures

Ensure that data stored temporarily maintains its integrity during all steps in an algorithm execution

Boundary conditions

Ensure that the module operates properly at boundary values established to limit or restrict processing

Independent paths (basis paths)

Paths are exercised to ensure that all statements in a module have been executed at least once

Error handling paths

Ensure that the algorithms respond correctly to specific error conditions

Common Computational Errors in Execution Paths

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

Other Errors to Uncover

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

Problems to uncover in Error Handling

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

Drivers and Stubs for Unit Testing

Driver

A simple main program that accepts test case data, passes such data to the component being tested, and prints the returned results

Stubs

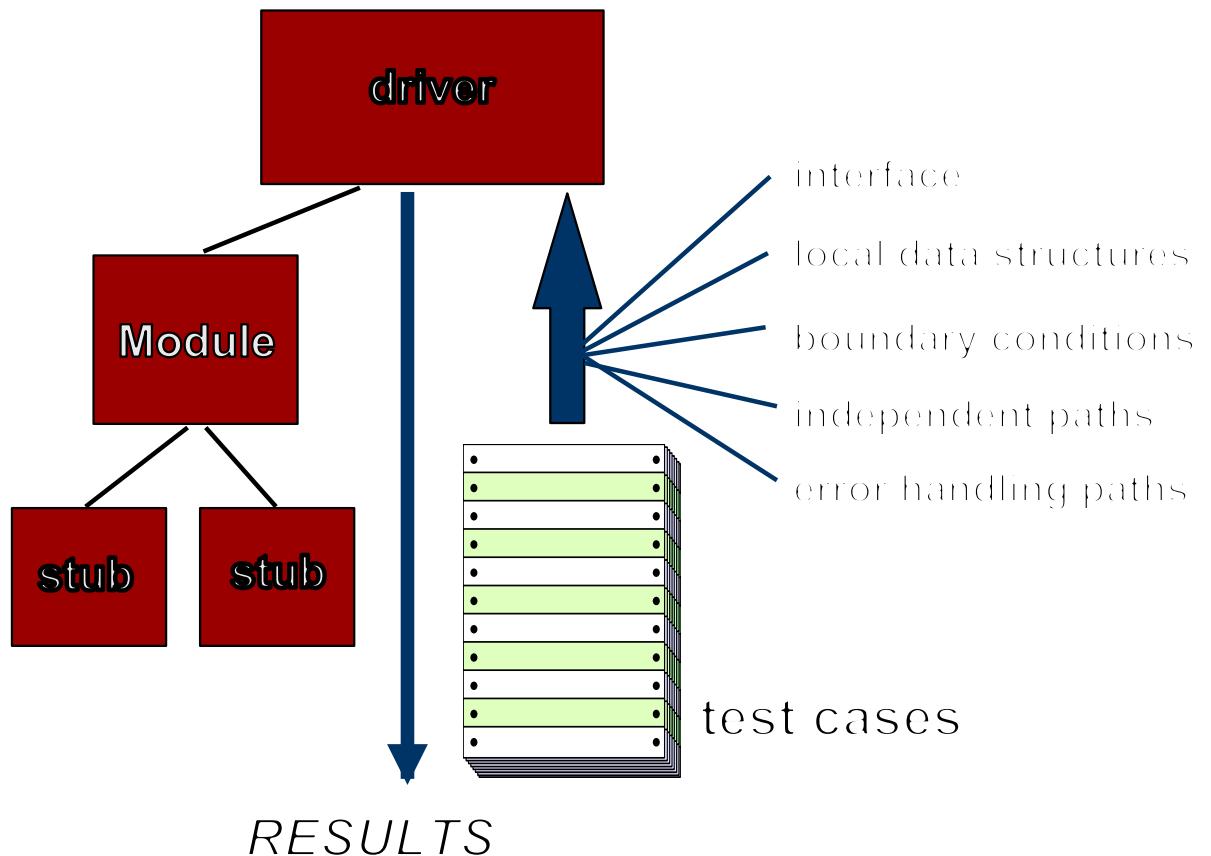
Serve to replace modules that are subordinate to (called by) the component to be tested

It uses the module's exact interface, may do minimal data manipulation, provides verification of entry, and returns control to the module undergoing testing

Drivers and stubs both represent overhead

Both must be written but don't constitute part of the installed software product

Unit Test Environment



Integration Testing

Defined as a systematic technique for constructing the software architecture

At the same time integration is occurring, conduct tests to uncover errors associated with interfaces

Objective is to take unit tested modules and build a program structure based on the prescribed design

Two Approaches

Non-incremental Integration Testing

Incremental Integration Testing

Non-incremental Integration Testing

Commonly called the “Big Bang” approach

All components are combined in advance

The entire program is tested as a whole

Chaos results

Many seemingly-unrelated errors are encountered

Correction is difficult because isolation of causes is complicated

Once a set of errors are corrected, more errors occur, and testing appears to enter an endless loop

Incremental Integration Testing

Three kinds

Top-down integration

Bottom-up integration

Sandwich integration

The program is constructed and tested in small increments

Errors are easier to isolate and correct

Interfaces are more likely to be tested completely

A systematic test approach is applied

Top-down Integration

Modules are integrated by moving downward through the control hierarchy, beginning with the main module

Subordinate modules are incorporated in either a depth-first or breadth-first fashion

- DF: All modules on a major control path are integrated

- BF: All modules directly subordinate at each level are integrated

Advantages

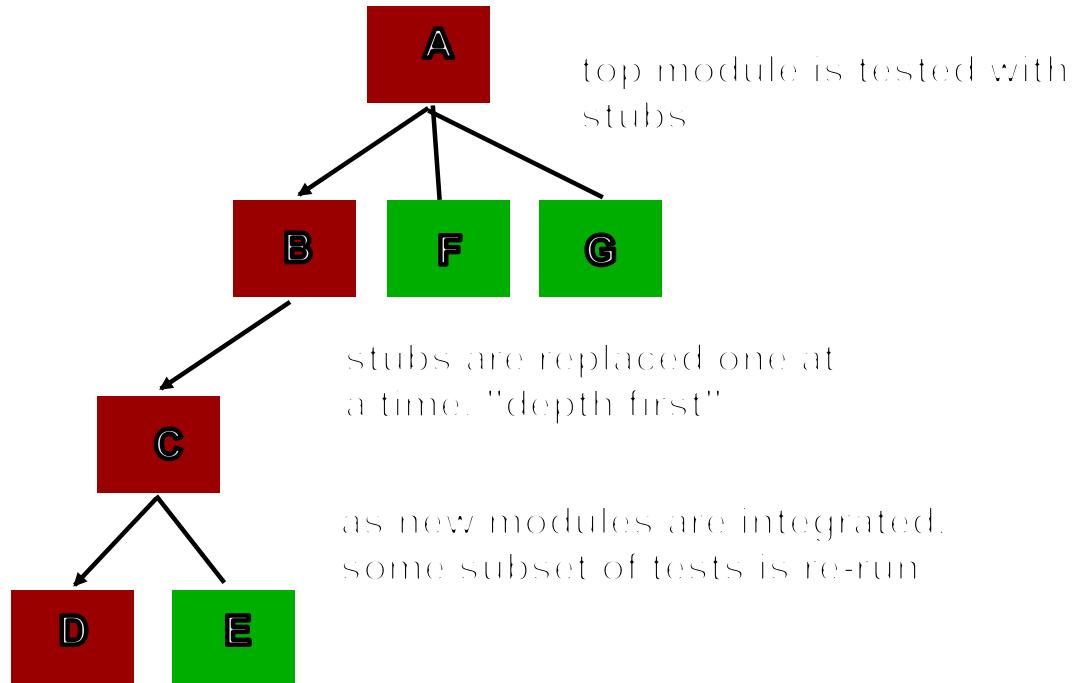
This approach verifies major control or decision points early in the test process

Disadvantages

Stubs need to be created to substitute for modules that have not been built or tested yet; this code is later discarded

Because stubs are used to replace lower level modules, no significant data flow can occur until much later in the integration/testing process

Top Down Integration



Bottom-up Integration

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

Advantages

- This approach verifies low-level data processing early in the testing process

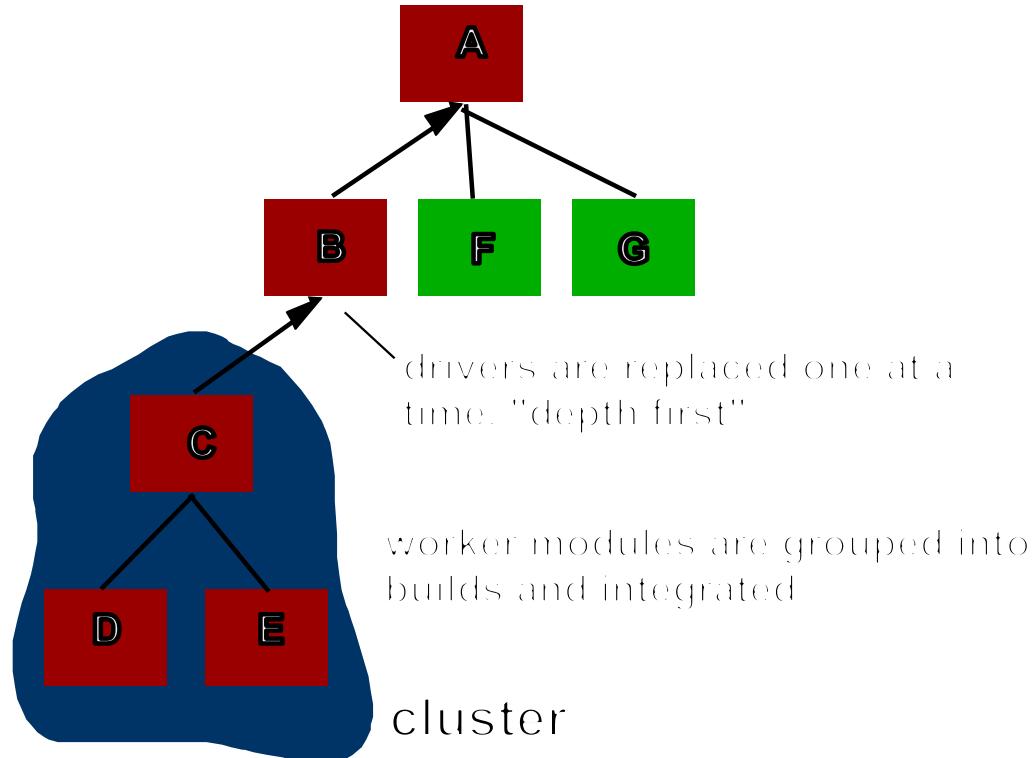
- Need for stubs is eliminated

Disadvantages

- Driver modules need to be built to test the lower-level modules; this code is later discarded or expanded into a full-featured version

- Drivers inherently do not contain the complete algorithms that will eventually use the services of the lower-level modules; consequently, testing may be incomplete or more testing may be needed later when the upper level modules are available

Bottom-Up Integration



Sandwich Integration

Consists of a combination of both top-down and bottom-up integration

Occurs both at the highest-level modules and also at the lowest level modules

Proceeds using functional groups of modules, with each group completed before the next

High and low-level modules are grouped based on the control and data processing they provide for a specific program feature

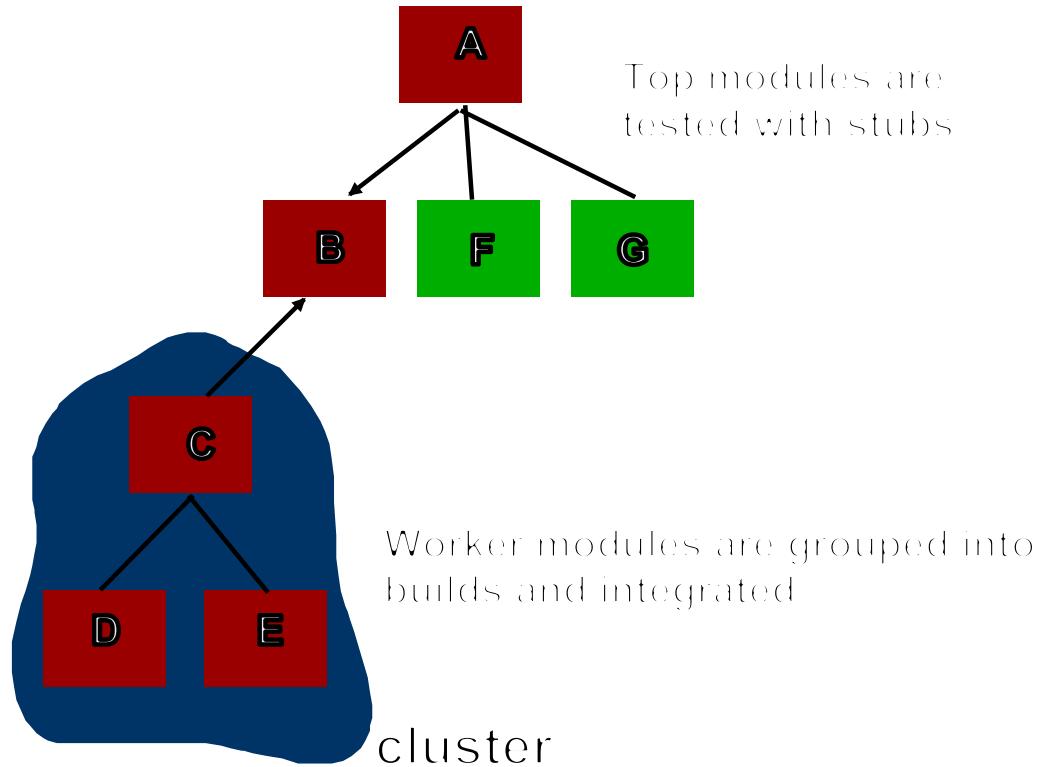
Integration within the group progresses in alternating steps between the high and low level modules of the group

When integration for a certain functional group is complete, integration and testing moves onto the next group

Reaps the advantages of both types of integration while minimizing the need for drivers and stubs

Requires a disciplined approach so that integration doesn't tend towards the "big bang" scenario

Sandwich Testing



Regression Testing

Each new addition or change to baselined software may cause problems with functions that previously worked flawlessly

Regression testing re-executes a small subset of tests that have already been conducted

- Ensures that changes have not propagated unintended side effects
- Helps to ensure that changes do not introduce unintended behavior or additional errors

- May be done manually or through the use of automated capture/playback tools

Regression test suite contains three different classes of test cases

- A representative sample of tests that will exercise all software functions

- Additional tests that focus on software functions that are likely to be affected by the change

- Tests that focus on the actual software components that have been changed

Smoke Testing

A common approach for creating “daily builds” for product software

Smoke testing steps:

Software components that have been translated into code are integrated into a “build.”

- A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.

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

- The intent should be to uncover “show stopper” errors that have the highest likelihood of throwing the software project behind schedule.

The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.

- The integration approach may be top down or bottom up.

Benefits of Smoke Testing

Integration risk is minimized

Daily testing uncovers incompatibilities and show-stoppers early in the testing process, thereby reducing schedule impact

The quality of the end-product is improved

Smoke testing is likely to uncover both functional errors and architectural and component-level design errors

Error diagnosis and correction are simplified

Smoke testing will probably uncover errors in the newest components that were integrated

Progress is easier to assess

As integration testing progresses, more software has been integrated and more has been demonstrated to work

Managers get a good indication that progress is being made

Test Strategies for Object-Oriented Software

Test Strategies for Object-Oriented Software

With object-oriented software, you can no longer test a single operation in isolation (conventional thinking)

Traditional top-down or bottom-up integration testing has little meaning

Class testing for object-oriented software is the equivalent of unit testing for conventional software

Focuses on operations encapsulated by the class and the state behavior of the class

Drivers can be used

To test operations at the lowest level and for testing whole groups of classes

To replace the user interface so that tests of system functionality can be conducted prior to implementation of the actual interface

Stubs can be used

In situations in which collaboration between classes is required but one or more of the collaborating classes has not yet been fully implemented

Test Strategies for Object-Oriented

Two different object-oriented testing strategies

Software (continued)

Thread-based testing

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

Use-based testing

- First tests the independent classes that use very few, if any, server classes
- Then the next layer of classes, called dependent classes, are integrated
- This sequence of testing layer of dependent classes continues until the entire system is constructed

Validation Testing

Background

Validation testing follows integration testing

The distinction between conventional and object-oriented software disappears

Focuses on user-visible actions and user-recognizable output from the system

Demonstrates conformity with requirements

Designed to ensure that

- All functional requirements are satisfied

- All behavioral characteristics are achieved

- All performance requirements are attained

- Documentation is correct

- Usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability)

After each validation test

The function or performance characteristic conforms to specification and is accepted

A deviation from specification is uncovered and a deficiency list is created

A configuration review or audit ensures that all elements of the software configuration have been properly developed, cataloged, and have the necessary detail for entering the support phase of the software life cycle

Alpha and Beta Testing

Alpha testing

Conducted at the developer's site by end users

Software is used in a natural setting with developers watching intently

Testing is conducted in a controlled environment

Beta testing

Conducted at end-user sites

Developer is generally not present

It serves as a live application of the software in an environment that cannot be controlled by the developer

The end-user records all problems that are encountered and reports these to the developers at regular intervals

After beta testing is complete, software engineers make software modifications and prepare for release of the software product to the entire customer base

System Testing

Different Types

Recovery testing

Tests for recovery from system faults

Forces the software to fail in a variety of ways and verifies that recovery is properly performed

Tests reinitialization, checkpointing mechanisms, data recovery, and restart for correctness

Security testing

Verifies that protection mechanisms built into a system will, in fact, protect it from improper access

Stress testing

Executes a system in a manner that demands resources in abnormal quantity, frequency, or volume

Performance testing

Tests the run-time performance of software within the context of an integrated system

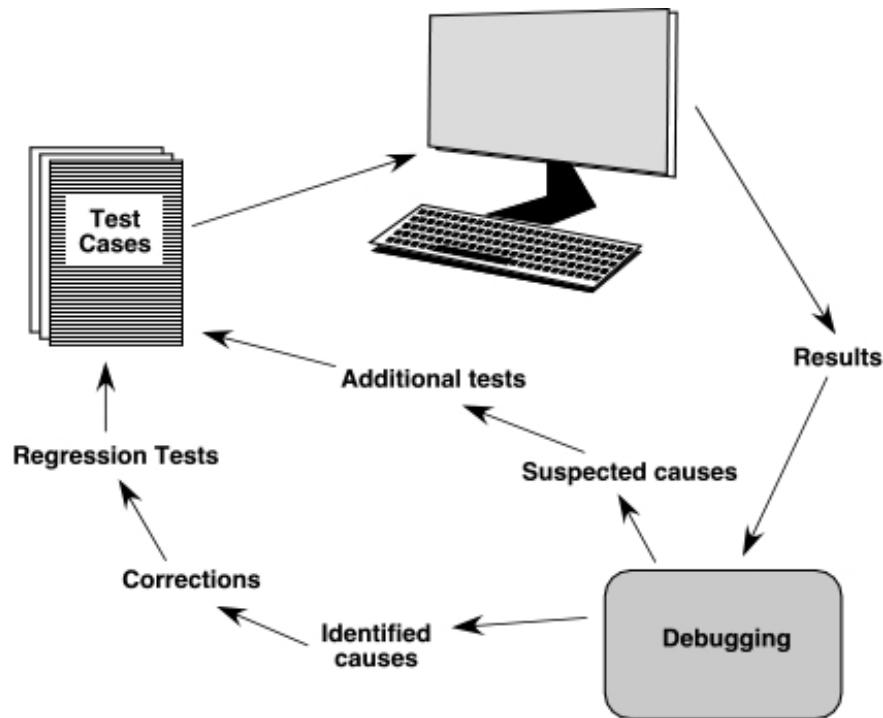
Often coupled with stress testing and usually requires both hardware and software instrumentation

Can uncover situations that lead to degradation and possible system failure

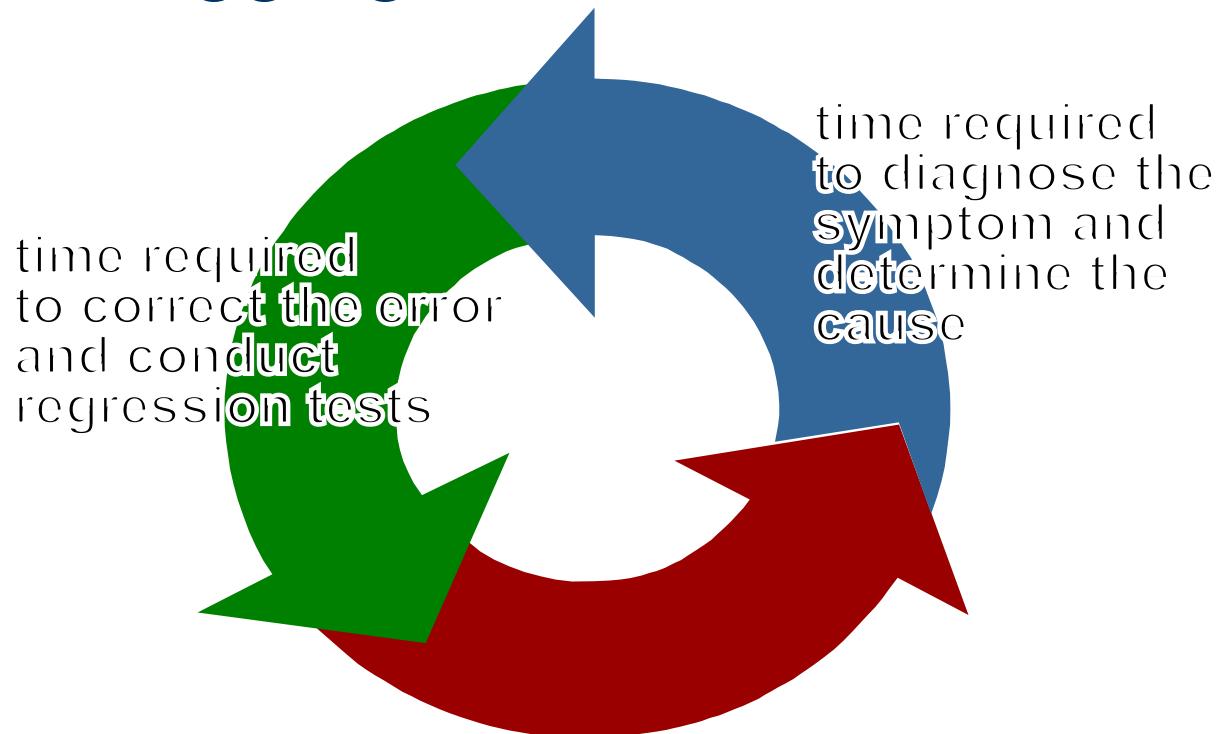
Debugging: A Diagnostic Process



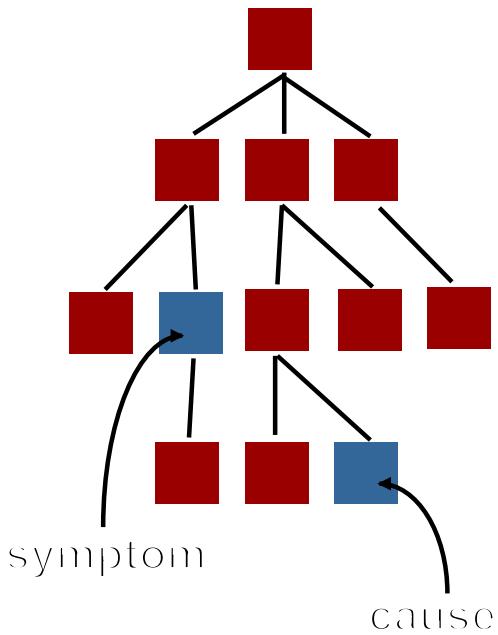
The Debugging Process



Debugging Effort

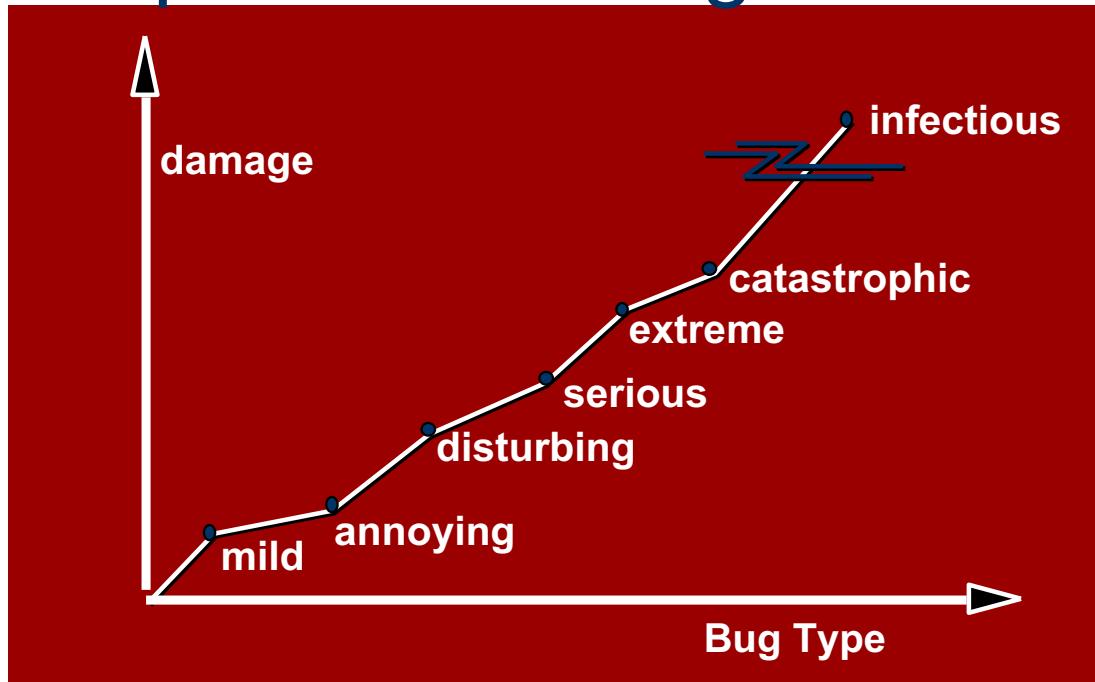


Symptoms & Causes



- ❑ symptom and cause may be geographically separated
- ❑ symptom may disappear when another problem is fixed
- ❑ cause may be due to a combination of non-errors
- ❑ cause may be due to a system or compiler error
- ❑ cause may be due to assumptions that everyone believes
- ❑ symptom may be intermittent

Consequences of Bugs



Bug Categories: function-related bugs,
system-related bugs, data bugs, coding bugs,
design bugs, documentation bugs, standards
violations, etc.

Debugging Techniques

Brute Force: Let computer find the error

Backtracking: Source code is traced backward until the cause or error will found

Cause Elimination: A cause hypothesis

Induction or Deduction

Data is used to prove or disprove the hypothesis

Automated Debugging

Different tools

Integrated Development Environments (IDEs)

Debugging compilers

Dynamic Debugging aids

Automated test case generators

Cross reference mapping tools

Correcting the Error

Is the cause of the bug reproduced in another part of the program?

In many situations, a program defect is caused by an erroneous pattern of logic that may be reproduced elsewhere.

What "next bug" might be introduced by the fix I'm about to make?

Before the correction is made, the source code (or, better, the design) should be evaluated to assess coupling of logic and data structures.

What could we have done to prevent this bug in the first place?

This question is the first step toward establishing a statistical software quality assurance approach. If you correct the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs.

Final Thoughts

Think -- before you act to correct

Use tools to gain additional insight

If you're at an impasse, get help from someone else

Once you correct the bug, use regression testing to uncover any side effects

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?

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) 3
 - 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 plan⁷ 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?"

RISK ITEM CHECKLIST

- Used as one way to identify risks
- Focuses on known and predictable risks in specific subcategories (see next slide)
- Can be organized in several ways
 - A list of characteristics relevant to each risk subcategory
 - Questionnaire that leads to an estimate on the impact of each risk
 - A list containing a set of risk component and drivers and their probability of occurrence

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

The background of the slide features a dark purple gradient with several light gray, semi-transparent circular arrows of varying sizes. Some of these circles contain small white arrows pointing clockwise or counter-clockwise. Interspersed among the circles are numerical values: 50, 160, 170, 180, 190, 200, 210, 220, 230, 240, 250, and 260, which appear to be part of a larger, continuous sequence.

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

<u>Risk Summary</u>	<u>Risk Category</u>	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

Risks	Category	Probability	Impact	RMM
<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>Components developed separately cannot be integrated easily, requiring redesign</u>	<u>DE</u>	<u>25%</u>	<u>3</u>	-
<u>Development of the wrong software functions requires redesign and implementation</u>	<u>DE</u>	<u>25%</u>	<u>3</u>	-
<u>Development of extra software functions that are not needed</u>	<u>DE</u>	<u>20%</u>	<u>3</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>Operation in unfamiliar software environment causes unforeseen problems</u>	<u>EV</u>	<u>25%</u>	<u>4</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>	-

ASSESSING RISK IMPACT

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



The background features a dark purple gradient with several light gray circular arrows of varying sizes. Some arrows have small downward-pointing chevrons, while others have upward-pointing chevrons. Interspersed among the arrows are numerical values: 50, 160, 170, 180, 190, 200, 210, 220, 230, 240, 250, and 260. These numbers are positioned near the center and towards the bottom right of the slide.

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 project²⁸

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	

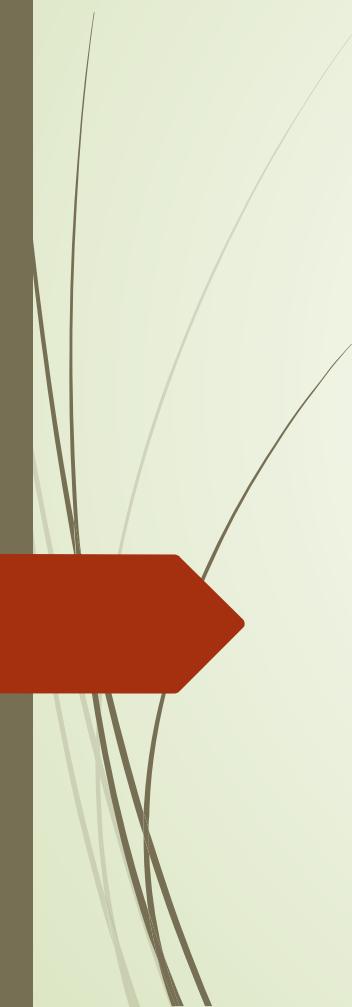
SEVEN PRINCIPLES OF RISK MANAGEMENT

- **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





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



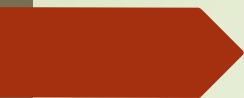
Why we need to measure?

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.

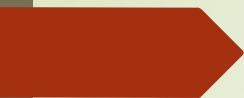
Process metrics provide a set of process indicators that lead to long-term software process improvement

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

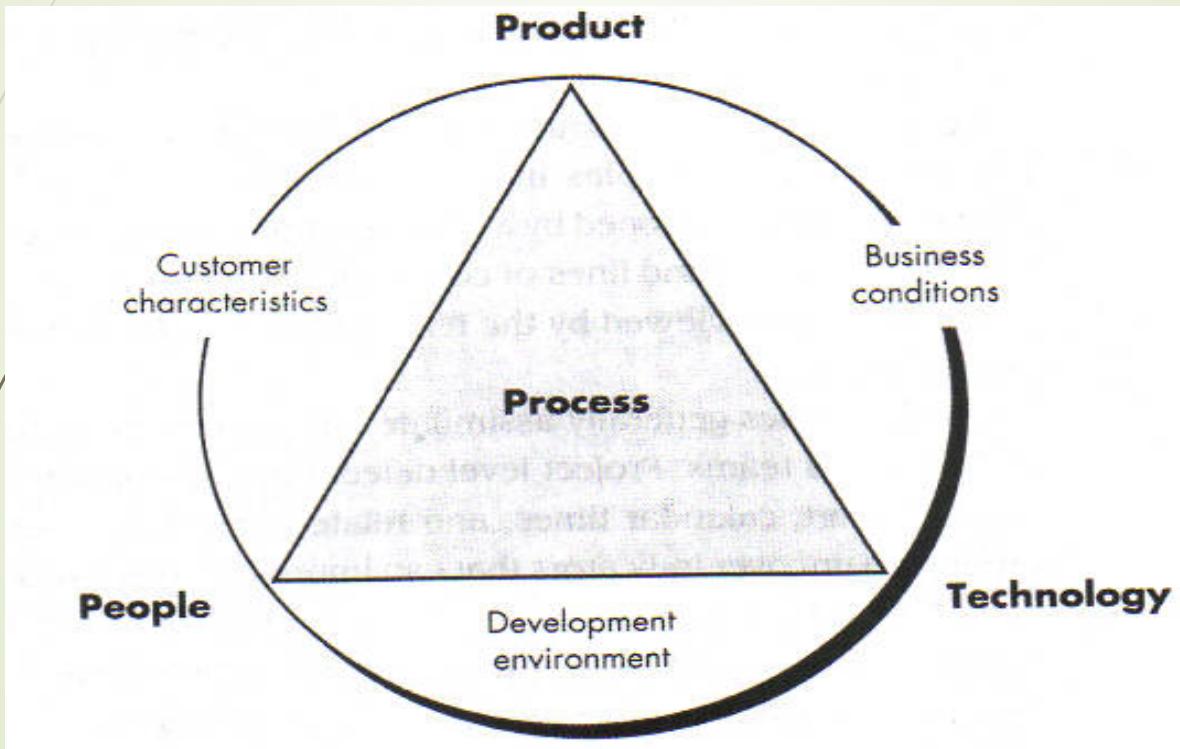
Used by project manager and a software team to adapt project workflow and technical activities.

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

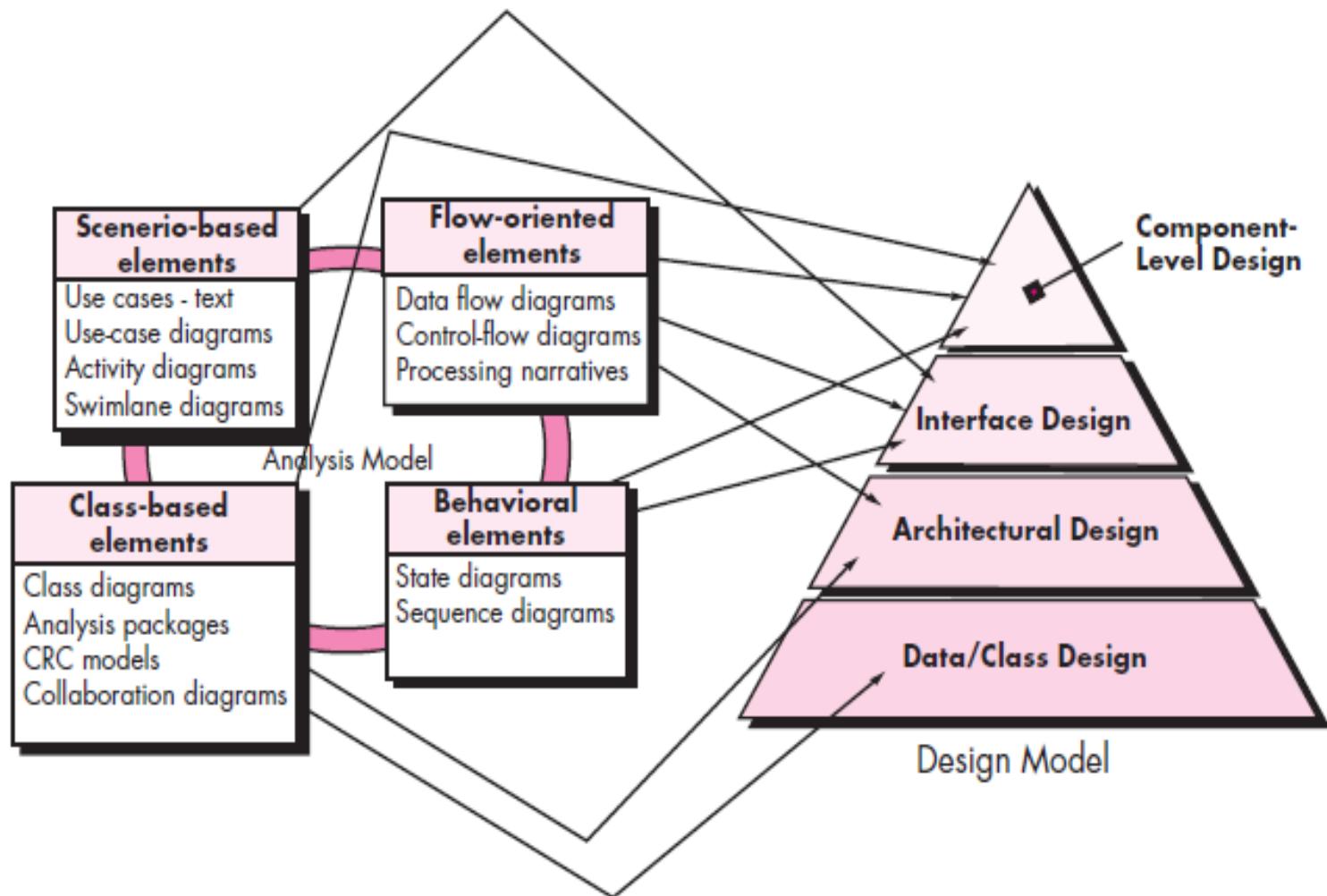
DRE encourages a software team to institute techniques for finding as many errors as possible before delivery

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

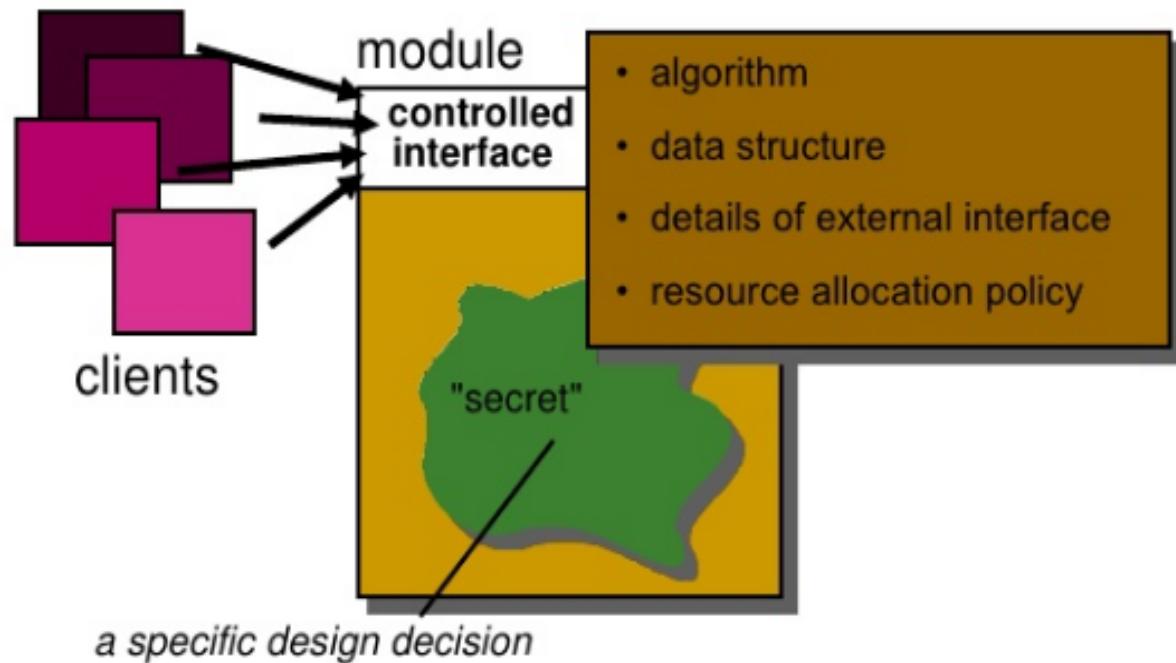
Complexity of the overall problem can be reduced if the design method provides a systematic mechanism to decompose a problem into sub problems

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

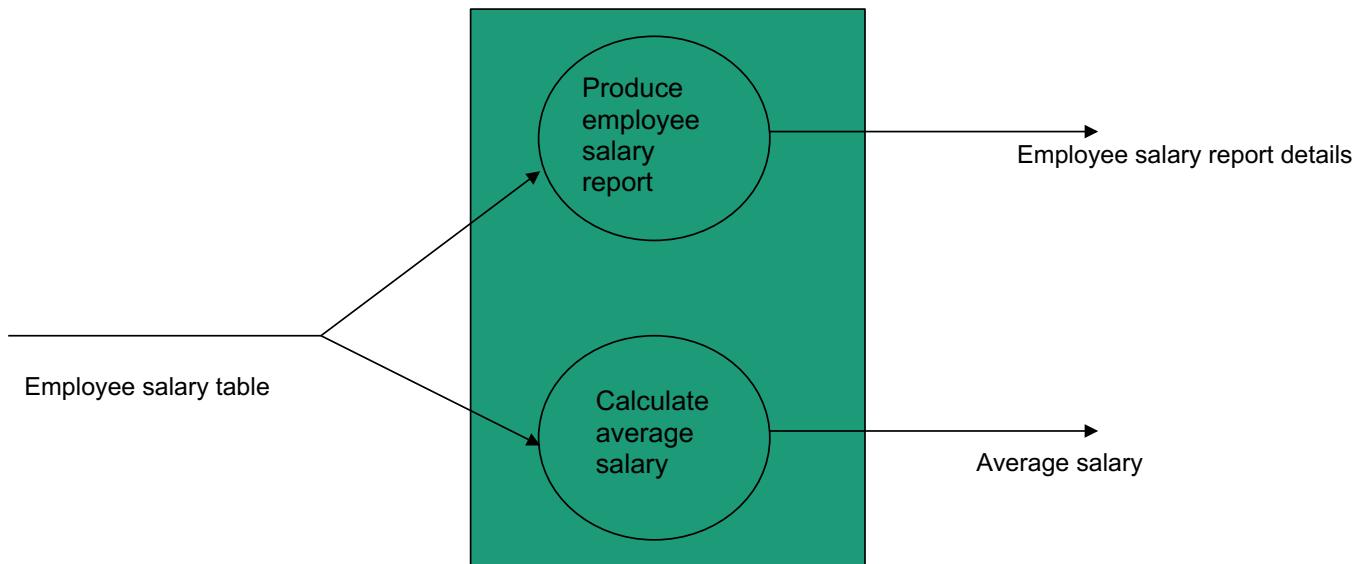
For instance, we may wish to

- 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” communicational cohesion.

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

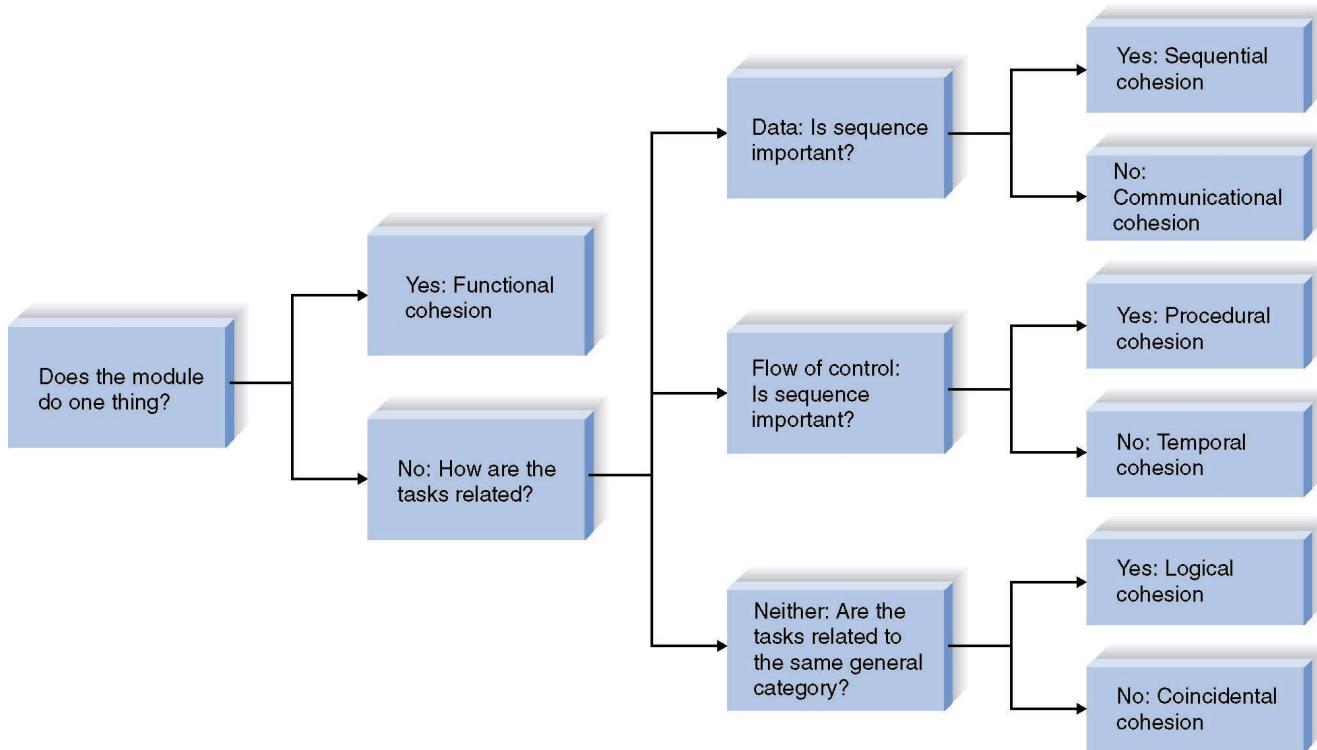
7. Coincidental Cohesion (Lowest)

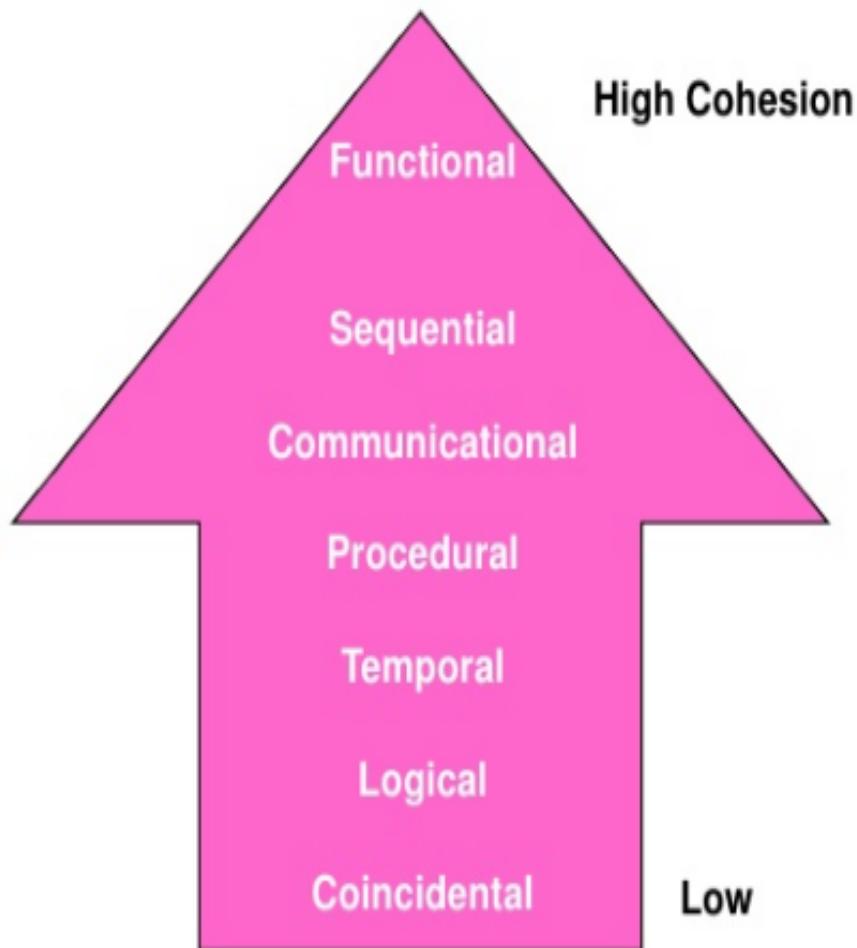
A coincidentally cohesive module is one whose elements contribute to activities with no meaningful relationship to one another.

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.

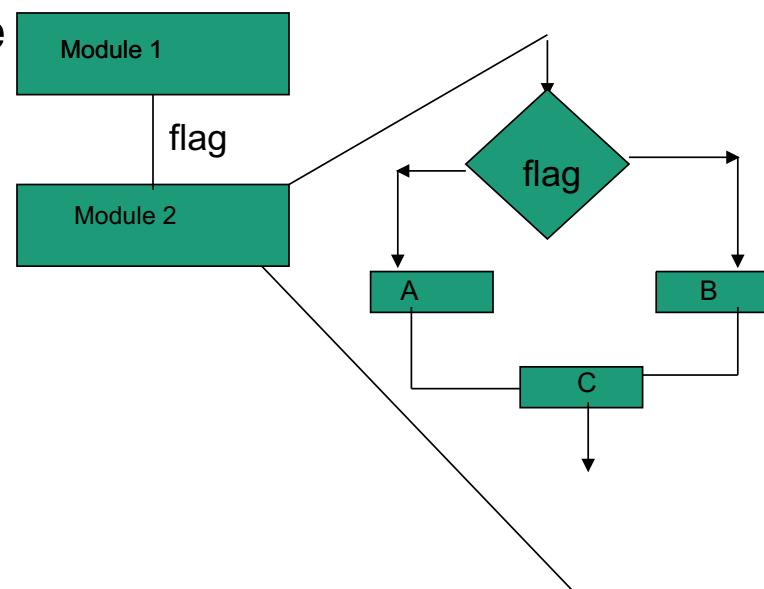
Coupling

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.

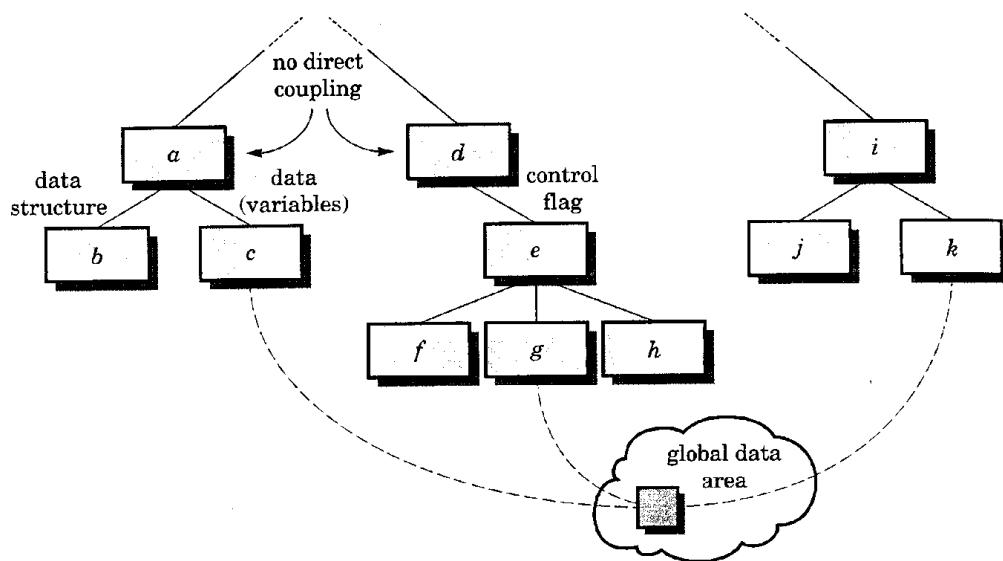


FIGURE 13.8. Types of coupling

Modules c, g and k exhibit common coupling

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;
```

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

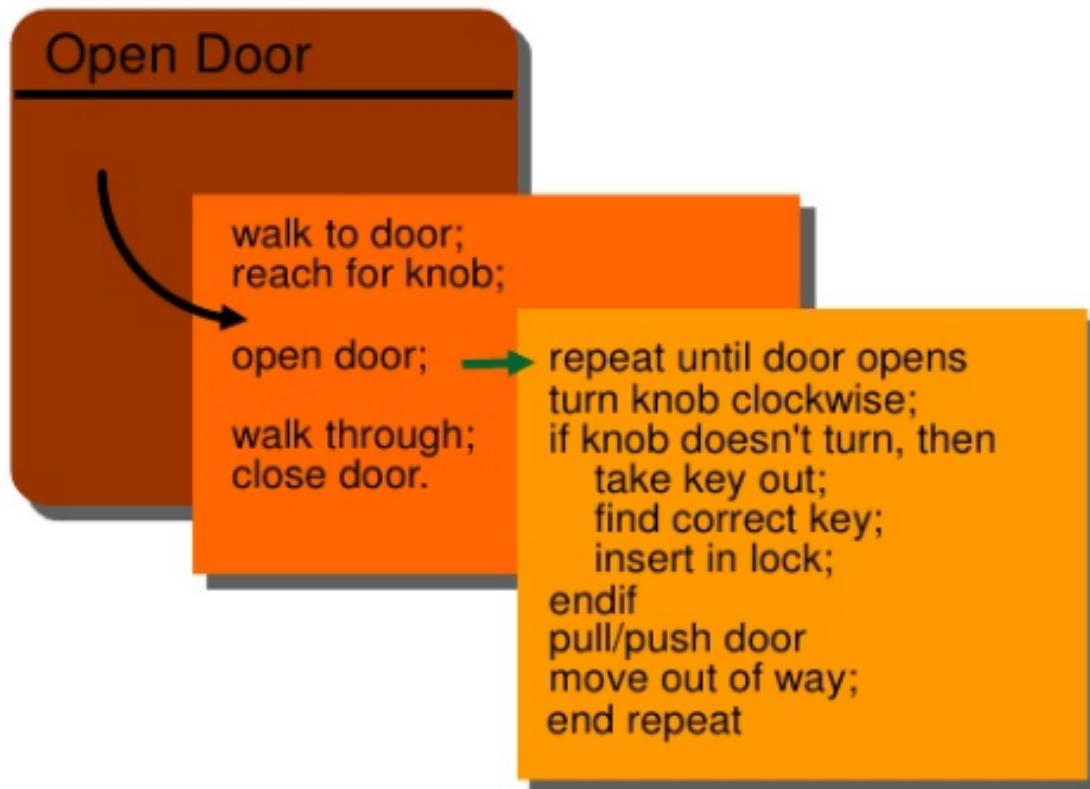
When customer not found, component adds customer by directly modifying the contents of the data structure containing customer data.

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.

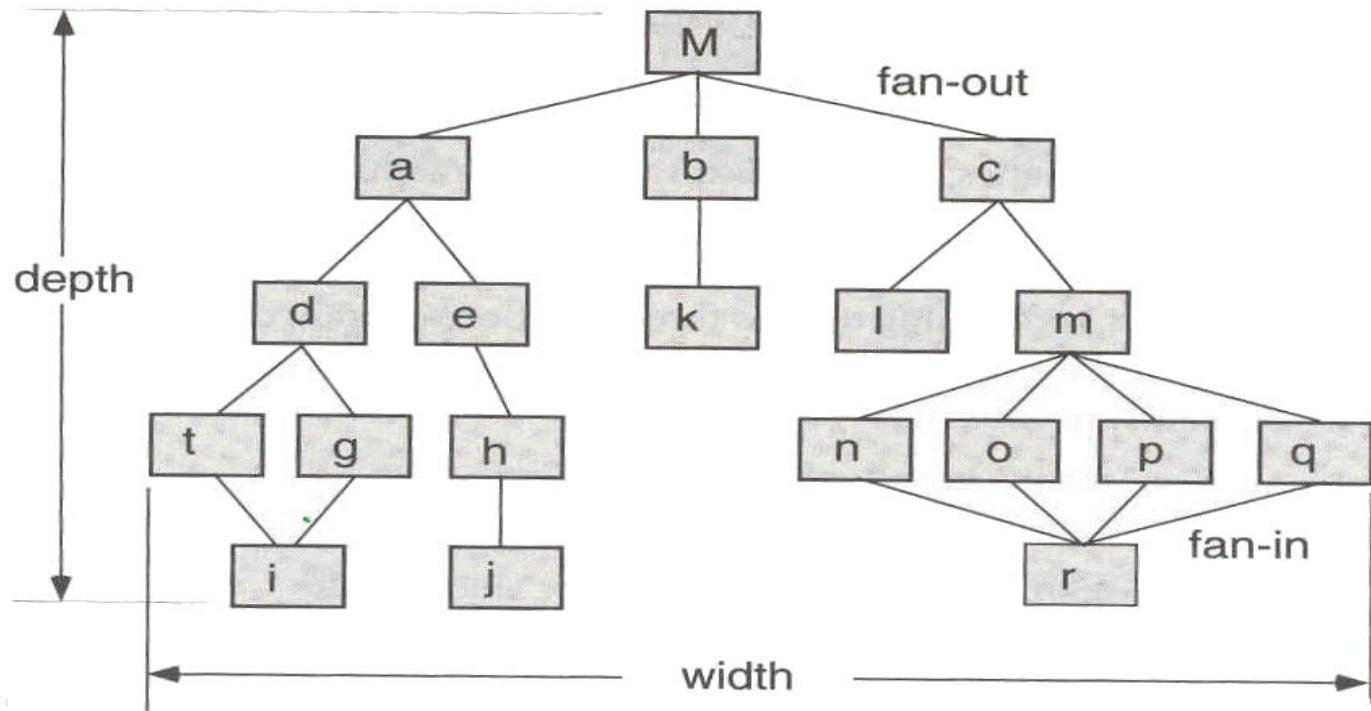
- *Persistent classes*
- represent data stores (e.g., a database) that will persist beyond the execution of the software.
- *System classes*
- implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world

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

Design Concepts

Data structure

is a representation of the logical relationship among individual elements of data.

- **Scalar item –**
A single element of information that may be addressed by an identifier .
- Scalar item organized as a list- **array**
- Data structure that organizes non-contiguous scalar items-**linked list**

Design heuristics for effective modularity

- Evaluate 1st iteration to reduce coupling & improve cohesion
- Minimize structures with high fan-out
- Keep scope of effect of a module within scope of control of that module

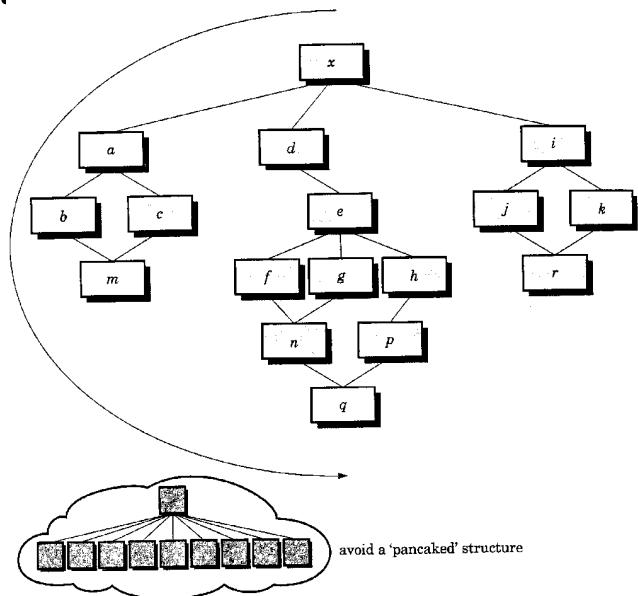


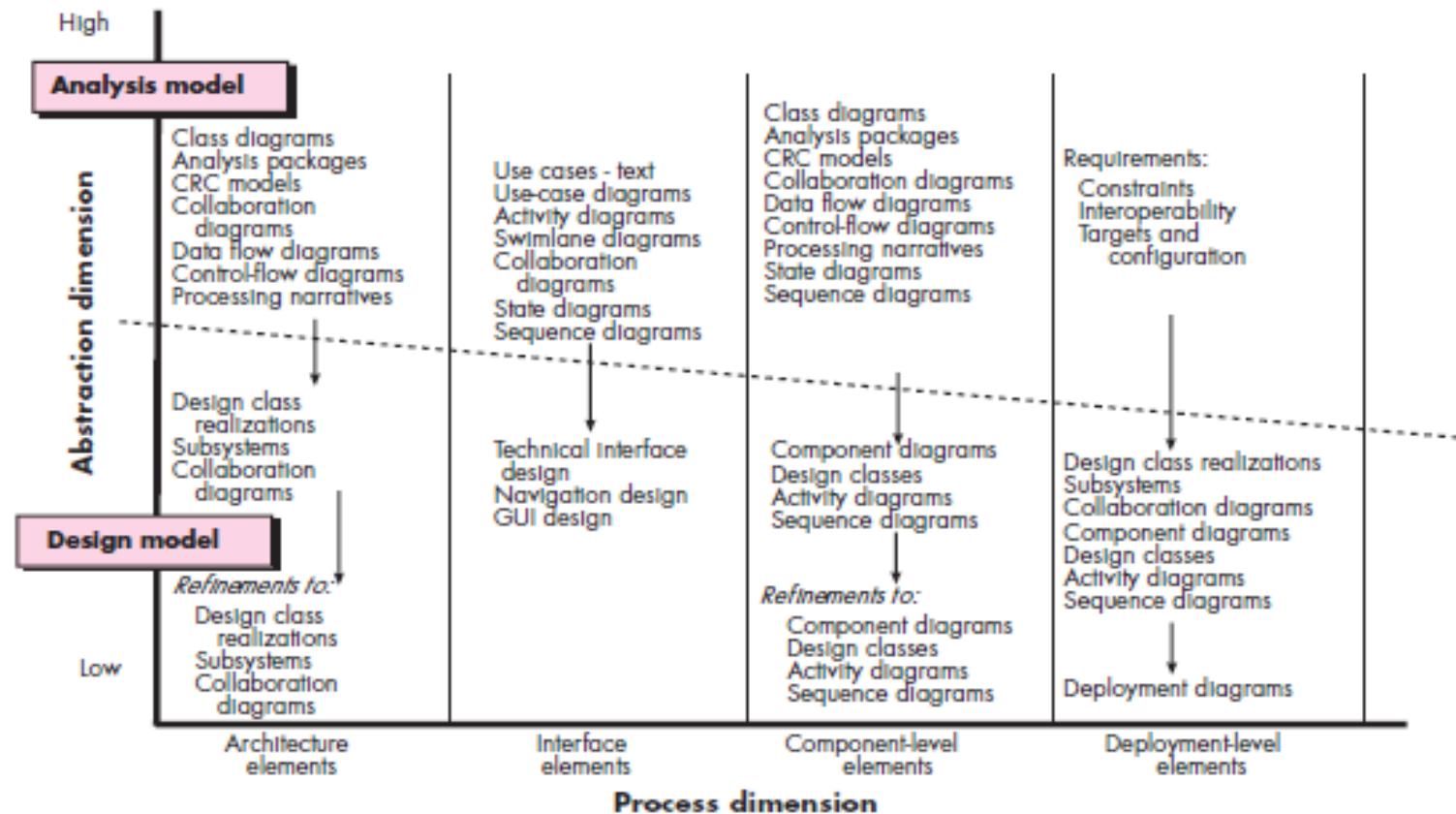
FIGURE 13.9. Program structures

Design heuristics for effective modularity

- Evaluate interfaces to reduce complexity
- Define modules with predictable function
- Strive for controlled entry -- no jumps into the middle

Design Model

FIGURE 8.4 Dimensions of the design model



Data Design Elements

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

Architectural Design Elements

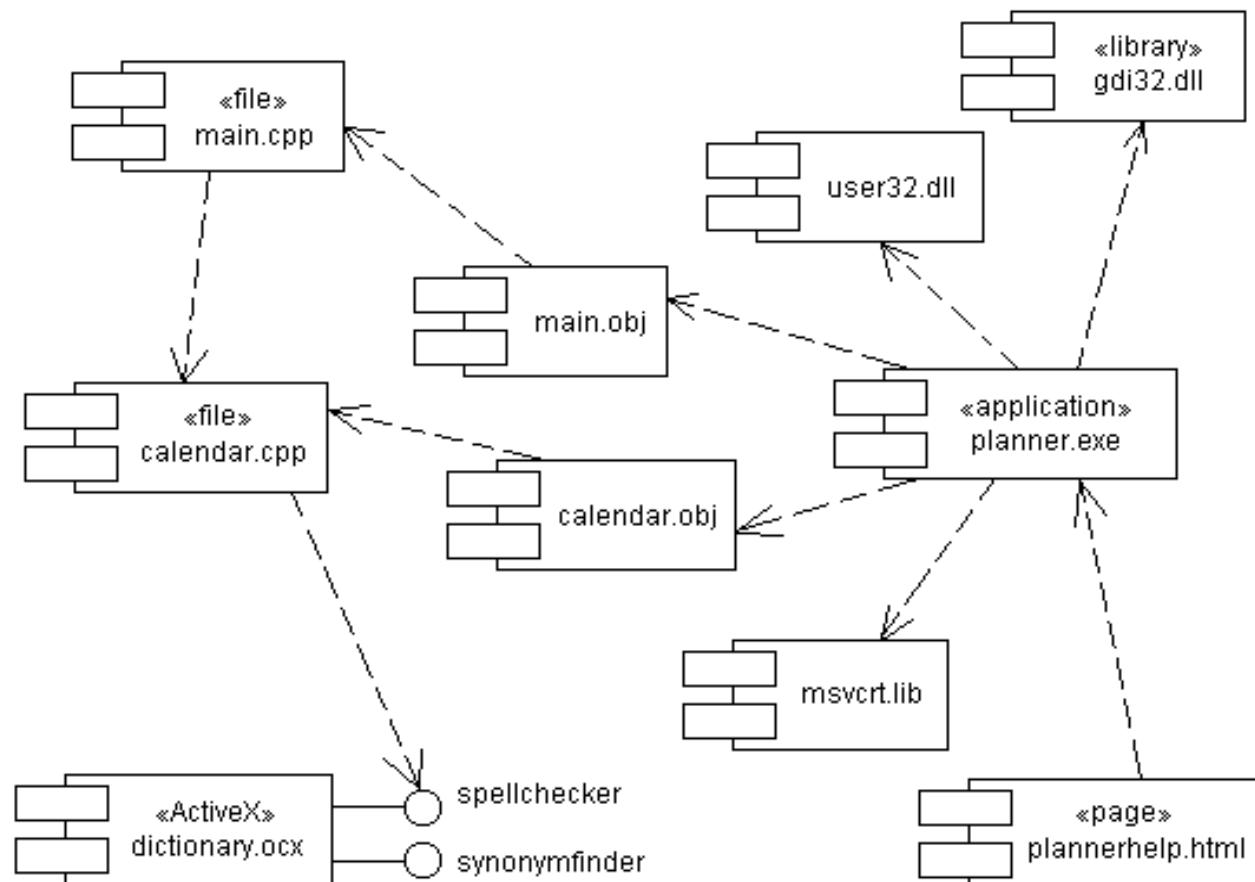
- The *architectural design* for software is the equivalent to the floor plan of a house
- The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model.
- Each subsystem may have it's own architecture (e.g.,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

Interface Design Elements

- The interface design elements for software depict information flows into and out of the system and how it is communicated among the components defined as part of the architecture.
- There are three important elements of interface design: (1) the user interface (UI); (2) external interfaces to other systems, devices, networks, or other producers or consumers of information; and (3) internal interfaces between various design components.
- These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

Component level design elements

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



Deployment level design elements

- Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.

