

Software Engineering

Lecture 01

Topic Covered

- Evolving Role of Software
- Hardware vs. Software
- Intro. to Software Engineering
- Software characteristics
- Software Components
- Software Applications
- Evolution of Software
- Software Myths
- Software Development Life Cycle (SDLC)
- SDLC Phases

Software is a set of instructions, data or programs used to operate computers and execute specific task

Evolving Role of Software

Software is a product

P M A Mo Di T

- Transforms information - produces, manages, acquires, modifies, displays, or transmits information
- Delivers computing potential of hardware and networks

Software is a vehicle for delivering a product

- Controls other programs (operating system)
- Effects communications (networking software)
- Helps build other software (software tools & environments)

What is Software ?

Software can define as:

- **Instruction** – executed provide desire **features, function & performance**.
- **Data structure** – to adequately manipulate operation.
- **Documents** – operation and use of the program.

Software products may be developed for a particular customer or may be developed for a general market.

- Software products may be
 - **Generic** - developed to be **sold to a range of different customers** e.g. PC software such as Excel or Word.
 - **Bespoke (custom)** - developed for a single customer according to their **specification**.

Hardware vs. Software

Hardware

- Manufactured
- wear out
- Built using components
- Relatively simple

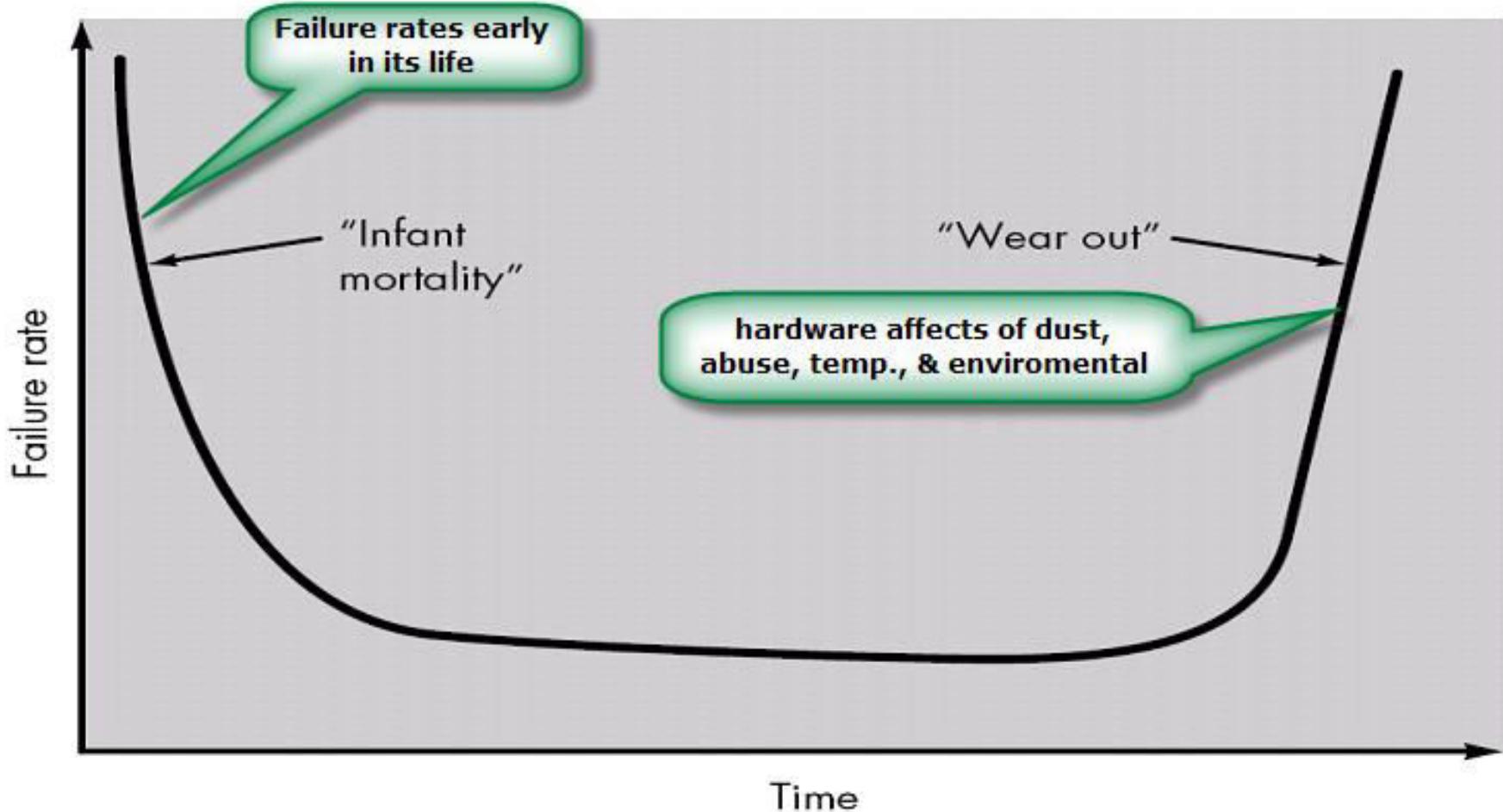
Software

- Developed/ engineered
- deteriorate (worsen)
- Custom built
- Complex

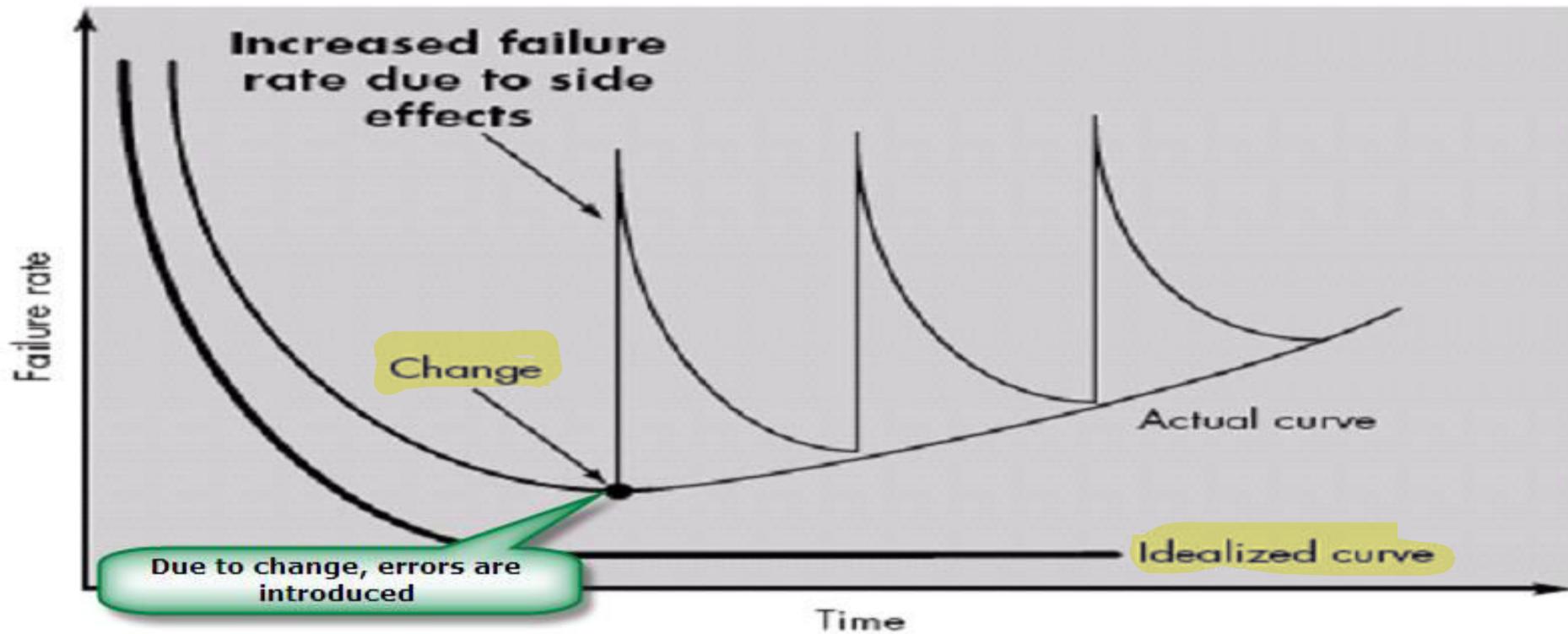
Manufacturing vs. Development

- Once a hardware product has been manufactured, it is difficult or impossible to modify. In contrast, software products are routinely modified and upgraded.
- In hardware, hiring more people allows you to accomplish more work, but the same does not necessarily hold true in software engineering.
- Unlike hardware, software costs are concentrated in design rather than production.

Failure curve for Hardware



Failure curve for Software



When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, software maintenance involves considerably more complexity

Component Based vs. Custom Built

- Hardware products typically employ many **standardized** design components.
- Most software continues to be **custom built**.
- The software industry does seem to be moving (slowly) toward component-based construction.

What is Software Engineering?

- The process of solving customers' problems by the systematic development and evolution of large, high-quality software systems within cost, time and other constraints
- Note:
 - Process, systematic (not ad hoc), evolutionary...
for this specific purpose
 - Constraints: high quality, cost, time, meets user requirements

Analysis of the Definition:

- Systematic development and evolution
 - An engineering process involves applying well understood techniques in a organized and disciplined way
 - Many well-accepted practices have been formally standardized
 - e.g. by the IEEE or ISO
 - Most development work is evolutionary

Analysis of the Definition:

- Large, high quality software systems
 - Software engineering techniques are needed because large systems cannot be completely understood by one person
 - Teamwork and co-ordination are required
 - Key challenge: Dividing up the work and ensuring that the parts of the system work properly together
 - The end-product that is produced must be of sufficient quality

Analysis of the Definition:

► Cost, time and other constraints

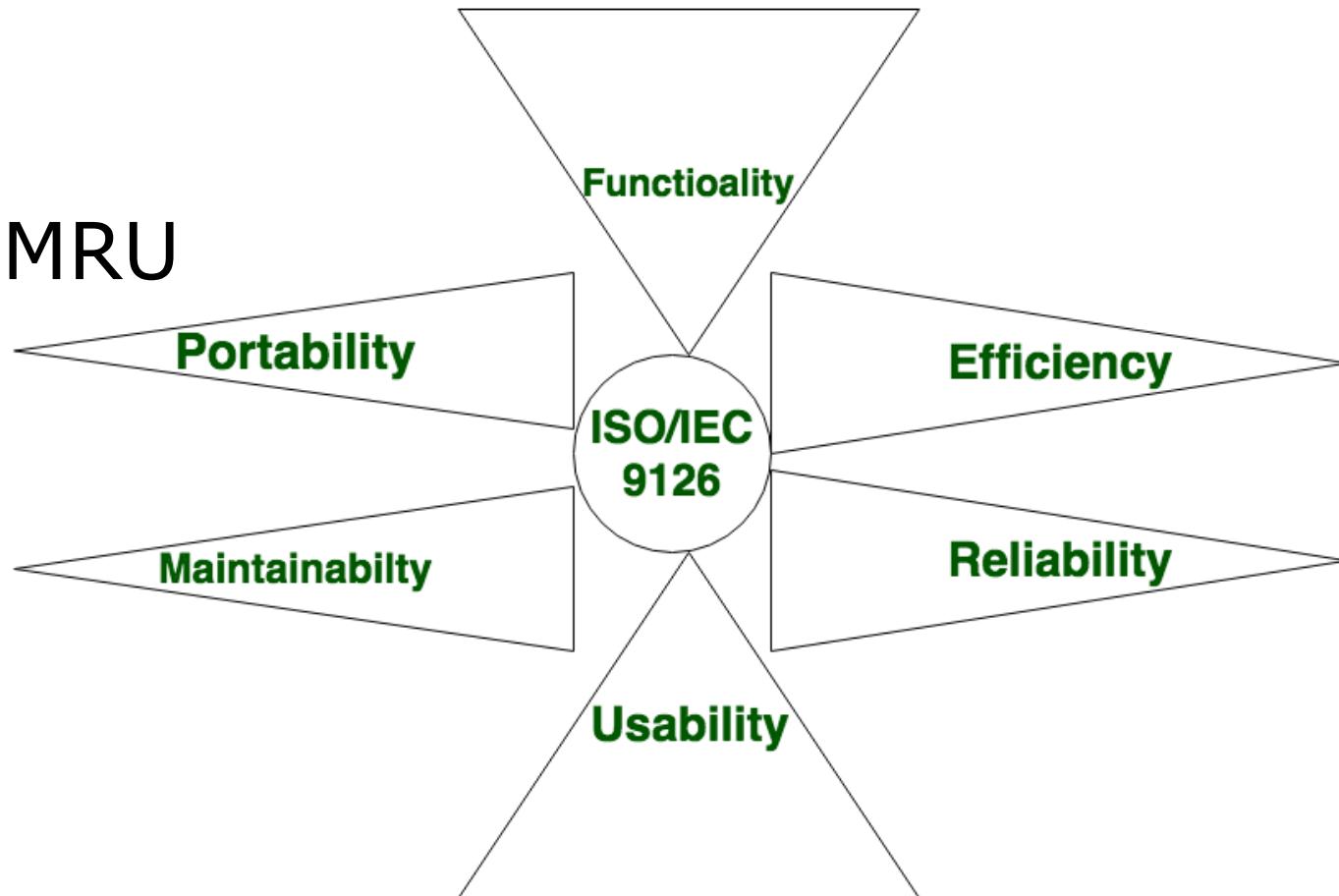
- Finite resources
- The benefit must outweigh the cost
- Others are competing to do the job cheaper and faster
- Inaccurate estimates of cost and time have caused many project failures

Software characteristics

- Software is developed or engineered; it is **not** manufactured.
- Software does not “wear out” but it does deteriorate.
- Software continues to be custom built, as industry is moving toward component based construction.

Software Components

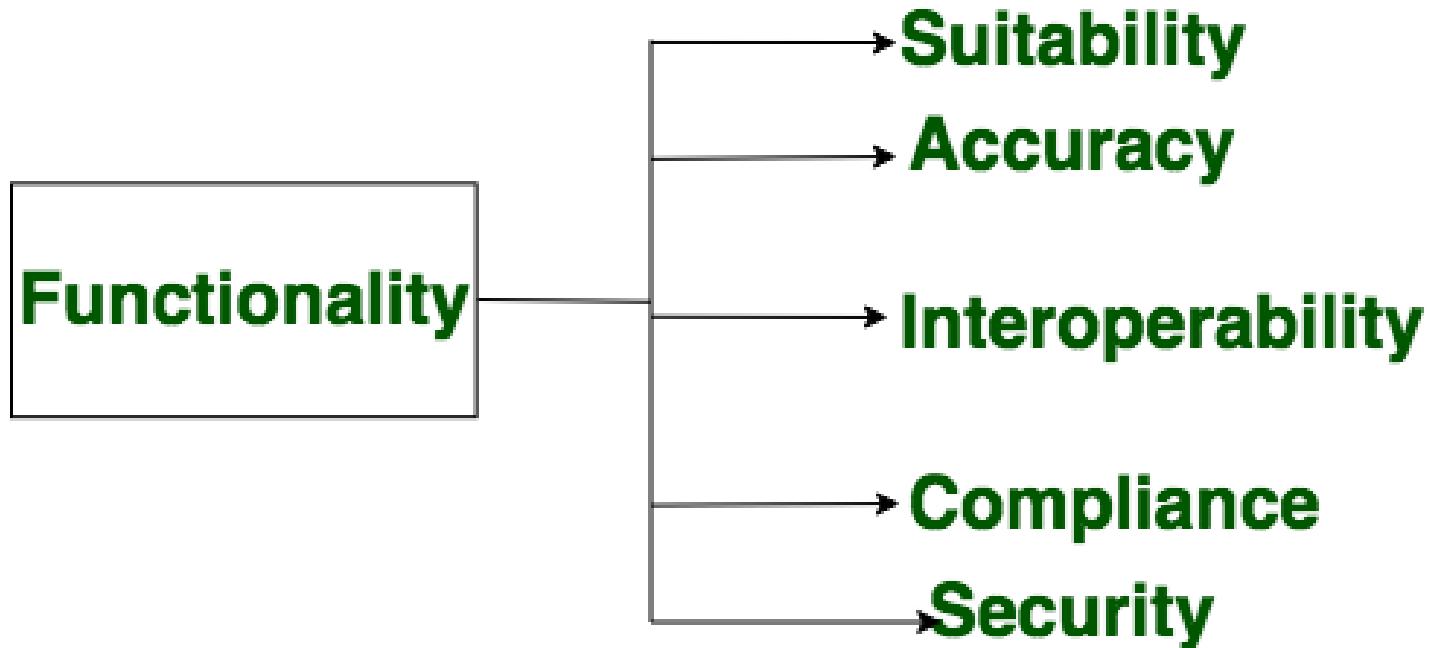
FPEMRU



Source: <https://www.geeksforgeeks.org/software-engineering-software-characteristics/>

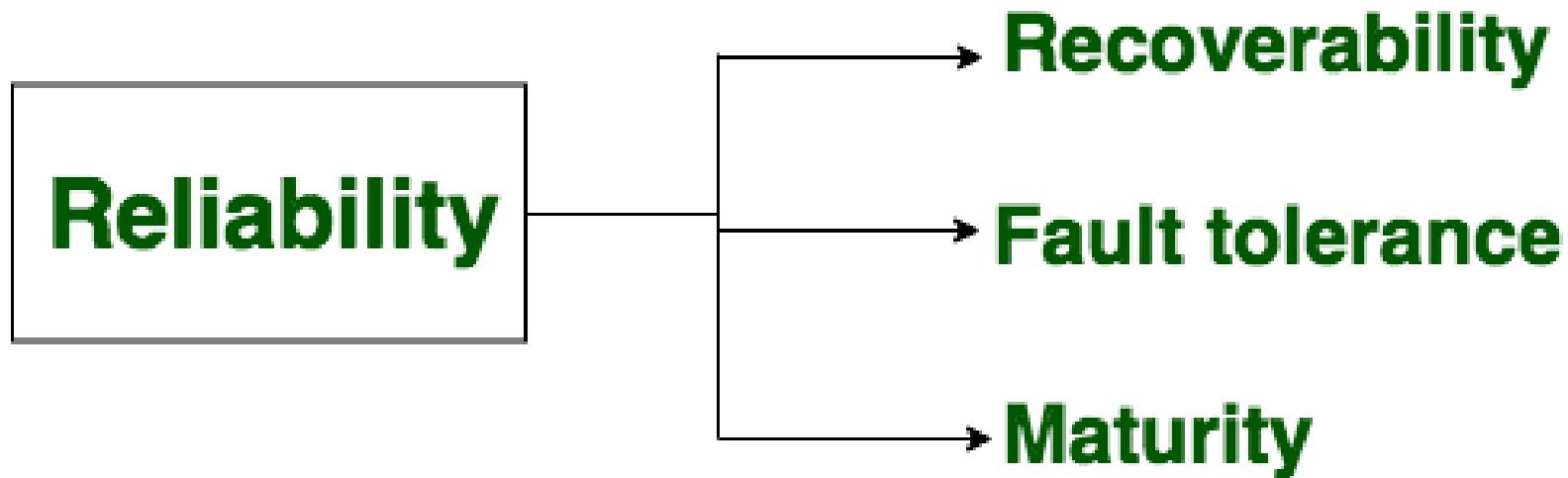
Functionality

- It refers to the degree of **performance** of the software **against** its intended purpose.



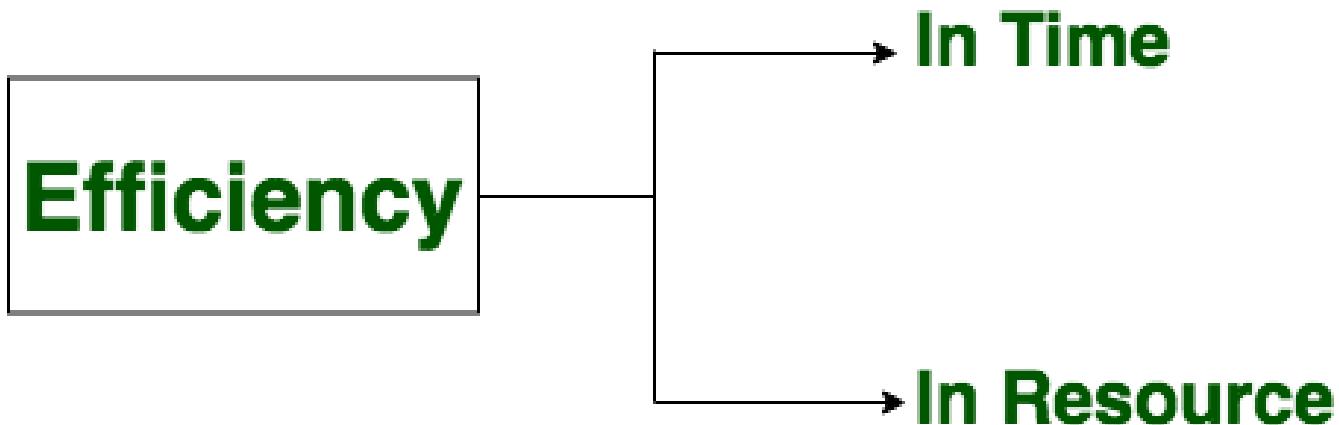
Reliability

- A set of attributes that bears on the **capability** of software to **maintain its level of performance** under the given condition for a stated period of time.



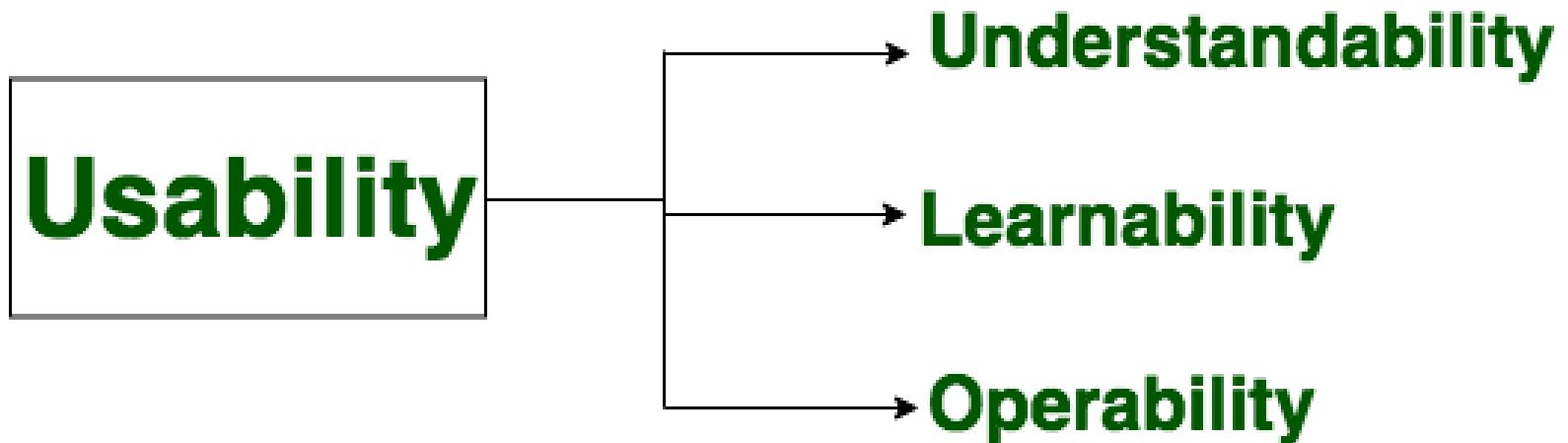
Efficiency

- It refers to the **ability** of the software to use system resources in the **most effective and efficient manner**. The software should make effective use of storage space and execute command as per desired timing requirements.



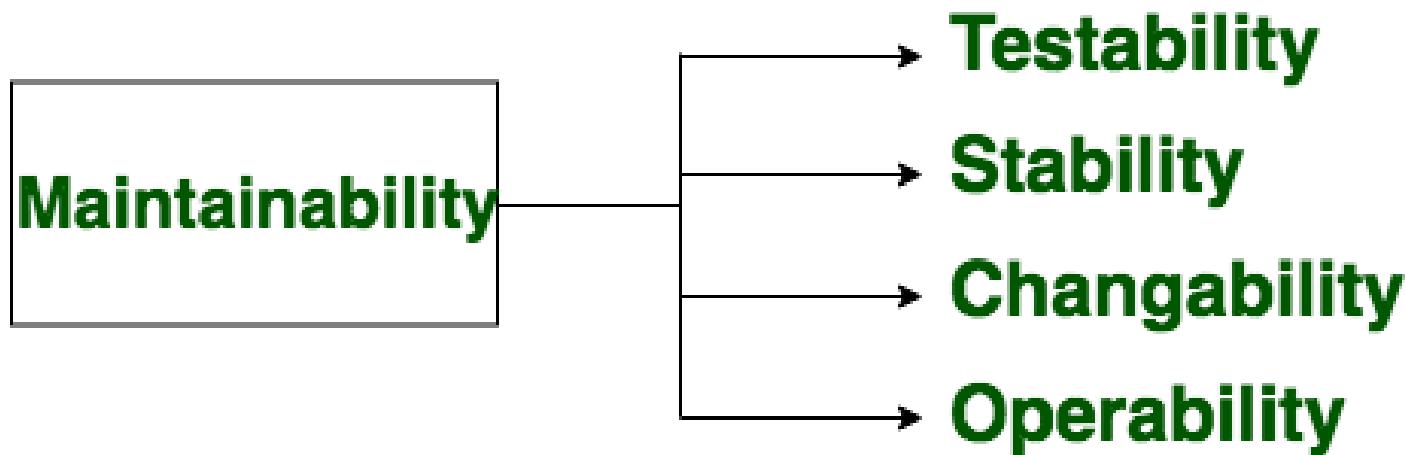
Usability

- It refers to the extent to which the software can be used with ease. the amount of effort or time required **to learn** how to use the software.



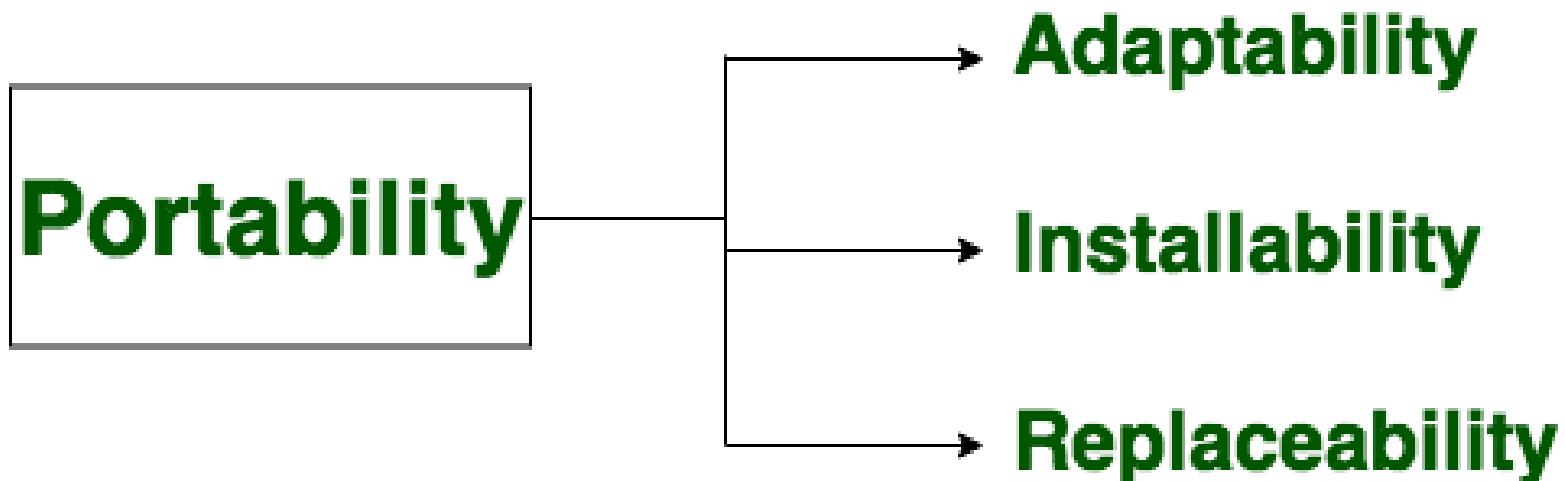
Maintainability

- It refers to the ease with which the **modifications** can be made in a software system to extend its functionality, improve its performance, or correct errors.



Portability

- A set of attributes that bears on the ability of software to be transferred from one environment to another, without or minimum changes.



Software Applications

- System software
 - Application software
 - Engineering/scientific software
 - Embedded software
 - Product line software
 - Web applications
 - Artificial intelligence software
- SAEPAW

System Software:

- System software is a collection of programs written to service other programs.
- It is characterized by **heavy interaction with computer hardware**; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and **sophisticated process management**; **complex data structures**; and **multiple external interfaces**.

Ex. Compilers, operating system, drivers etc.

Application Software :

- Application software consists of **standalone programs** that solve a **specific business need**.
- Application software is used to control the business function in real-time.

Engineering /Scientific software:

- Characterized by "number crunching" algorithms.
- Applications range from astronomy to volcano logy, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing.

Ex. Computer Aided Design (CAD), system stimulation etc.

Embedded Software:

- It resides in **read-only memory** and is used to control products and systems
- Embedded software can perform limited and esoteric functions.

Ex. keypad control for a microwave oven.

Product line software:

- Designed to provide a **specific capability for use by many different customers**, product line software can focus on a limited and **esoteric** marketplace.

Ex. Word processing, spreadsheet, CG, multimedia, etc.

intended for or likely
to be understood by
only a small number
of people with a
specialized
knowledge or
interest

Web Applications:

- Web apps can be little more than a set of linked hypertext files.
- It evolving into sophisticated computing environments that not only provide standalone features, functions but also integrated with corporate database and business applications.

Artificial Intelligence software

- AI software makes use of **non-numerical algorithms to solve complex problems** that are not amenable to computation or straightforward analysis

Ex. Robotics, expert system, game playing, etc.

Software Crisis Problem

- Increasing cost of Computers
- Increasing product complexity
- Lack of programmers
- Slow programmer's productivity growth
- Lack of funding for software engineering research
- Rising demand for software
- Lack of caffeine in software development organizations

Software Crisis Problem

- Increasing cost of Computers
- **Increasing product complexity**
- Lack of programmers
- **Slow programmer's productivity growth**
- Lack of funding for software engineering research
- **Rising demand for software**
- Lack of caffeine in software development organizations

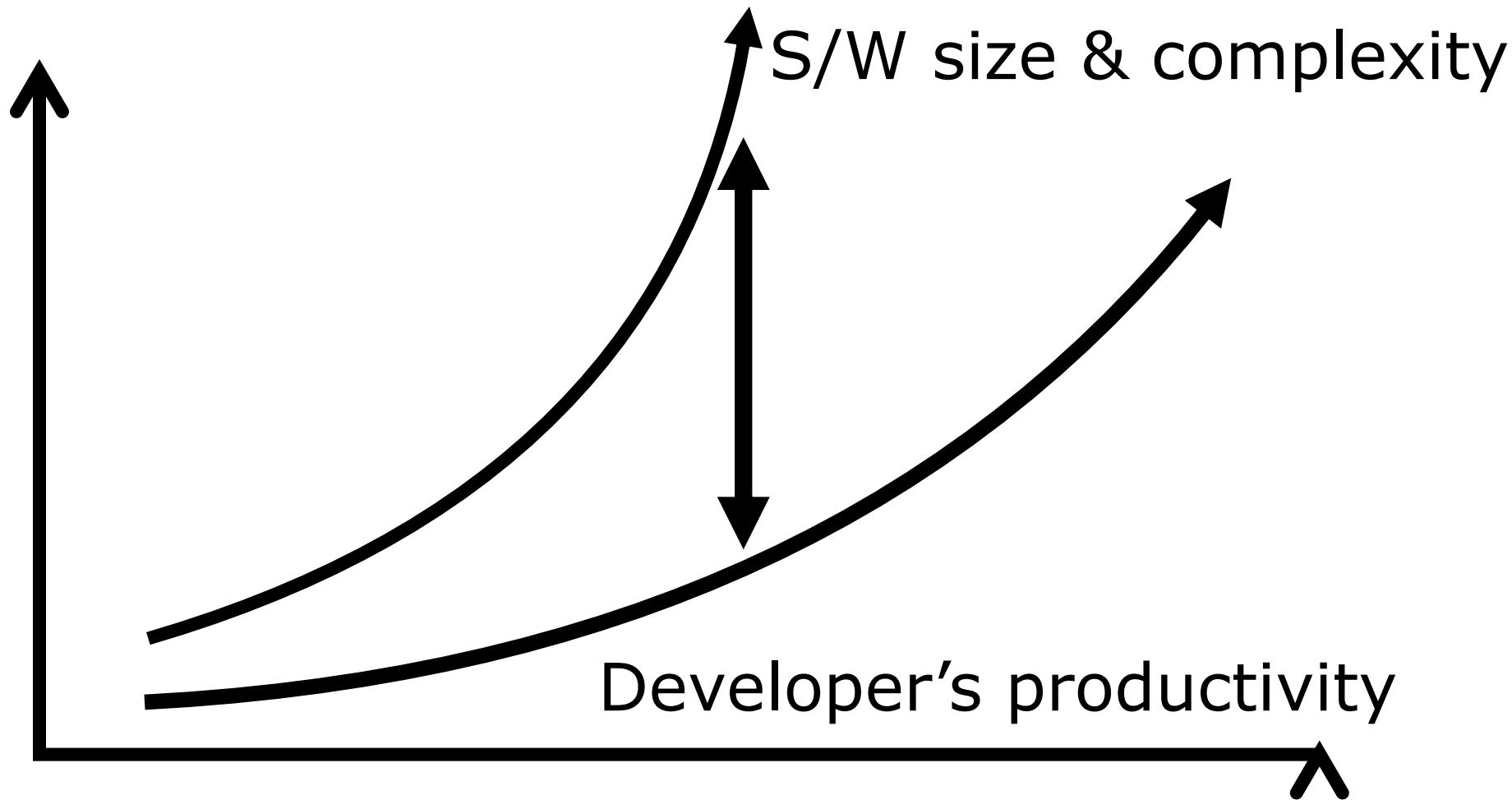
Development Effort

Size (LOC)	Example
10^2	Class exercise
10^3	Small Project
10^4	Term Project
10^5	Word processor
10^6	Operating System
10^7	Distributed System

Programming effort

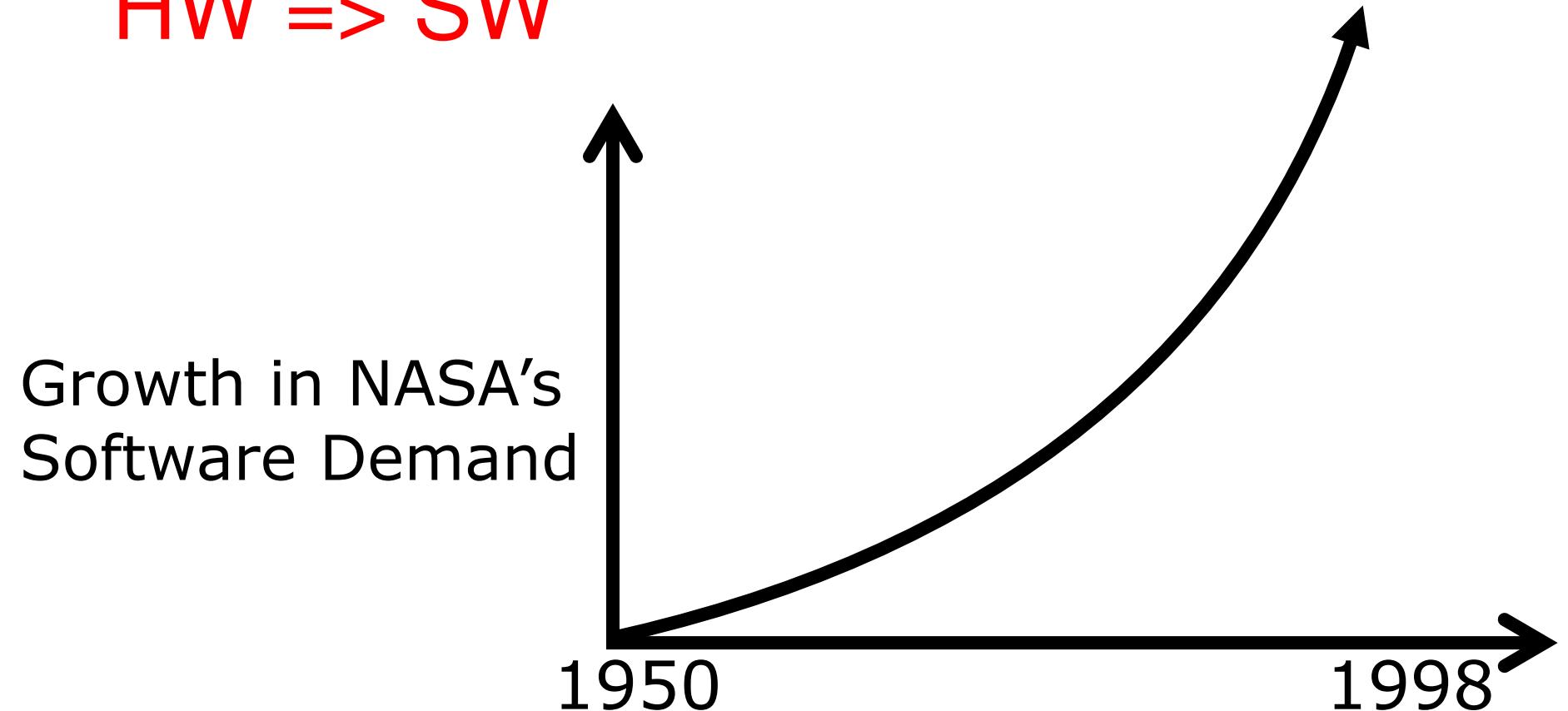
Software Engineering effort

Developers Productivity Growth



Rising Demand for Software

HW => SW



Software Evolution

- **The Law of Continuing Change (1974):** E-type (Real world implemented) systems must be continually adapted else they become progressively less satisfactory.
- **The Law of Increasing Complexity (1974):** As an E-type system evolves its complexity increases unless work is done to maintain or reduce it.
- **The Law of Self Regulation (1974):** The E-type system evolution process is self-regulating with distribution of product and process measures close to normal.
- **The Law of Conservation of Organizational Stability (1980):** The average effective global activity rate in an evolving E-type system is invariant over product lifetime.

Software Evolution

- **The Law of Conservation of Familiarity (1980):** As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behavior to achieve satisfactory evolution.
- **The Law of Continuing Growth (1980):** The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime.
- **The Law of Declining Quality (1996):** The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.
- **The Feedback System Law (1996):** E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

Software Myths

Definition: Beliefs about software and the process used to build it. Myths have number of attributes that have made them insidious (i.e. dangerous).

- Misleading Attitudes - caused serious problem for managers and technical people.

Management myths

Managers in most disciplines, are often under pressure to maintain budgets, keep schedules on time, and improve quality.

Myth1: 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 :

- Are software practitioners aware of existence standards?
- Does it reflect modern software engineering practice?
- Is it complete? Is it streamlined to improve time to delivery while still maintaining a focus on quality?

Myth2: If we get behind schedule, we can add more programmers and catch up

Reality: Software development is not a mechanistic process like manufacturing.

Adding people to a late software project makes it later.

- People can be added but only in a planned and well-coordinated manner

Myth3: 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 outsource software projects

Customer Myths

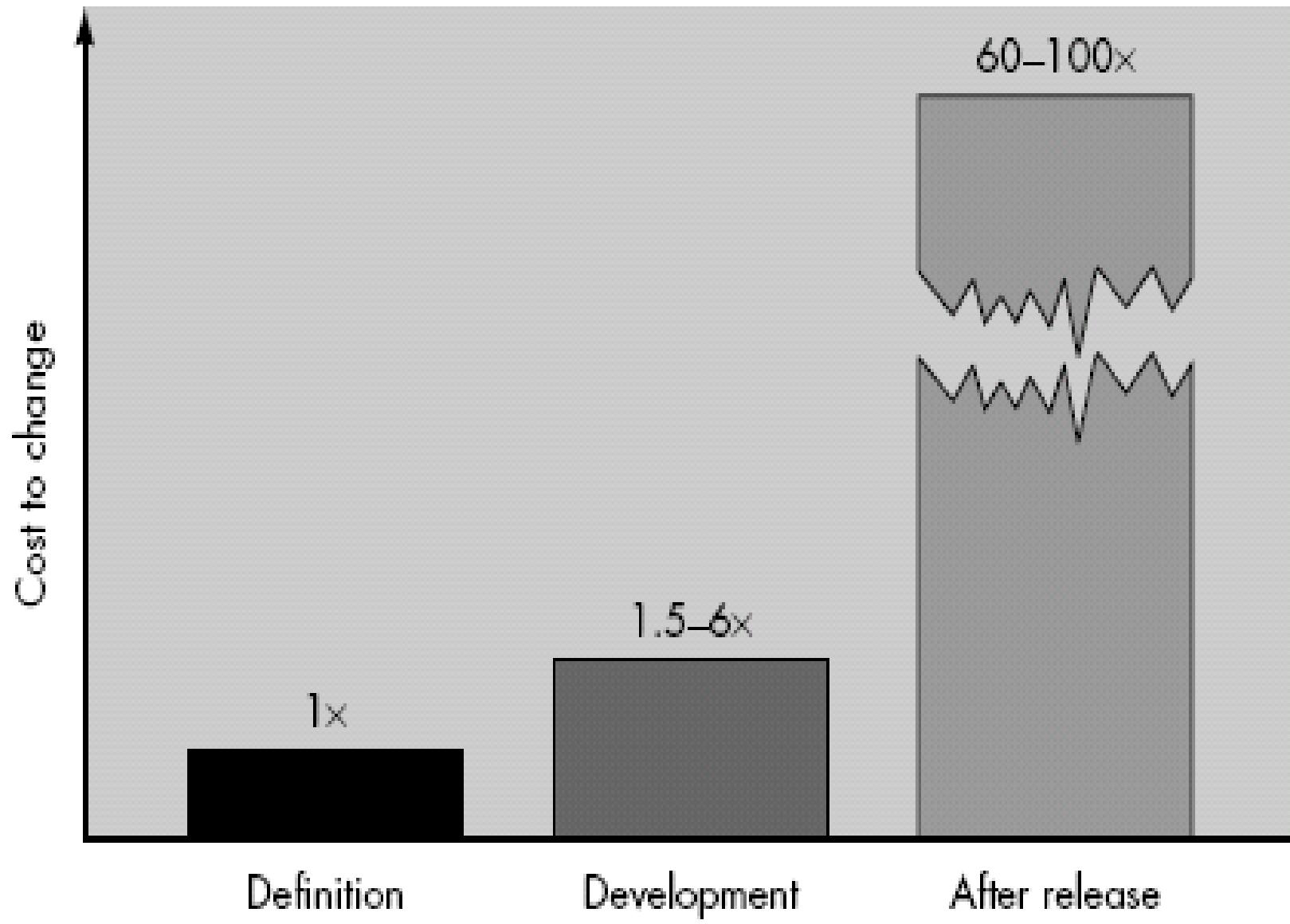
Customer may be a person from inside or outside the company that has requested software under contract.

Myth: A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.

Reality: A poor up-front definition is the major cause of failed software efforts. A formal and detailed description of the information domain, function, behavior, performance, interfaces, design constraints, and validation criteria is essential. These characteristics can be determined only after thorough communication between customer and developer.

Myth: Project requirements continually change, but change can be easily accommodated because software is flexible.

Reality: Customer can review requirements and recommend modifications with relatively little impact on cost. When changes are requested during software design, the cost impact grows rapidly. Below mentioned *figure* for reference.



Practitioner's myths

Myth1: 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.

Myth2: 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 formal technical review.

Myth3: The only deliverable work product for a successful project is the working program.



the establishment or starting point of an institution or activity

Reality: A working program is only one part of a software configuration that includes many elements. Documentation provides a foundation for successful engineering and, more important, guidance for software support.

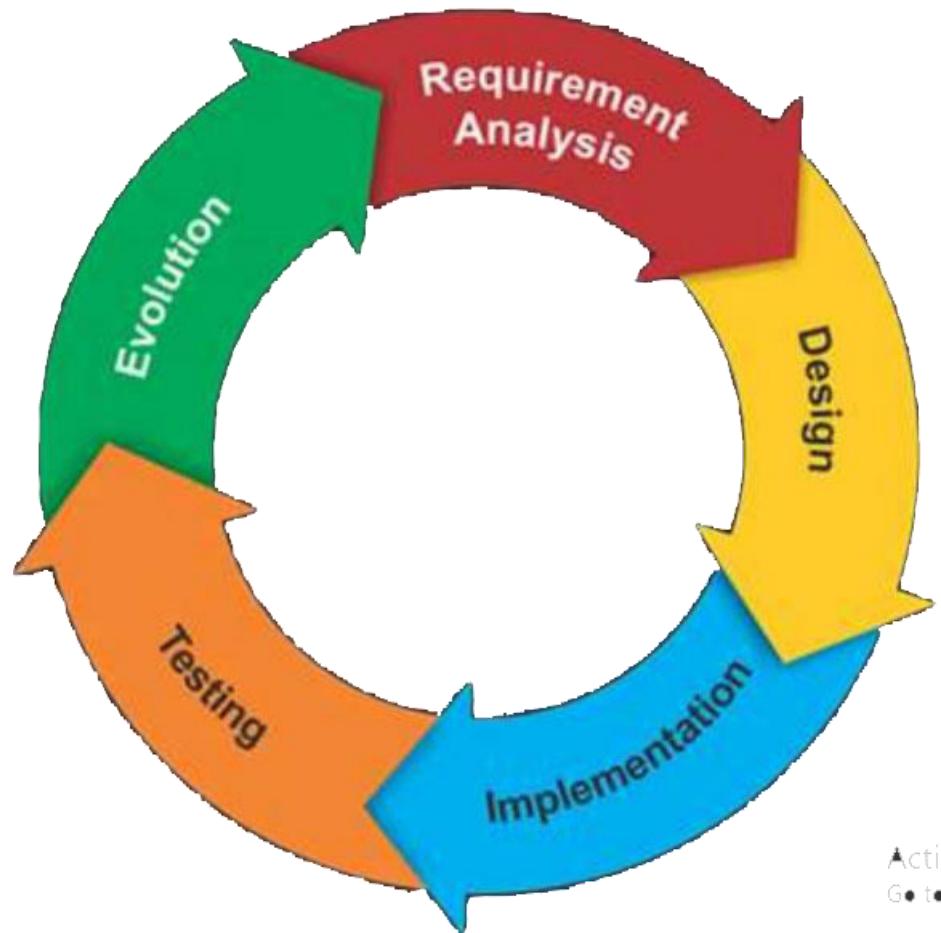
Myth4 : Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.

Reality: Software engineering is not about creating documents. It is about creating quality. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

Software Development Life Cycle (SDLC)

- The SDLC is a framework that describes the activities performed at each stage of a software development project.
- SDLC process is used by the software industry to design, develop and test high quality software. It aims to produce the quality software that meets or exceeds customer expectations, reaches completion within time and budget.

SDLC Phases



Acti
Go to

E A D I T

SDLC Phases

1. Planning and Requirements Analysis
2. Defining Requirements
3. Designing the Software
4. Building or Developing the Software
5. Testing the Software
6. Deployment and Maintenance

P,A
DR
DS
BDS
TS
DM

1. Planning & Requirement Analysis

- Requirement analysis is the most important.
- It is performed by the senior members of the team with inputs from all the *stakeholders* and *domain experts* or *SMEs* in the industry.
subject matter expert
- Planning for the *quality assurance requirements* and *identification of the risks associated* with the project is also done at this stage.

Requirements Analysis (cont.)

- Business Requirements
- Stakeholder Requirements
- Solution Requirements
 - Functional Requirements
 - Non-functional Requirements
- Transition Requirements

2. Defining Requirements

Once the requirement analysis is done the next step is to clearly define and document the software requirements and get them **approved from the project stakeholders.**

This is done through '**SRS**' – Software Requirement Specification document which consists of all the product requirements to be designed and developed during the project life cycle.

Defining Requirements (cont.)

- Enterprise Analysis
- Business Analysis Planning & Monitoring
- Elicitation evoke or draw out (a reaction, answer, or fact) from someone
- Requirements Analysis
- Requirements Management & Communication
- Solution Assessment & Validation

3. Designing the Software

- Based on the requirements specified in SRS, usually more than one design approach for the product architecture is proposed and documented in a DDS - **Design Document Specification**.
- This DDS is reviewed by all the stakeholders and based on various parameters as risk assessment, design modularity , budget and time constraints , the best design approach is selected for the software.

4. Developing the Software

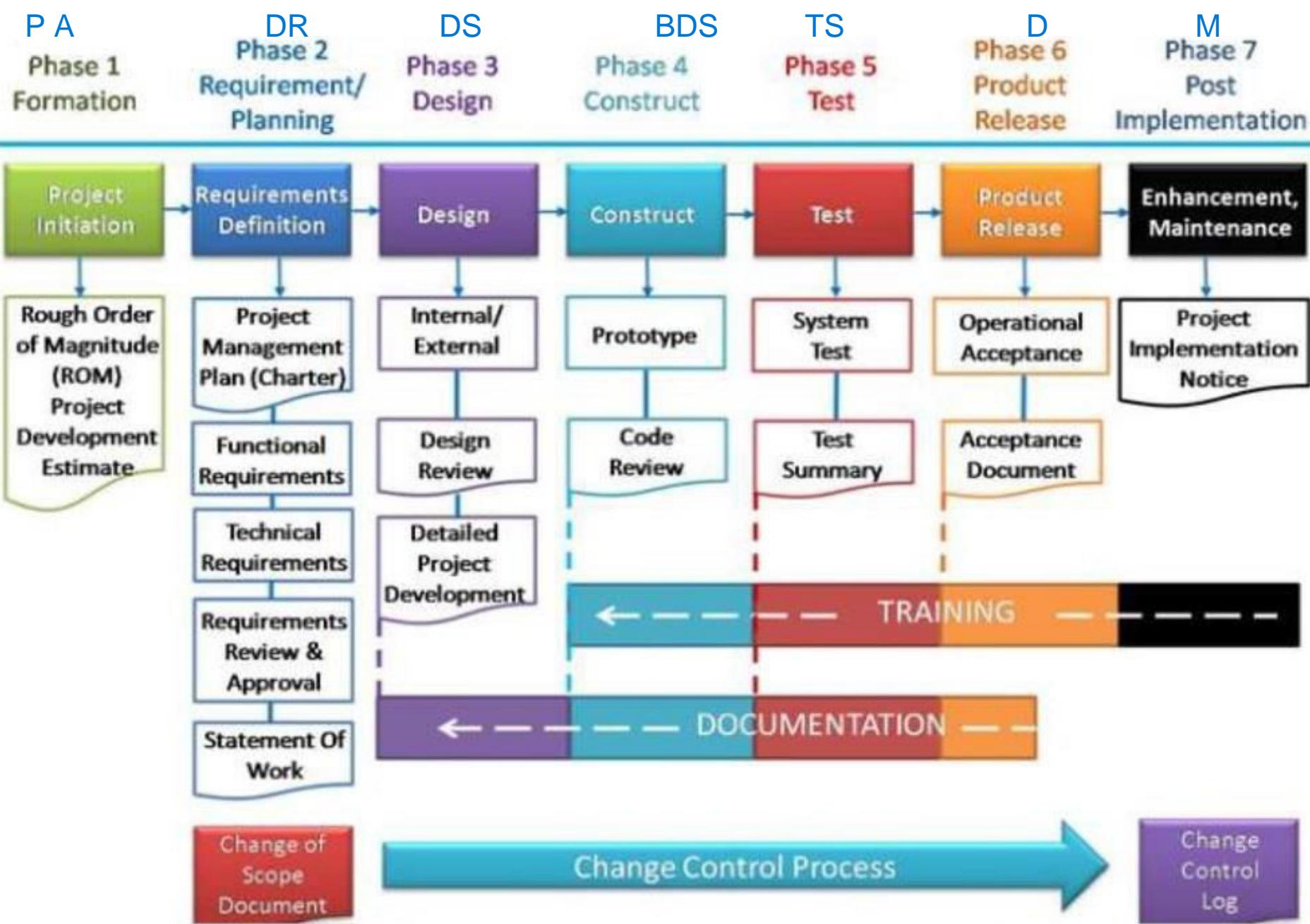
- In this stage of SDLC the actual **development starts** and the product is built. The programming code is generated as per DDS during this stage.
- Developers have to follow **the coding guidelines defined by their organization** and programming tools like compilers, interpreters, debuggers etc are used to generate and implement the code.

5. Testing the Software

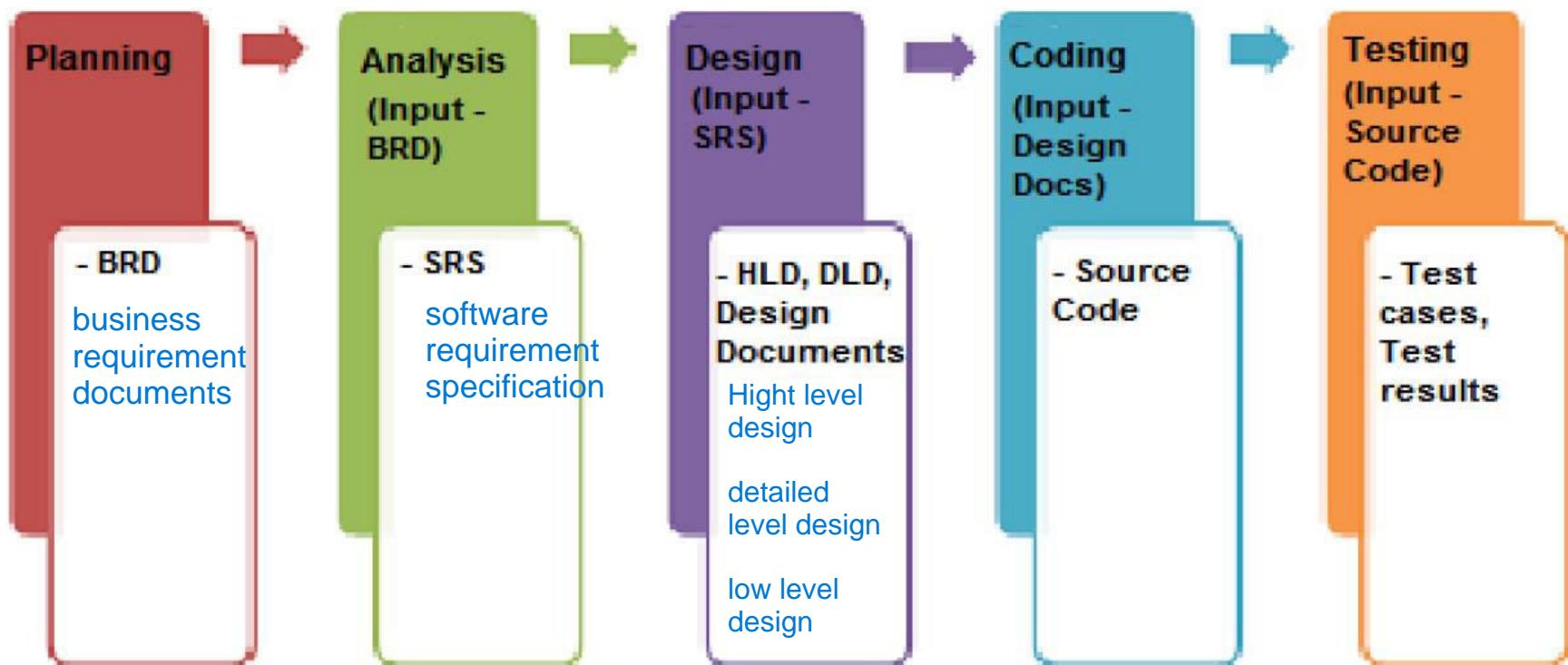
- This stage is usually a **subset** of all the stages as in the modern SDLC models, the testing activities are mostly involved in all the stages of SDLC.
- However this stage refers to the testing only that stage of the software where defects are **reported, tracked, fixed and retested**, until the software reaches the quality standards defined in the SRS.

6. Deployment and Maintenance

- Once the software is tested and no bugs or errors are reported then it is deployed.
- Then based on the feedback, the software may be released as it is or with suggested enhancements in the target segment.
- After the software is deployed then its maintenance starts.



SDLC Stages & Documents



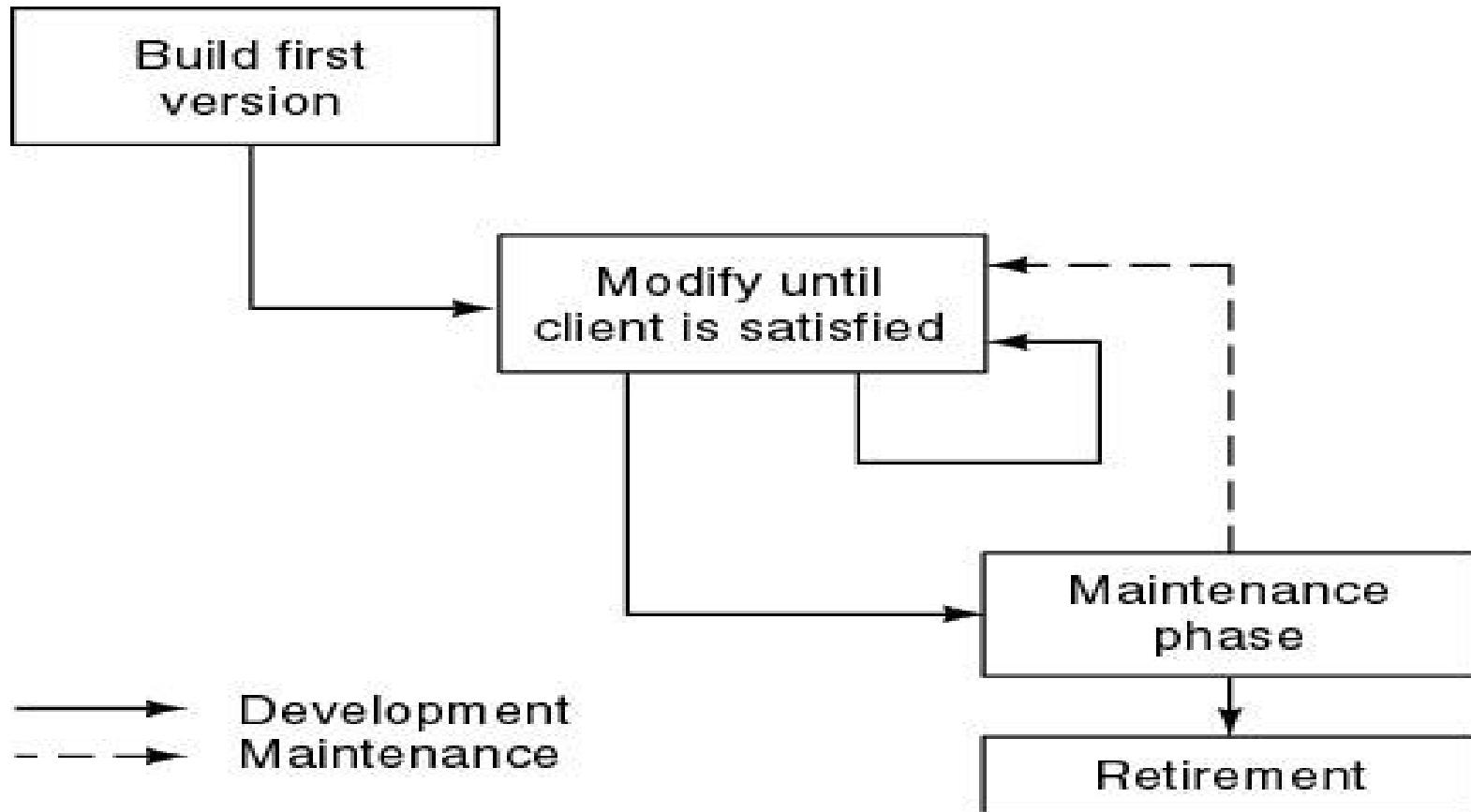
Software Engineering

Lecture 02 Software Process Model

Software process model

- Process models prescribe a distinct set of activities, actions, tasks, milestones, and work products required to engineer high quality software.
- Process models are not perfect, but provide roadmap for software engineering work.
- Software models provide stability, control, and organization to a process that if not managed can easily get out of control
- Software process models are adapted to meet the needs of software engineers and managers for a specific project.

Build and Fix Model



Build and Fix Model

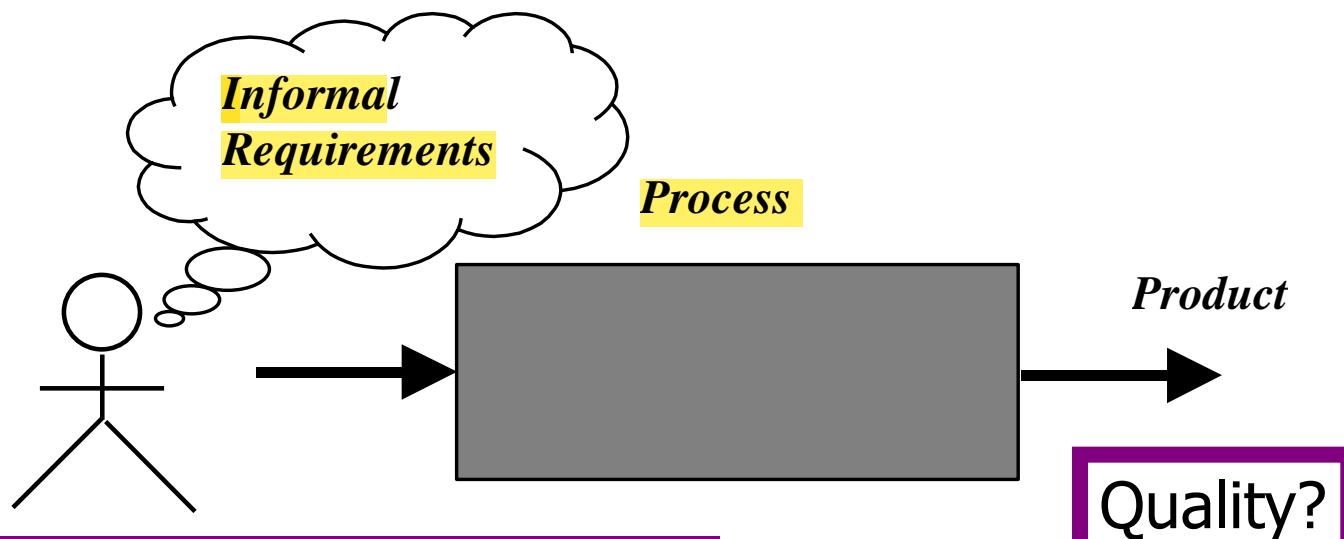
The earlier approach

- Product is constructed **without specification** or any attempt at **design**.
- developers simply **build a product that is reworked** as many times as necessary to satisfy the client.
- model may work **for small projects** but is totally unsatisfactory for products of any reasonable size.
- **Maintenance is high.**
- Source of difficulties and deficiencies
 - impossible to predict
 - impossible to manage

Why Models are needed?

- Symptoms of inadequacy: the software crisis
 - scheduled time and cost exceeded
 - user expectations not met
 - poor quality

Process as a "black box"

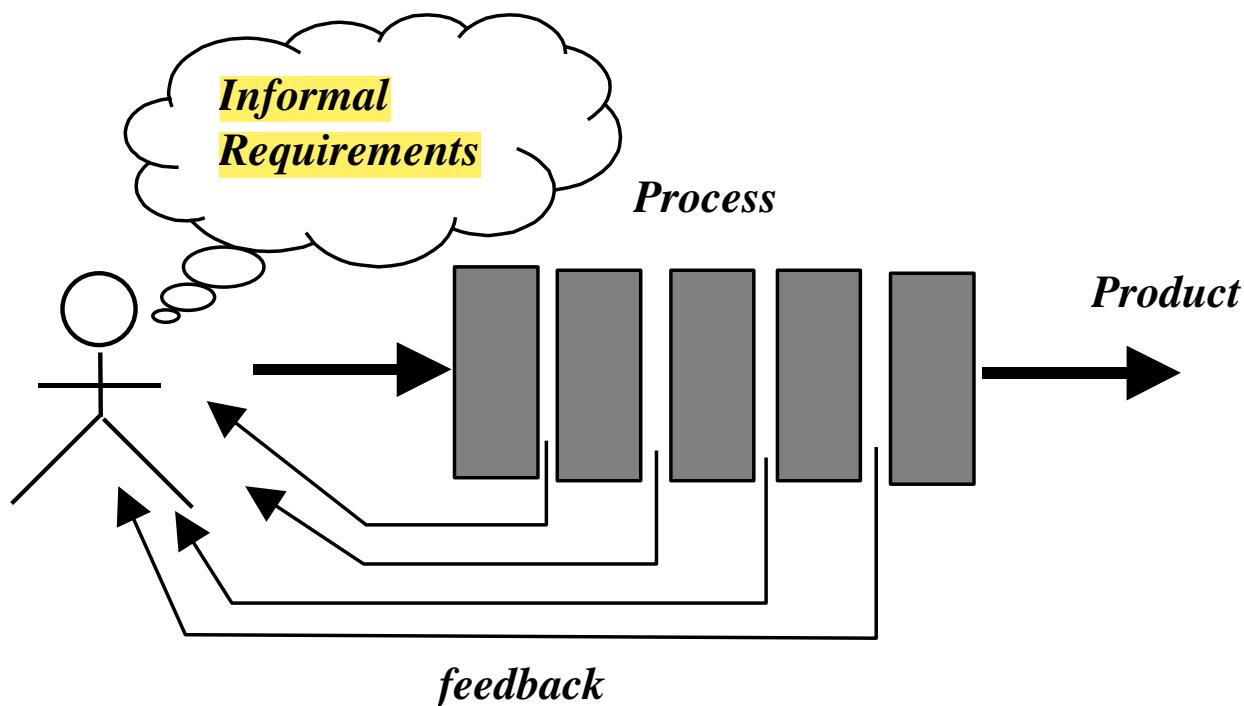


Uncertain /
Incomplete requirement
In the beginning

Problems

- The assumption is that requirements can be fully understood prior to development
- Interaction with the customer occurs only at the beginning (requirements) and end (after delivery)
- Unfortunately the assumption almost never holds

Process as a "white box"



Advantages

- Reduce risks by improving visibility
- Allow project changes as the project progresses
 - based on feedback from the customer

having become legally established or accepted by long usage or the passage of time

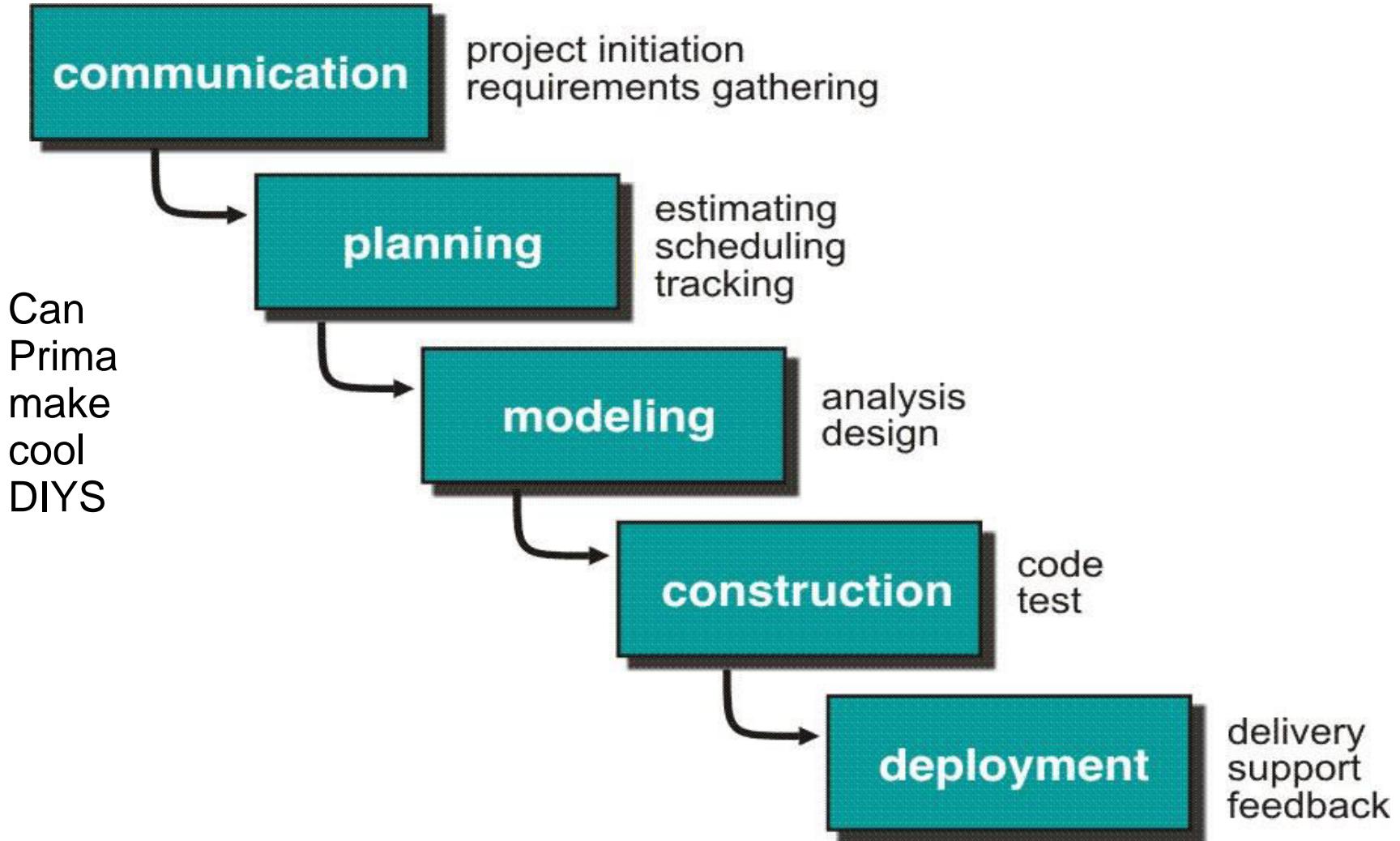
Prescriptive Model

- Prescriptive process models advocate an **orderly approach** to software engineering
 - **Organize framework** activities in a certain order
- **Process framework** activity with set of software engineering actions.
- Each action in terms of a task set that identifies the work to be accomplished to meet the goals.
- The resultant process model should be adapted to accommodate the nature of the specific project, people doing the work, and the work environment.
- Software engineer choose process framework that includes activities like;
 - **Communication**
 - **Planning**
 - **Modeling**
 - **Construction**
 - **Deployment**

Prescriptive Model

- Calling this model as “Prescribe” because it recommend a set of process elements, activities, action task, work product & quality.
- Each elements are inter related to one another (called workflow).

Waterfall Model or Classic Life Cycle



Waterfall Model or Classic Life Cycle

- Requirement Analysis and Definition: What - The systems **services, constraints and goals** are defined by customers with system users.
- Scheduling tracking -
 - **Evaluating progress** against the project plan.
 - Require action to maintain schedule.
- System and Software Design: How –It establishes and overall system architecture. Software design involves fundamental system **abstractions and their relationships**.
- Integration and system testing: The individual program unit or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.
- Operation and Maintenance: Normally this is the longest phase of the software life cycle. The system is installed and put into practical use. Maintenance **involves correcting errors** which were not discovered in earlier stages of the life-cycle.

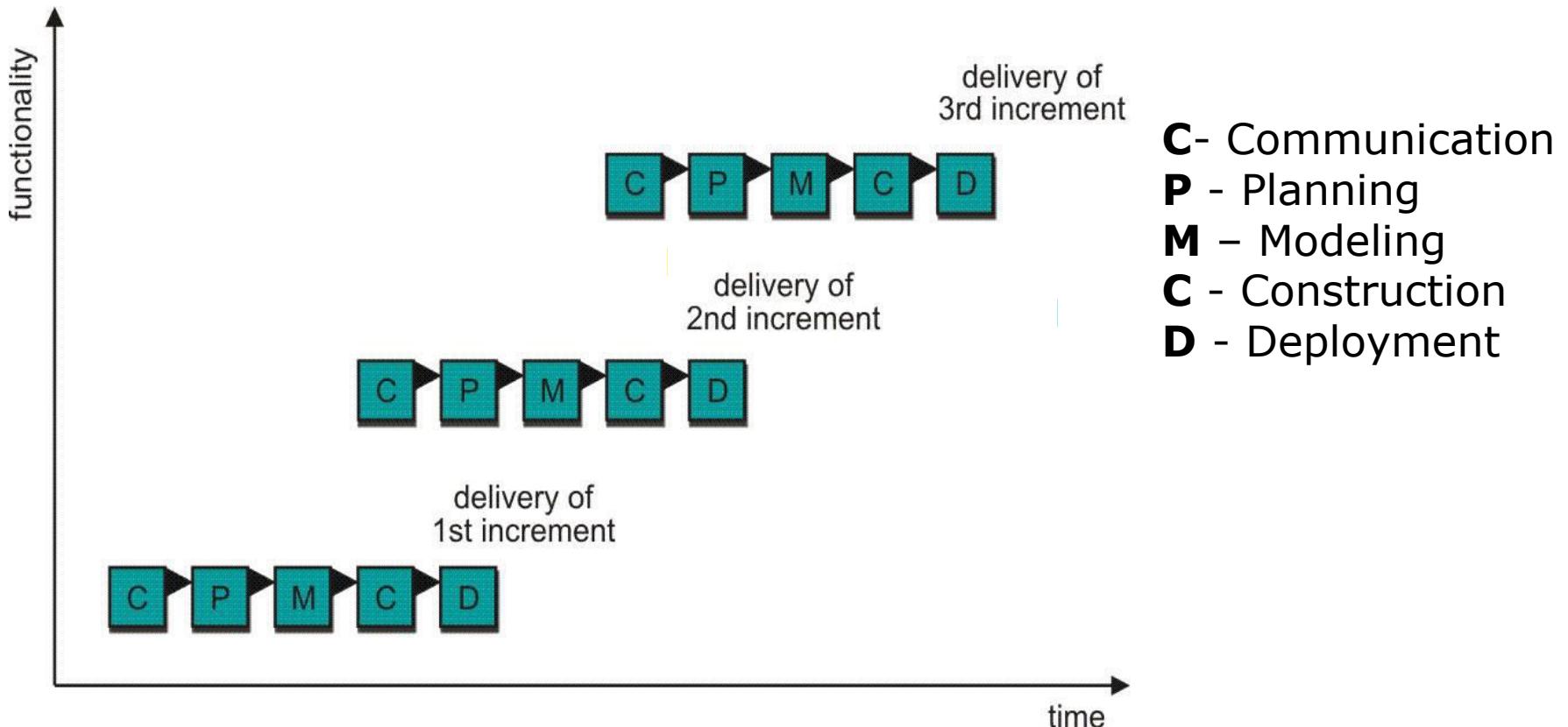
Limitations of the waterfall model

- ❑ The nature of the requirements will not change very much During development; during evolution
- ❑ The model implies that you should attempt to complete a given stage before moving on to the next stage
 - ❑ Does not account for the fact that requirements constantly change.
 - ❑ It also means that customers can not use anything until the entire system is complete.
- ❑ The model implies that once the product is finished, everything else is maintenance.
- ❑ Surprises at the end are very expensive
- ❑ Some teams sit ideal for other teams to finish
- ❑ Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.

Problems:

1. Real projects are rarely follow the sequential model.
2. Difficult for the customer to state all the requirement explicitly.
3. Assumes patience from customer - working version of program will not available until programs not getting change fully.

Incremental Process Model



Delivers software in small but usable pieces, each piece builds on pieces already delivered

The Incremental Model

- Rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality.
- First Increment is often core product
 - Includes basic requirement
 - Many supplementary features (known & unknown) remain undelivered
- A plan of next increment is prepared
 - Modifications of the first increment
 - Additional features of the first increment
- It is particularly useful when enough staffing is not available for the whole project
- Increment can be planned to manage technical risks.
- Incremental model focus more on delivery of operation product with each increment.

The Incremental Model

- User requirements are prioritised and the highest priority requirements are included in early increments.
- Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve.
- Customer value can be delivered with each increment so system functionality is available earlier.
- Early increments act as a prototype to help elicit requirements for later increments.
- Lower risk of overall project failure.
- The highest priority system services tend to receive the most testing.

Rational Unified Process

- Rational Unified Process, or RUP, is a configurable software development process platform that delivers practices and a configurable architecture
- RUP is a method of managing OO Software Development
- Enables the developers to select and deploy only the process components they need for each stage of their project

Rational Unified Process

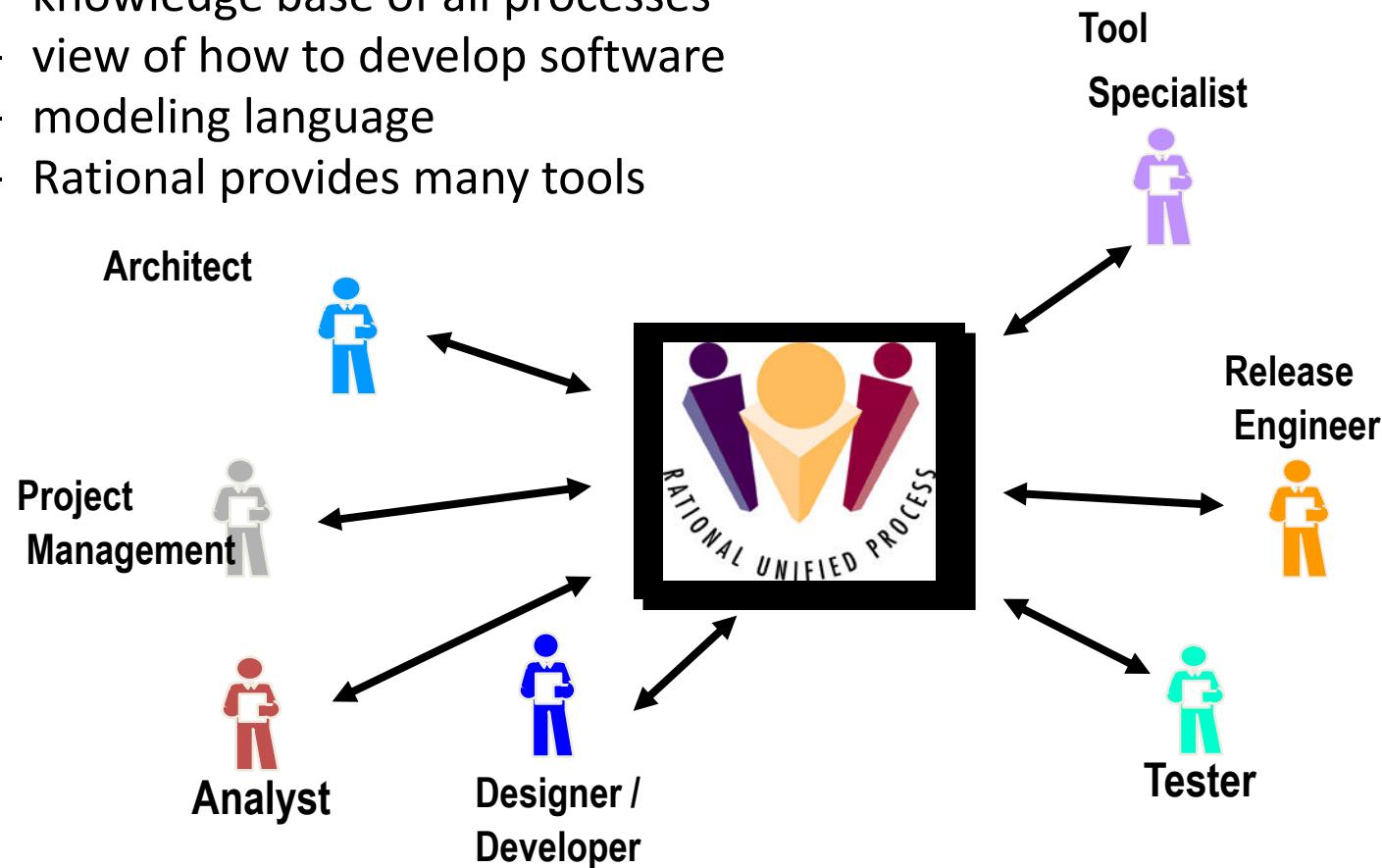
- It can be viewed as a Software Development Framework which is extensible and features:
 - Iterative Development
 - Requirements Management
 - Component-Based Architectural Vision
 - Visual Modeling of Systems
 - Quality Management
 - Change Control Management

- Team-Unifying Approach

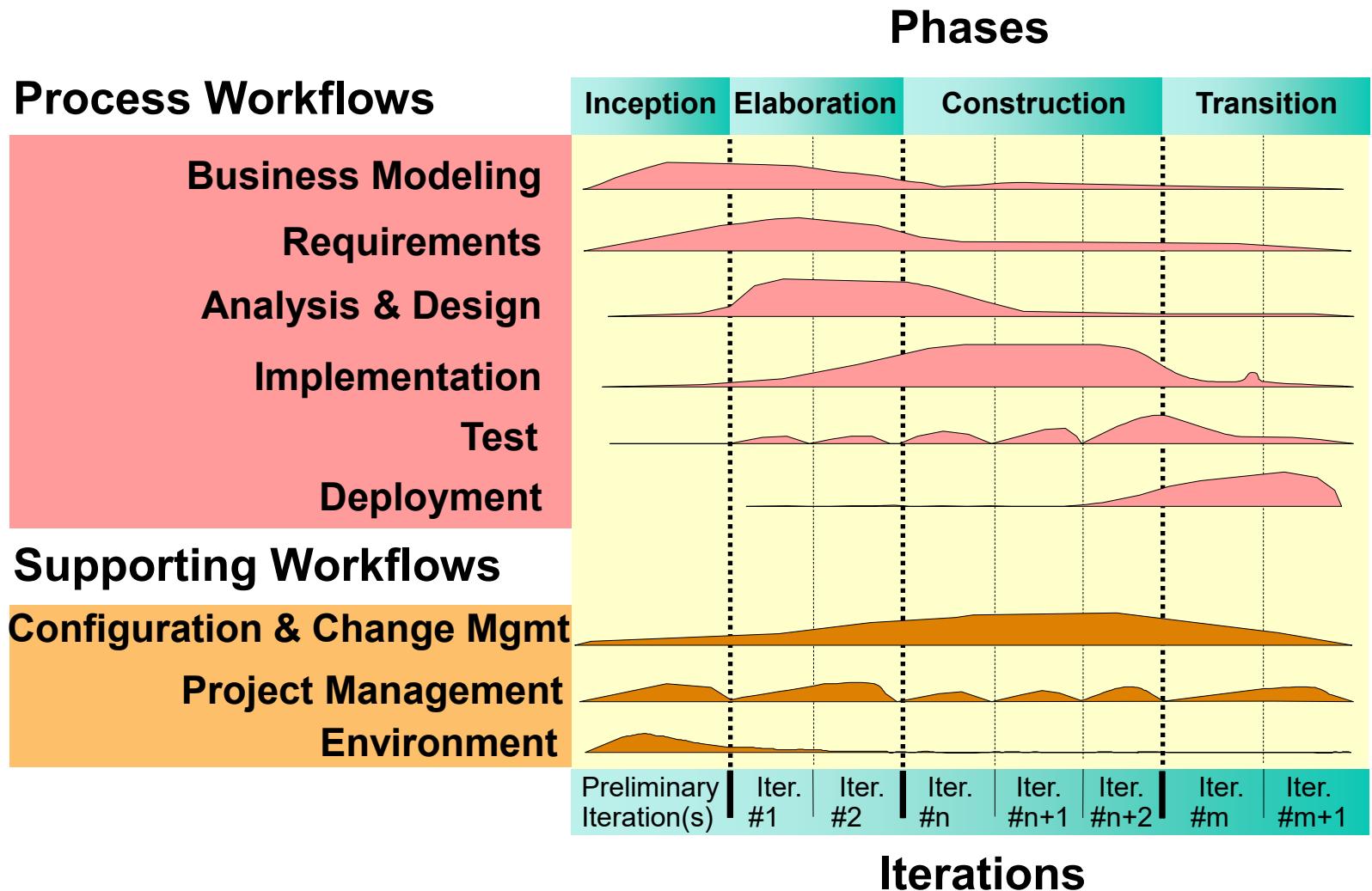
The RUP unifies a software team by providing a common view of the development process and a shared vision of a common goal

- Increased Team Productivity

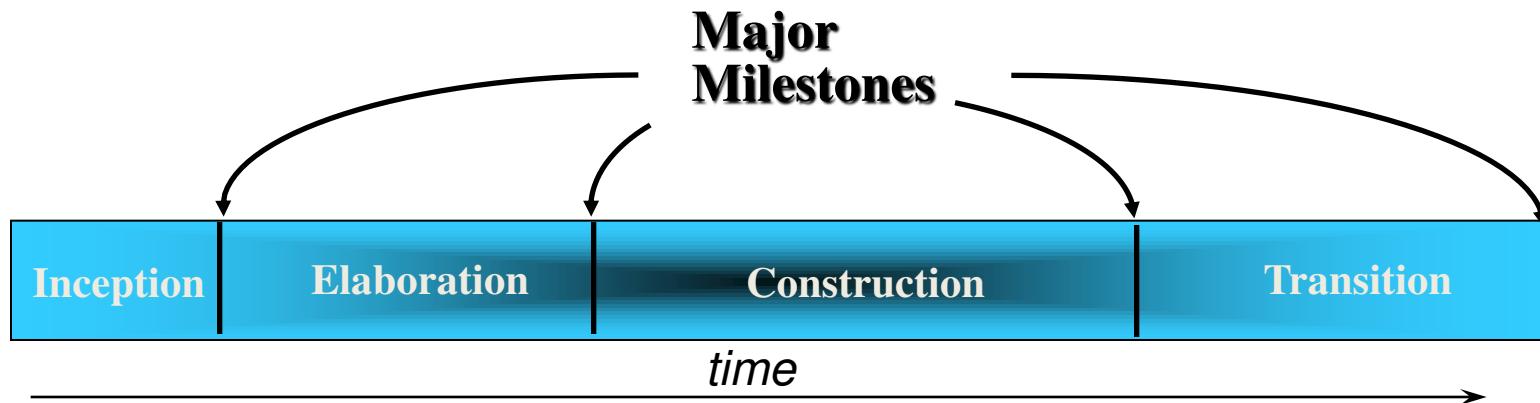
- knowledge base of all processes
- view of how to develop software
- modeling language
- Rational provides many tools



Rational Unified Process (RUP)



Phases in RUP



The Rational Unified Process has four phases:

- Inception - Define the scope of project
- Elaboration - Plan project, specify features, baseline architecture
- Construction - Build the product
- Transition - Transition the product into end user community

starting

Inception phase

- Establishing the project's software scope and boundary conditions, including an operational vision, acceptance criteria and what is intended to be in the product and what is not.
- Discriminating the critical use cases of the system, the primary scenarios of operation that will drive the major design tradeoffs.
- Exhibiting, and maybe demonstrating, at least one candidate architecture against some of the primary scenarios.
- Estimating the overall cost and schedule for the entire project (and more detailed estimates for the elaboration phase that will immediately follow).
- Estimating potential risks (the sources of unpredictability)
- Preparing the supporting environment for the project.

Explanation

Elaboration phase

- Defining, validating and baselining the architecture as rapidly as practical.
- Refining the Vision, based on new information obtained during the phase, establishing a solid understanding of the most critical use cases that drive the architectural and planning decisions.
- Creating and baselining detailed iteration plans for the construction phase.
- Refining the development case and putting in place the development environment, including the process, tools and automation support required to support the construction team.
- Refining the architecture and selecting components. Potential components are evaluated and the make/buy/reuse decisions sufficiently understood to determine the construction phase cost and schedule with confidence. The selected architectural components are integrated and assessed against the primary scenarios.

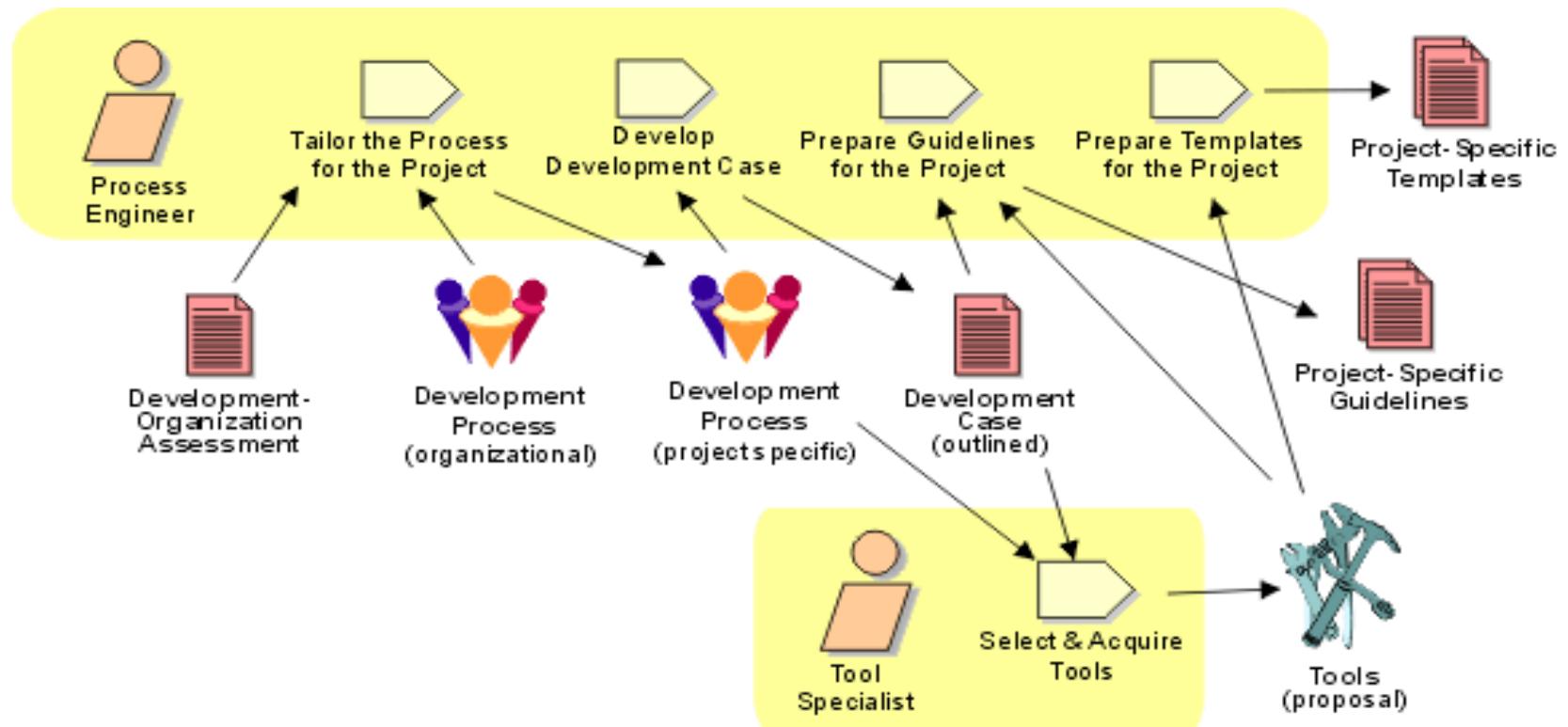
Construction phase

- Resource management, control and process optimization
- Complete component development and testing against the defined evaluation criteria
- Assessment of product releases against acceptance criteria for the vision.

Transition phase

- Executing deployment plans.
- Finalizing end-user support material.
- Testing the deliverable product at the development site.
- Creating a product release.
- Getting user feedback.
- Fine-tuning the product based on feedback.
- Making the product available to end users.

Workflow Detail: Prepare Environment for Project



Workflows - 3 key elements

- Three key elements of each workflows:
 - Artifacts
 - Roles
 - Activities

Agnes
Rinaldo
always

ARA

Artifacts

A piece of information that:

- Is produced, modified, or used by a process
- Defines an area of responsibility
- Is subject to version control.

An artifact can be a *model*, a *model element*, or a *document*. A document can enclose other documents.

Roles

- Represent a role that an individual may play on the project
- Responsible for producing artefacts
- Distinct from actors

Activities

- Tasks performed by people representing particular roles in order to produce artifacts

Brief summary of process workflows

- Business Modelling
- Requirements
- Analysis & Design
- Implementation
- Test
- Deployment

Both R And I Test Done

Business Modelling

- Understand **structure & dynamics** of organization in which system is to be deployed
- Understand **current problems** in the target organization & identify improvement potential
- Ensure customers, end users & developers have common understanding of target organisation
- Derive system requirements to support target organisation

Analysis & Design

- Transform requirements into a design of the system
- Evolve a robust architecture for the system
- Adapt design to match the implementation environment, designing it for performance

Implementation

- Define organization of the code, in terms of implementation subsystems organized in layers
- Implement classes & objects in terms of components
- Test developed components as units
- Integrate results into an executable system

Test

- Verify interaction between objects
- Verify proper integration of all components of the software
- Verify that all requirements have been correctly implemented
- Identify & ensure defects are addressed prior to deployment

Deployment

- Provide custom installation
- Provide shrink wrap product offering
- Provide software over internet

Brief summary of supporting workflows

- Configuration & Change Management
- Project Management
- Environment

Configuration & Change Management

- Supports development methods
- Maintains product integrity
- Ensures completeness & correctness of configured product
- Provides stable environment within which to develop product
- Restricts changes to artifacts based on project policies
- Provides an audit trail on why, when & by whom any artifact was changed

Project Management

- A framework for managing software-intensive projects
- Practical guidelines for planning, staffing, executing & monitoring projects
- A framework for managing risk

Environment

- Design, implement and manage the project's required technical environments
- Define the technical architectures for the development, system validation, testing & staging/release management environments
- When possible, standard architectural models for given types of platforms should be utilized when defining the production environment

Bringing It All Together...

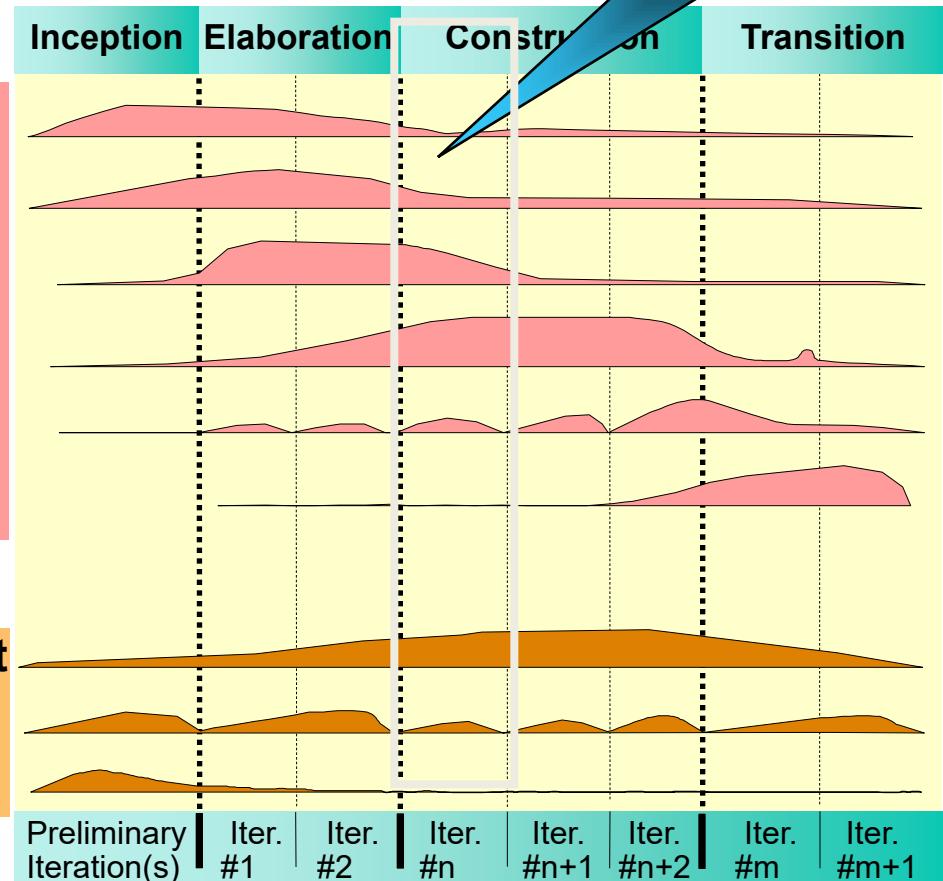
In an iteration, you walk through all workflows

Process Workflows

- Business Modeling
- Requirements
- Analysis & Design
- Implementation
- Test
- Deployment

Supporting Workflows

- Configuration & Change Mgmt
- Project Management
- Environment

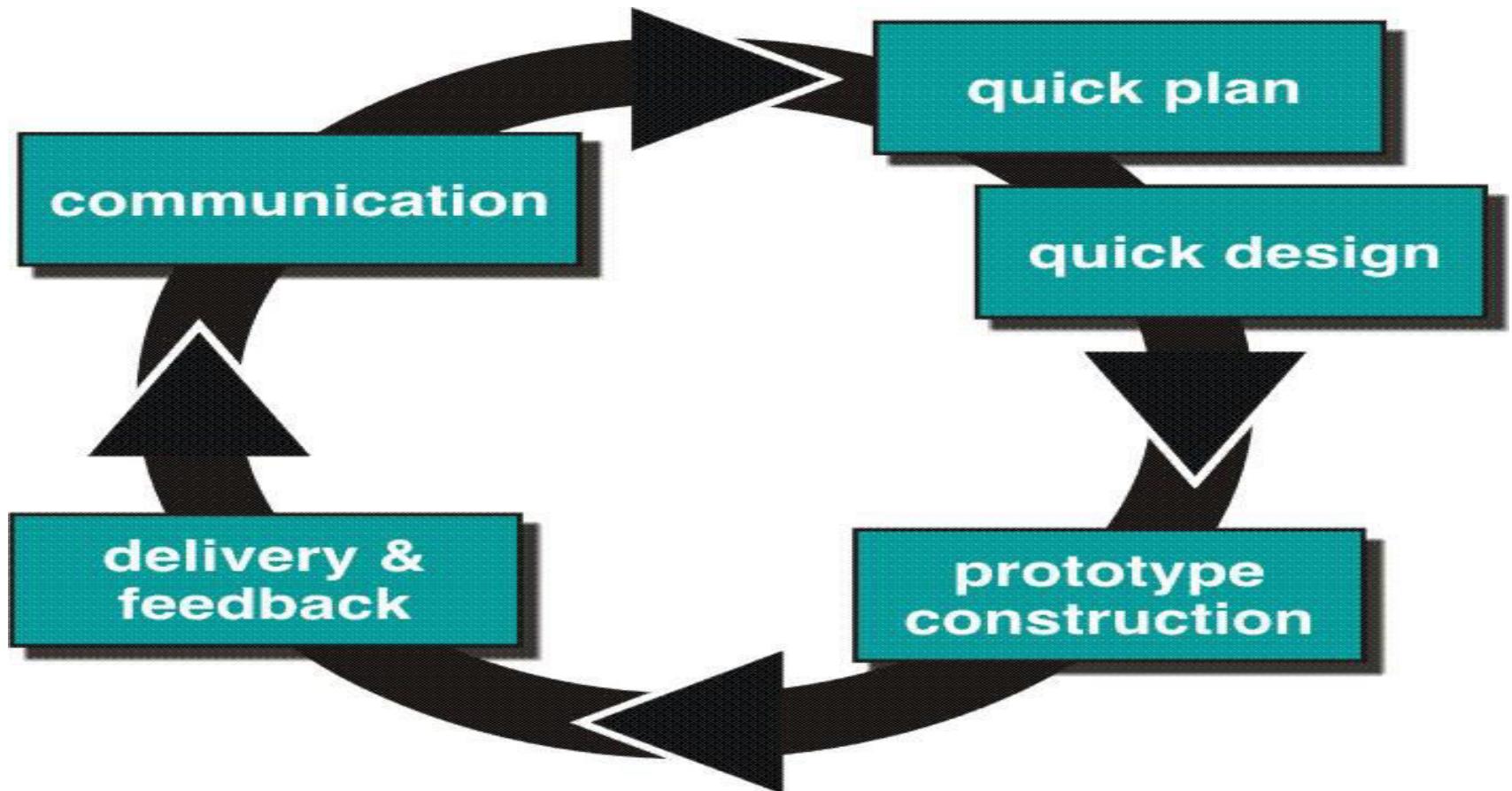


Iterations

Evolutionary Process Model

- Produce an increasingly more complete version of the software with each iteration.
- Evolutionary Models are iterative.
- Evolutionary models are:
 - Prototyping
 - Spiral Model
 - Concurrent Development Model
 - Fourth Generation Techniques (4GT)

Evolutionary Process Models : Prototyping



Prototyping cohesive sticking together

- **Best approach when:**
 - Objectives defines by customer are general but does not have details like input, processing, or output requirement.
 - Developer may be unsure of the efficiency of an algorithm, O.S., or the form that human machine interaction should take.
- It can be used as standalone process model.
- Model assist software engineer and customer to better understand what is to be built when requirement are fuzzy.
- Prototyping start with communication, between a customer and software engineer to define overall objective, identify requirements and make a boundary.
- Going ahead, planned quickly and modeling (software layout visible to the customers/end-user) occurs.
- Quick design leads to prototype construction.
- Prototype is deployed and evaluated by the customer/user.
- Feedback from customer/end user will refine requirement and that is how iteration occurs during prototype to satisfy the needs of the customer.

Prototyping (cont..)

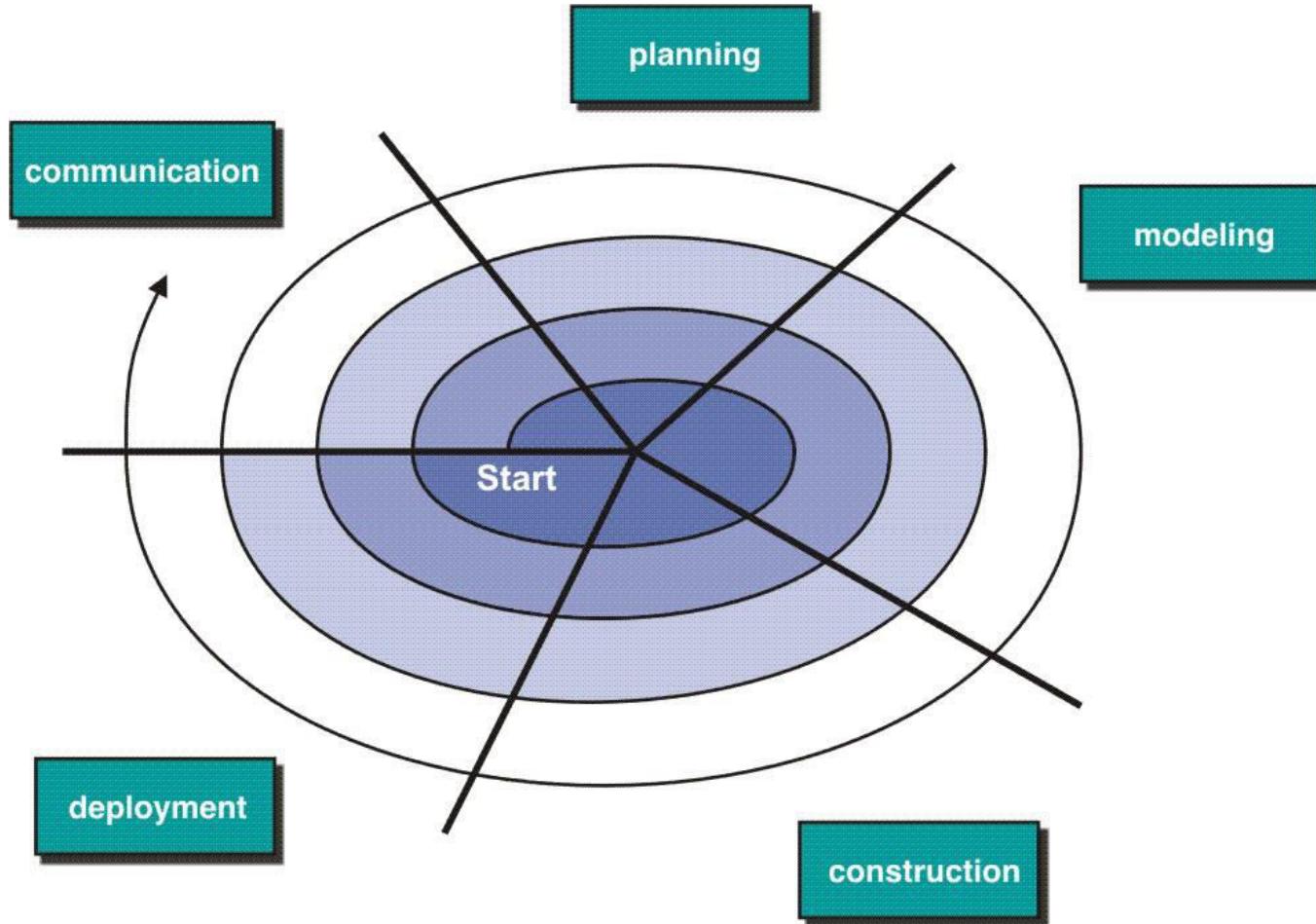
- Prototype can be serve as “**the first system**”.
- Both customers and developers like the prototyping paradigm.
 - Customer/End user gets a feel for the actual system
 - Developer get to build something immediately.

Problem Areas:

- Customer cries foul and demand that “a few fixes” be applied to make the prototype a working product, due to that software quality suffers as a result.
- Developer often makes implementation in order to get a prototype working quickly without considering other factors in mind like OS, Programming language, etc.

Customer and developer both must be agree that the prototype is built to serve as a mechanism for defining requirement.

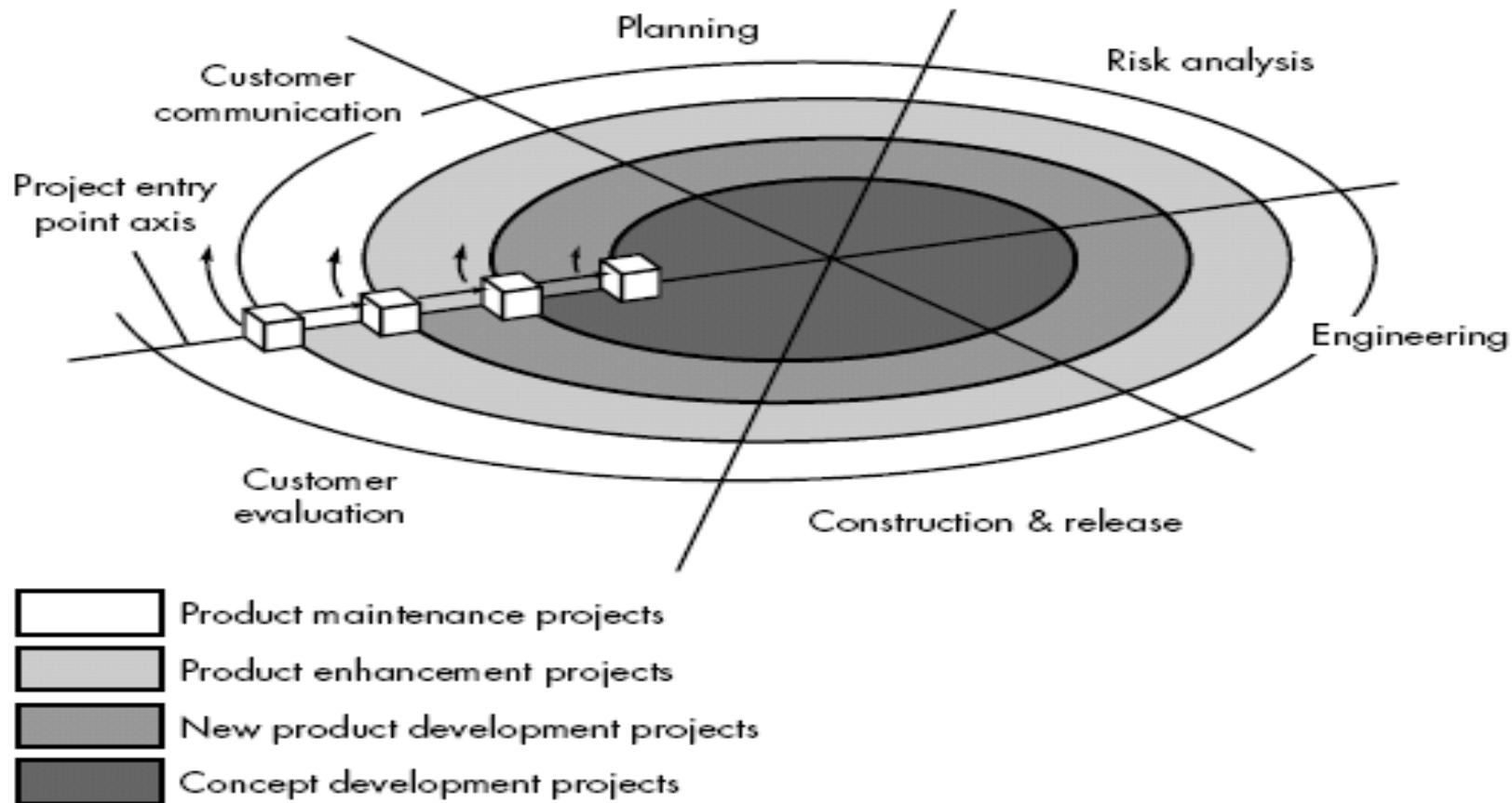
Evolutionary Model: Spiral Model



Spiral Model

- ❑ Couples iterative nature of prototyping with the controlled and systematic aspects of the linear sequential model
- It provide potential for rapid development of increasingly more complete version of the software.
- Using spiral, software developed in as series of evolutionary release.
 - Early iteration, release might be on paper or prototype.
 - Later iteration, more complete version of software.
- Divided into framework activities (C,P,M,C,D). Each activity represent one segment.
- Evolutionary process begins in a clockwise direction, beginning at the center risk.
- First circuit around the spiral might result in development of a product specification. Subsequently, develop a prototype and then progressively more sophisticated version of software.
- Unlike other process models that end when software is delivered.
- It can be adapted to apply throughout the life of software.

Spiral Model



Spiral Model (cont.)

Concept Development Project:

- Start at the core and continues for multiple iterations until it is complete.
- If concept is developed into an actual product, the process proceeds outward on the spiral.

New Product Development Project:

- New product will evolve through a number of iterations around the spiral.
- Later, a circuit around spiral might be used to represent a “Product Enhancement Project”

Product Enhancement Project:

- There are times when process is dormant or software team not developing new things but change is initiated, process start at appropriate entry point.

- Spiral models uses prototyping as a risk reduction mechanism but, more important, enables the developer to apply the prototyping approach at each stage in the evolution of the product.
- It maintains the systematic stepwise approach suggested by the classic life cycle but also incorporates it into an iterative framework activity.
- If risks cannot be resolved, project is immediately terminated

Problem Area:

- It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable.
- If a major risk is not uncovered and managed, problems will undoubtedly occur.

quick

Agile Process

- 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

- Is 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

Agility Principles

1. Our highest priority is to satisfy the customer through **early and continuous delivery of valuable software**.
2. Welcome **changing requirements, even late in development**. Agile processes harness change for the customer's competitive advantage.
3. **Deliver working software frequently**, from a couple of weeks to a couple of months, with a preference to the **shorter timescale**.
4. Business people and developers must **work together** daily throughout the project.
5. **Build projects around motivated individuals**. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is **face-to-face conversation**.

Agility Principles (cont.)

7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity – the art of maximizing the amount of work not done – is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Unified Process Model

- Unified Process is component-based, which means that the software system being built is made up of software components interconnected via well-defined interfaces in UML

Choosing a SDLC Model



Requirements
Understanding



Expected
lifeTime



RISK



Schedule
Constraints



Interaction with
management/customers

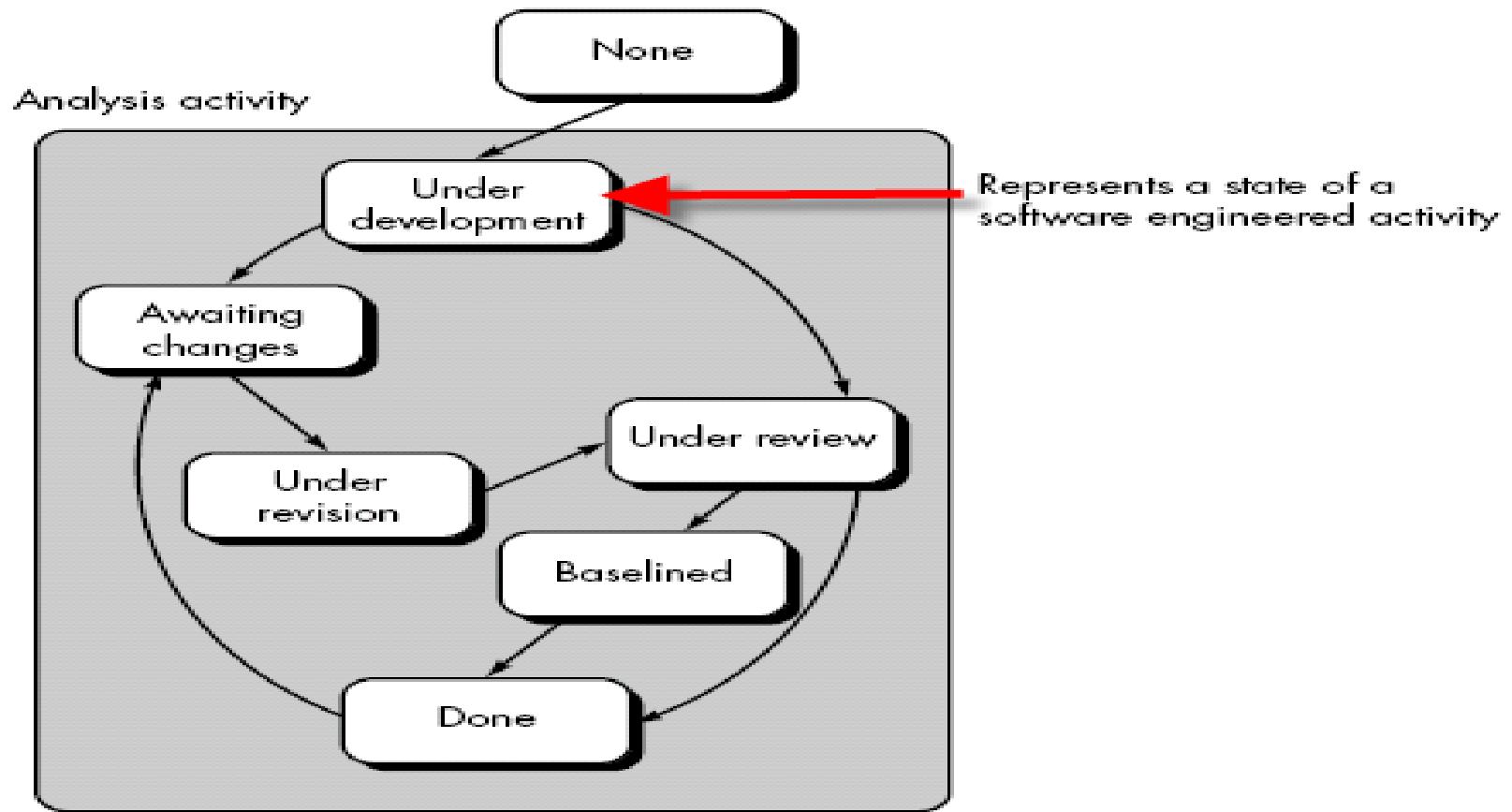


Expertise

Choosing a SDLC Model

- Is the SDLC suitable for the size of our team and their skills?
- Is the SDLC suitable for the selected technology we use for implementing the solution?
- Is the SDLC suitable for client and stakeholders concerns and priorities?
- Is the SDLC suitable for the geographical situation (distributed team)?
- Is the SDLC suitable for the size and complexity of our software?
- Is the SDLC suitable for the type of projects we do?
- Is the SDLC suitable for our software engineering capability?
- Is the SDLC suitable for the project risk and quality insurance?

Concurrent Development Model



Concurrent Development Model

- It represented schematically as series of major technical activities, tasks, and their associated states.
- It is often more appropriate for system engineering projects where different engineering teams are involved.
- The activity-modeling may be in any one of the states for a given time.
- All activities exist concurrently but reside in different states.

E.g.

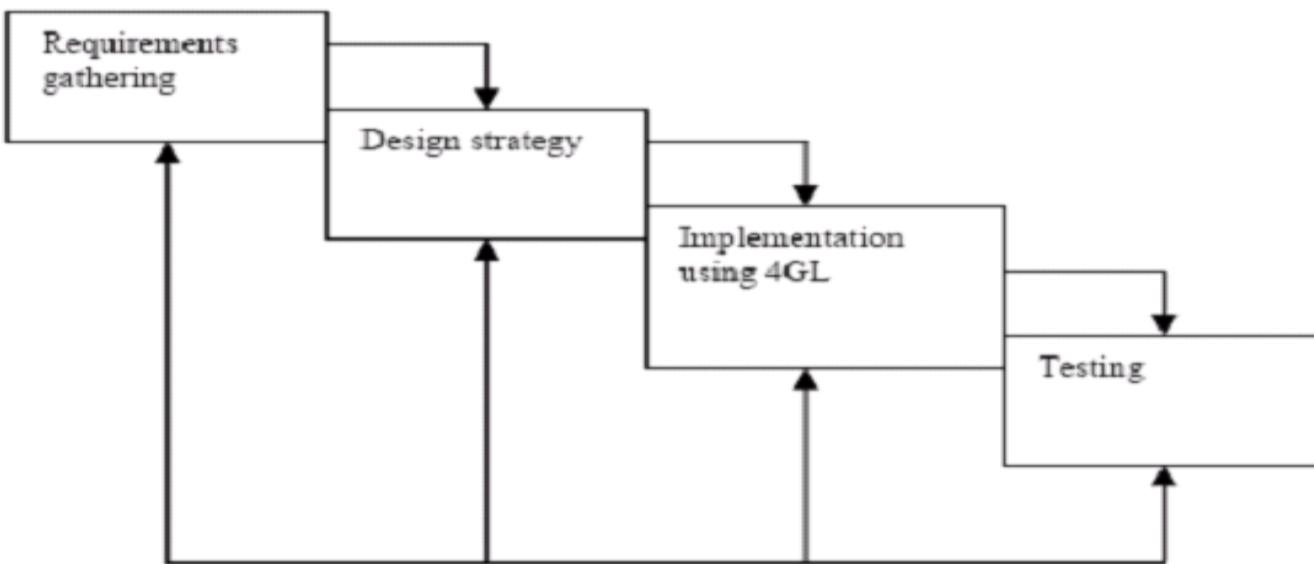
- The *analysis* activity (existed in the **none** state while initial customer communication was completed) now makes a transition into the **under development** state.
- *Analysis* activity moves from the **under development** state into the **awaiting changes** state only if customer indicates changes in requirements.
- Series of event will trigger transition from state to state.

E.g. During initial stage there was inconsistency in design which was uncovered. This will triggers the analysis action from the **Done** state into **Awaiting Changes** state.

Concurrent Development (Cont.)

- Visibility of current state of project
- It define network of activities
- Each activities, actions and tasks on the network exists simultaneously with other activities ,actions and tasks.
- Events generated at one point in the process network trigger transitions among the states.

Fourth Generation Techniques(4GT)



Fourth Generation Techniques

4GT

- Like all other models, 4GT begins with a requirements gathering phase.
- Ideally, the customer would describe the requirements, which are directly translated into an operational prototype.
- Practically, however, the client may be unsure of the requirements, may be ambiguous in his specs or may be unable to specify information in a manner that a 4GT tool can use.
- For small applications, it may be possible to move directly from the requirements gathering phase to the implementation phase using a nonprocedural fourth generation language.
- However for larger projects a design strategy is necessary. Otherwise, the same difficulties are likely to arise as with conventional approaches.

4GT

- To transform a 4GT implementation into a product, the developer must conduct thorough testing, develop meaningful documentation.
- In addition, the 4GT developed software must be built in a manner that enables maintenance to be performed quickly.

Merits:

- Dramatic reduction in software development time. (For small and intermediate application)
- Improved productivity for software developers.

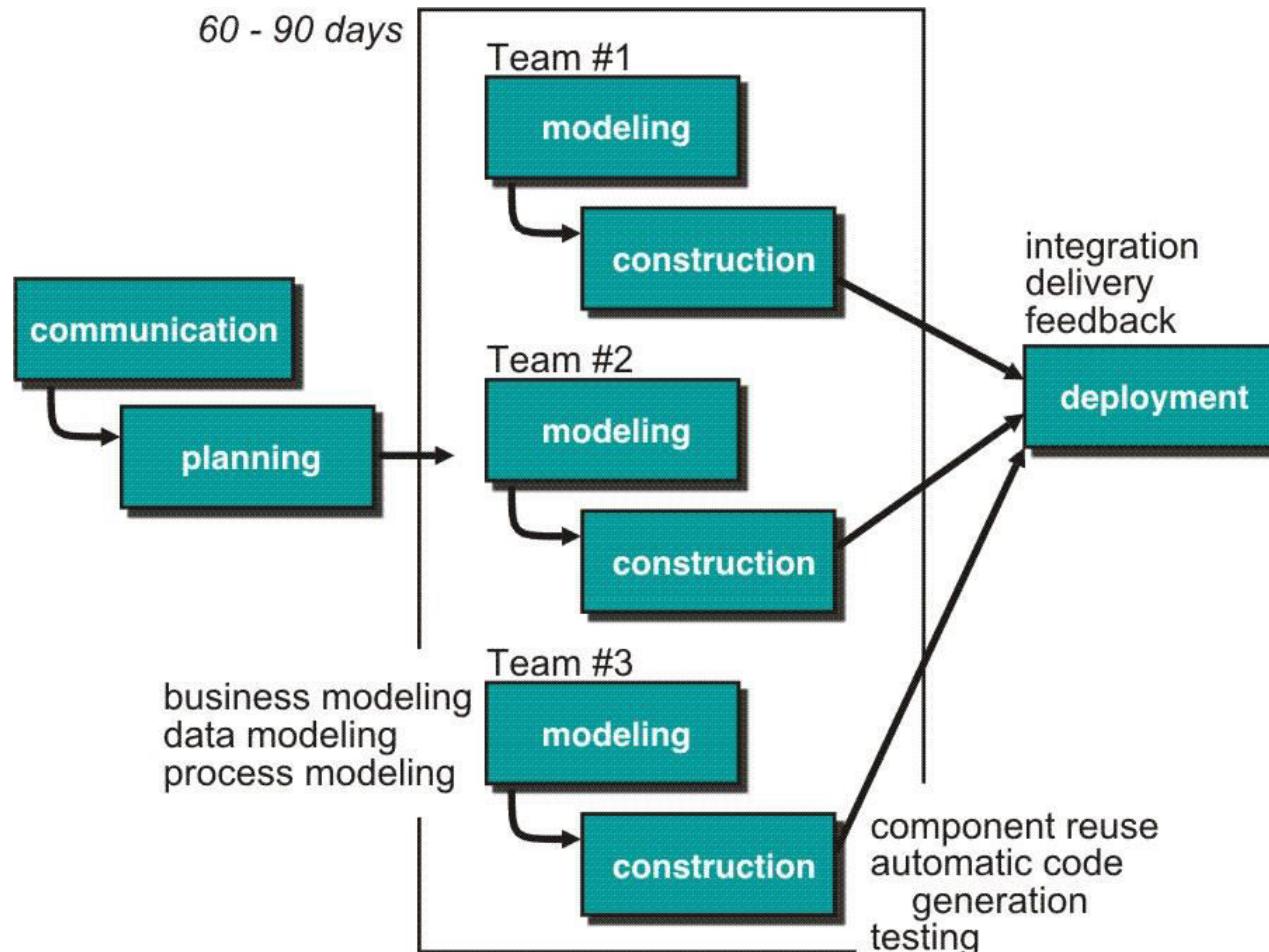
Demerits:

- Not much easier to use as compared to programming languages
- The maintainability of large software systems built using 4GT is open to question.

4GT

- 4GT Software tool is used to generate the source code for a software system from a high level specification representation
- Commonly used 4GT in development models are mentioned below:
 - Report Generation
 - Data base query language
 - Data Manipulation
 - Screen definition and interaction
 - Code Generation
 - Web engineering Tools
 - high-level graphics

Rapid Application Development (RAD) Model



Makes heavy use of reusable software components with an extremely short development cycle

RAD model

- **Communication** – to understand business problem.
- **Planning** – multiple s/w teams works in parallel on diff. system.
- **Modeling** –
 - **Business modeling** – Information flow among business is working.
Ex. What kind of information drives?
 - Who is going to generate information?
 - From where information comes and goes?
 - **Data modeling** – Information refine into set of data objects that are needed to support business.
 - **Process modeling** – Data object transforms to information flow necessary to implement business.

- **Construction** — it highlighting the use of pre-existing software component.
- **Deployment** — Deliver to customer basis for subsequent iteration.
- RAD model emphasize a **short development cycle**.
- “High speed” edition **of linear sequential model**.
- If requirement are well understood and project scope is constrained then it enable development team to create “ fully functional system” within a very short time period.

RAD Model

- If application is **modularized** (“Scalable Scope”), each major function to be completed in less than three months.
- Each major function can be addressed by a separate team and then integrated to form a whole.

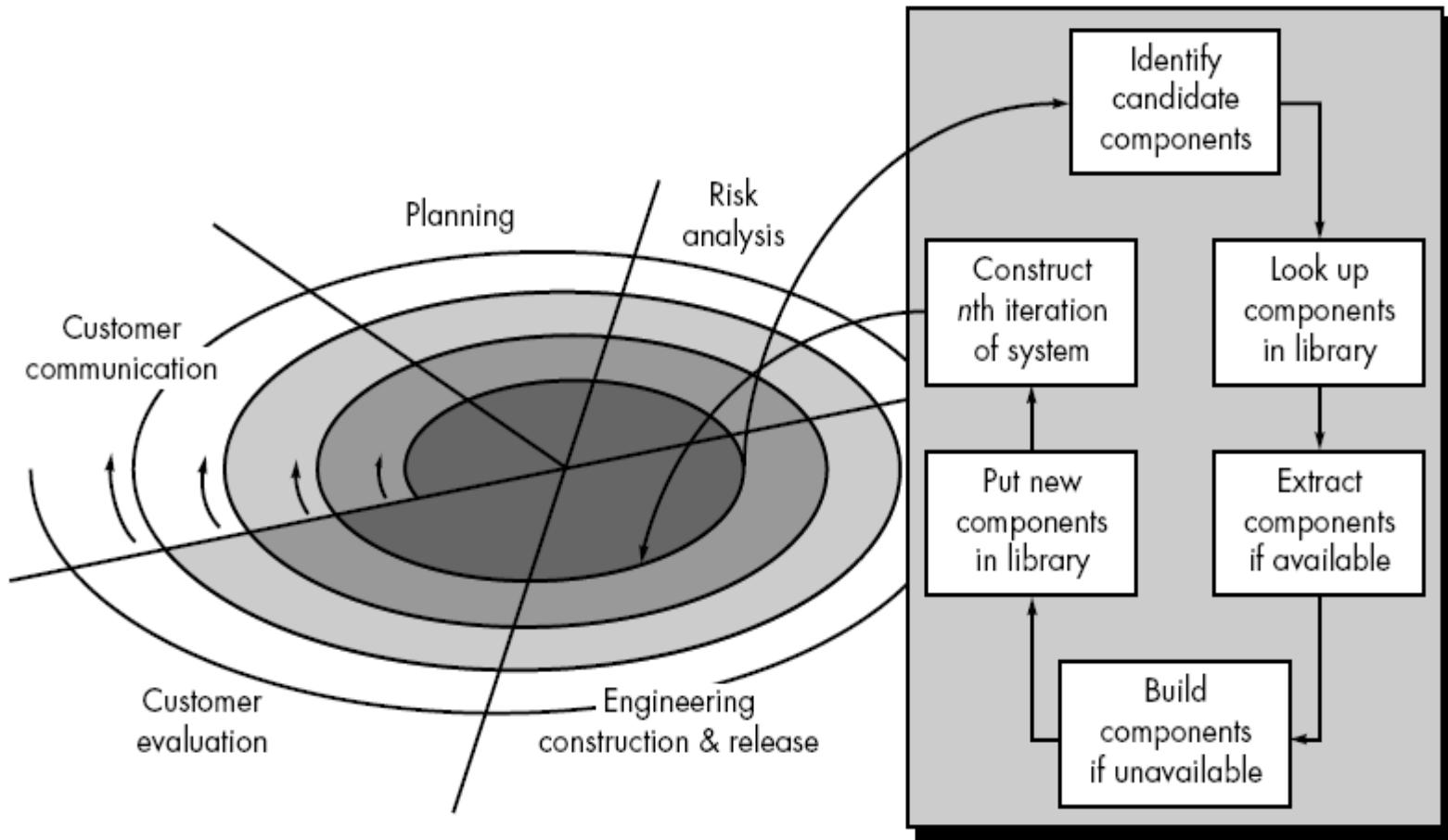
Drawback:

- For large but scalable projects
 - RAD requires sufficient human resources
- Projects fail if developers and customers are not committed in a much shortened time-frame
- Problematic if system can not be modularized
- Not appropriate when technical risks are high (heavy use of new technology)

Component Based Development

- component-based development (CBD) model incorporates many of the characteristics of the spiral model.
- It is **evolutionary by nature** and iterative approach to create software.
- CBD model creates applications from **prepackaged software components** (called *classes*).

CBD Model



CBD model (cont.)

- Modeling and construction activities begin with identification of candidate components.
- Classes created in past software engineering projects are stored in a class library or repository.
- Once candidate classes are identified, the class library is searched to determine if these classes already exist.
- If class is already available in library extract and reuse it.
- If class is not available in library, it is engineered or developed using **object-oriented methods**.
- Any new classes built to meet the unique needs of the application.
- Now process flow return to the spiral activity.

CBD model (cont.)

- CBD model leads to software reusability.
- Based on studies, CBD model leads to 70 % reduction in development cycle time.
- 84% reduction in project cost.
- Productivity is very high.

Software Engineering

Lecture 03 **Requirement Engineering**

Requirements Engineering

- **Requirement:** A function, constraint or other property that the system must provide to fill the needs of the system's intended user(s)
- **Engineering:** implies that systematic and repeatable techniques should be used
- **Requirement Engineering** means that requirements for a product are defined, managed and tested systematically

Requirements Engineering

- It is essential that the software engineering team **understand the requirements of a problem** before the team tries to solve the problem.
- In some cases requirements engineering may **be abbreviated**, but it is **never abandoned**.
- RE is software engineering actions that start with **communication activity** and continues into the **modeling activity**.
- RE establishes a **solid base for design and construction**. Without it, resulting software has a high probability of not meeting customer needs.

Characteristics of a Good Requirement

- **Clear and Unambiguous**
 - standard structure
 - has only one possible interpretation
 - Not more than one requirement in one sentence
- **Correct**
 - A requirement contributes to a real need
- **Understandable**
 - A reader can easily understand the meaning of the requirement
- **Verifiable**
 - A requirement can be tested
- **Complete**
- **Consistent**
- **Traceable**

CUCUVCCT

Why is Getting Good Requirements Hard?

- Stakeholders don't know what they really want.
- Stakeholders express requirements in their own terms.
- Different stakeholders may have conflicting requirements.
- Organisational and political factors may influence the system requirements.
- The requirements change during the RE process.
New stakeholders may emerge and the business environment change.

Types of Requirements

Functional requirements

- Statements of **services the system should provide**, how the system should **react to particular inputs** and how the system should **behave in particular situations**.
- May state **what the system should not do**.

Non-functional requirements

- Constraints on the services** or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
- Often apply to **the system as a whole** rather than individual features or services.

Domain requirements

- Constraints on the system **from the domain of operation**

Functional Requirements

- Describe functionality or system services.
- Depend on the type of software, expected users and the type of system where the software is used.
- Functional user requirements may be high-level statements of what the system should do.
- Functional system requirements should describe the system services in detail.
- Essentially, these are the ‘what’s’ of the system that we often refer to. These are not ‘all that there is,’ but these should describe the overall functionality of the system.

Non-functional Requirements

- These define system properties and constraints e.g. **reliability, response time, maintainability, scalability, portability, and storage requirements.**
- Constraints are I/O device capability, system representations, etc.
- Process requirements may also be specified mandating a particular IDE, programming language or development method.
- (Often internal to an organization or required for fit / compatibility with other comparable systems.)
- Non-functional requirements may **be more critical than functional requirements**. If these are **not met**, the system may be useless.

Non-functional Requirements Implementation

- ❑ Non-functional requirements **may affect the overall architecture of a system** rather than the individual components.
 - ❑ For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.
- ❑ A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that **define system services that are required**.
 - ❑ It **may also generate requirements that restrict existing requirements**.

Metrics for specifying nonfunctional requirements

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure (MTTF) Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure (MTTR) Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

User requirements

- Statements in natural language plus diagrams of the services the system provides and its operational constraints.
- Written for customers.

System requirements

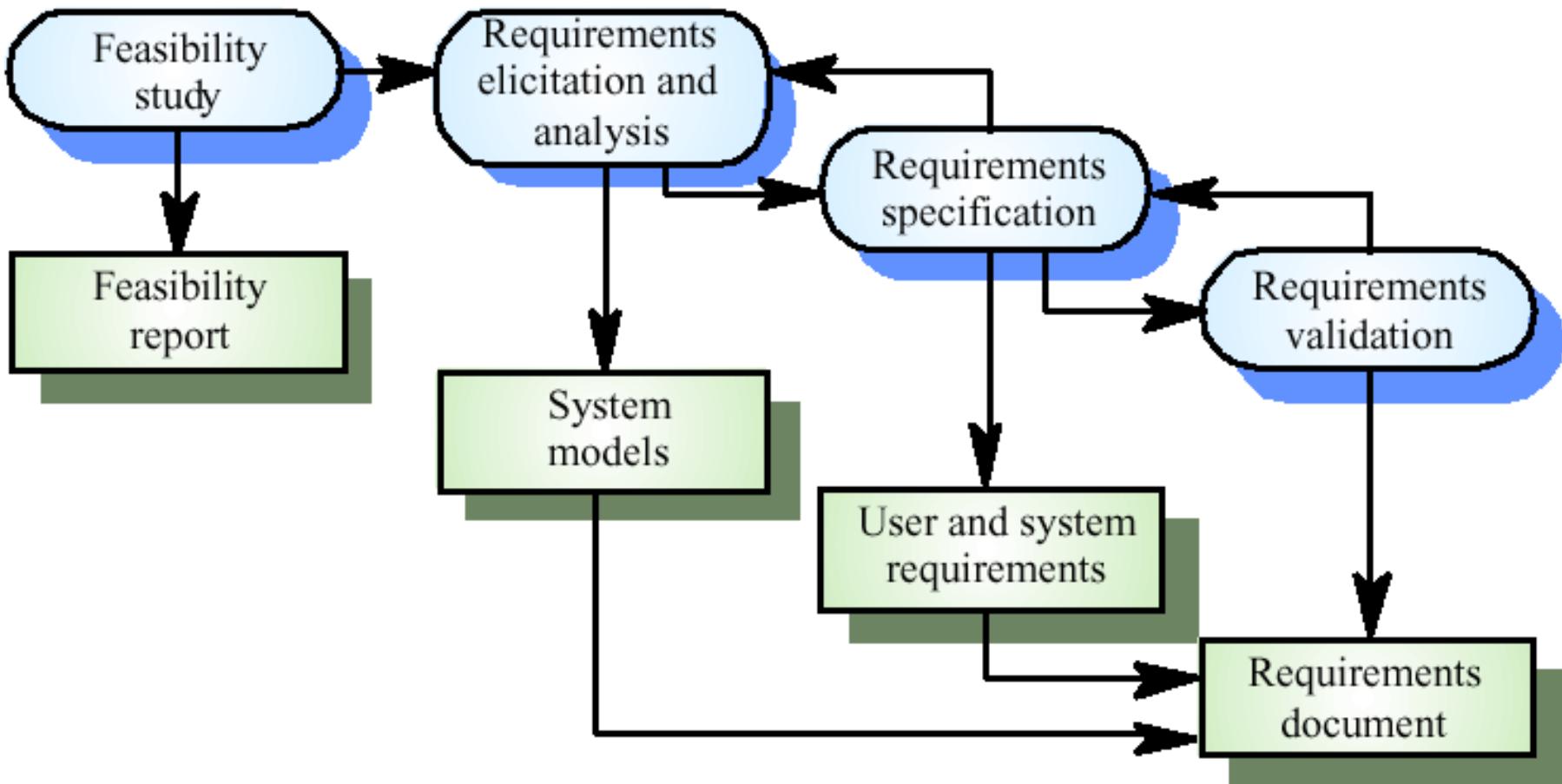
- A structured document setting out detailed descriptions of the system's functions, services and operational constraints.
- Defines what should be implemented so may be part of a contract between client and contractor.
- Whom do you think these are written for?
- These are higher level than functional and non-functional requirements, which these may subsume.

Requirements engineering processes

- Requirements elicitation
- Requirements analysis
- Requirements validation
- Requirements management

E
A
V
M

The requirements engineering process



Requirements Engineering Tasks

- Inception —Establish a basic understanding of the problem and the nature of the solution.
- Elicitation —Draw out the requirements from stakeholders.
- Elaboration (Highly structured)—Create an analysis model that represents information, functional, and behavioral aspects of the requirements.
- Negotiation—Agree on a deliverable system that is realistic for developers and customers.
- Specification—Describe the requirements formally or informally.
- Validation —Review the requirement specification for errors, ambiguities, omissions, and conflicts.
- Requirements management —Manage changing requirements.

Inception

- Inception— Ask “context-free” questions that establish ...
 - Basic understanding of the problem
 - The people who want a solution
 - The nature of the solution that is desired, and
 - The effectiveness of preliminary communication and collaboration between the customer and the developer

Elicitation

- Elicitation - elicit requirements from customers, users and others.
 - Find out from customers, users and others what the product objectives are
 - what is to be done
 - how the product fits into business needs, and
 - how the product is used on a day to day basis

Why Requirement elicitation is difficult?

- Problems of scope:
 - The boundary of the system is ill-defined.
 - Customers/users specify unnecessary technical detail that may confuse rather than clarify objectives.
- Problem of understanding:
 - Customers are not completely sure of what is needed.
 - Customers have a poor understanding of the capabilities and limitations of the computing environment.
 - Customers don't have a full understanding of their problem domain.
 - Customers have trouble communicating needs to the system engineer.
 - Customers omit detail that is believed to be obvious.
 - Customers specify requirements that conflict with other requirements.
 - Customers specify requirements that are ambiguous or not able to test.
- Problems of volatility:
 - Requirement change over time.

Initiating Requirements Engineering Process

- **Identify stakeholders**
 - Stakeholder can be “anyone who benefits in a direct or indirect way from the system which is being developed”
Ex. Business manager, project manager, marketing people, software engineer, support engineer, end-users, internal-external customers, consultants, maintenance engineer.
 - Each one of them has different view of the system.
- **Recognize multiple points of view**
 - Marketing group concern about feature and function to excite potential market. To sell easily in the market.
 - Business manager concern about feature built within budget and will be ready to meet market.
 - End user – Easy to learn and use.
 - SE – product functioning at various infrastructure support.
 - Support engineer – Maintainability of software.

Role of RE is to categorize all stakeholder information in a way that there could be no inconsistent or conflict requirement with one another

Initiating Requirements Engineering Process (cont.)

- **Work toward collaboration**
 - RE identify areas of commonality (i.e. Agreed requirement) and areas of conflict or inconsistency.
 - It does not mean requirement defined by committee. It may happened they providing just view of their requirement.
 - Business manager or senior technologist may make final decision.
- **Asking the first questions**
 - 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?

These questions will help – stakeholder interest in the software & measurable benefit of successful implementation.

Asking the question

Next set of questions – better understanding of the problem.

- What business problem (s) will this solution address?
- Describe business environment in which the solution will be used?
- Will performance or productivity issues affect the solution is approached?

Final set of questions – Effectiveness of communication

- Are my questions relevant to the problem?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

Eliciting Requirement

Approach for eliciting requirement:

- Collaborative Requirement Gathering
- Quality Function Deployment
- User Scenarios
- Elicitation Work Products

Collaborative Requirement Gathering

- Meetings are attended by all interested stakeholders.
- Rules established for preparation and participation.
- Agenda should be formal enough to cover all important points, but informal enough to encourage the free flow of ideas.
- A facilitator controls the meeting.
- A definition mechanism (blackboard, flip charts, etc.) is used.
- During the meeting:
 - The problem is identified.
 - Elements of the solution are proposed.
 - Different approaches are negotiated.
 - A preliminary set of solution requirements are obtained.
 - The atmosphere is collaborative and non-threatening.
- Flow of event – Outline the sequence of events occurs
 - Requirement gathering meeting (initial meeting)
 - During meeting
 - Follow the meeting.

Collaborative requirement gathering (contd.)

- In initial meeting, distribute “Product request” (defined by stakeholder) to all attendee.
- Based on product request, each attendee is asked to make
 - List of objects (Internal or external system objects)
 - List of services(Processes or functions)
 - List of constraints (cost, size, business rules) and performance criteria(speed, accuracy) are developed.
- Collect lists from everyone and combined.
- Combined list eliminates redundant entries, add new ideas , but does not delete anything.
- Objective is to develop a consensus list in each topic area (objects, services, constraints and performance).
- Based on lists, team is divided into smaller sub-teams : each works to develop mini-specification for one or more entries on each of the lists.

Collaborative requirement gathering (Contd.)

- Each sub-team presents its mini-specification to all attendees for discussion. Addition, deletion and further elaboration are made.
- Now each team makes a list of validation criteria for the product and present to team.
- Finally, one or more participants is assigned the task of writing a complete draft specification.

Quality Function Deployment

- It is a technique that translates the needs of the customer into technical requirements for software.
- Concentrates on maximizing customer satisfaction.
- QFD emphasizes – what is valuable to the customer and then deploys these values throughout the engineering process.

Three types of requirement:

1. Normal Requirements – reflect objectives and goals stated for product. If requirements are present in final products, customer is satisfied.
2. Expected Requirements – customer does not explicitly state them. Customer assumes it is implicitly available with the system.
3. Exciting Requirements- Features that go beyond the customer's expectation.

During meeting with customer –

Function deployment determines the “value” of each function required of the system.

Information deployment identifies data objects and events and also ties with functions.

Task deployment examines the behavior of the system.

Value analysis determines the priority of requirements during these 3 deployments.

User Scenario

- It is difficult to move into more software engineering activities until s/w team understands how these functions and features will be used by diff. end-users.
- Developers and users create a set of usage threads for the system to be constructed
- A use-case scenario is a story about how someone or something external to the software (known as an **actor**) interacts with the system.
- Describe how the system will be used
- Each scenario is described from the point-of-view of an “actor”—a person or device that interacts with the software in some way

Elicitation Work Products

Elicitation work product will vary depending upon the size of the system or product to be built.

- Statement of **need** and **feasibility**.
- Statement of **scope**.
- List of **participants** in requirements elicitation.
- Description of the system's technical **environment**.
- List of **requirements** and associated domain **constraints**.
- List of usage **scenarios**.
- Any **prototypes** developed to refine requirements.

Elaboration

- Focuses on developing a refined technical model of software functions, features, and constraints using the information obtained during inception and elicitation
- Create an analysis model that identifies data, function and behavioral requirements.
- It is driven by the creation and refinement of user scenarios that describe how the end-user will interact with the system.
- Each event parsed into extracted.
- End result defines informational, functional and behavioral domain of the problem

Negotiation

- Negotiation - agree on a deliverable system that is realistic for developers and customers
 - Requirements are categorized and organized into subsets
 - Relations among requirements identified
 - Requirements reviewed for correctness
 - Requirements prioritized based on customer needs
 - Negotiation about requirements, project cost and project timeline.
 - There should be no winner and no loser in effective negotiation.

Specification

- Specification – Different things to different people.
- It can be –
 - Written Document
 - A set of graphical models,
 - A formal mathematical models
 - Collection of usage scenario.
 - A prototype
 - Combination of above.
- The Formality and format of a specification varies with the size and the complexity of the software to be built.
- For large systems, written document, language descriptions, and graphical models may be the best approach.
- For small systems or products, usage scenarios

Specification Principles

- May be viewed as representation process.
1. Separate functionality from implementation.
 2. Develop a model of the desired behavior of a system.
 3. Establish the context in which software operates by specifying the manner.
 4. Define the environment in which the system operates and indicate how.

Specification Principles (cont.)

5. Create a cognitive model rather than a design or implementation model.
The cognitive model describes a system as perceived by its user community.
6. The specifications must be tolerant of incompleteness and augmentable.
7. Establish the content and structure of a specification in a way that will enable it to be amenable to change.

open to

Specification Representation

- **Representation format and content should be relevant to the problem.**

- For example, a specification for a manufacturing automation system might use different symbology, diagrams and language than the specification for a programming language compiler.

- **Information contained within the specification should be nested (layered).**

- Paragraph and diagram numbering schemes should indicate the level of detail that is being presented.
 - It is sometimes worthwhile to present the same information at different levels of abstraction to aid in understanding.

- **Diagrams and other notational forms should be restricted in number and consistent in use.**

- Confusing or inconsistent notation, whether graphical or symbolic, degrades understanding and fosters errors.

- **Representations should be revisable.**

SRS

Software Requirements Specification

- It contains a complete information description, a detailed functional description, a representation of system behavior, an indication of performance requirements and design constraints, appropriate validation criteria, and other information pertinent to requirements.

Format of SRS:

Introduction of the software requirements specification states the goals and objectives of the software, **describing it in the context** of the computer-based system.

Information content, flow, and structure are documented. Hardware, software, and human interfaces are described for external system elements and internal software functions.

Functional Description A processing **narrative** is provided for each function, design constraints are stated and justified & performance characteristics are stated

Behavioral Description **operation** of the software as a **consequence of** external events and internally generated control characteristics.

Software Requirements Specification (Cont.)

Validation Criteria is probably the most important and, ironically, the most often neglected section of the *Software Requirements Specification (SRS)*. **Testing or validating each user-scenario.**

Finally, the specification includes a **Bibliography and Appendix**. The *bibliography* contains references to all documents that relate to the software. The *appendix* contains information that supplements the specifications

Validation

- Requirements Validation - formal technical review mechanism that looks for
 - Errors in content or interpretation
 - Areas where clarification may be required
 - Missing information
 - Inconsistencies (a major problem when large products or systems are engineered)
 - Conflicting or unrealistic (unachievable) requirements.

Requirements validation techniques

- Requirements reviews
 - Systematic manual analysis of the requirements
- Prototyping
 - Using an executable model of the system to check requirements.
- Test-case generation
 - Developing tests for requirements to check testability
- Automated consistency analysis
 - Checking the consistency of a structured requirements description

Requirements Review

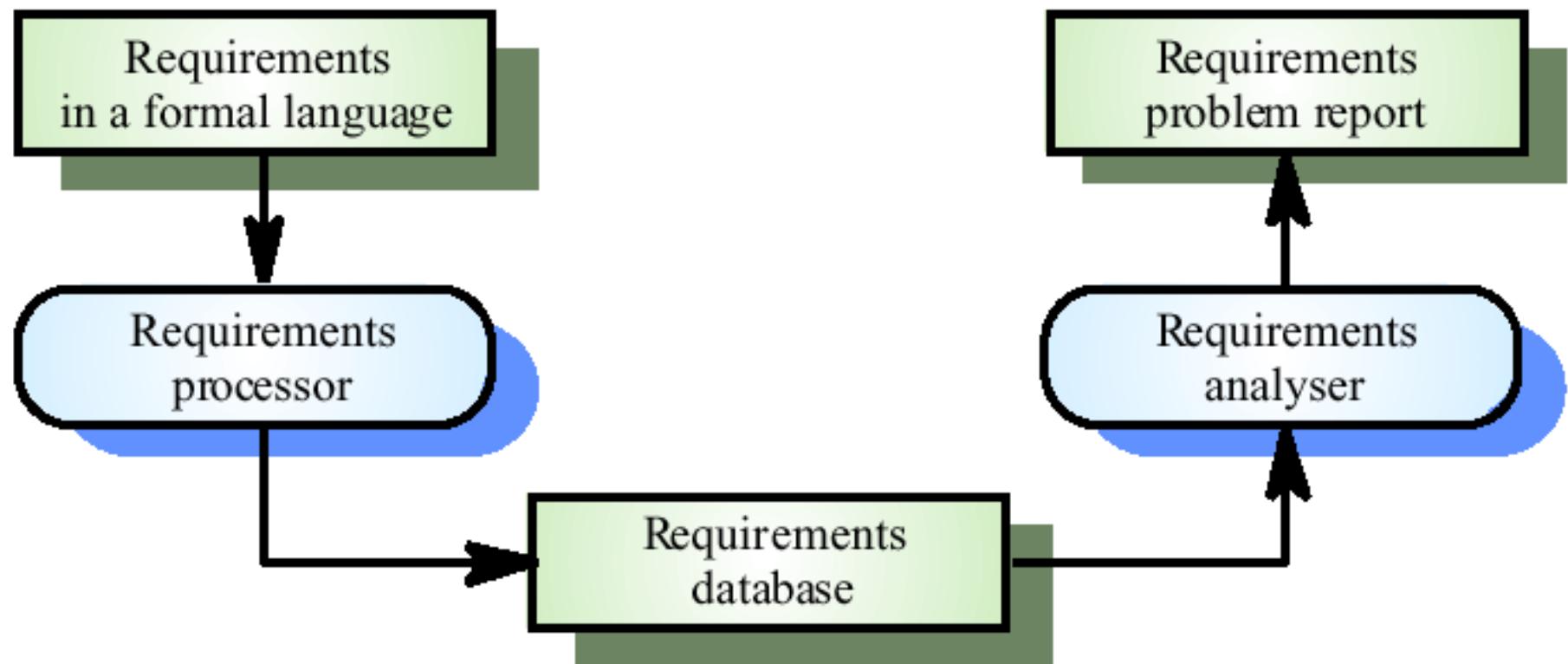
- A review of the SRS (and/or prototype) is conducted by both the software developer and the customer.
- Conducted at a macroscopic level
 - Ensure that specification is complete
 - Consistent
 - Accurate (Information, functional and behavioral domain considered).
- Review becomes more detailed while examining Information, functional and behavioral domain.
- Examining not only broad descriptions but the way in which requirement worded.

E.g. Terms like “Vague ” (some, sometimes, often, usually) should be flag by reviewer for further clarification.

Requirements Review (cont.)

- Once review is complete – SRS “signed off” by both customer and developer. (“contract” for software development)
- Requests for changes in requirements after the specification is finalized will not be eliminated.
- Change is an extension of software scope and therefore can increase cost and/or delivery of product.
- During the review, changes to the specification may be recommended.
- It can be extremely difficult to assess the global impact of a change; that is, how a change in one function affects requirements for other functions

Automated consistency checking



Requirement Management

- Set of activities that help project team to identify, control, and track requirements and changes as project proceeds
- Requirements begin with identification. Each requirement is assigned a unique identifier. Once requirements have been identified, traceability tables are developed.

Traceability Table:

- **Features traceability table** - shows how requirements relate to customer observable features
- **Source traceability table** - identifies source of each requirement
- **Dependency traceability table** - indicates relations among requirements
- **Subsystem traceability table** - requirements categorized by subsystem
- **Interface traceability table** - shows requirement relations to internal and external interfaces

It will help to track, if change in one requirement will affect different aspects of the system.

Software Engineering

Lecture 04 System Design

Design Engineering

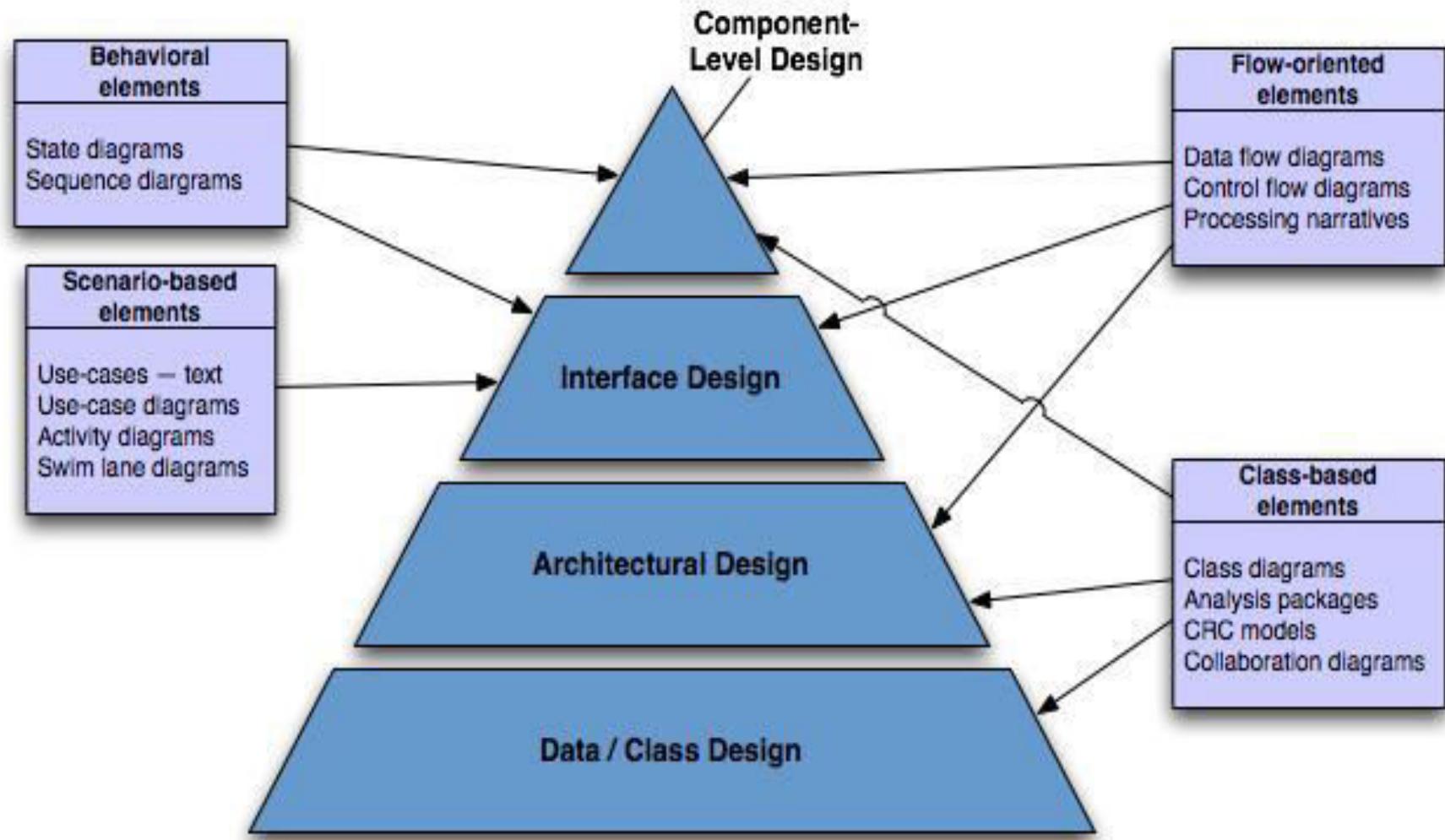
- It covers the set of principles, concepts, and practices that lead to the development of a high quality system or product.
- Goal of design engineering is to produce a model or representation that depict:
 - Firmness – program should not have any bug that inhibits its functions.
 - Commodity – suitable to its intended use.
 - Delight - pleasurable to use
- The design model provides detail about software data structures, architecture, interfaces, and components that are necessary to implement the system.

Software Design

- Software design model consists of 4 designs:
 - Data/class Design Arch
 - Architectural Design Comp
 - Interface Design Inter
 - Component Design Data

ACID

Translating Analysis → Design



- **Data/class design** - Created by transforming the analysis model **class-based elements** into **classes and data structures** required to implement the software
- **Architectural design** - defines the **relationships** among **the major structural elements** of the software, it is derived from the class-based elements and flow-oriented elements of the analysis model

- **Interface design** - describes how the software elements, hardware elements, and end-users communicate with one another, it is derived from the analysis model scenario-based elements, flow-oriented elements, and behavioral elements
- **Component-level design** - created by transforming the structural elements defined by the software architecture into a procedural description of the software components using information obtained from the analysis model class-based elements, flow-oriented elements, and behavioral elements

Why design is so important?

- It is place where quality is fostered.
- It provides us with representation of software that can be assessed for quality.
- Only way that can accurately translate a customer's requirements into a finished software product.
- It serves as foundation for all software engineering activities.
- Without design difficult to assess:
 - Risk
 - Test
 - Quality

Design Process and Design Quality

- S/w design is an iterative process through which requirements are translated into a “blueprint” for constructing the s/w.
- As design iteration occur, subsequent refinement leads to design representation at much lower levels of abstraction.

Goal of design process

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

Quality Guidelines

Characteristics of good design

- A design should exhibit an architecture that
 - as been created using recognizable architectural styles or patterns,
 - is composed of components that exhibit good design characteristics and
 - can be implemented in an evolutionary fashion
- A design should be modular; that is, the software should be logically partitioned into elements or subsystems
- A design should contain distinct representations of data, architecture, interfaces, and components.
- A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
- A design should lead to components that exhibit independent functional characteristics

Quality Guidelines (contd.)

- A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
- A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
- A design should be represented using a notation that effectively communicates its meaning.

Design Principles

- S/W design is both a process and a model.
- *Design process* - sequence of steps that enable the designer to describe all aspects of the software to be built.
- *Design model* - created for software provides a variety of different views of the computer software

Design Principles (cont.)

- The design process should not suffer from ‘tunnel vision.’ - Designer should consider alternative approaches.
- The design should be traceable to the analysis model - a single element of the design model often traces to multiple requirements, it is necessary to have a means for tracking how requirements have been satisfied by the design model.
- The design should not reinvent the wheel- use already exists design pattern because time is short and resource are limited.
- The design should “minimize the intellectual distance” between the software and the problem as it exists in the real world. – design should be self-explanatory

Design Principles (cont.)

- The design should exhibit uniformity and integration – before design work begins rules of styles and format should be defined for a design team.
- The design should be structured to accommodate change
- The design should be structured to degrade gently, even when unusual data, events, or operating conditions are encountered.
- Design is not coding, coding is not design.
- The design should be assessed for quality as it is being created, not after the fact
- The design should be reviewed to minimize conceptual (semantic) errors.

Design concepts

- Design concepts provide the necessary framework for “to get the thing on right way”.
 - Abstraction
 - Refinement
 - Modularity
 - Architecture
 - Control Hierarchy
 - Structural Partitioning
 - Data Structure
 - Software Procedure
 - Information Hiding

AMRA C-I-D-SS

Abstraction

- At the highest level of abstraction – a solution is stated in broad terms
- At lower level of abstraction – a more detailed description of the solution is provided.
- Two types of abstraction:
- Procedural abstraction: Sequence of instructions that have a specific and limited function.

Ex. Open a door

open implies long sequence of activities (e.g. walk to the door, grasp knob, turn knob and pull the door, etc).

- Data abstraction: collection of data that describes a data object.

Ex. Open a door. – **door** is data object.

- Data abstraction for **door** would encompass a set of attributes that describe the door. (E.g. door type, swing direction, opening mechanism, etc.)

Refinement

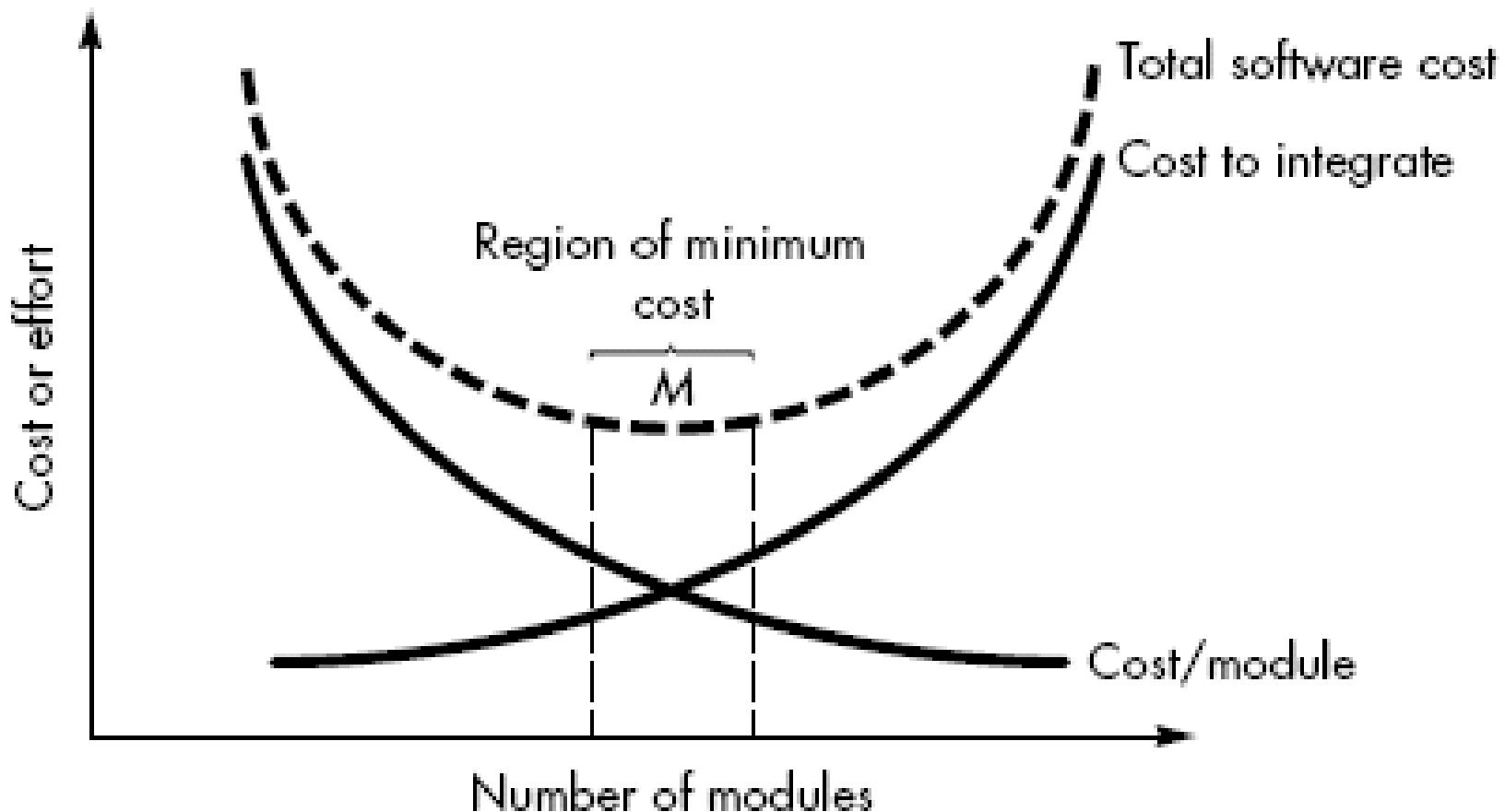
- Refinement is actually a process of *elaboration*.
- begin with a statement of function (or description of information) that is defined at a high level of abstraction.
- That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information.
- Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.
- Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low-level details.
- Refinement helps the designer to expose low-level details as design progresses.

Modularity

- Architecture and design pattern embody modularity.
- Software is divided into separately named and addressable components, sometimes called modules, which are integrated to satisfy problem requirement.
- modularity is the single attribute of software that allows a program to be intellectually manageable
- It leads to a “divide and conquer” strategy. – it is easier to solve a complex problem when you break into a manageable pieces.
- Refer fig. that state that effort (cost) to develop an individual software module does decrease if total number of modules increase.
- However as the no. of modules grows, the effort (cost) associated with integrating the modules also grows.



Modularity and software cost



- Under-modularity and over-modularity should be avoided. But how do we know the vicinity of M?
- We modularize a design so that development can be more easily planned.
- Software increments can be defined and delivered.
- Changes can be more easily accommodated.
- Testing and debugging can be conducted more efficiently and long-term maintained can be conducted without serious side effects.

Architecture

- Software architecture suggest “ the overall structure of the software and the ways in which that structure provides conceptual integrity for a system.

Architecture Representation

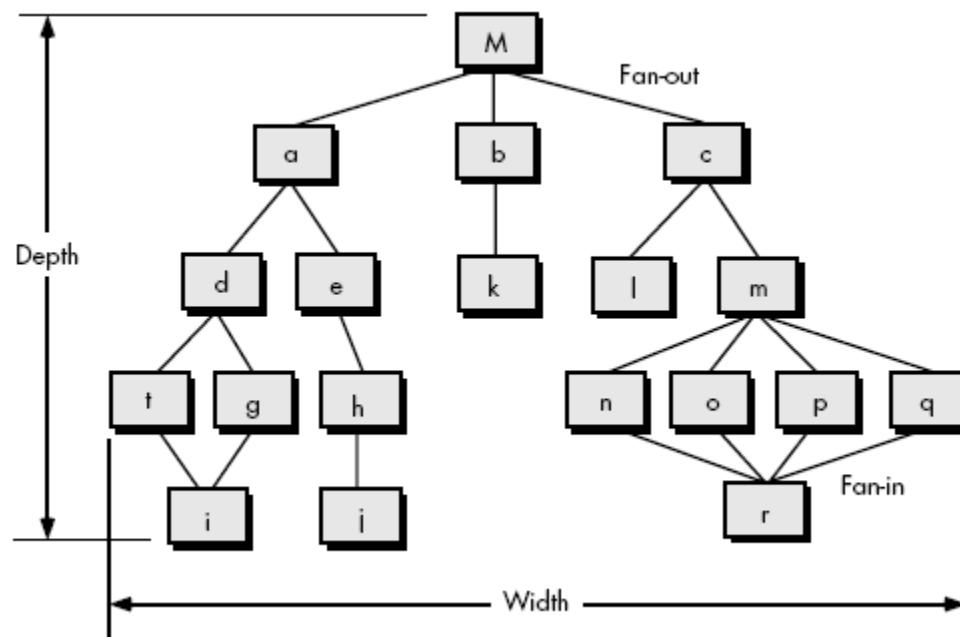
SPDFF

- **Structural Model** – represent architecture as an organized collection of components
- **Framework model** – Increase level of design abstraction by identifying repeatable architectural design framework.
- **Dynamic model** – address behavior of the program architecture and indicating how system or structure configuration may change as a function.
- **Process Model** – focus on design of the business or technical process that the system must accommodate.
- **Functional models** – used to represent the functional hierarchy of a system.

Control Hierarchy

- *Control hierarchy*, represents the organization of program components (modules) and implies a hierarchy of control.
- Different notations are used to represent control hierarchy for those architectural styles that are amenable to this representation.
- The most common is the treelike diagram that represents hierarchical control for call and return architectures.

Control Hierarchy

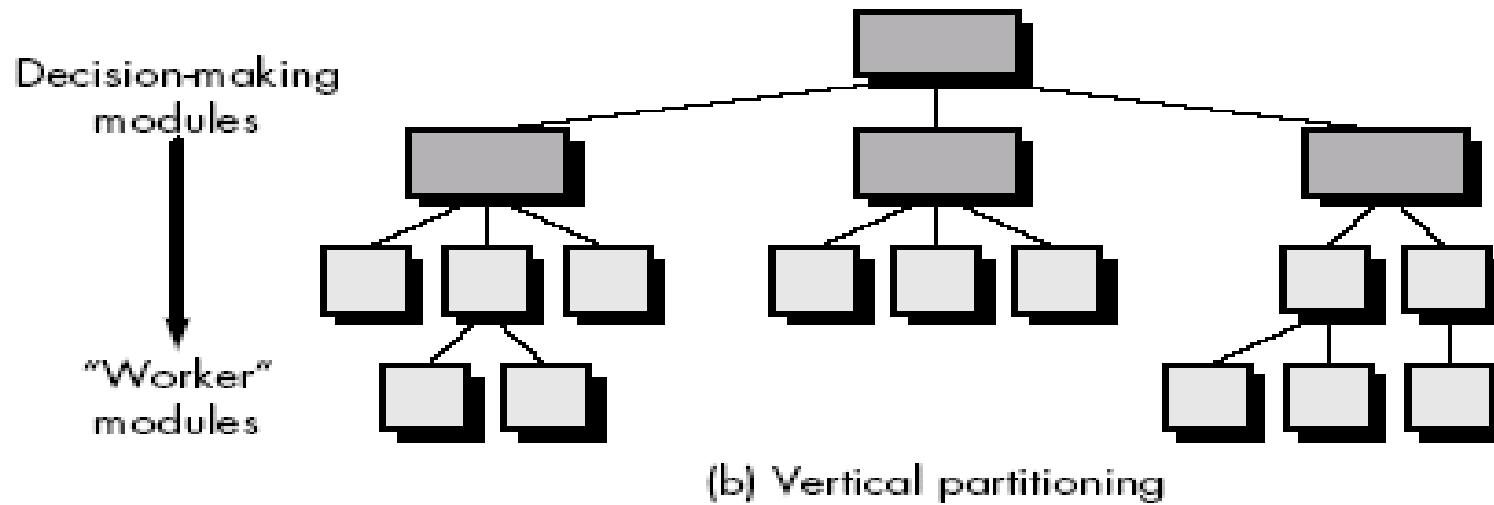
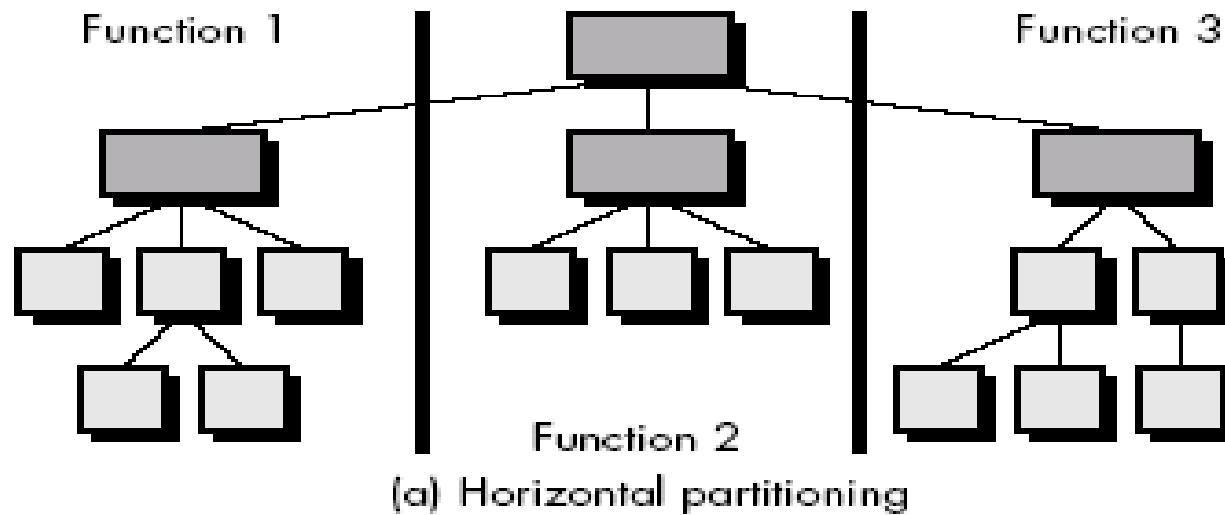


- In fig. *depth* and *width* provide an indication of the number of levels of control and overall *span of control*.
- *Fan-out* is a measure of the number of modules that are directly controlled by another module. *Fan-in* indicates how many modules directly control a given module.
- A module that controls another module is said to be *super-ordinate* to it, and conversely, a module controlled by another is said to be *sub-ordinate* to the controller.

Ex. Module *M* is super-ordinate to modules *a*, *b*, and *c*.

Module *h* is subordinate to module *e*

Structural Partitioning



Structure partitioning

- Two types of Structure partitioning:
 - Horizontal Partitioning
 - Vertical Partitioning

Horizontal partitioning

- **Horizontal partitioning** defines separate branches of the modular hierarchy for each major program function.
- Control modules, represented in a darker shade are used to coordinate communication & execution between its functions.
- **Horizontal partitioning** defines three partitions—**input**, **data transformation** (often called *processing*) and **output**.
- Benefits of Horizontal partitioning:
 - software that is easier to test
 - software that is easier to maintain
 - propagation of fewer side effects
 - software that is easier to extend

Vertical partitioning

- On the negative side (Drawback), horizontal partitioning often causes more data to be passed across module interfaces and can complicate the overall control of program flow.
- Vertical partitioning, often called *factoring*, suggests that control (decision making) and work should be distributed top-down in the program structure.
- Top-level modules should perform control functions and do little actual processing work.
- Modules that reside low in the structure should be the workers, performing all input, computation, and output tasks.

Vertical partitioning (cont.)

- A change in a control module (high in the structure) will have a higher probability of propagating side effects to modules that are subordinate to it.
- A change to a worker module, given its low level in the structure, is less likely to cause the propagation of side effects.
- For this reason vertically partitioned structures are less likely to be susceptible to side effects when changes are made and will therefore be more maintainable.

Data Structure

- *Data structure* is a representation of the logical relationship among individual elements of data
- A *scalar item* is the simplest of all data structures. It represents a single element of information that may be addressed by an identifier.
- When scalar items are organized as a list or contiguous group, a *sequential vector* is formed.
- When the *sequential vector* is extended to two, three, and ultimately, an arbitrary number of dimensions, an *n-dimensional space* is created. Most common n-dimensional space is the two-dimensional matrix
- A *linked list* is a data structure that organizes contiguous scalar items, vectors, or spaces in a manner (called *nodes*) that enables them to be processed as a list.
- A *hierarchical data structure* is implemented using multilinked lists that contain scalar items, vectors, and possibly, *n*-dimensional spaces.

Software Procedure

- Software procedure focuses on the processing details of each module individually.
- Procedure must provide a precise specification of processing, including sequence of events, exact decision points, repetitive operations, and even data organization and structure.

Information Hiding

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

Information Hiding

- The intent of information hiding is to **hide** the details of data structure and procedural processing behind a module interface.
- It gives benefits when modifications are required during testing and maintenance because data and procedure are hiding from other parts of software, unintentional errors introduced during modification are less.

Effective Modular Design

- Effective modular design consist of three things:
 - Functional Independence
 - Cohesion
 - Coupling

Functional Independence

- Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.
- In other words - each module addresses a specific sub-function of requirements and has a simple interface when viewed from other parts of the program structure.
- Independence is important –
 - Easier to develop
 - Easier to Test and maintain
 - Error propagation is reduced
 - Reusable module.

Functional Independence

- To summarize, functional independence is a key to good design, and design is the key to software quality.
- To measure independence, have two qualitative criteria: cohesion and coupling
- *Cohesion* is a measure of the relative functional strength of a module.
- *Coupling* is a measure of the relative interdependence among modules.

Cohesion

- We always strive for high cohesion, although the mid-range of the spectrum is often acceptable.
- Low-end cohesiveness is much "worse" than middle range, which is nearly as "good" as high-end cohesion.
- So. designer should avoid low levels of cohesion when modules are designed.

Cohesion

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

Cohesion

- When processing elements of a module are related and must be executed in a specific order, *procedural cohesion* exists.
- When all processing elements concentrate on one area of a data structure, *communicational cohesion* is present.
- High cohesion is characterized by a module that performs one distinct procedural task.

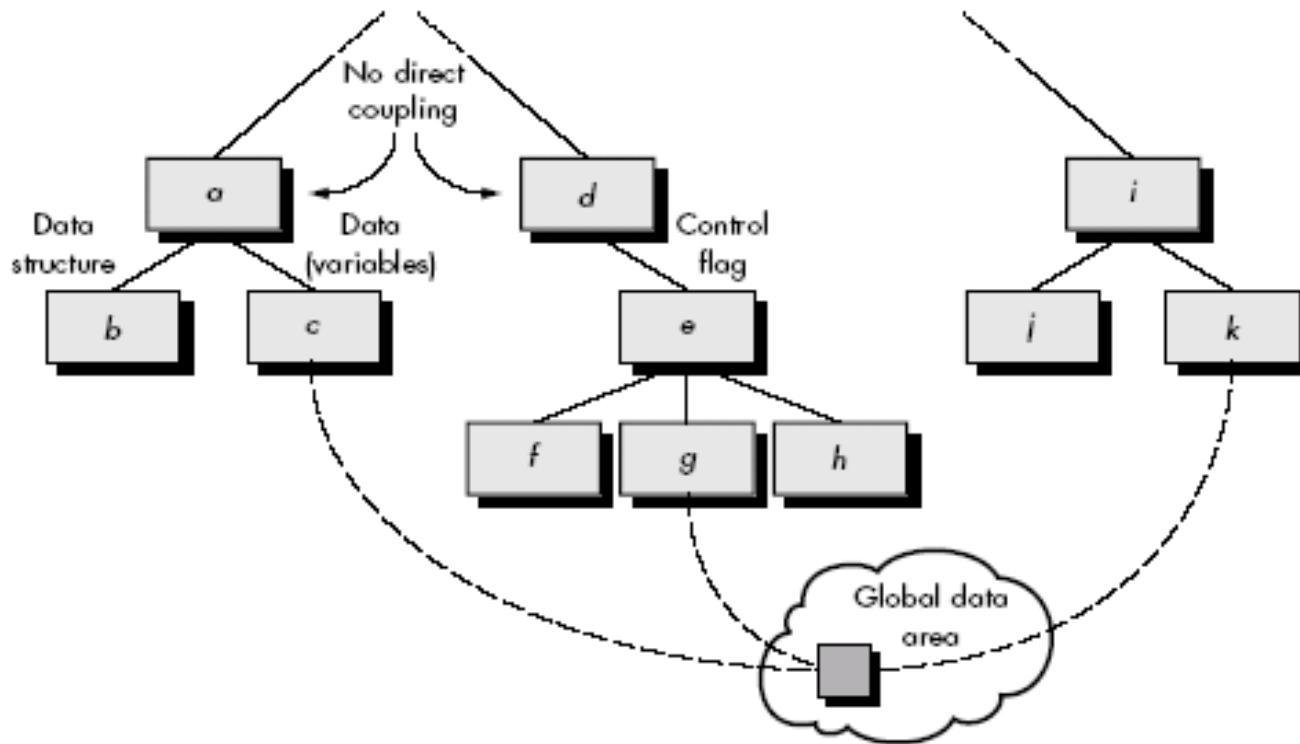
Types of cohesion

- A module that performs tasks that are related logically is *logically cohesive*.
- When a module contains tasks that are related by the fact that all must be executed with the same span of time, the module exhibits *temporal cohesion*.
- At the low-end of the spectrum, a module that performs a set of tasks that relate to each other loosely, called *coincidentally cohesive*.

Coupling

- Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface
- In software design, we **strive for lowest possible coupling**. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect" caused when errors occur at one location and propagate through a system.
- It occurs because of design decisions made when structure was developed.

Coupling



Coupling

- Coupling is characterized by passage of control between modules.
- “Control flag” (a variable that controls decisions in a sub-ordinate or super-ordinate module) is passed between modules d and e (called control coupling).
- Relatively high levels of coupling occur when modules communicate with external to software.
- External coupling is essential, but should be limited to a small number of modules with a structure.

Coupling

- High coupling also occurs when a number of modules reference a global data area.
- Common coupling, no. of modules access a data item in a global data area
- So it does not mean “use of global data is bad”. It does mean that a software designer must be take care of this thing.

Evolution of Software Design

- Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a top-down manner.
- Later work proposed methods for the translation of data flow or data structure into a design definition.
- Today, the emphasis in software design has been on software architecture and the design patterns that can be used to implement software architectures.

Characteristics are common to all design methods

- A mechanism for the translation of analysis model into a design representation,
- A notation for representing functional components and their interfaces.
- Heuristics for refinement and partitioning
- Guidelines for quality assessment.

Design quality attributes

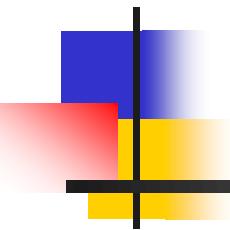
- Acronym FURPS –
 - Functionality
 - Usability
 - Reliability
 - Performance
 - Supportability

FURry PetS

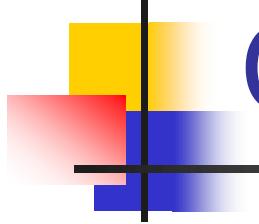
- **Functionality** – is assessed by evaluating the feature set and capabilities of the program.
 - Functions that are delivered and security of the overall system.
- **Usability** – assessed by considering human factors, consistency & documentation.
- **Reliability** – evaluated by
 - measuring the frequency and severity of failure.
 - Accuracy of output results.
 - Ability to recover from failure and predictability of the program.

- **Performance** - measured by processing speed, response time, resource consumption, efficiency.
- **Supportability** – combines the ability to extend the program (extensibility), adaptability and serviceability.

Software Engineering

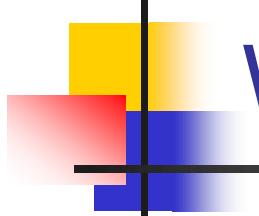


Lecture 05 **UML**



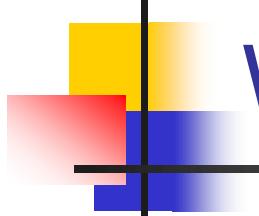
Outline

- What is UML and why we use UML?
- How to use UML diagrams to design software system?
- What UML Modeling tools we use today?



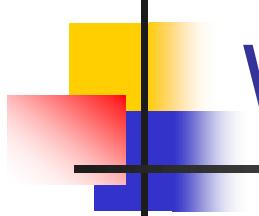
What is UML and Why we use UML?

- UML → “Unified Modeling Language”
 - Language: express idea, not a methodology
 - Modeling: Describing a software system at a high level of abstraction
 - Unified: UML has become a world standard
Object Management Group (OMG): www.omg.org



What is UML and Why we use UML?

- More description about UML:
 - It is a industry-standard graphical language for specifying, visualizing, constructing, and documenting the artifacts of software systems
 - The UML uses mostly graphical notations to express the Object Oriented analysis and design of software projects.
 - Simplifies the complex process of software design



What is UML and Why we use UML?

- **Why we use UML?**

- Use graphical notation: more clearly than natural language (imprecise) and code (too detailed).
- Help acquire an overall view of a system.
- UML is *not* dependent on any one language or technology.
- UML moves us from fragmentation to standardization.

Overview of UML Diagrams

Structural

: element of spec. irrespective of time

- Class
- Component
- Deployment
- Object
- *Composite structure*
- *Package*

Behavioral

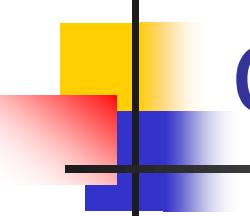
: behavioral features of a system / business process

- Activity
- State machine
- Use case
- *Interaction*

Interaction

: emphasize object interaction

- Communication(collaberation)
- Sequence
- *Interaction overview*
- *Timing*



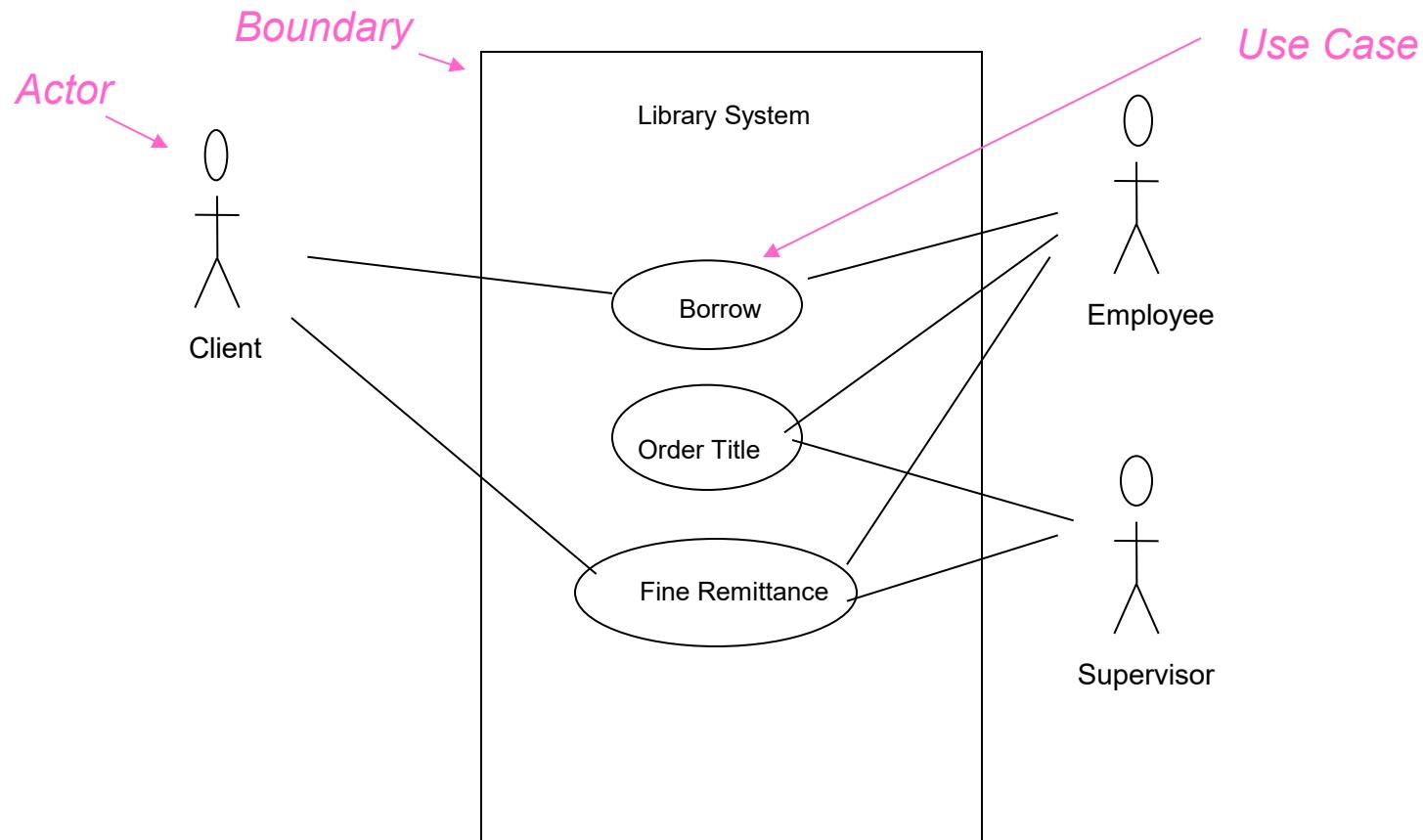
How to use UML diagrams to design software system?

■ Types of UML Diagrams:

- Use Case Diagram
- Class Diagram
- Sequence Diagram
- Collaboration Diagram
- State Diagram

This is only a subset of diagrams ... but are most widely used

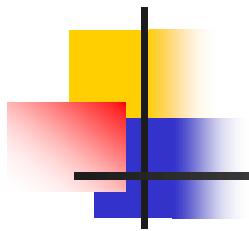
Use-Case Diagrams



Use-Case Diagrams

- **Actors:** A role that a user plays with respect to the system, including **human users and other systems**. e.g., inanimate **physical objects** (e.g. robot); an external system that needs some information from the current system.
- **Use case:** A set of scenarios that describing an interaction between a user and a system, including alternatives.
- **System boundary:** **rectangle** diagram representing the boundary between the actors and the system.



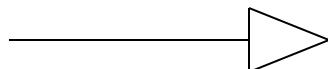


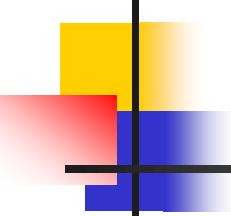
Use-Case Diagrams

- Association:

communication between an actor and a use case; Represented by a solid line.

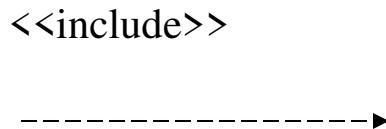
- Generalization: relationship between one general use case and a special use case (used for defining special alternatives) Represented by a line with a triangular arrow head toward the parent use case.



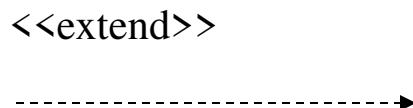


Use-Case Diagrams

Include: a dotted line labeled <<include>> beginning at base use case and ending with an arrow pointing to the include use case. **The include relationship occurs when a chunk of behavior is similar across more than one use case.** Use “include” in stead of copying the description of that behavior.



Extend: a dotted line labeled <<extend>> with an arrow toward the base case. **The extending use case may add behavior to the base use case.** The base class declares “extension points”.



Use-Case Diagrams

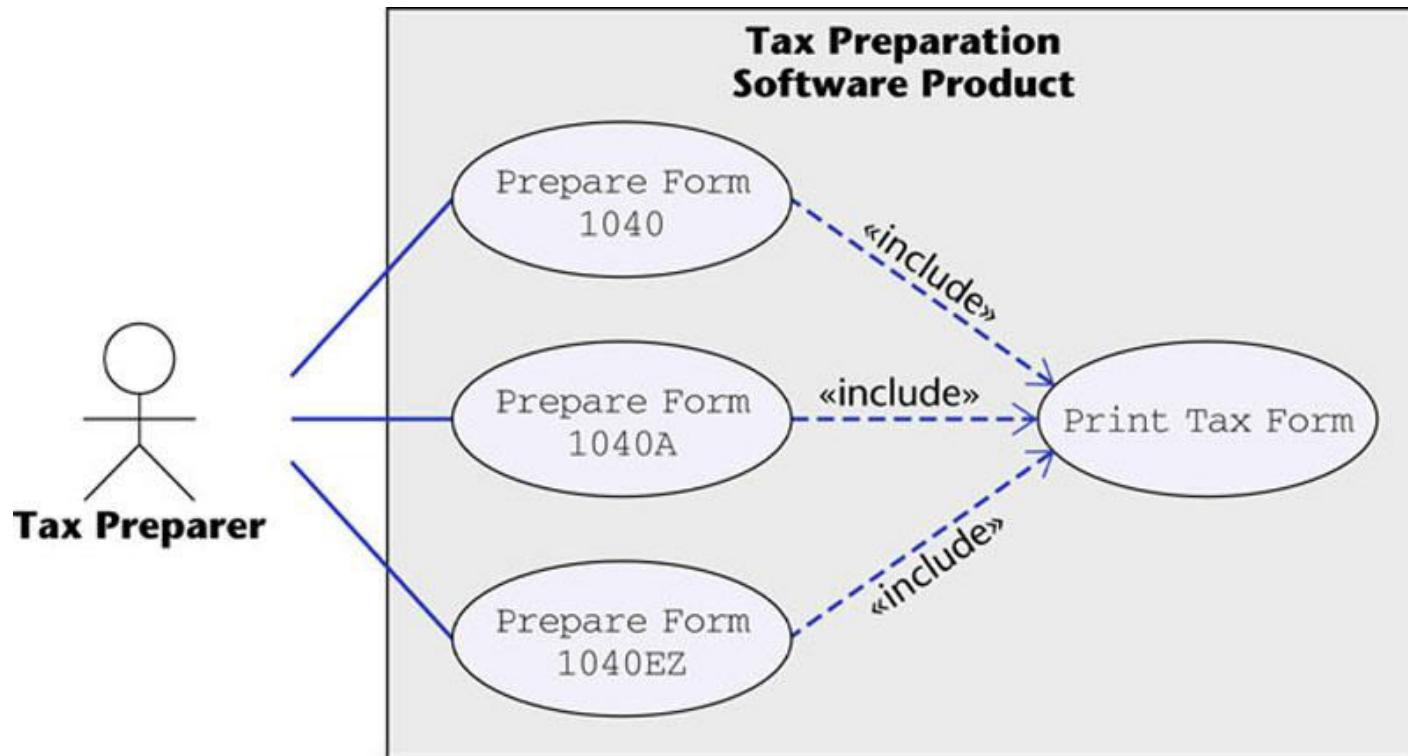
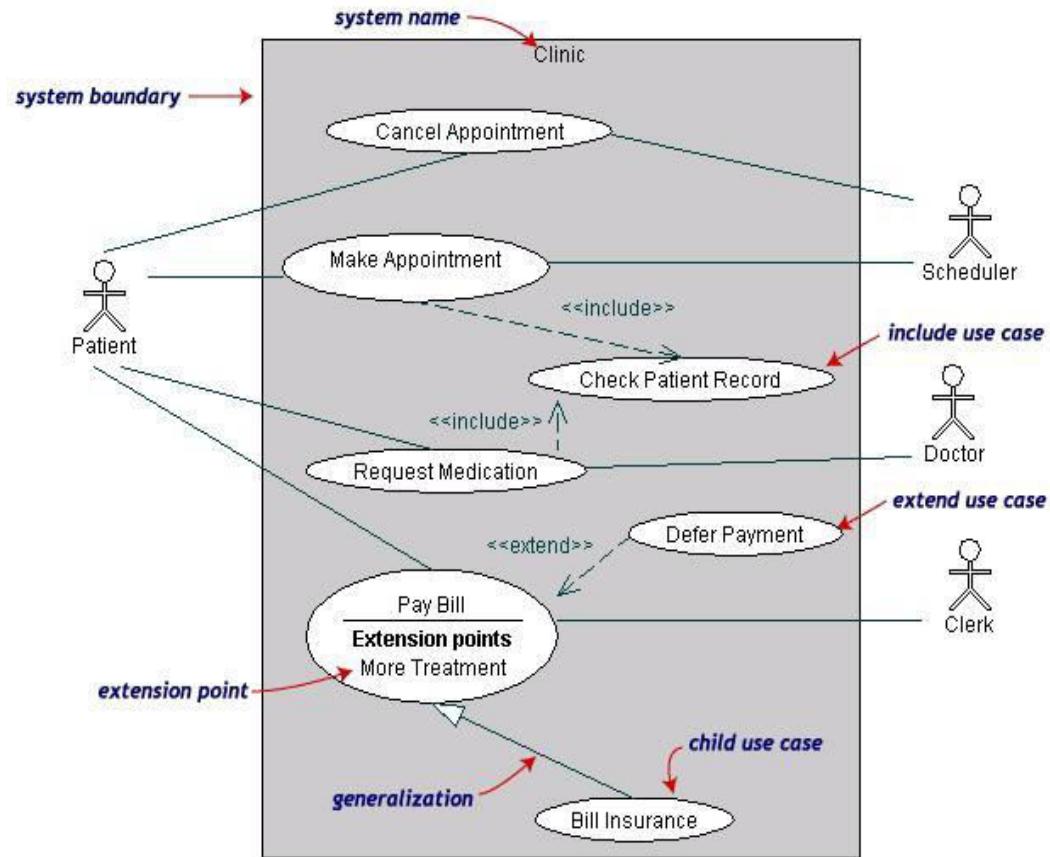


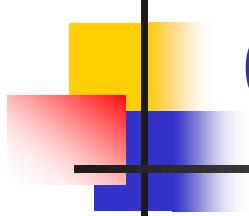
Figure 16.12

Use-Case Diagrams

- Both **Make Appointment** and **Request Medication** include **Check Patient Record** as a subtask (include)
- The **extension point** is written inside the base case **Pay bill**; the extending class **Defer payment** adds the behavior of this extension point. (extend)
- **Pay Bill** is a parent use case and **Bill Insurance** is the child use case. (generalization)

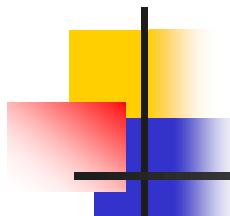


(TogetherSoft, Inc)



Class diagram

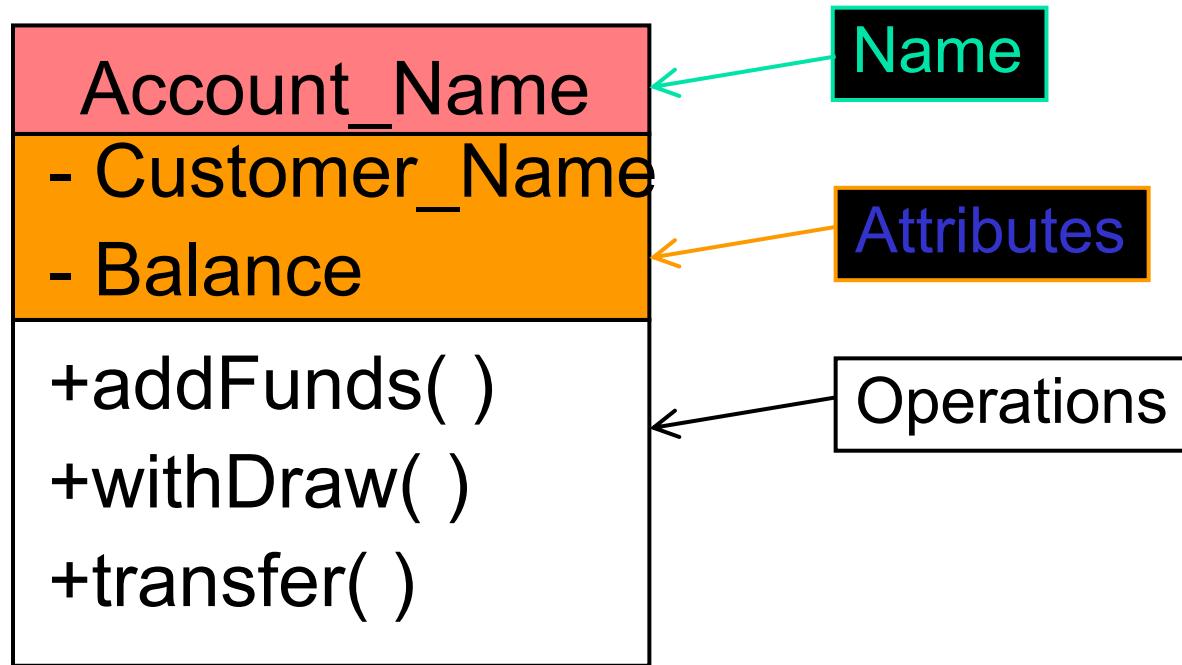
- A class diagram depicts classes and their interrelationships
- Used for describing **structure and behavior** in the use cases
- Provide a conceptual model of the system in terms of entities and their relationships
- Used for requirement capture, end-user interaction
- Detailed class diagrams are used for developers

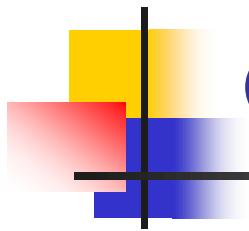


Class diagram

- Each class is represented by a rectangle subdivided into three compartments
 - Name
 - Attributes
 - Operations
- Modifiers are used to indicate visibility of attributes and operations.
 - '+' is used to denote *Public* visibility (everyone)
 - '#' is used to denote *Protected* visibility (friends and derived)
 - '-' is used to denote *Private* visibility (no one)
- By default, attributes are hidden and operations are visible.

Class diagram



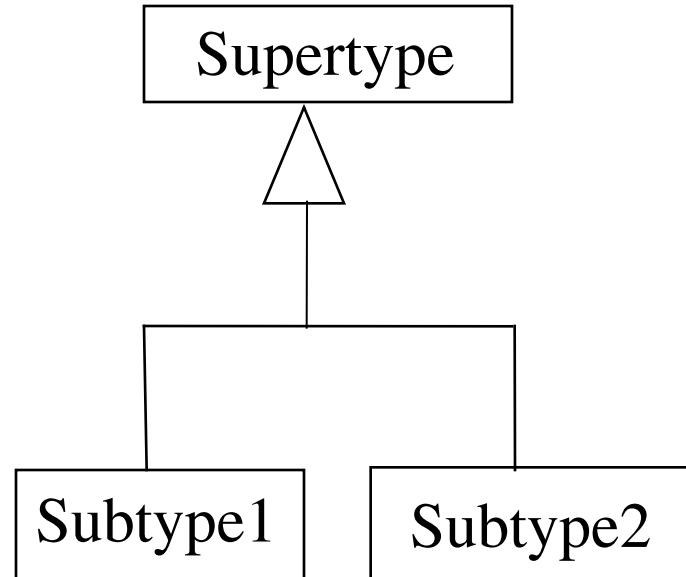


OO Relationships

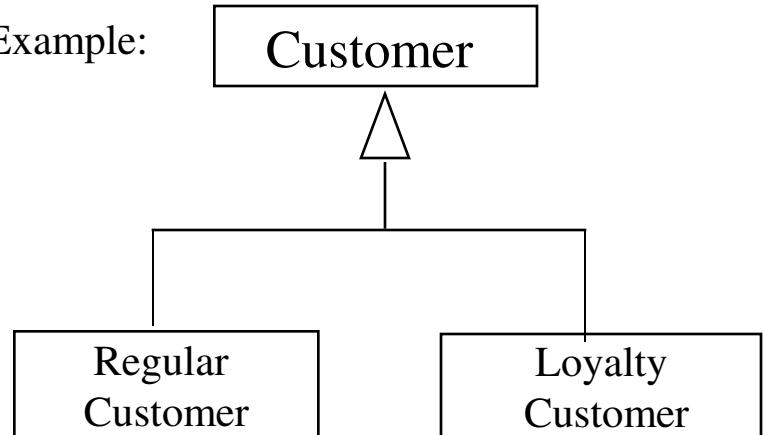
- There are two kinds of Relationships
 - Generalization (parent-child relationship)
 - Association (student enrolls in course)

- Associations can be further classified as
 - Aggregation
 - Composition

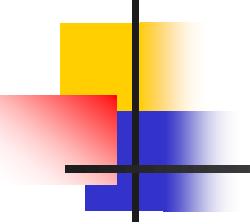
OO Relationships: Generalization



Example:



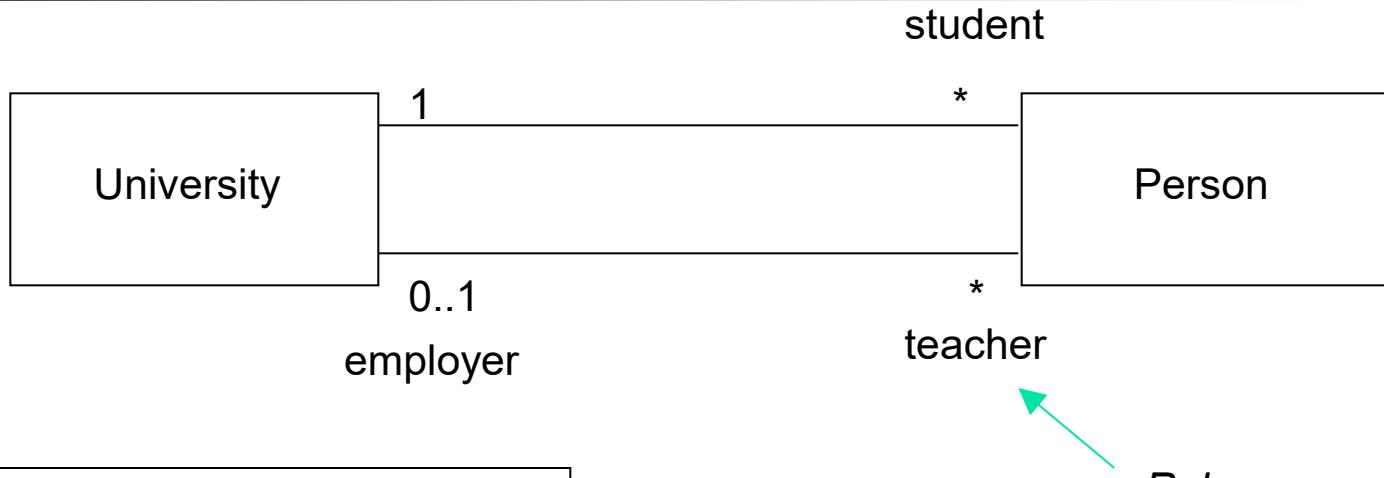
- Inheritance is a required feature of object orientation
- Generalization expresses a parent/child relationship among related classes.
- Used for abstracting details in several layers



OO Relationships: Association

- Represent relationship between instances of classes
 - Student enrolls in a course
 - Courses have students
 - Courses have exams
 - Etc.
- Association has two ends
 - Role names (e.g. enrolls)
 - Multiplicity (e.g. One course can have many students)
 - Navigability (unidirectional, bidirectional)

Association: Multiplicity and Roles

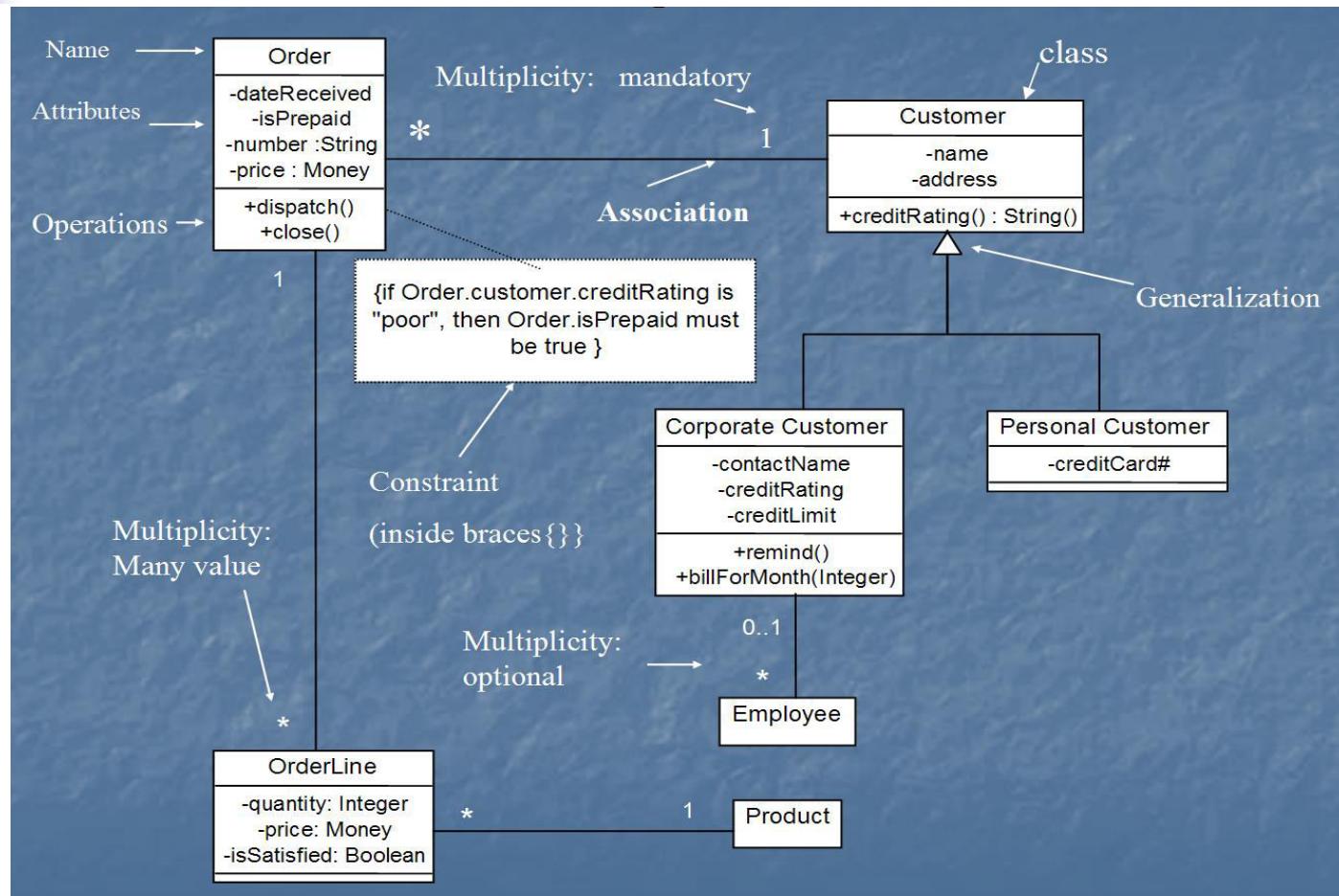


Multiplicity	
Symbol	Meaning
1	One and only one
0..1	Zero or one
M..N	From M to N (natural language)
*	From zero to any positive integer
0..*	From zero to any positive integer
1..*	From one to any positive integer

Role

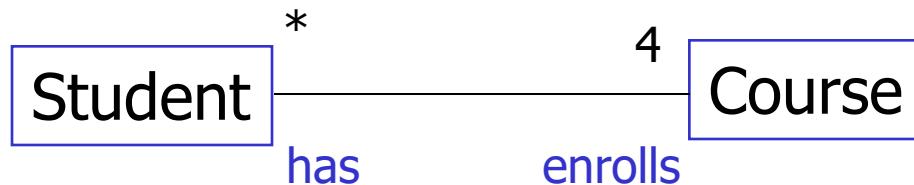
“A given university groups many people; some act as students, others as teachers. A given student belongs to a single university; a given teacher may or may not be working for the university at a particular time.”

Class diagram



[from *UML Distilled Third Edition*]

Association: Model to Implementation

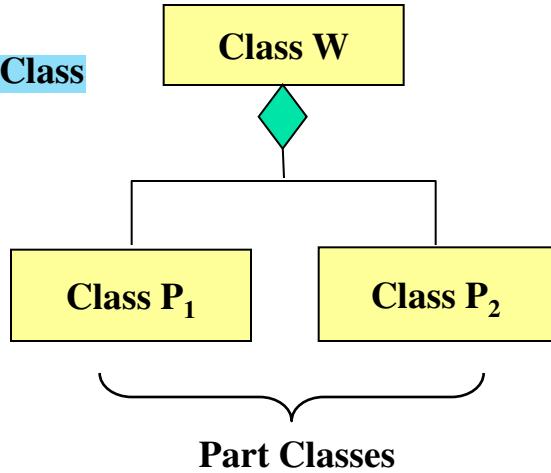


```
Class Student {  
    Course enrolls[4];  
}
```

```
Class Course {  
    Student have[];  
}
```

OO Relationships: Composition

Whole Class



[From Dr.David A. Workman]

Example

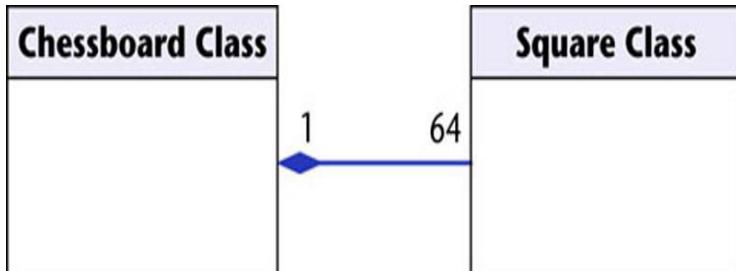


Figure 16.7

Association

Models the part–whole relationship

Composition

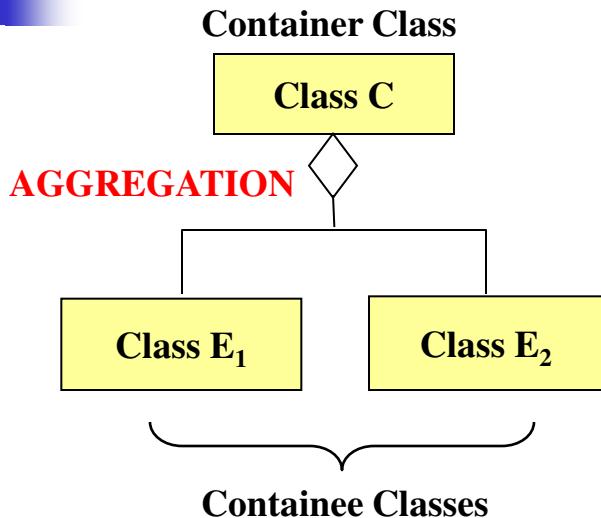
Also models the part–whole relationship but, in addition, Every part may belong to only one whole, and If the whole is deleted, so are the parts

Example:

A number of different chess boards: Each square belongs to only one board. If a chess board is thrown away, all 64 squares on that board go as well.

child class present only if parent class present.

OO Relationships: Aggregation



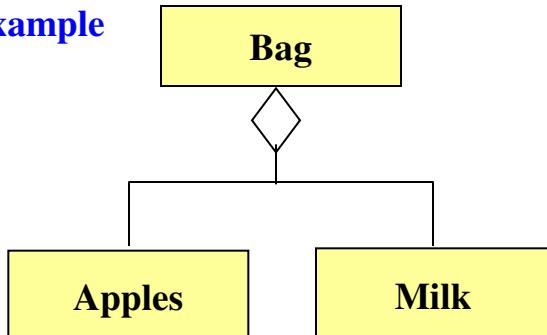
Aggregation:

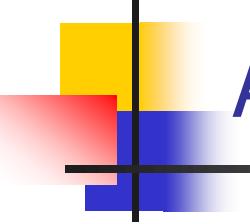
expresses a relationship among instances of related classes. It is a specific kind of Container-Containee relationship.

express a more informal relationship than composition expresses.

Aggregation is appropriate when Container and Containees have no special access privileges to each other.

Example





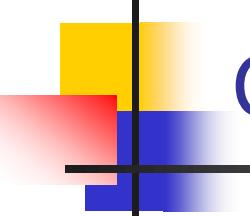
Aggregation vs. Composition

■ **Composition** is really a strong form of association

- components have only one owner
- components cannot exist independent of their owner
- components live or die with their owner
- e.g. Each car has an engine that can not be shared with other cars.

■ **Aggregations**

may form "part of" the association, but may not be essential to it. They may also exist independent of the aggregate. e.g. Apples may exist independent of the bag.

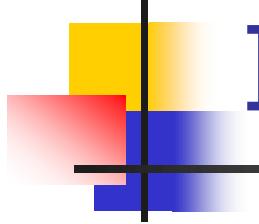


Good Practice: CRC Card

Class Responsibility Collaborator

- easy to describe how classes work by moving cards around; allows to quickly consider alternatives.

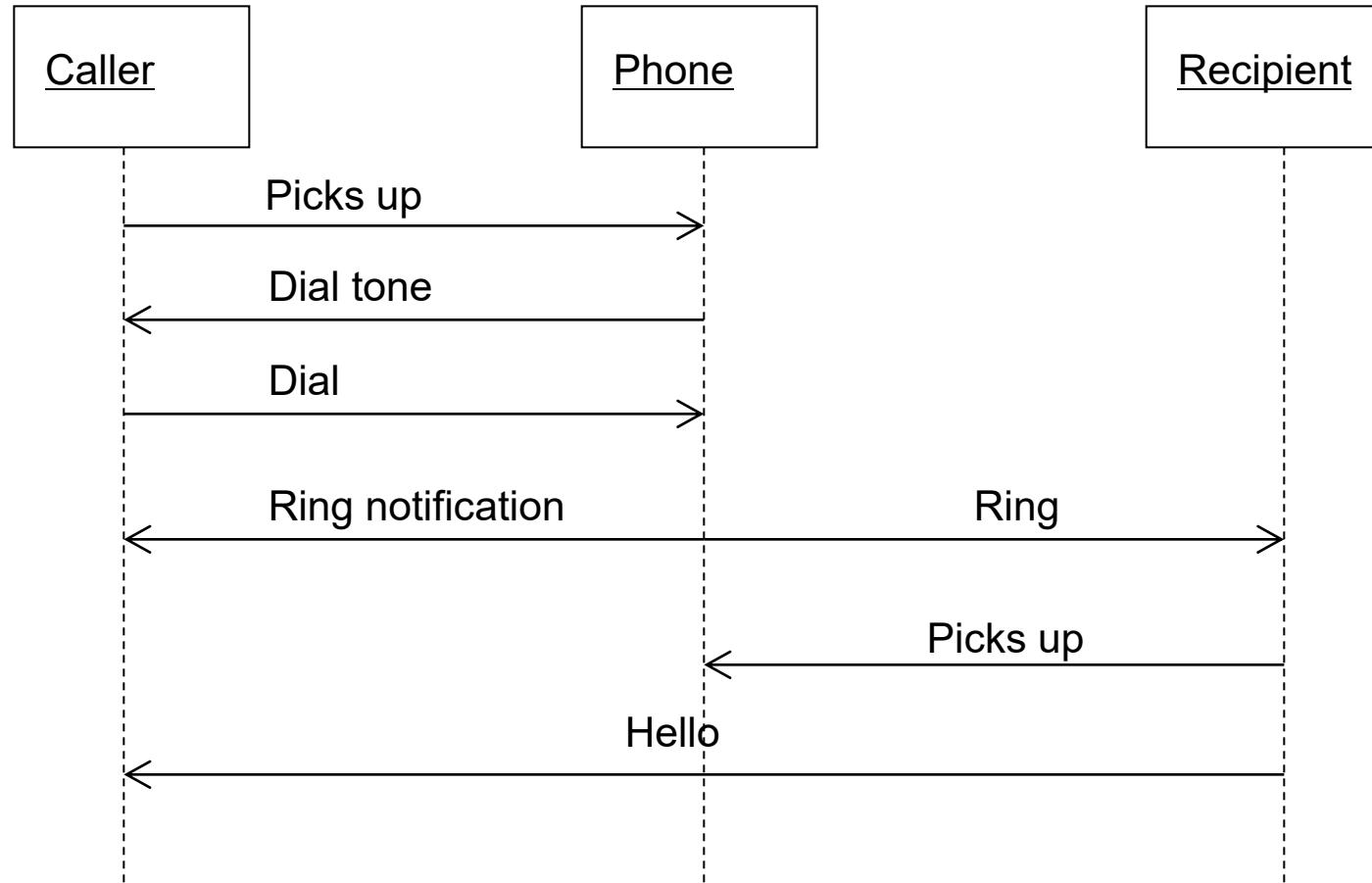
Class Reservations	Collaborators <ul style="list-style-type: none">▪ Catalog▪ User session
Responsibility <ul style="list-style-type: none">▪ Keep list of reserved titles▪ Handle reservation	



Interaction Diagrams

- show how objects interact with one another
- UML supports two types of interaction diagrams
 - Sequence diagrams
 - Collaboration diagrams

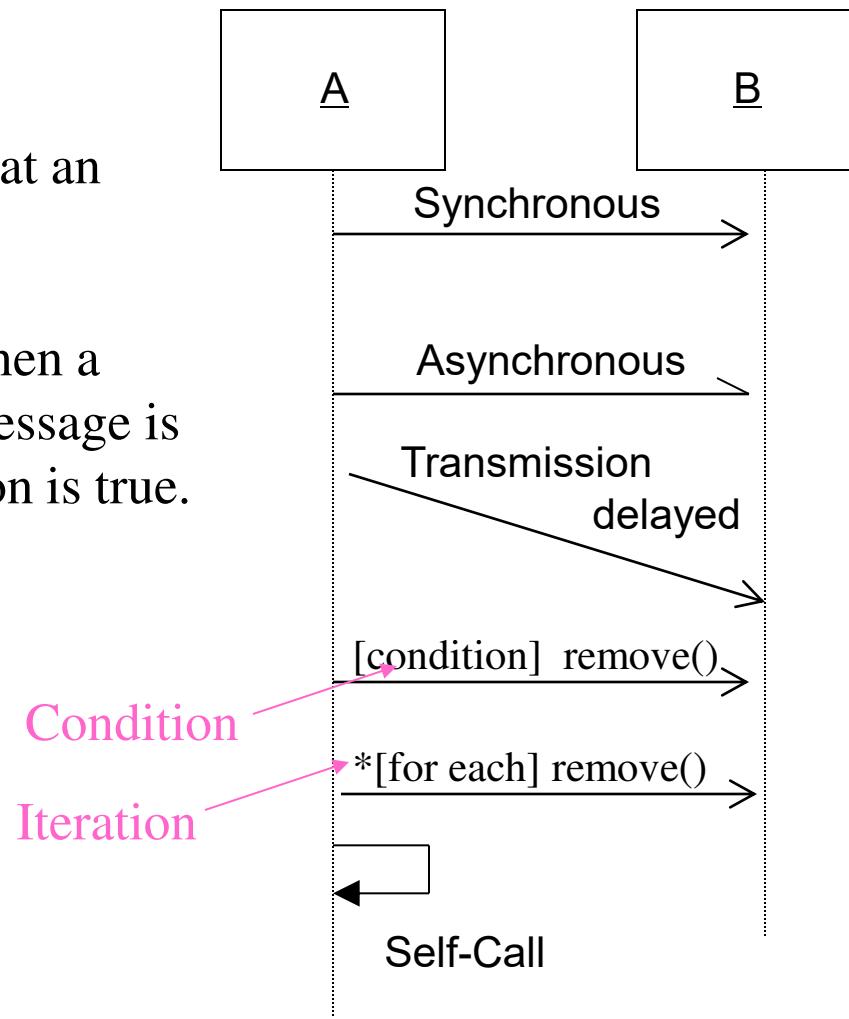
Sequence Diagram(make a phone call)



Sequence Diagram: Object interaction

Self-Call: A message that an Object sends to itself.

Condition: indicates when a message is sent. The message is sent only if the condition is true.



Sequence Diagrams – Object Life Spans

Creation

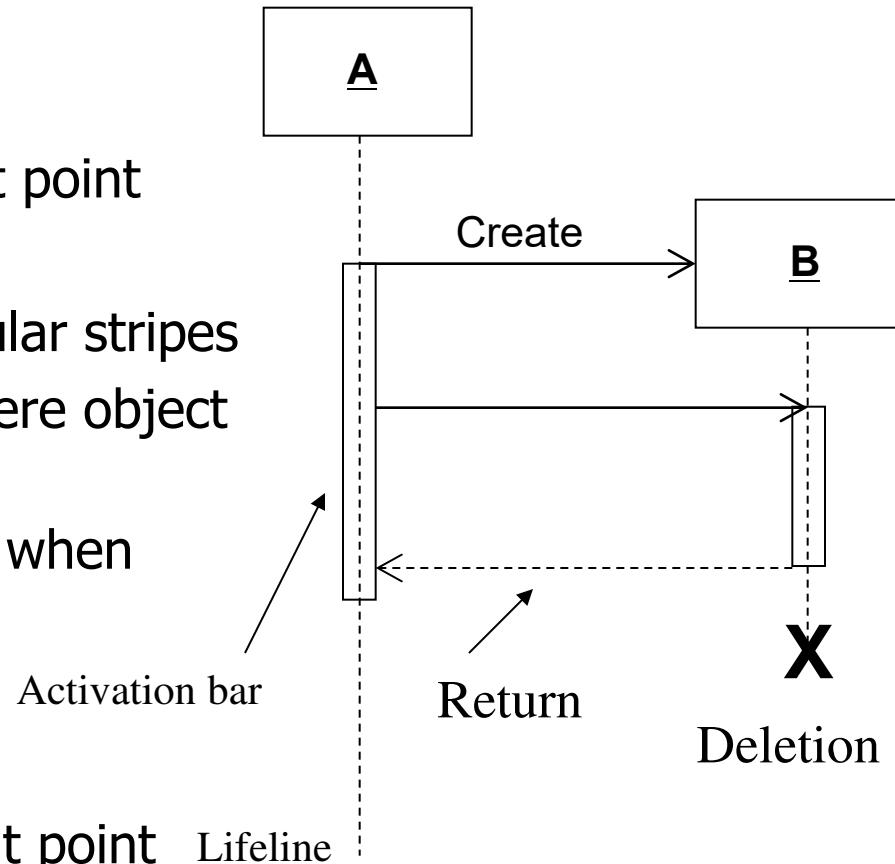
- Create message
- Object life starts at that point

Activation

- Symbolized by rectangular stripes
- Place on the lifeline where object is activated.
- Rectangle also denotes when object is deactivated.

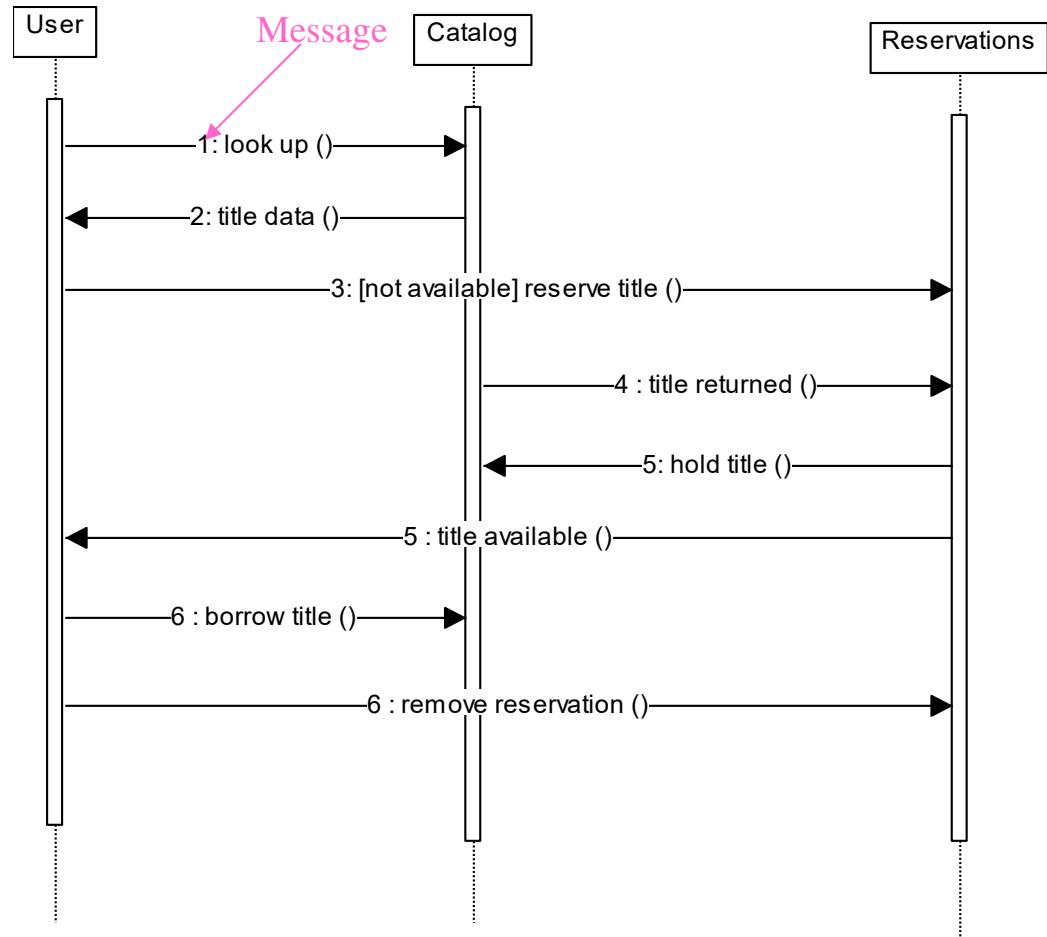
Deletion

- Placing an 'X' on lifeline
- Object's life ends at that point

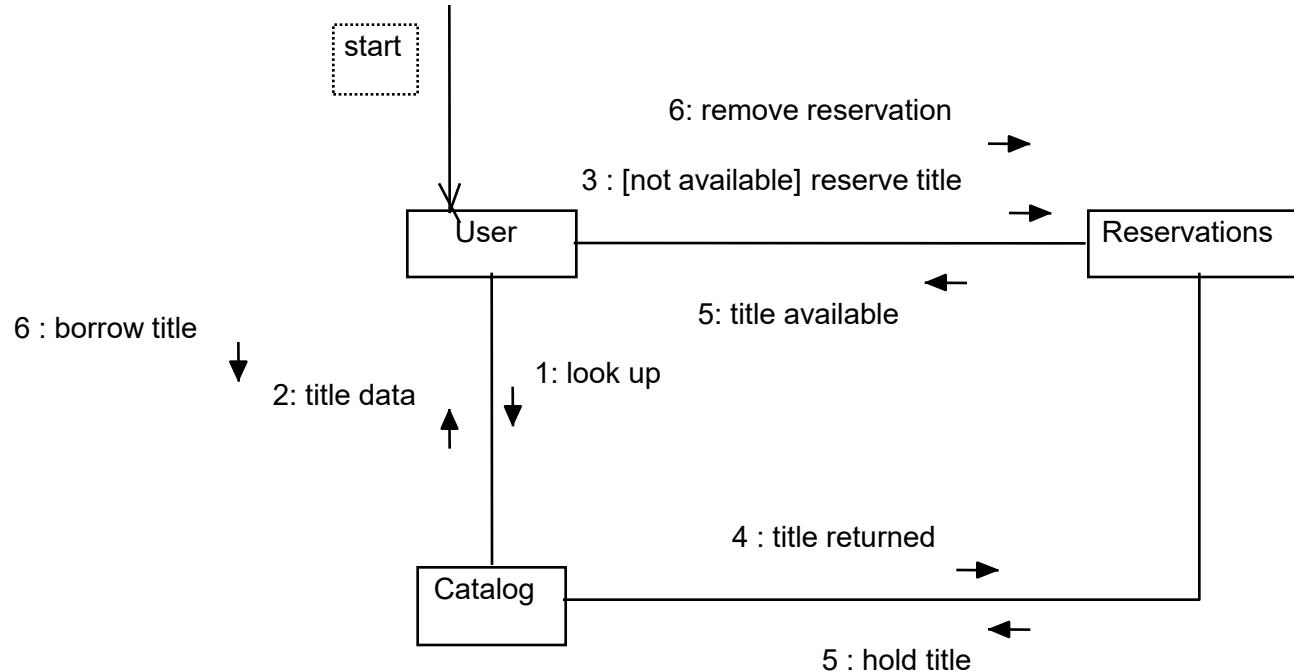


Sequence Diagram

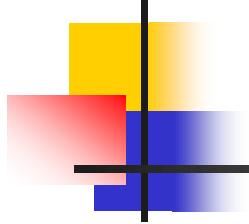
- Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass.
- The horizontal dimension shows the objects participating in the interaction.
- The vertical arrangement of messages indicates their order.
- The labels may contain the seq. # to indicate concurrency.



Interaction Diagrams: Collaboration diagrams

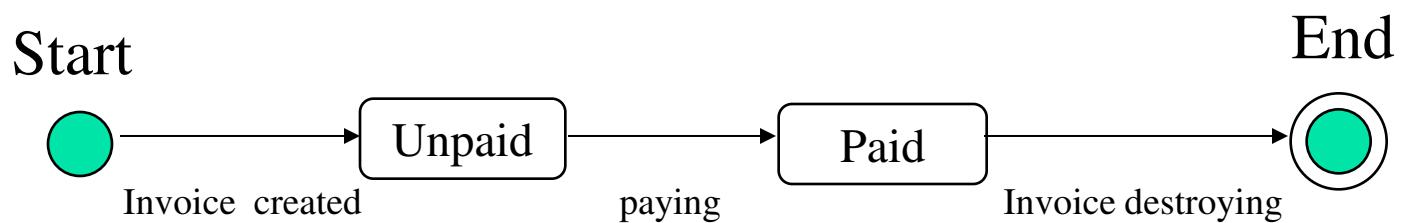


- Collaboration diagrams are equivalent to sequence diagrams. All the features of sequence diagrams are equally applicable to collaboration diagrams
- Use a sequence diagram when the transfer of information is the focus of attention
- Use a collaboration diagram when concentrating on the classes

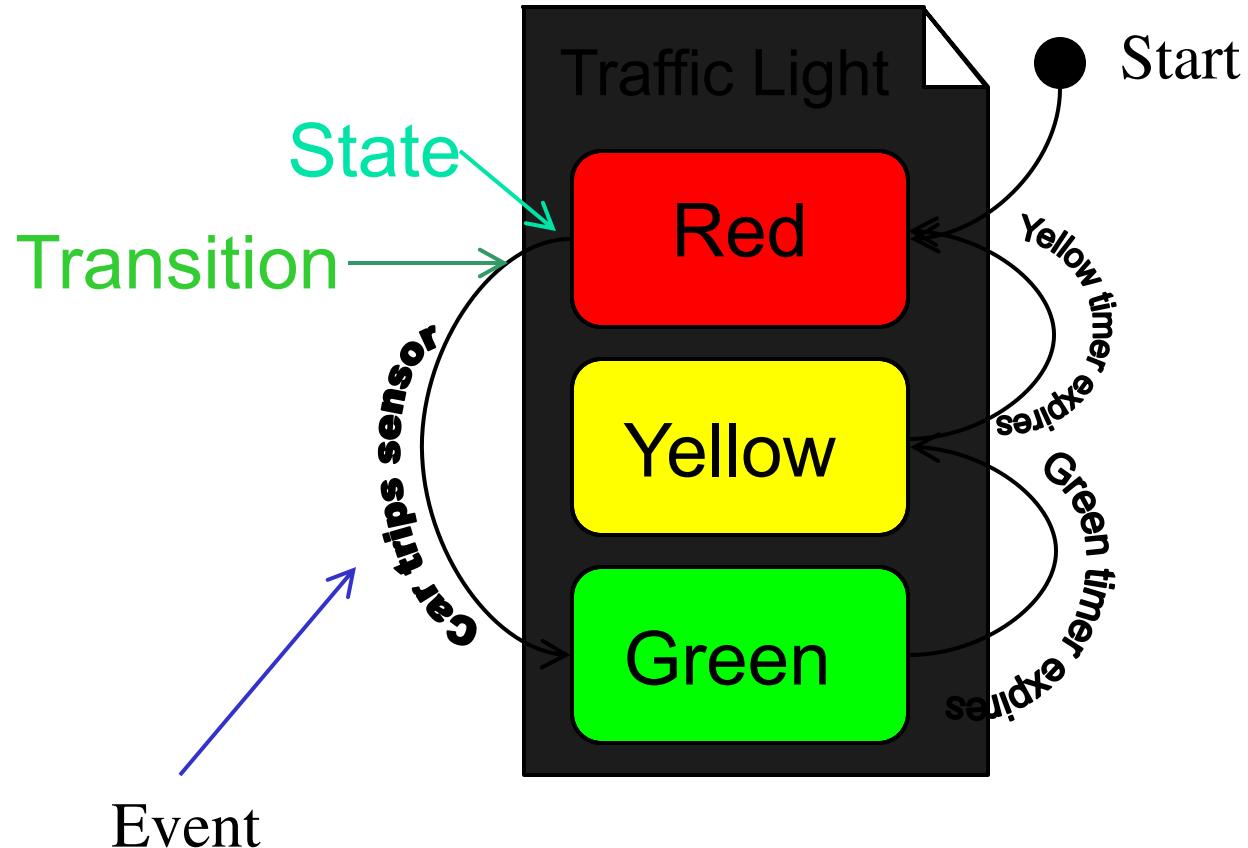


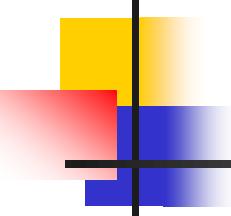
State Diagrams (Billing Example)

State Diagrams show the sequences of states an object goes through during its life cycle in response to stimuli, together with its responses and actions; an abstraction of all possible behaviors.



State Diagrams (Traffic light example)





Conclusion

- UML is a standardized specification language for object modeling
- Several UML diagrams:
 - use-case diagram: a number of use cases (use case models the interaction between actors and software)
 - Class diagram: a model of classes showing the static relationships among them including association and generalization.
 - Sequence diagram: shows the way objects interact with one another as messages are passed between them. Dynamic model
 - State diagram: shows states, events that cause transitions between states. Another dynamic model reflecting the behavior of objects and how they react to specific event
- There are several UML tools available

Software Engineering

Lecture 06 **Architecture & User Interface Design**

Software Architecture

- “The software architecture of a program or computing system is the structure or structures of the system, which comprise the software components, the externally visible properties of those components, and the relationships among them.”
- It is not operational software but it is representation that enables software engineer to
 - Analyze the effectiveness of design in meeting its stated requirement.
 - consider architectural alternatives at a stage when making design changes is still relatively easy,
 - reduce the risks associated with the construction of the software.

Importance of Software Architecture

- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development
- Architecture highlights early design decisions that will have a profound impact on all software engineering work that follows.
- Architecture “constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together”

Data Design

- The structure of data has always been an important part of software design.
- *Component level* – the design of *data structures* and the associated *algorithms* required to manipulate them is essential to the creation of high-quality applications.
- *Application level* – the *translation* of a data model into a *database design*

Data Design at architectural design

- Today, business (i.e. irrespective of its size large or small) have dozens of database serving many applications encompassing hundreds of gigabytes of data.
- Challenge is to extract useful information from its data environment, particularly when the information desired is cross-functional.
- To solve this challenge, developed technique called
 - Data Mining (also called *knowledge discovery in databases (KDD)*),
 - Data Warehouse
- *Data mining* that navigate through existing databases in an attempt to extract appropriate business-level information

- However, the existence of multiple databases, their different structures, the degree of detail contained with the databases, and many other factors make data mining difficult within an existing database environment.
- A data warehouse is a separate data environment that is not directly integrated with day-to-day applications but encompasses all data used by a business
- A data warehouse is a large, independent database that encompasses some, but not all, of the data that are stored in databases that serve the set of applications required by a business.

Data design at the component level

- Component level focuses on the representation of data structures that are directly accessed by one or more software components.

Principles are applicable to data design

1. *The systematic analysis principles applied to function and behavior should also be applied to data.*
2. *All data structures and the operations to be performed on each should be identified.*
3. *A data dictionary should be established and used to define both data and program design (operations)*
4. *Low-level data design decisions should be deferred until late in the design process.*

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

Architectural Style

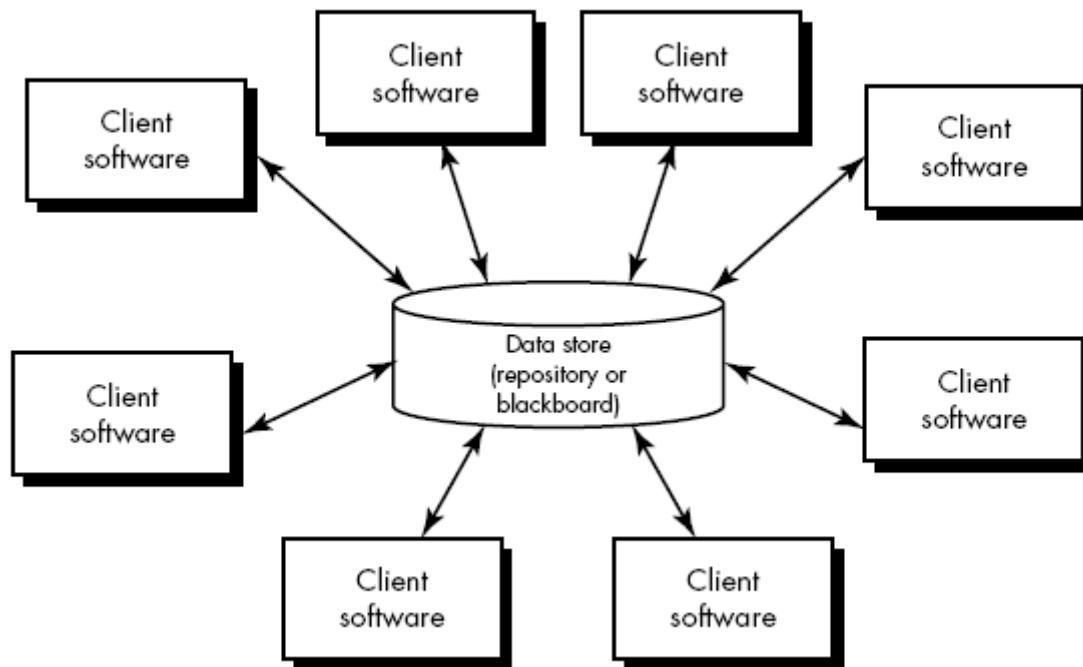
- Style describes a system category that encompasses
 - 1. A set of **components** (e.g., a database, computational modules) that perform a function required by a system;
 - 2. **a set of connectors** that enable “communication, co-ordinations and cooperation” among components;
 - 3. **constraints** that define how components can be integrated to form the system
 - 4. **semantic models** that enable a designer to understand the overall properties of a system

It can be represent by

- Data-centered architecture
- Data flow architecture
- Call and return architecture
- Object oriented architecture
- Layered architecture.

C-F-CR-OO-L

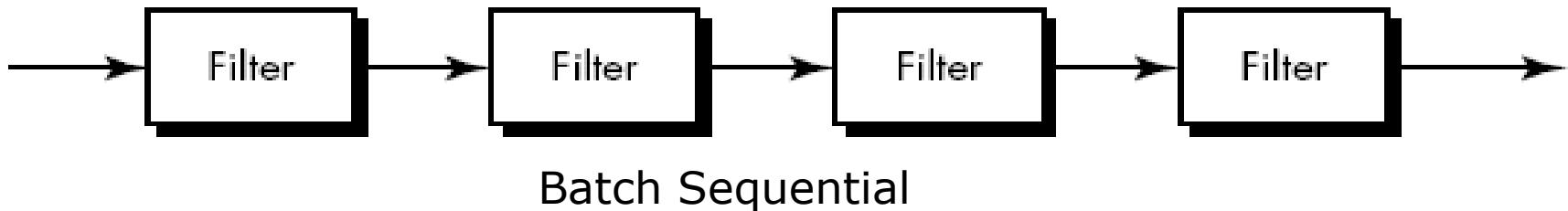
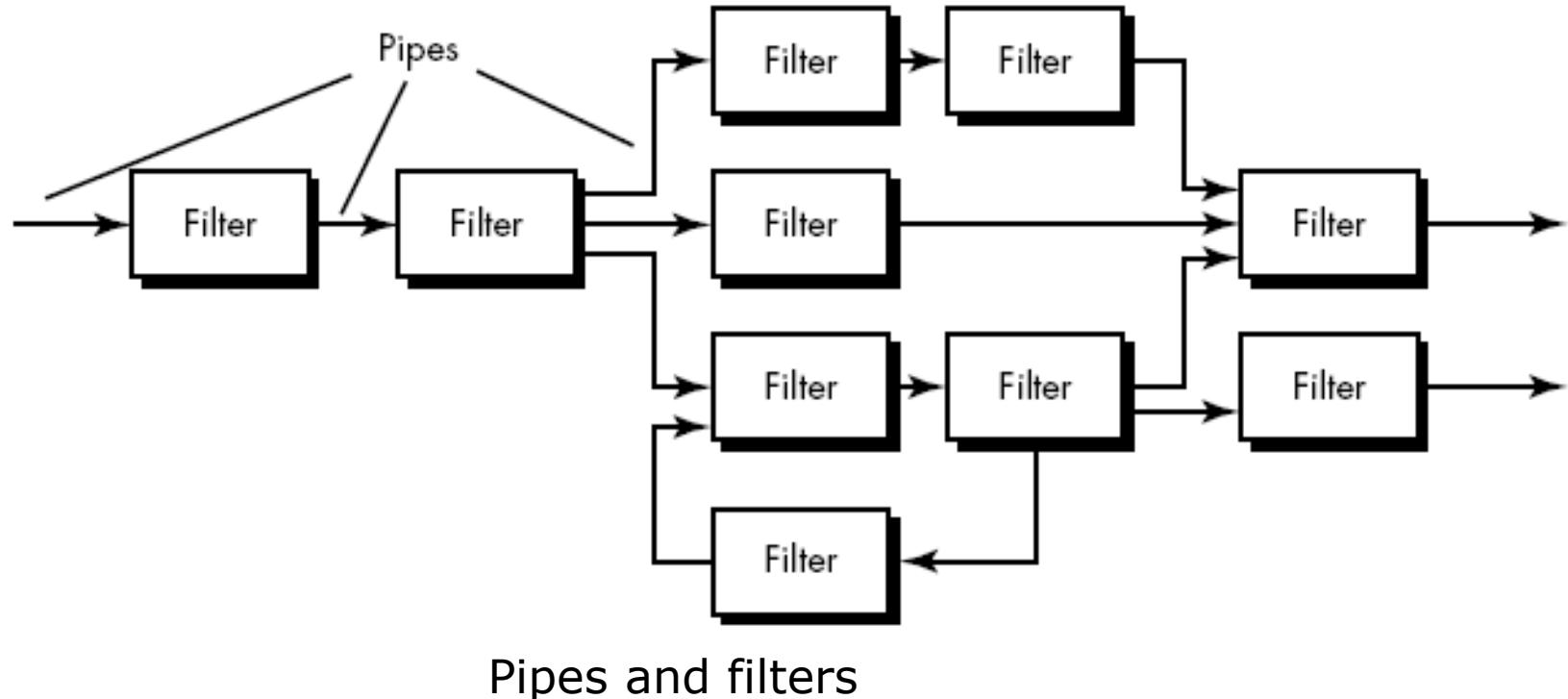
Data-centered architecture



Data-centered architecture

- A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store.
- Client software accesses a central repository which is in passive state (in some cases).
- Client software accesses the data independent of any changes to the data or the actions of other client software.
- So, in this case transform the repository into a “Blackboard”.
- A blackboard sends notification to subscribers when data of interest changes, and is thus active.
- Data-centered architectures promote *integrity*.
- Existing components can be changed and new client components can be added to the architecture without concern about other clients.
- Data can be passed among clients using the blackboard mechanism. So Client components independently execute processes

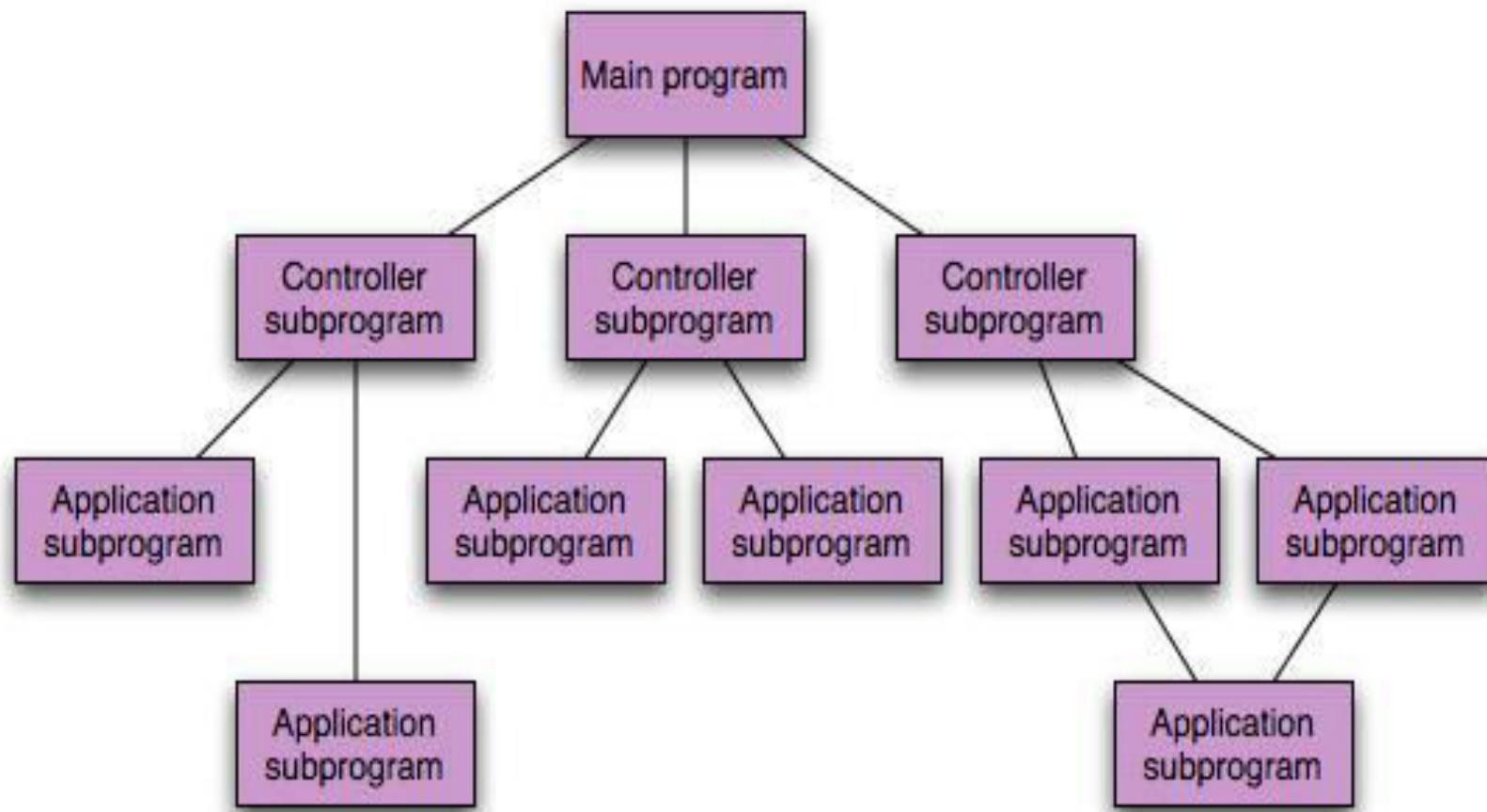
Data Flow architecture



Data Flow architecture

- This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data.
- A *pipe and filter pattern* (Fig .1) has a set of components, called *filters*, connected by pipes that transmit data from one component to the next.
- Each filter works independently (i.e. upstream, downstream) and is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form.
- the filter does not require knowledge of the working of its neighboring filters.
- If the data flow degenerates into a single line of transforms, it is termed *batch sequential*.

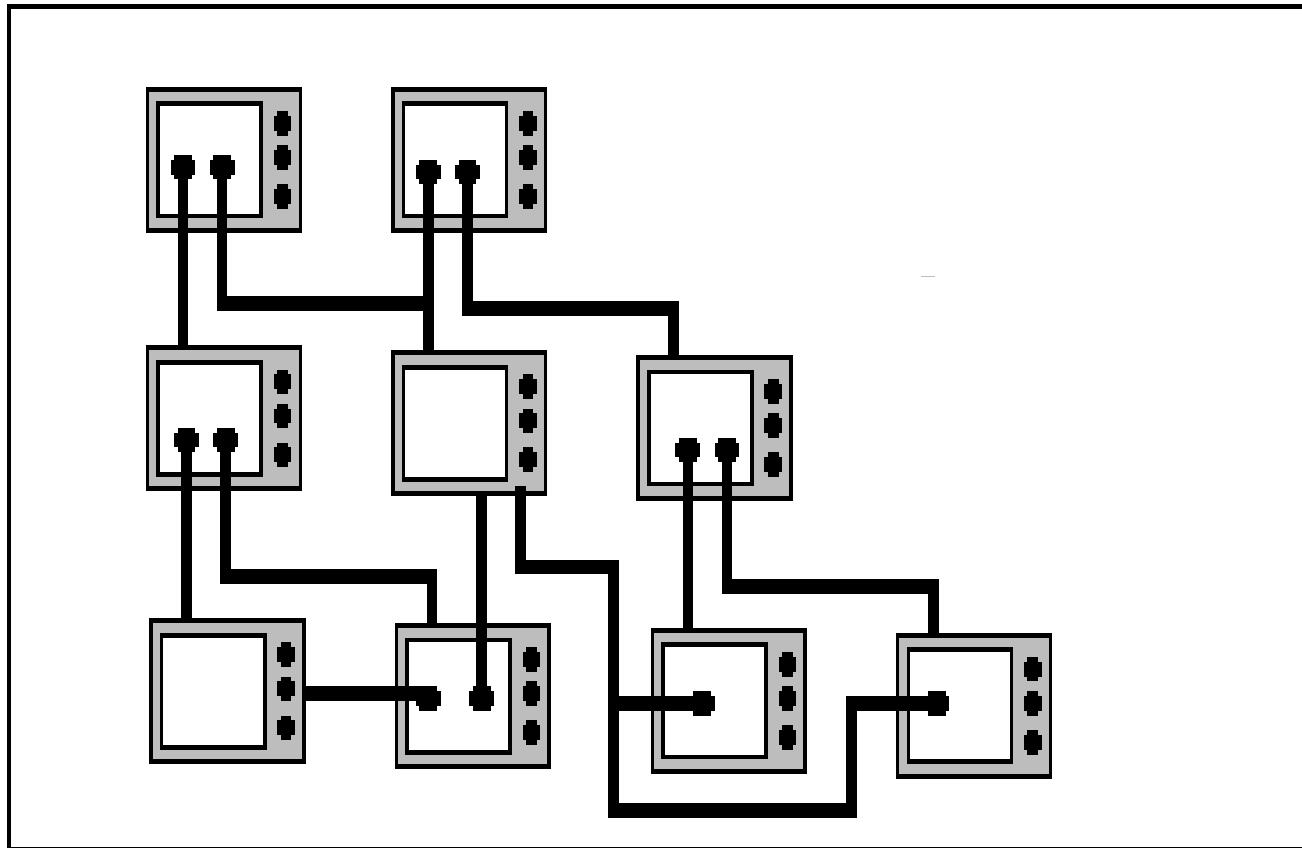
Call and return architecture



Call and return architecture

- Architecture style enables a software designer (system architect) to achieve a program structure that is **relatively easy to modify and scale**.
- Two sub-styles exist within this category:
 1. *Main/sub program architecture:*
 - Program structure **decomposes function into a control hierarchy** where a “main” program invokes a number of program components, which in turn may invoke still other components.
 2. *Remote procedure Call architecture:*
 - The **components of a main program/subprogram architecture are distributed across multiple computers on a network**

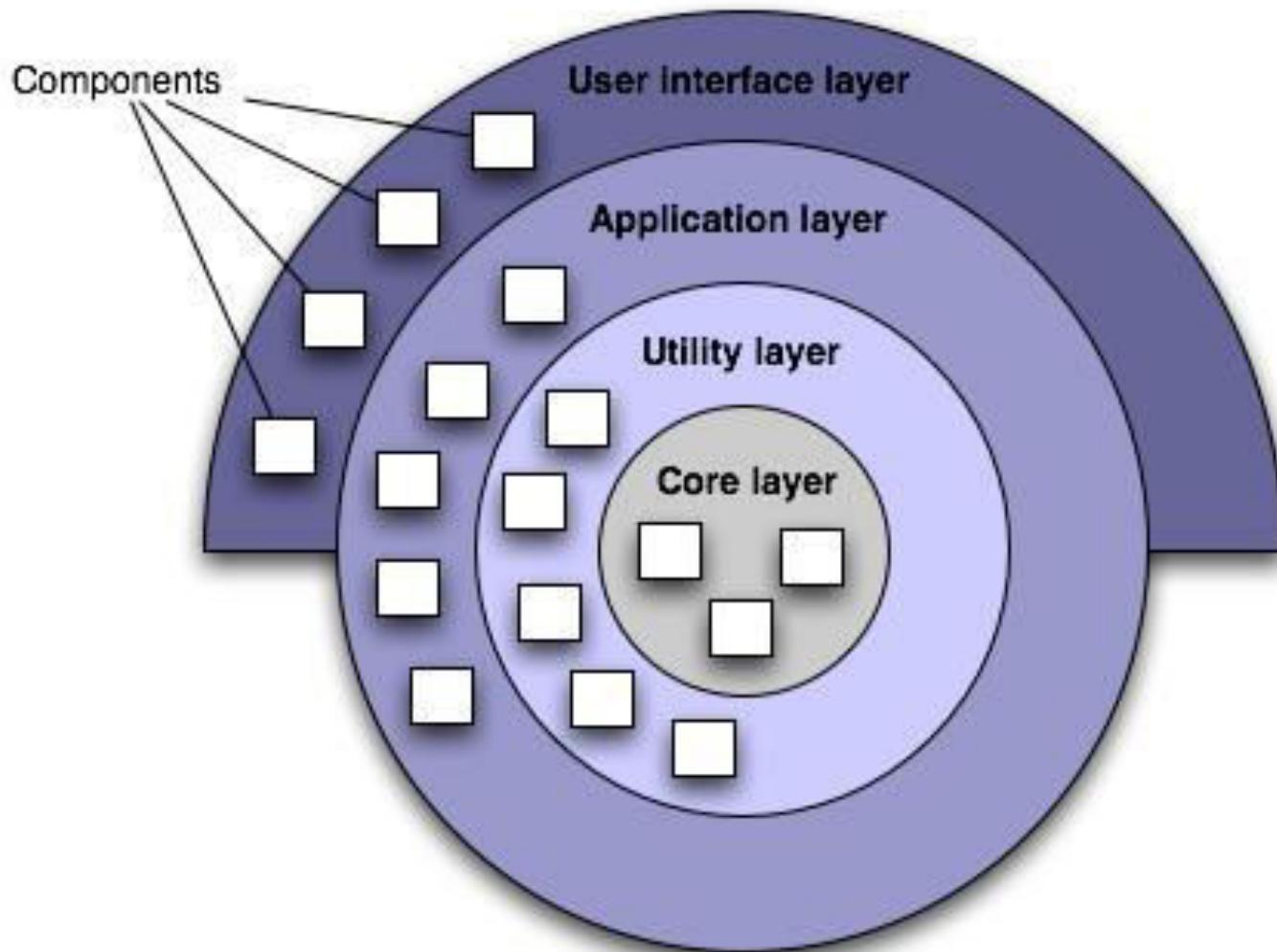
Object-oriented architecture



Object-oriented architecture

- The object-oriented paradigm, like the **abstract data type** paradigm from which it evolved, emphasizes the bundling of data and methods to manipulate and access that data (Public Interface).
- Components of a **system** summarize data and the **operations** that must be **applied to manipulate the data**.
- Communication and coordination between components is **accomplished via message passing**.

Layered Architecture



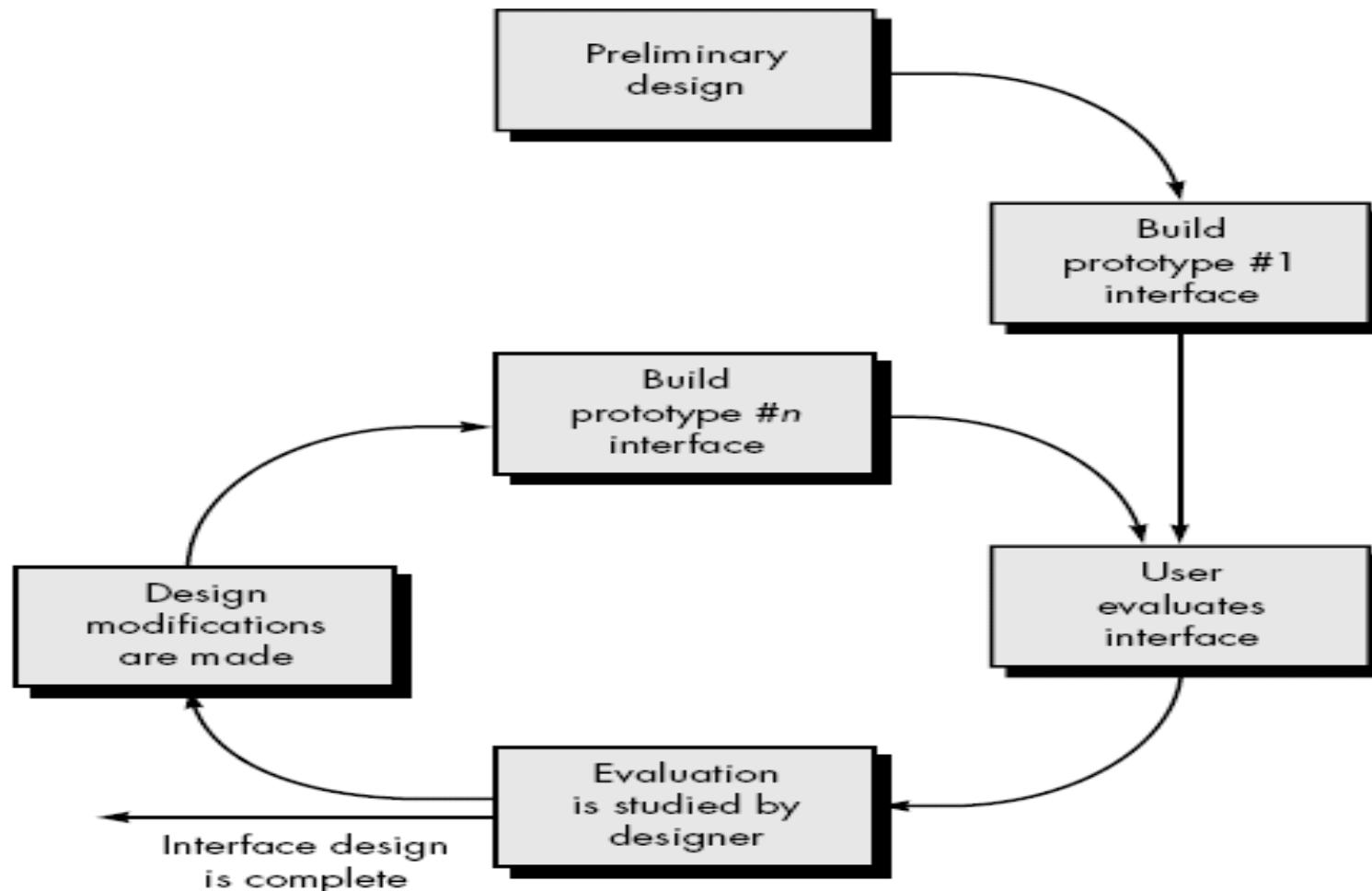
Layered Architecture

- A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
- At the outer layer, components examine user interface operations.
- At the inner layer, components examine operating system interfacing.
- Intermediate layers provide utility services and application software functions.

User Interface Design

- User interface design creates an effective communication medium between a human and a computer.
- Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

Design Evaluation



user interface evaluation cycle

- After the design model has been completed, a first-level prototype is created.
- The prototype is evaluated by the user, who provides the designer with direct comments about the efficiency of the interface.
- In addition, if formal evaluation techniques are used (e.g., questionnaires, rating sheets), the designer may extract information from these data.
- Design modifications are made based on user input and the next level prototype is created.

- The evaluation cycle continues until no further modifications to the interface design are necessary.
- The prototyping approach is effective, but is it possible to evaluate the quality of a user interface before a prototype is built?
- If potential problems uncovered and corrected early, the number of loops through the evaluation cycle will be reduced and development time will shorten.

- Evaluation criteria can be applied during early design reviews:
 1. The length and complexity of the written specification of the system and its interface provide an indication of the amount of learning required by users of the system.
 2. The number of user tasks specified and the average number of actions per task provide an indication of interaction time and the overall efficiency of the system.
 3. The number of actions, tasks, and system states indicated by the design model imply the memory load on users of the system.
 4. Interface style, help facilities, and error handling protocol provide a general indication of the complexity of the interface and the degree to which it will be accepted by the user.

- Once the first prototype is built, the designer can collect a variety of qualitative and quantitative data that will assist in evaluating the interface.
- To collect qualitative data, questionnaires can be distributed to users of the prototype.
- Questions can be all
 - simple yes/no response,
 - numeric response,
 - scaled (subjective) response,
 - percentage (subjective) response.
 - Liker scale (e.g. strongly disagree, somewhat agree).
 - Open-minded.

Example

1. Were the icons self-explanatory? If not, which icons were unclear?
2. Were the actions easy to remember and to invoke?
3. How many different actions did you use?
4. How easy was it to learn basic system operations (scale 1 to 5)?
5. Compared to other interfaces you've used, how would this rate—top 1%, top 10%, top 25%, top 50%, bottom 50%?

- If quantitative data desired, a form of time study analysis can be conducted.
- Users are observed during interaction, and data such as
 - number of tasks correctly completed over a standard time period,
 - frequency & sequence of actions,
 - time spent "looking" at the display,
 - number and types of errors,
 - error recovery time,
 - time spent using help, and number of help references per standard time period.

Software Engineering

Lecture 07 **Project Management Concept**

Project management concerns

Manager concerns about following issues:

- Product quality
- Risk Assessment
- Measurement
- Cost Estimation
- Project Schedule
- Customer Communication
- Staffing
- Other Resources
- Project Monitoring

Why Project Fail?

- Changing customer requirement
- Ambiguous/Incomplete requirement
- Unrealistic deadline
- An honest underestimate of effort
- Predictable and/or unpredictable risks
- Technical difficulties
- Miscommunication among project staff

Management Spectrum

- Effective project management focuses on four aspects of the project known as the 4 P's:
 - People - **The most important element of a successful project.** (recruiting, selection, performance
 - management, training, compensation, career development, organization, work design, team/culture development)
 - Product - **The software to be built** (product objectives, scope, alternative solutions, constraint)
 - Process - **The set of framework activities and software engineering tasks to get the job done** (framework activities populated with tasks, milestones, work products, and QA points)
 - Project - **All work required to make the product a reality.** (planning, monitoring, controlling)

People

Player of the project:

- The Stakeholders
- Team leaders
- The Software Team
- Agile Team (Implementer)
- Coordination and Communication Issues.

Stakeholders

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

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

Team Leaders

MOI model for leadership

-
- **Motivation** The ability to encourage (by “push or pull”) technical people to produce to their best ability.
 - **Organization** The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.
 - **Ideas or Innovation.** The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

Characteristics of effective project managers (problem solving, managerial identity, achievement, influence and team building)

Software Teams

How to lead?

How to organize?

How to collaborate?



How to motivate?

How to create good ideas?

Software Teams

The following factors must be considered when selecting a software project team structure ...

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

a typical example or pattern of something; a model

Organizational Paradigms

- Closed paradigm— structures a team along a traditional hierarchy of authority. Less likely to be innovative when working within the closed paradigm.
- Random paradigm— structures a team loosely and depends on individual initiative of the team members. It struggles when “orderly performance” is required.
- Open paradigm— attempts to structure a team in a manner that achieves some of the controls associated with the closed paradigm but also much of the innovation that occurs when using the random paradigm
- Synchronous paradigm— relies on the natural compartmentalization of a problem and organizes team members to work on pieces of the problem with little active communication among themselves

Agile Team

Small, Highly motivated project team also called Agile Team, adopts many of the characteristics of successful software projects.

- Team members must have trust in one another.
- The distribution of skills must be appropriate to the problem.
- Unconventional person may have to be excluded from the team, if team organized is to be maintained.
- Team is “self-organizing”
 - An adaptive team structure
 - Uses elements of organizational paradigm’s random, open, and synchronous paradigms
 - Significant autonomy

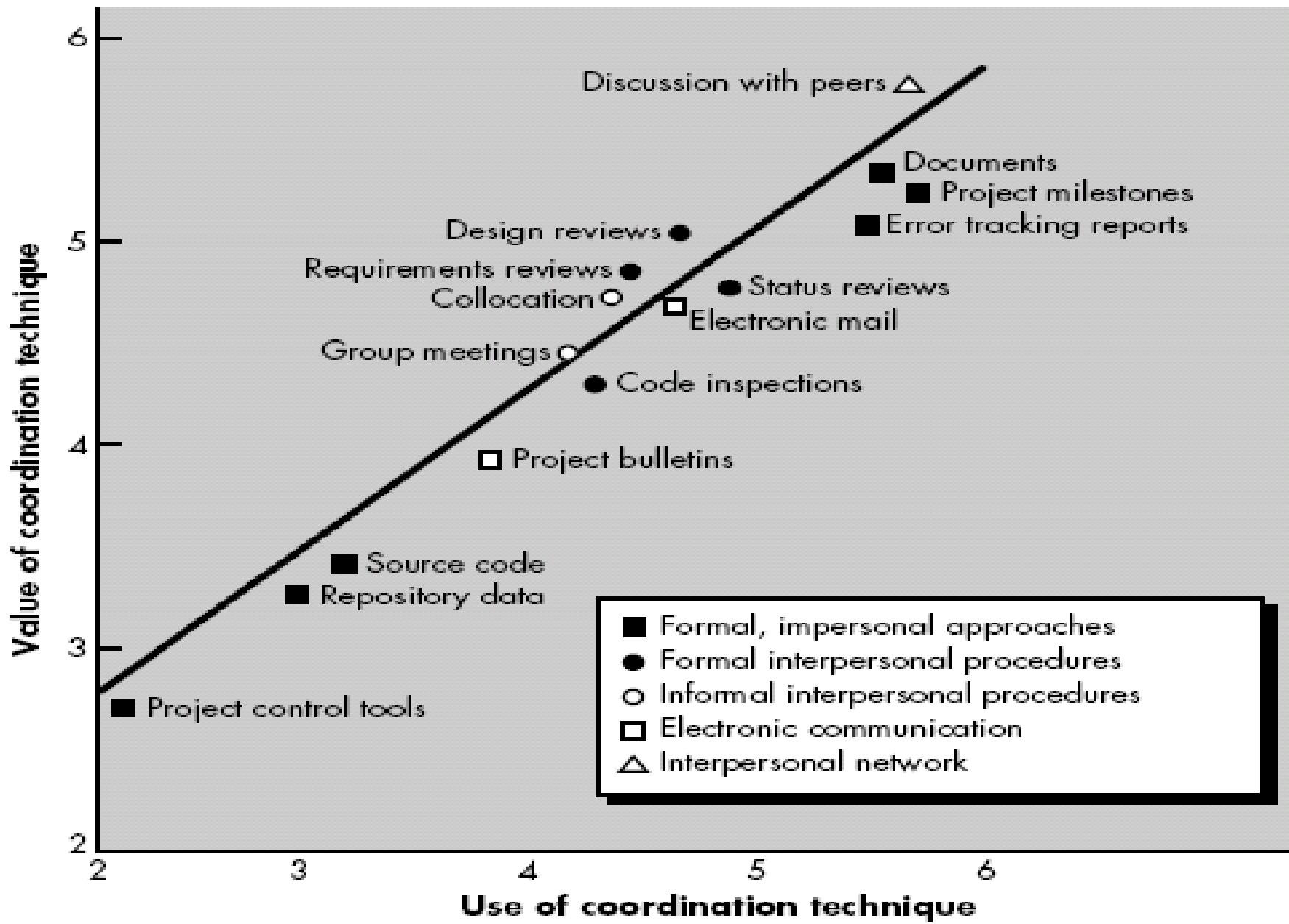
the right or condition of self-government

Team Coordination & Communication

- *Formal, impersonal approaches*
 - include software engineering documents and work products (including source code), technical memos, project milestones, schedules, and project control tools, change requests and related documentation, error tracking reports, and repository data.
- *Formal, interpersonal procedures*
 - focus on quality assurance activities applied to software engineering work products. These include status review meetings and design and code inspections.

Team Coordination & Communication

- *Informal, interpersonal procedures*
 - include group meetings for information dissemination and problem solving and spreading "collocation of requirements and development staff."
- *Electronic communication*
 - encompasses electronic mail, electronic bulletin boards, and by extension, video-based conferencing systems.
- *Interpersonal networking*
 - includes informal discussions with team members and those outside the project who may have experience or insight that can assist team members.



Product Scope

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

Problem Decomposition

- Sometimes called partitioning or problem elaboration
- Decomposition is applied in 2 major areas
 - **Functionality** that must be delivered.
 - Process that will be used to deliver it.
- Once scope is defined ...
 - It is decomposed into constituent functions
 - It is decomposed into user-visible data objects

or

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

The Process

- Process model chosen must be appropriate for the:
 - Customers and developers,
 - Characteristics of the product, and
 - Project development environment
- Once a process **framework** has been established
 - Consider project **characteristics**
 - Determine the degree of **thoroughness** required
 - Define a task set for each software engineering activity
 - Task set =
 - Software engineering tasks
 - Work products
 - Quality assurance points
 - Milestones

Melding the product and process

Melding the Product and Process

- Project planning begins with melding the product and the process
- Each function to be engineered must pass through the set of framework activities defined for a software organization
- The job of the project manager is to estimate the resources required to move each function through the framework activities to produce each work product

Process decomposition

Process decomposition begins when the project manager tries to determine how to accomplish each activity.

E.g. A small, relatively simple project might require the following work tasks for the communication activity:

1. Develop list of clarification issues.
2. Meet the customer to address clarification issues.
3. Jointly develop a statement of scope.
4. Review the statement of scope with all concerned.
5. Modify the statement of scope as required.

The Project

- Projects get into jeopardy when ...
 - Software people don't understand their customer's needs.
 - The product scope is poorly defined.
 - Changes are managed poorly.
 - The chosen technology changes.
 - Business needs change [or are ill-defined].
 - Deadlines are unrealistic.
 - Users are resistant.
 - Sponsorship is lost [or was never properly obtained].
 - The project team lacks people with appropriate skills.
 - Managers [and practitioners] avoid best practices and lessons learned.

Common-Sense Approach

- ***Start on the right foot.*** This is accomplished by working hard (very hard) to understand the problem that is to be solved and then setting realistic objectives and expectations.
- ***Maintain momentum.*** The project manager must provide incentives to keep turnover of personnel to an absolute minimum, the team should emphasize quality in every task it performs, and senior management should do everything possible to stay out of the team's way.
- ***Track progress.*** For a software project, progress is tracked as work products (e.g., models, source code, sets of test cases) are produced and approved (using formal technical reviews) as part of a quality assurance activity.
- ***Make smart decisions.*** In essence, the decisions of the project manager and the software team should be to "keep it simple."
- ***Conduct a postmortem analysis.*** Establish a consistent mechanism for extracting lessons learned for each project. Evaluate plan, schedule, analysis of project, customer feedback, etc in written form.

To Get to the Essence of a Project - W⁵HH Approach

- Boehm suggests an approach(W⁵HH) that addresses project objectives, milestones and schedules, responsibilities, management and technical approaches, and required resources.
- Why is the system being developed?
 - Enables all parties to assess the validity of business reasons for the software work
- What will be done?
 - Establish the task set that will be required.
- When will it be accomplished?
 - Project schedule to achieve milestone.
- Who is responsible?
 - Role and responsibility of each member.
- Where are they organizationally located?
 - Customer, end user and other stakeholders also have responsibility.
- How will the job be done technically and managerially?
 - Management and technical strategy must be define.
- How much of each resource is needed?
 - Develop estimation.

It applicable regardless of size or complexity of software project

Software Engineering

Lecture 08

SOFTWARE PROCESS AND PROJECT METRICS

Topic Covered

- Metrics in the process and project domains
- Process, project and measurement
- Process Metrics and Software Process Improvement

Process, project and measurement

Process Metrics:-

Are collected across all projects and over long periods of time. Their intent is to provide a set of process indicator that lead to long term software process improvement.

Project Metrics:-

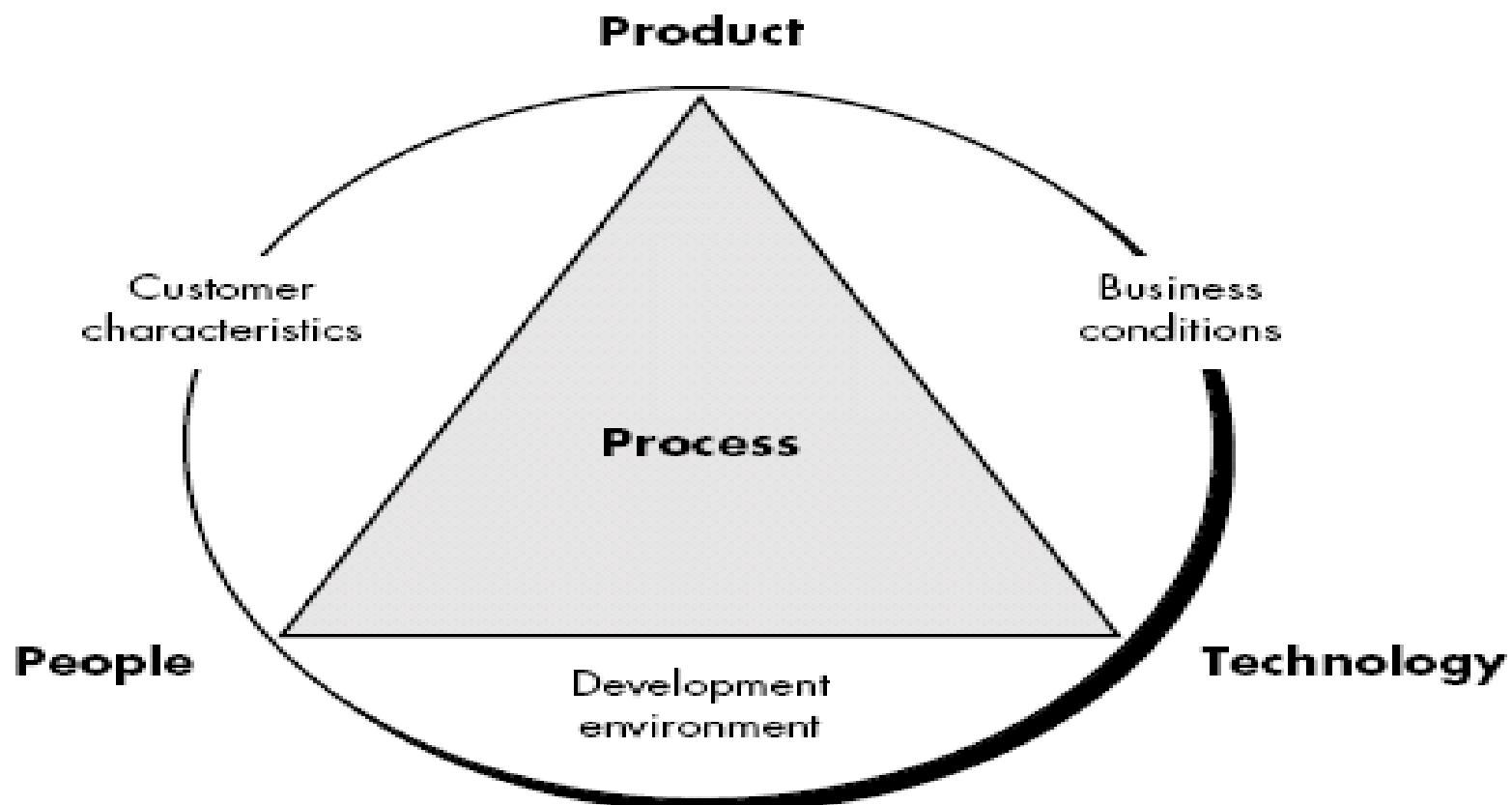
enables a software project manager to

- 1) Assess the status of an ongoing project
- 2) Track potential risks.
- 3) Uncover problem areas before they go “Critical”
- 4) Adjust work flow or tasks
- 5) Evaluate the project team’s ability to control quality of software work products.

Measurement :-

Are collected by a project team and converted into process metrics during software process improvement.

Process Metrics and Software Process Improvement



Process Metrics and Software Process Improvement

- Process at the center connecting 3 factors that have a profound influence on software quality and organizational performance.
- Process triangle exists within a circle of environmental conditions that include the development environment, business conditions and customer characteristics.
- We measure the efficiency of a software process indirectly.
 - That is, we derive a set of metrics based on the outcomes that can be derived from the process.
 - Outcomes include
 - measures of errors uncovered before release of the software
 - defects delivered to and reported by end-users
 - work products delivered (productivity)
 - human effort expended
 - calendar time expended
 - schedule conformance
 - other measures.
- We also derive process metrics by measuring the characteristics of specific software engineering tasks.

Process Metrics Guidelines

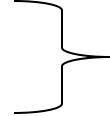
- Use common sense and organizational sensitivity when interpreting metrics data.
- Provide regular feedback to the individuals and teams who collect measures and metrics.
- Don't use metrics to appraise individuals.
- Work with practitioners and teams to set clear goals and metrics that will be used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

Project Metrics

- Used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks
- Used to assess product quality on an ongoing basis and, when necessary, modify the technical approach to improve quality.
- Every project should measure:
 - *Inputs*—measures of the resources (e.g., people, tools) required to do the work.
 - *Outputs*—measures of the deliverables or work products created during the software engineering process.
 - *Results*—measures that indicate the effectiveness of the deliverables.

Software Measurement

Categories in 2 ways:

- ***Direct measure*** of the software process & Product
 - E.g. Lines of code (LOC), execution speed, and defect)
 - ***Indirect measures*** of the product that include functionality, complexity, efficiency, reliability, maintainability etc.
 - Team A found : 342 errors
 - Team B found : 184 errors
 - It is depends on size or complexity of the projects.
- 
- Which team is more efficient ?

Size oriented metrics

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
:	:	:	:	:	:		
:	:	:	:	:	:		

The diagram illustrates the progression of project metrics over time. It features a horizontal timeline at the bottom with three distinct phases: 'analysis, design , code and test' (labeled with a green speech bubble), 'Before Release' (labeled with a green speech bubble), and 'After Release' (labeled with a green speech bubble). Above the timeline, a table lists project metrics for four projects: alpha, beta, gamma, and three unlabeled rows represented by ellipses. The metrics include LOC, Effort, cost in thousands of dollars, Pp. doc., Errors, Defects, and People. A green arrow points from the 'Before Release' phase towards the 'After Release' phase, indicating the transition of metrics post-release.

Size-Oriented metrics

Size-oriented metrics measures on LOC as normalization value.

- Errors per KLOC (thousand lines of code)
- Defects per KLOC
- \$ per LOC
- Pages of documentation per KLOC
- Errors per person-month
- Errors per review hour
- LOC per person-month
- \$ per page of documentation

Function-Oriented Metrics

- It use a measure of functionality delivered by the application as a normalization value.
- Since ‘functionality’ cannot be measured directly, it must be derived indirectly using other direct measures
- Function Point (FP) is widely used as function oriented metrics.
- FP derived using an empirical relationship based on countable (direct) measures of software’s information domain and assessments of software complexity.
- FP is based on characteristic of Software information domain and complexity.
- Like LOC measure, FP is controversial.
- FP is programming language independent.
- It is ideal for applications using conventional and nonprocedural languages.
only tell what to do (DON'T INCLUDE how to do)

Reconciling LOC and FP metric

- Relationship between lines of code and function points depends upon the programming language that is used to implement the software and the quality of the design.
- Following table provides rough estimates of the average number of LOC required to build one FP in various programming languages:

Programming Language	LOC/FP (average)
Assembly language	320
C	128
COBOL	106
FORTRAN	106
Pascal	90
C++	64
Ada95	53
Visual Basic	32
Smalltalk	22
Powerbuilder (code generator)	16
SQL	12

Software Engineering

Lecture 09 Testing Tactics

Testing Fundamental

- Software engineer attempts to build software from an abstract concept to a tangible product.
Next is Testing.
- The engineer creates a series of test cases that are intended to "demolish" the software that has been built.
- In fact, testing is the one step in the software process that could be viewed as destructive rather than constructive.

Testing Objective

- Primary Objective
 - Testing is a process of executing a program with the intent of finding an error.
 - A good test case is one that has a high probability of finding an as-yet undiscovered error.
 - A successful test is one that uncovers an as-yet-undiscovered error
- Objective is to design tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort.
- Testing demonstrates that software functions appear to be working according to specification, that behavioral and performance requirements appear to have been met.
- Data collected as testing is conducted provide a good indication of software reliability and some indication of software quality as a whole.
- But testing cannot show the absence of errors and defects, it can show only that software errors and defects are present.

Testing Principles

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

TIPSE

Software Testability

- S/w testability is simply how easily system or program or product can be tested.
- Testing must exhibit set of characteristics that achieve the goal of finding errors with a minimum of effort.

Characteristics of s/w Testability:

Choose OS OS DU
COODSSU

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

- **Observability** - "What you see is what you test."
 - Distinct output is generated for each input.
 - System states and variables are visible or queriable during execution.
 - Incorrect output is easily identified.
 - Internal errors are automatically detected & reported.
 - Source code is accessible.
- **Controllability** - "The better we can control the software, the more the testing can be automated and optimized."
 - Software and hardware states and variables can be controlled directly by the test engineer.
 - Tests can be conveniently specified, automated, and reproduced.
- **Decomposability** - By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting.
 - Independent modules can be tested independently.

- **Simplicity** - The less there is to test, the more quickly we can test it.
 - *Functional simplicity* (e.g., the feature set is the minimum necessary to meet requirements).
 - *Structural simplicity* (e.g., architecture is modularized to limit the propagation of faults).
 - *Code simplicity* (e.g., a coding standard is adopted for ease of inspection and maintenance).
- **Stability** - "The fewer the changes, the fewer the disruptions to testing."
 - Changes to the software are infrequent.
 - Changes to the software are controlled.
 - Changes to the software do not invalidate existing tests.
- **Understandability** – "The more information we have, the smarter we will test."
 - Dependencies between internal, external, and shared components are well understood.
 - Changes to the design are communicated to testers.
 - Technical documentation is instantly accessible, well organized, specific and detailed, and accurate.

Testing attributes

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

TEST CASE DESIGN

- Objectives of testing, we must design tests that have **the highest likelihood of finding the most errors with a minimum amount of time and effort.**
- Test case design methods provide a mechanism that can help to ensure the completeness of tests and provide the highest likelihood for uncovering errors in software.

Any product or system can be tested on two ways:

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

White box testing

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

White box testing

Using this method, SE can derive test cases that

1. Guarantee that all independent paths within a module have been exercised at least once
2. Exercise all logical decisions on their true and false sides,
3. Execute all loops at their boundaries and within their operational bounds
4. Exercise internal data structures to ensure their validity.

Basis path testing

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

Methods:

1. Flow graph notation
2. Independent program paths or Cyclomatic complexity
3. Deriving test cases
4. Graph Matrices

Flow Graph Notation

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

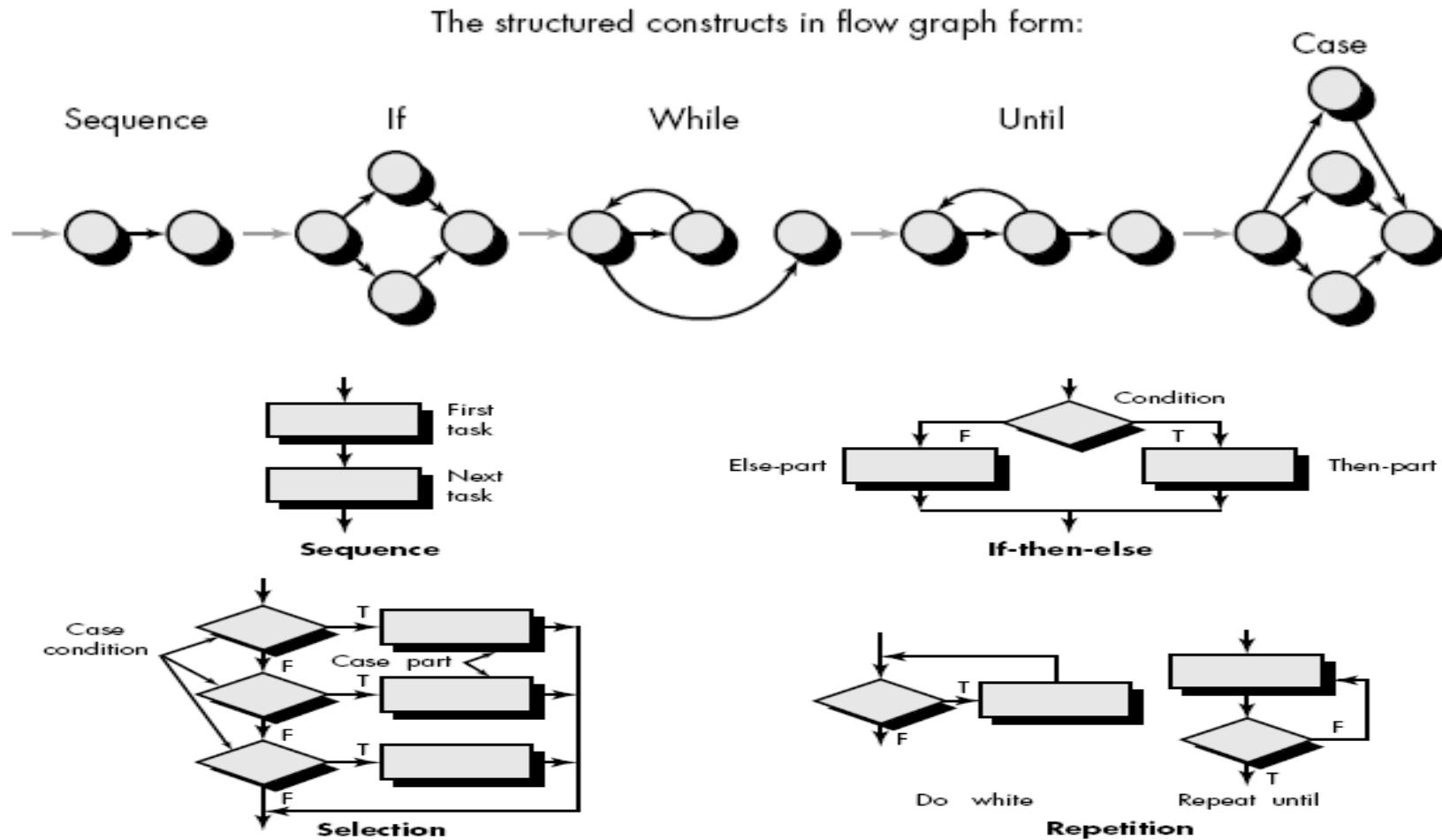
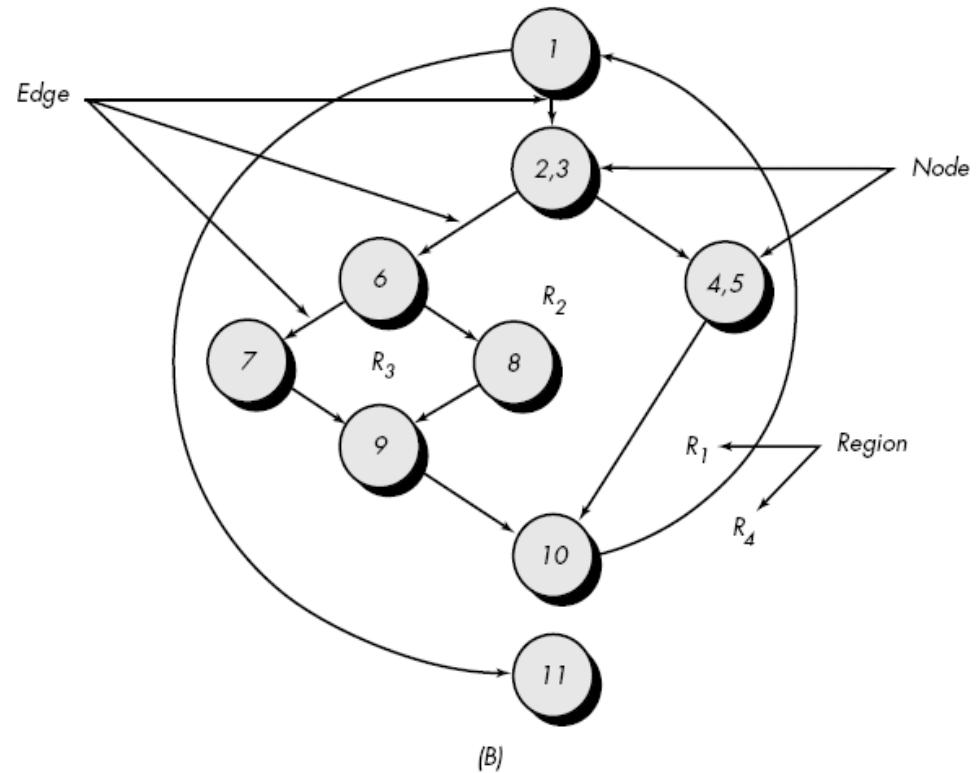
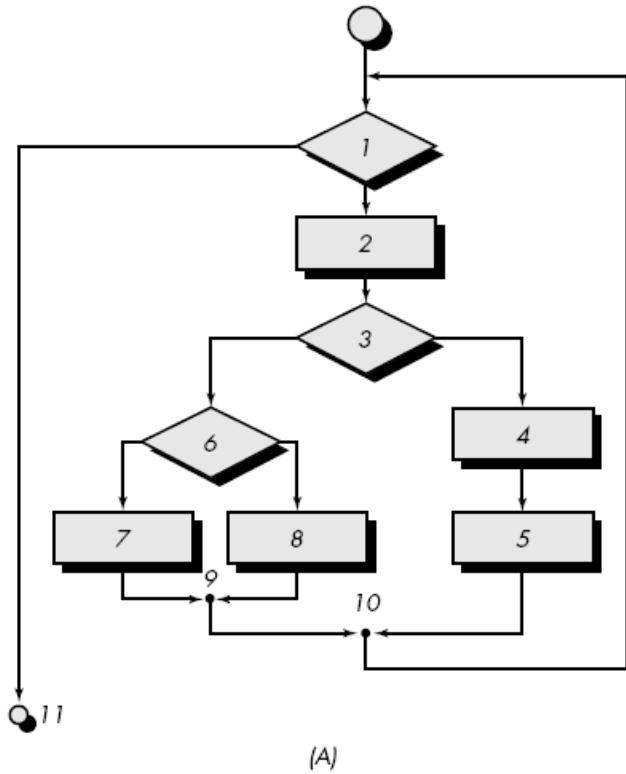
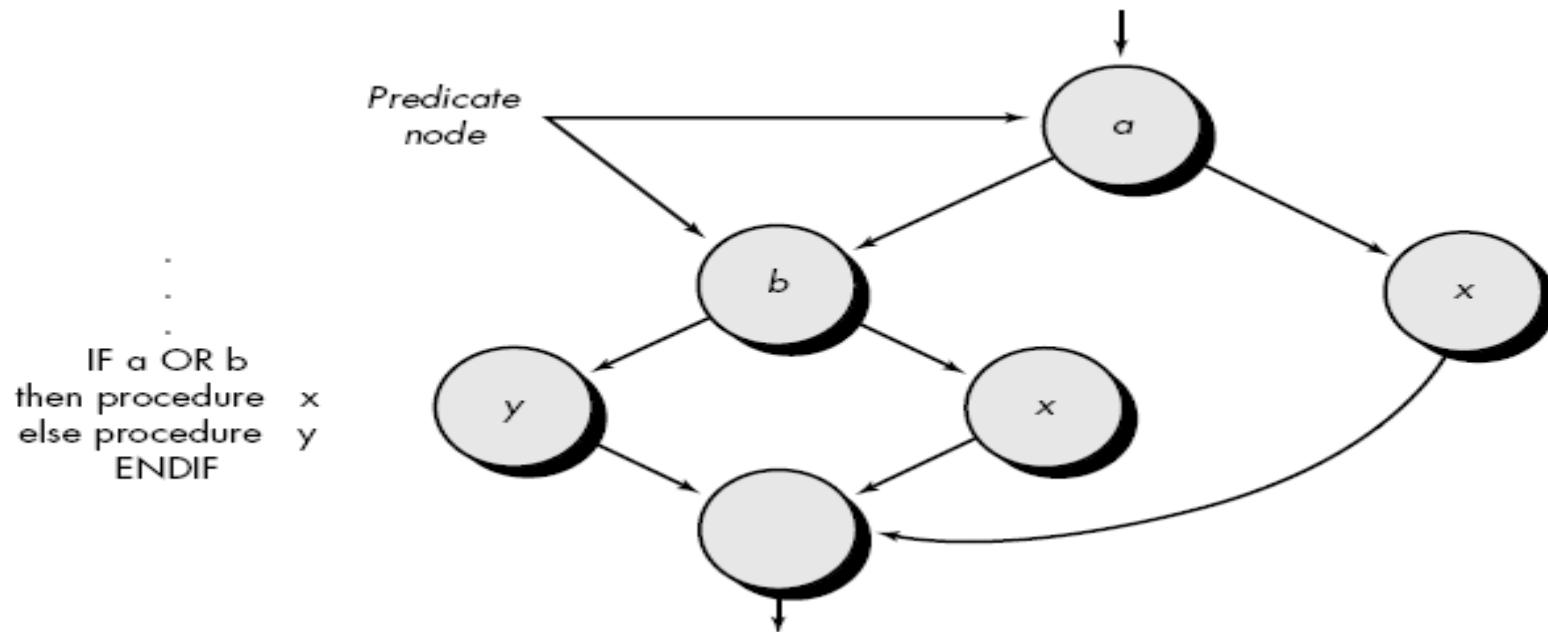


Fig. A represent program control structure and fig. B maps the flowchart into a corresponding flow graph.



In fig. B each circle, called flow graph node, represent one or more procedural statement.

- A sequence of process boxes and decision diamond can map into a single node.
- The arrows on the flow graph, called edges or links, represent flow of control and are parallel to flowchart arrows.
- An edge must terminate at a node, even if the node does not represent any procedural statement.
- Areas bounded by edges and nodes are called *regions*. When counting regions, we include the area outside the graph as a region.
- When compound condition are encountered in procedural design, flow graph becomes slightly more complicated.



- When we translating PDL segment into flow graph, separate node is created for each condition.
- Each node that contains a condition is called *predicate node* and is characterized by two or more edges comes from it.

Independent program paths or Cyclomatic complexity

- An *independent path* is any path through the program that introduces at least one *new set of processing statement or new condition*.
- For example, a set of independent paths for flow graph:
 - Path 1: 1-11
 - Path 2: 1-2-3-4-5-10-1-11
 - Path 3: 1-2-3-6-8-9-1-11
 - Path 4: 1-2-3-6-7-9-1-11
- Note that each new path introduces a new edge.
- The path 1-2-3-4-5-10-1-2-3-6-8-9-1-11 is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.
- Test cases should be designed to force execution of these paths (basis set).
- Every statement in the program should be executed at least once and every condition will have been executed on its true and false.

- How do we know how many paths to looks for ?
 - **Cyclomatic complexity** is a software metrics that provides a quantitative measure of the logical complexity of a program.
 - It defines no. of independent paths in the basis set and also provides number of test that must be conducted.
 - One of three ways to compute cyclomatic complexity:
 1. The *no. of regions* corresponds to the cyclomatic complexity.
 2. Cyclomatic complexity, $V(G)$, for a flow graph, G , is defined as
$$V(G) = E - N + 2$$
where E is the number of flow graph edges, N is the number of flow graph nodes.
 3. Cyclomatic complexity, $V(G)$, for a flow graph, G , is also defined as
$$V(G) = P + 1$$
where P is the number of predicate nodes edges
- So the value of $V(G)$ provides us with upper bound of test cases.

Deriving Test Cases

- It is a series of steps method.
- The *procedure average* depicted in PDL.
- *Average*, an extremely simple algorithm, contains compound conditions and loops.

Program Design Language

To derive basis set, follow the steps.

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

- A flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes.

Deriving Test Cases

PROCEDURE average;

- * This procedure computes the average of 100 or fewer numbers that lie between bounding values; it also computes the sum and the total number valid.

INTERFACE RETURNS average, total.input, total.valid;

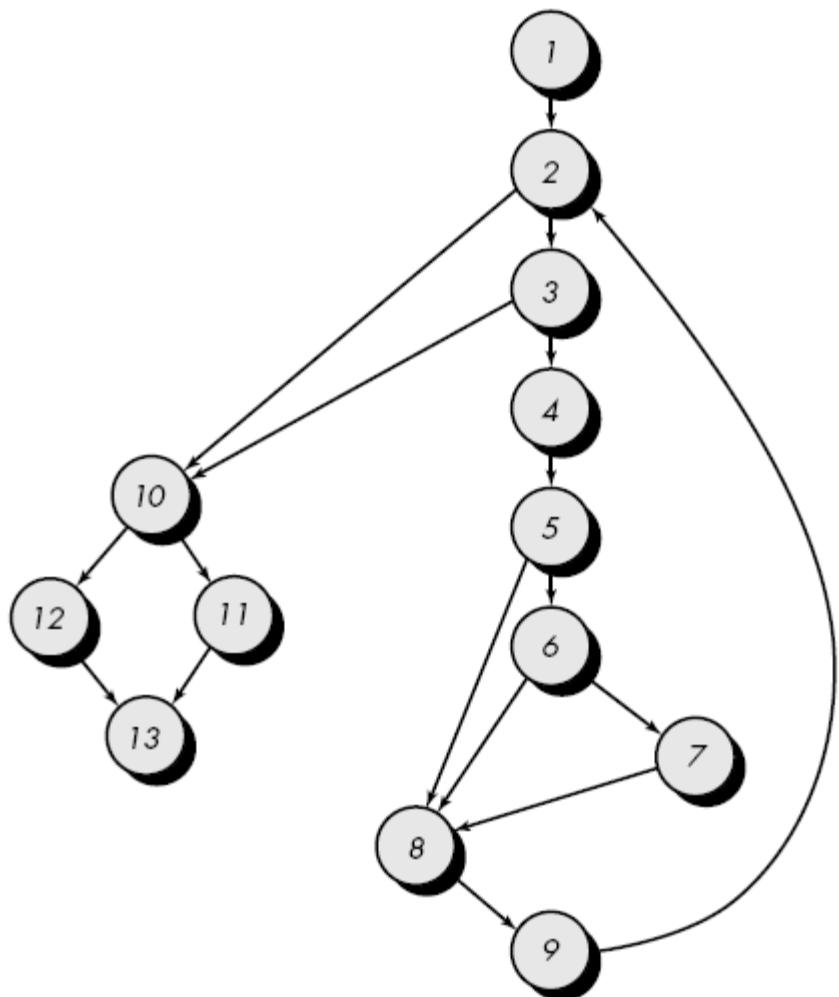
INTERFACE ACCEPTS value, minimum, maximum;

TYPE value[1:100] **IS SCALAR ARRAY**;

TYPE average, total.input, total.valid;
minimum, maximum, sum **IS SCALAR**;

TYPE i **IS INTEGER**:

```
i = 1;
total.input = total.valid = 0;
sum = 0;
DO WHILE value[i] <> -999 AND total.input < 100
    4 increment total.input by 1;
    IF value[i] >= minimum AND value[i] <= maximum
        5 THEN increment total.valid by 1;
        6 sum = sum + value[i]
        7 ELSE skip
    ENDIF
    8 increment i by 1;
9 ENDDO
IF total.valid > 0
    10 11 THEN average = sum / total.valid;
    ELSE average = -999;
12 13 ENDIF
END average
```



Flow graph for the procedure average

2. Determine the cyclomatic complexity of the resultant flow graph.

2. $V(G)$ can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure *average*, compound conditions count as two) and adding 1
3. $V(G) = 6$ regions
4. $V(G) = 17$ edges - 13 nodes + 2 = 6
5. $V(G) = 5$ predicate nodes + 1 = 6

3. Determine a basis set of linearly independent paths

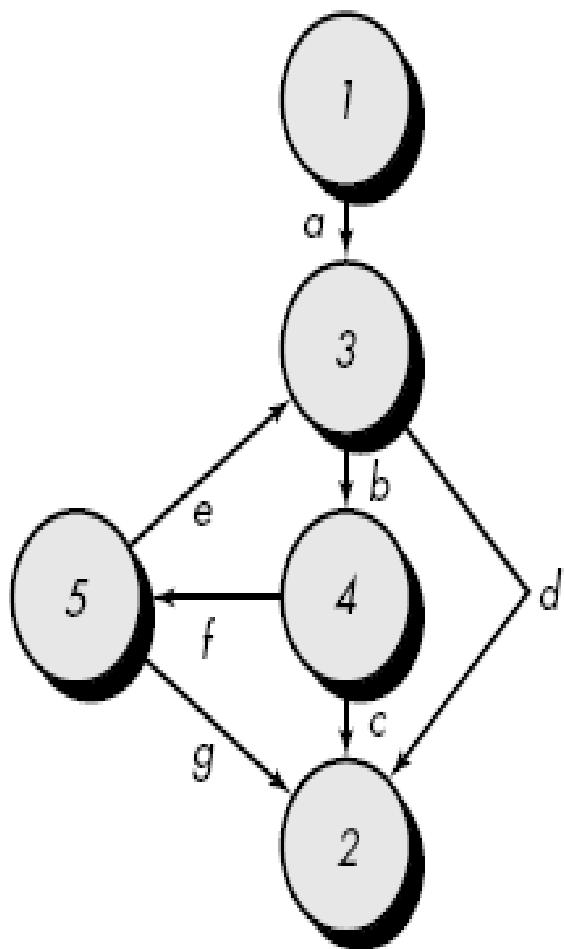
3. The value of $V(G)$ provides the number of linearly independent paths through the program control structure.
4. path 1: 1-2-10-11-13
5. path 2: 1-2-10-12-13
6. path 3: 1-2-3-10-11-13
7. path 4: 1-2-3-4-5-8-9-2-...
8. path 5: 1-2-3-4-5-6-8-9-2-...
9. path 6: 1-2-3-4-5-6-7-8-9-2-...
10. The ellipsis (...) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable.

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

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

Graph Matrices

- A *graph matrix* is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph.
- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.
- Each node on the flow graph is identify by numbers, while each edge is identify by letters.
- The graph matrix is nothing more than a tabular representation of a flow graph.
- By adding a *link weight* to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing.
- The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist).



Flow graph

Connected to node

Node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5	g	e			

Graph matrix

Connection matrix

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

Graph matrix

- Each letter has been replaced with a 1, indicating that a connection exists (this graph matrix is called a *connection matrix*).
- In fig.(connection matrix) each row with two or more entries represents a predicate node.
- We can directly measure cyclomatic complexity value by performing arithmetic operations
- Connections = Each row Total no. of entries – 1.
- $V(G)$ = Sum of all connections + 1

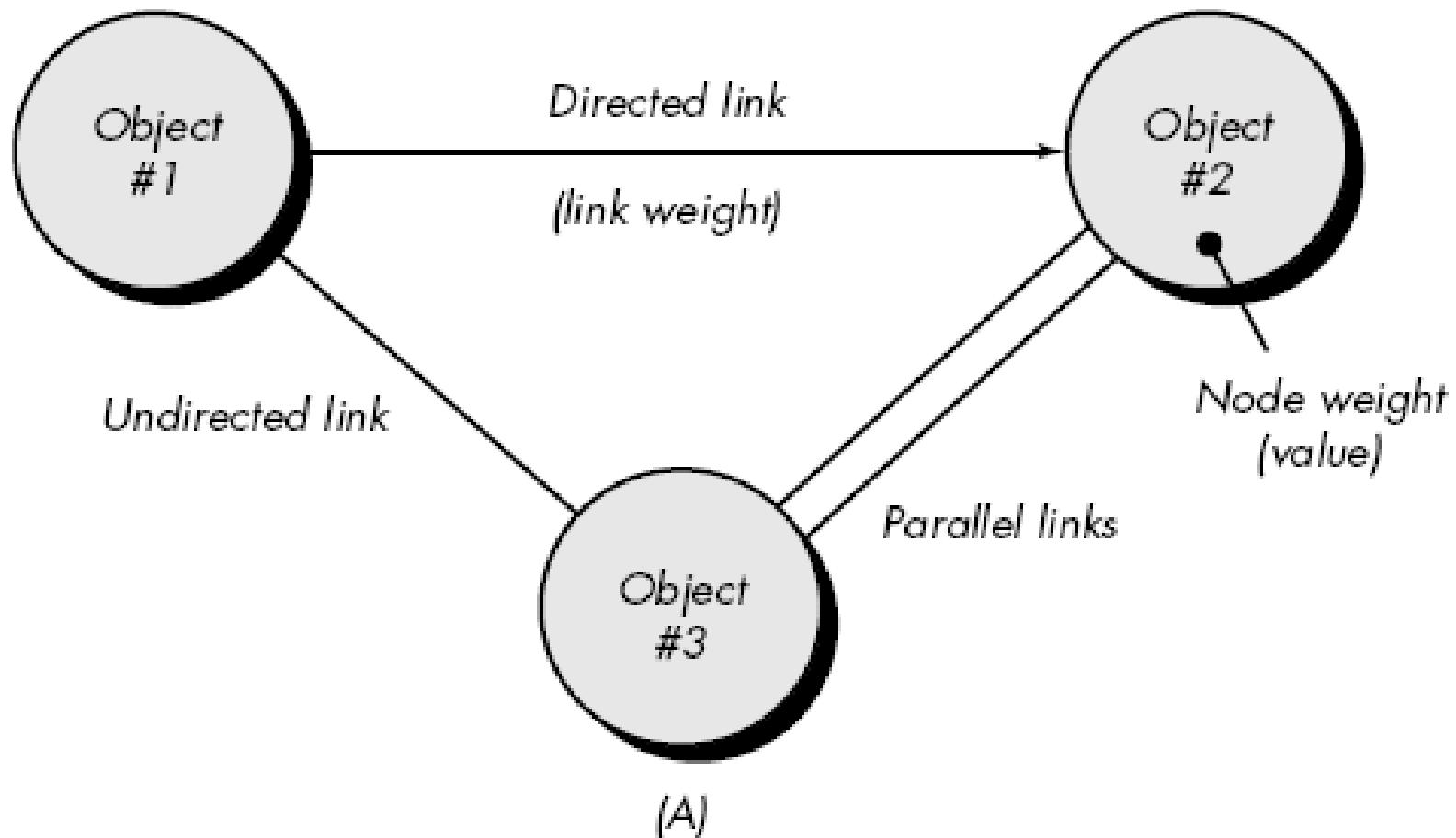
Black box testing

- Also called *behavioral testing*, focuses on the functional requirements of the software.
- It enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program.
- Black-box testing is not an alternative to white-box techniques but it is complementary approach.
- Black-box testing attempts to find errors in the following categories:
 - Incorrect or missing functions,
 - Interface errors,
 - Errors in data structures or external data base access.
 - Behavior or performance errors,
 - Initialization and termination errors.

- Black-box testing purposely ignored control structure, attention is focused on the information domain. Tests are designed to answer the following questions:
 - How is functional validity tested?
 - How is system behavior and performance tested?
 - What classes of input will make good test cases?
- By applying black-box techniques, we derive a set of test cases that satisfy the following criteria
 - Test cases that reduce the number of additional test cases that must be designed to achieve reasonable testing (i.e minimize effort and time)
 - Test cases that tell us something about the presence or absence of classes of errors
- Black box testing methods
 - Graph-Based Testing Methods
 - Equivalence partitioning
 - Boundary value analysis (BVA)
 - Orthogonal Array Testing

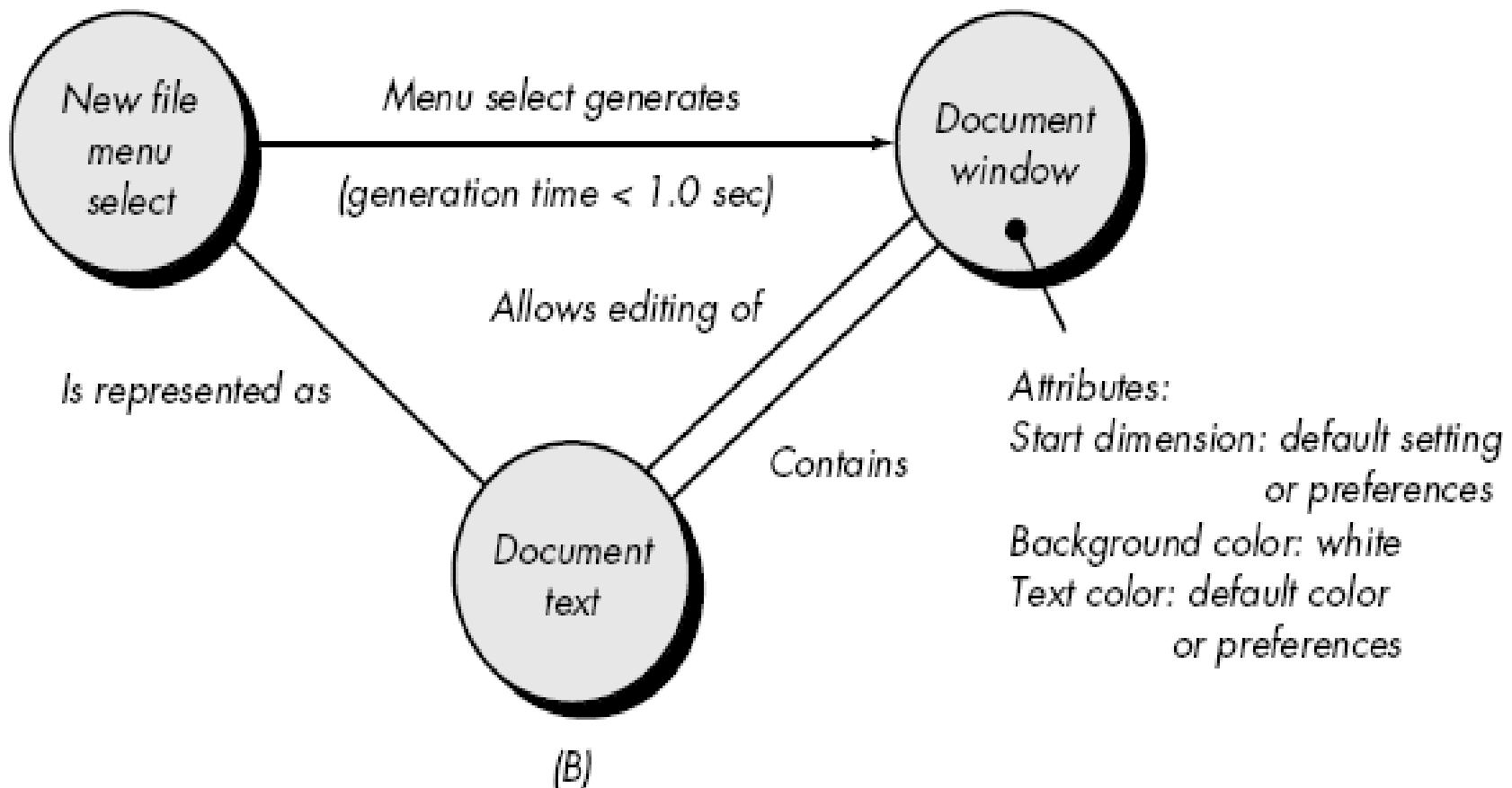
Graph-Based Testing Methods

- To understand the objects that are modeled in software and the relationships that connect these objects.
- Next step is to define a series of tests that verify “all objects have the expected relationship to one another.
- Stated in other way:
 - Create a graph of important objects and their relationships
 - Develop a series of tests that will cover the graph
- So that each object and relationship is exercised and errors are uncovered.
- **Begin by creating graph –**
 - a collection of nodes that represent objects
 - links that represent the relationships between objects
 - node weights that describe the properties of a node
 - link weights that describe some characteristic of a link.



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

Example



- *Object #1 = new file menu select*
- *Object #2 = document window*
- *Object #3 = document text*

Referring to example figure, a menu select on **new file** generates a **document window**.

- The link weight indicates that the window must be generated in less than 1.0 second.
- The node weight of **document window** provides a list of the window attributes that are to be expected when the window is generated.
- An undirected link establishes a symmetric relationship between the **new file menu select** and **document text**,
- parallel links indicate relationships between **document window** and **document text**

- Number of behavioral testing methods that can make use of graphs:
- **Transaction flow modeling.**
 - The nodes represent steps in some transaction and the links represent the logical connection between steps
- **Finite state modeling.**
 - The nodes represent different user observable states of the software and the links represent the transitions that occur to move from state to state. (Starting point and ending point)
- **Data flow modeling.**
 - The nodes are data objects and the links are the transformations that occur to translate one data object into another.
- **Timing modeling.**
 - The nodes are program objects and the links are the sequential connections between those objects.
 - Link weights are used to specify the required execution times as the program executes.

Equivalence Partitioning

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

To define equivalence classes follow the guideline

1. If an input condition specifies a *range*, one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific *value*, one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
4. If an input condition is *Boolean*, one valid and one invalid class are defined.

Example

- area code—blank or three-digit number
- prefix—three-digit number not beginning with 0 or 1
- suffix—four-digit number
- password—six digit alphanumeric string
- commands— check, deposit, bill pay, and the like

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

Boundary Value Analysis (BVA)

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

Guidelines for BVA

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

Orthogonal Array Testing

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

Example

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

Test case	Test parameters			
	P ₁	P ₂	P ₃	P ₄
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Software Engineering

Lecture 10 **Testing Strategies**

Strategic approach to software testing

- Generic characteristics of strategic software testing:
 - To perform effective testing, a software team should conduct effective *formal technical reviews*. By doing this, many errors will be eliminated *before testing start*.
 - Testing begins at the *component level* and works "*outward*" toward the integration of the entire computer-based system.
 - Different *testing techniques* are appropriate at different points in *time*.
 - Testing is conducted by the developer of the software and (for large projects) an independent test group.
 - *Testing and debugging are different activities*, but debugging must be accommodated in any testing strategy.

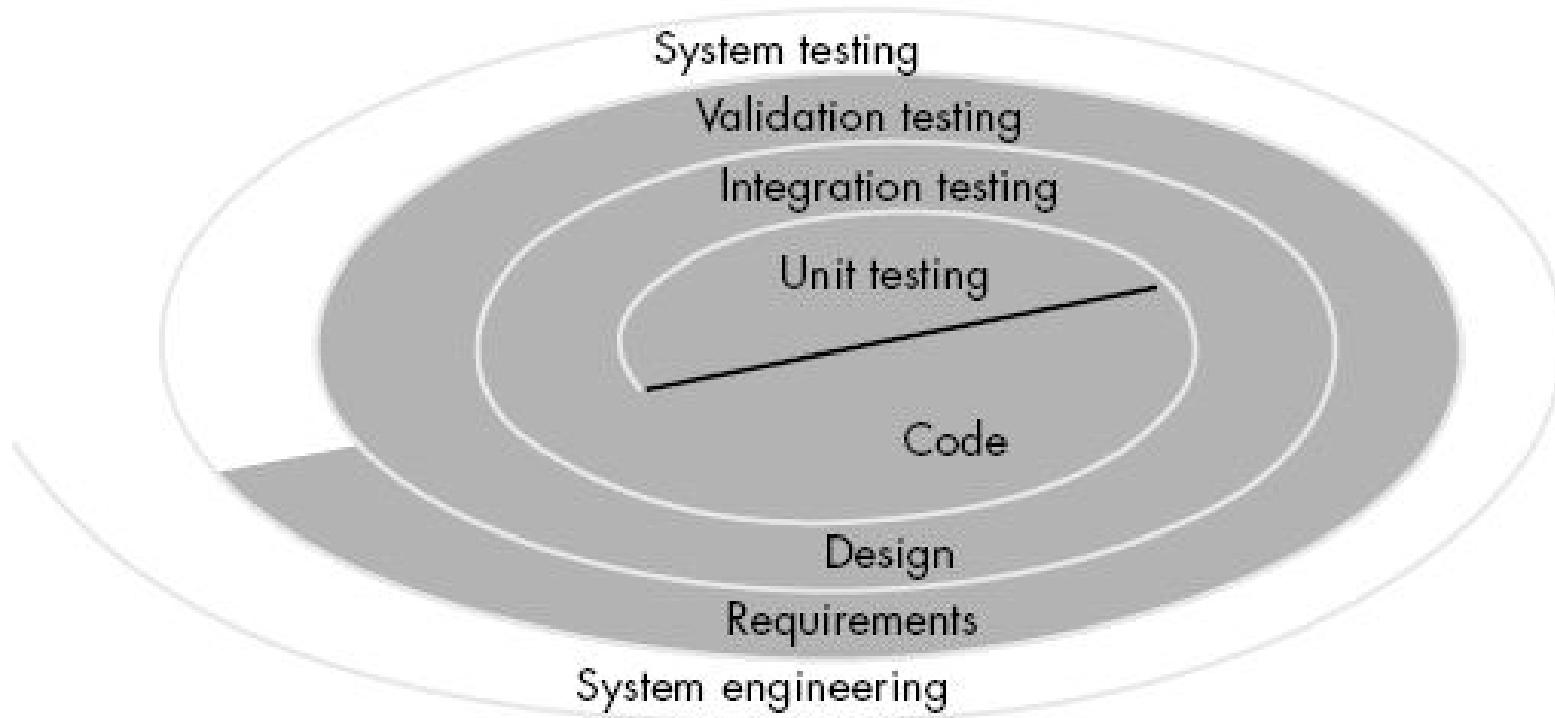
Verification and Validation

- Testing is one element of a broader topic that is often referred to as *verification and validation* (V&V).
- *Verification* refers to the set of activities that ensure that software correctly implements a specific function.
- *Validation* refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements.
- State another way:
 - *Verification*: "Are we building the product right?"
 - *Validation*: "Are we building the right product?"
- The definition of V&V encompasses many of the activities that are similar to *software quality assurance* (SQA).

- V&V encompasses a wide array of SQA activities that include
 - **Formal technical reviews,**
 - quality and configuration audits,
 - performance monitoring,
 - simulation,
 - feasibility study,
 - documentation review,
 - database review,
 - algorithm analysis,
 - development testing,
 - qualification testing, and installation testing
- Testing does provide the last bastion from which quality can be assessed and, more pragmatically, errors can be uncovered.
- Quality is not measure only by no. of error but it is also measure on **application methods**, process model, tool, formal technical review, etc will lead to quality, that is confirmed during testing.

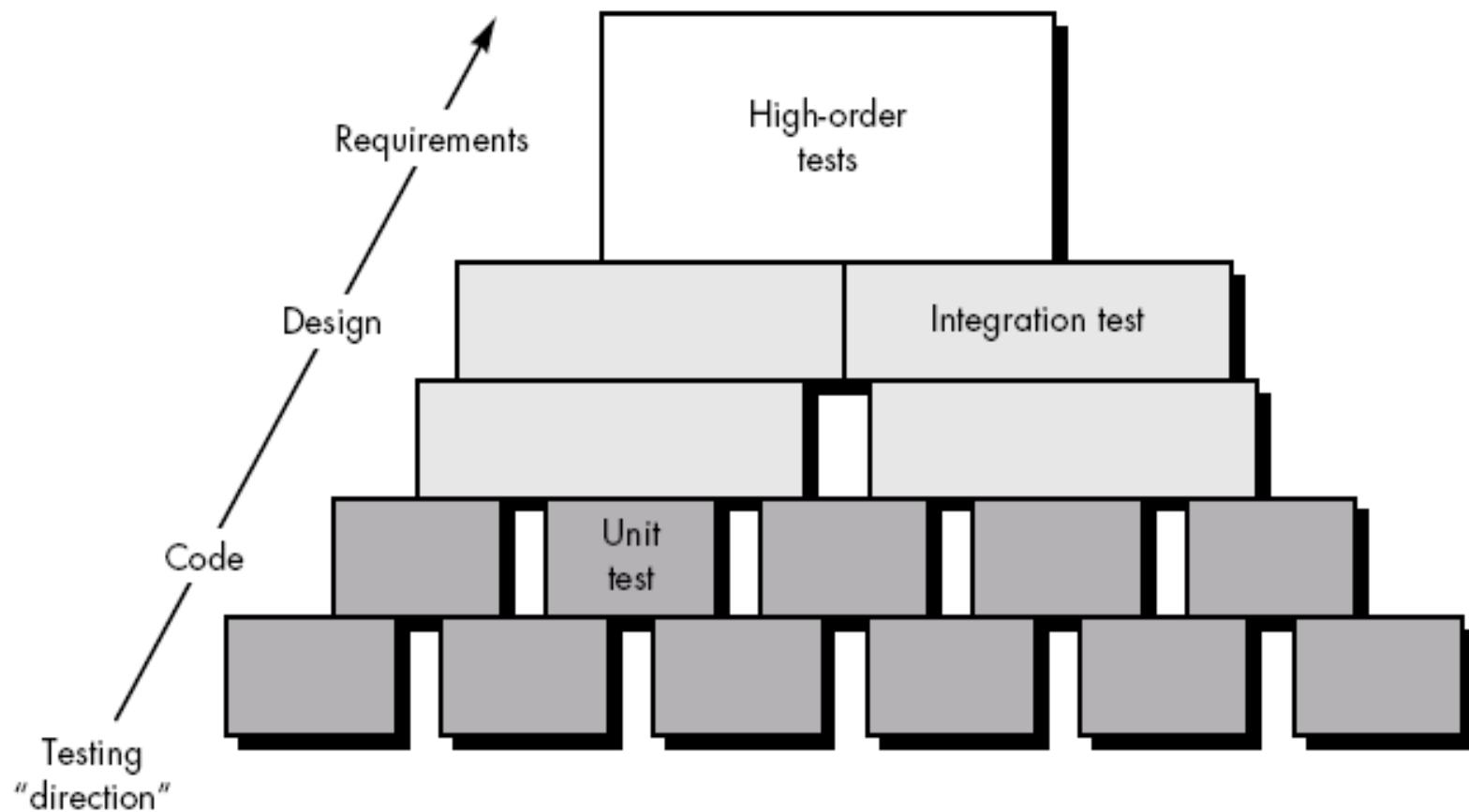
Formal Quality Performance simulate feasible documents of database and algorithm about development and qualification installation

Software Testing Strategy for conventional software architecture



U C I Do Very Regret Testing and Engineering

- A *Software process & strategy for software testing* may also be viewed in the context of the *spiral*.
- *Unit testing* begins at the vortex of the spiral and concentrates on each unit (i.e., component) of the software.
- Testing progresses by moving outward along the spiral to *integration testing*, where the focus is on design and the construction.
- Another turn outward on the spiral, we encounter *validation testing*, where requirements established as part of software requirements analysis are validated against the software.
- Finally, we arrive at *system testing*, where the software and other system elements are tested as a whole.



- Software process from a procedural point of view; a series of four steps that are implemented sequentially.

- Initially, tests focus on each component individually, ensuring that it functions properly as a unit.
- Unit testing makes heavy use of white-box testing techniques, exercising specific paths in a module's control structure.
- Integration testing addresses the issues associated with the dual problems of verification and program construction.
- Black-box test case design techniques are the most prevalent during integration.
- Now, validation testing provides final assurance that software meets all functional, behavioral, and performance requirements.
- Black-box testing techniques are used exclusively during validation.
- once validated, must be combined with other system elements (e.g., hardware, people, databases). System testing verifies that all elements mesh properly and that overall system function / performance is achieved.

Criteria for Completion of Testing

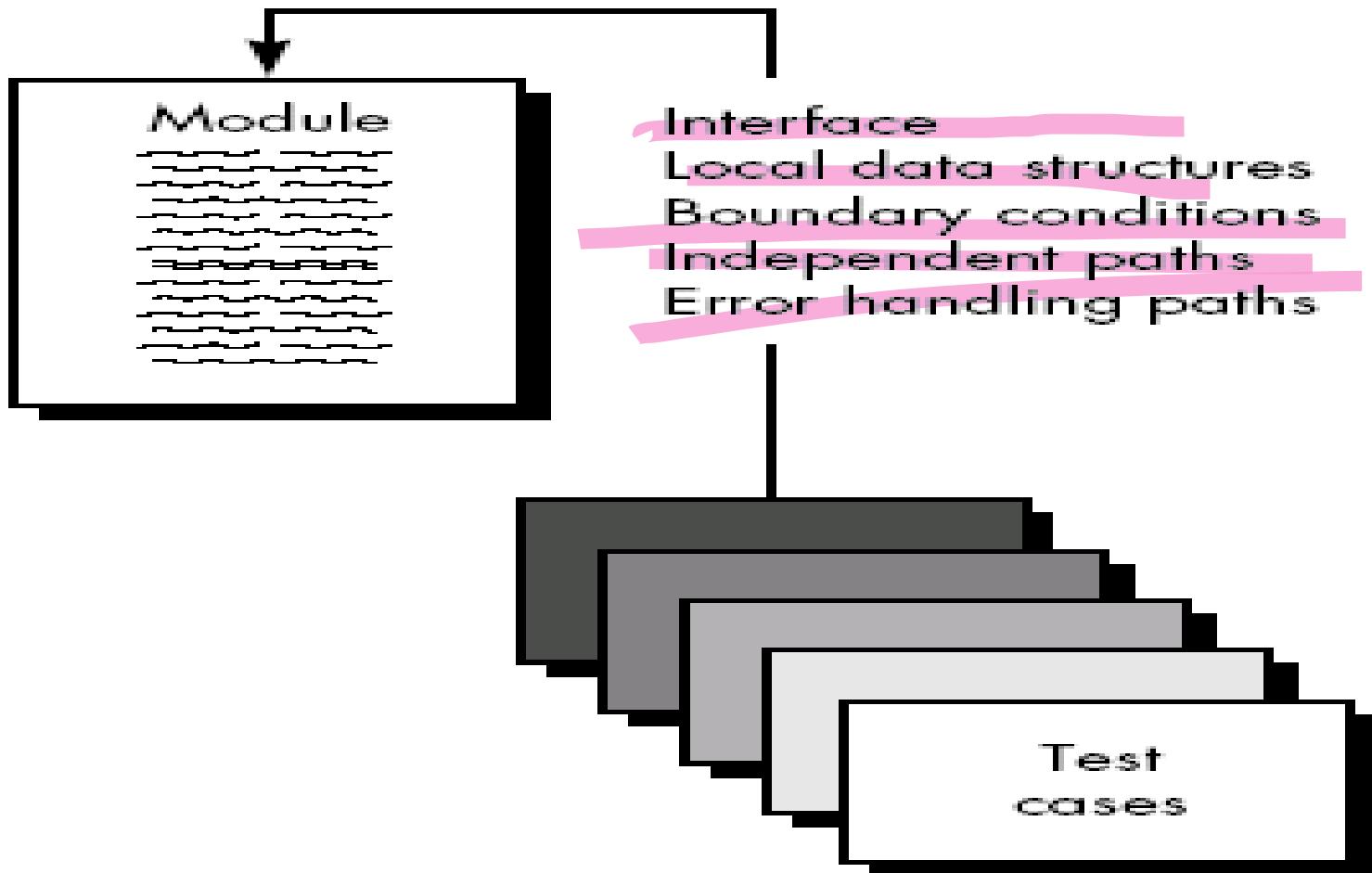
- There is no definitive answer to state that “we have done with testing”.
- One response to the question is: "You're never done testing, the burden simply shifts from you (the software engineer) to your customer." Every time the customer/ user executes a computer program, the program is being tested.
- Another response is: "You're done testing when you run out of time (deadline to deliver product to customer) or you run out of money (*spend so much money on testing*)."

- But few practitioners would argue with these responses, a software engineer needs more rigorous criteria for determining when sufficient testing has been conducted.
- Response that is based on statistical criteria: "No, we cannot be absolutely predict that the software will never fail, but relative to a theoretically sound and experimentally validated statistical model, we have done sufficient testing to say with 95 percent confidence that program will not fail."

Unit testing strategies for conventional software

- Focuses verification effort on the smallest unit of software design – component or module.
- Using the component-level design description as a guide
 - important control paths are tested to uncover errors within the boundary of the module.
- Unit test is white-box oriented, and the step can be conducted in parallel for multiple components.
- Unit test consists of
 - Unit Test Considerations
 - Unit Test Procedures

Unit Test Considerations



Contd.

- *Module interface* - information properly flows into and out of the program unit under test.
- *local data structure* - data stored temporarily maintains its integrity.
- *Boundary conditions* -module operates properly at boundaries established to limit or restrict processing
- Independent paths - all statements in a module have been executed at least once.
- And finally, all *error handling paths* are tested.

- *Module interface* are required before any other test is initiated because If data do not enter and exit properly, all other tests are debatable.
- In addition, *local data structures* should be exercised and the local impact on global data should be discover during unit testing.
- Selective testing of *execution paths* is an essential task during the unit test. Test cases should be designed to uncover errors due to
 - Computations,
 - Incorrect comparisons, or
 - Improper control flow
- *Basis path* and loop testing are effective techniques for uncovering a broad array of *path errors*.

Errors are commonly found during unit testing

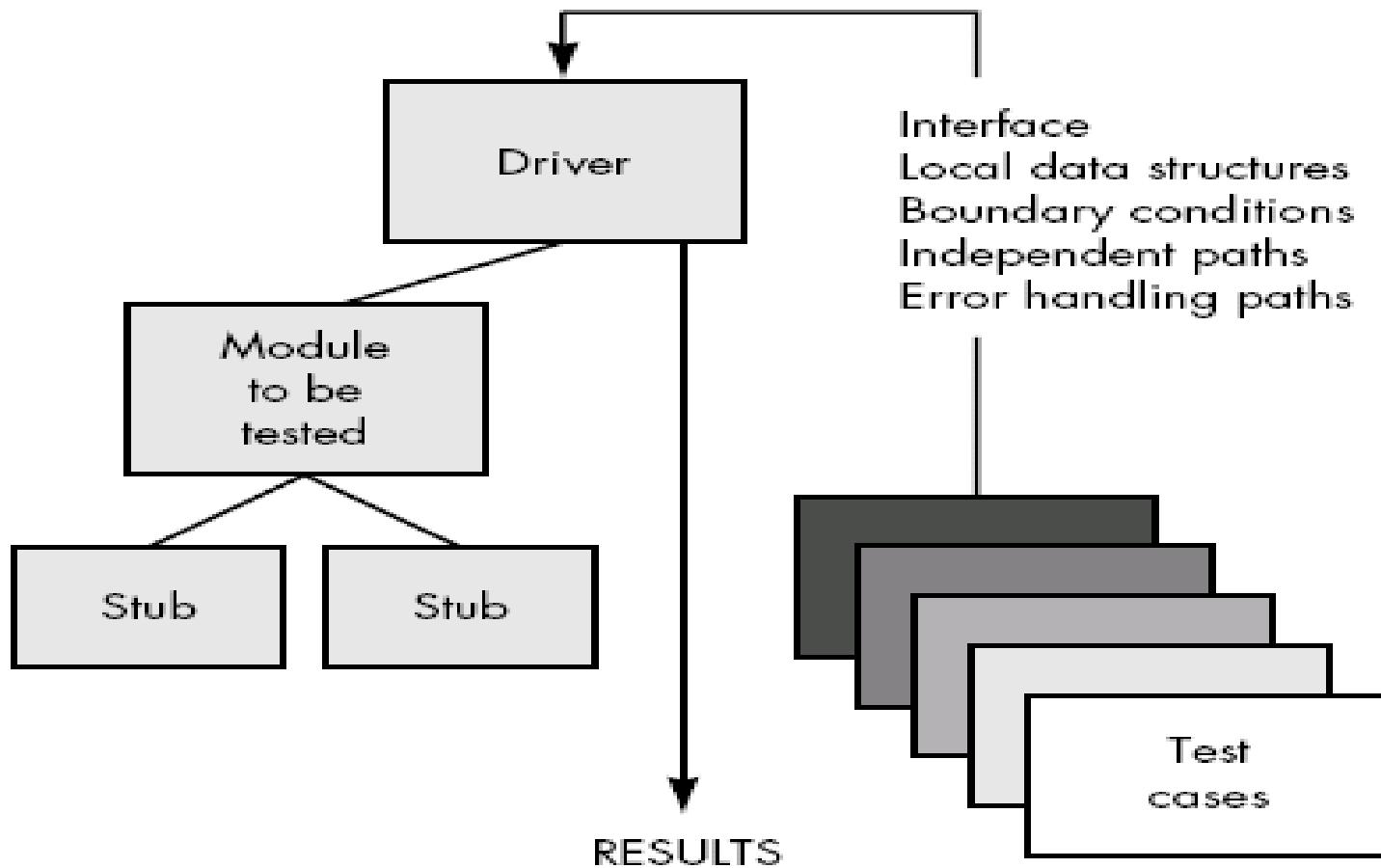
- More common errors in computation are
 - misunderstood or incorrect arithmetic precedence
 - mixed mode operations,
 - incorrect initialization,
 - precision inaccuracy,
 - incorrect symbolic representation of an expression.
- Comparison and control flow are closely coupled to one another
 - Comparison of different data types,
 - Incorrect logical operators or precedence,
 - Incorrect comparison of variables
 - Improper or nonexistent loop termination,
 - Failure to exit when divergent iteration is encountered
 - improperly modified loop variables.

- Potential errors that should be tested when error handling is evaluated are
 - Error description is unintelligible.
 - Error noted does not correspond to error encountered.
 - Error condition causes 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.
- Software often fails at its boundaries. That is, errors often occur when the n th element of an n -dimensional array is processed or when the maximum or minimum allowable value is encountered.
- So BVA test is always be a last task for unit test.
- Test cases that exercise data structure, control flow, and data values just below, at, and just above maxima and minima are very likely to uncover errors.

Unit Test Procedures

- Perform before coding or after source code has been generated.
- A review of design information provides guidance for establishing test cases. Each test case should be coupled with a set of expected results.
- Because a component is not a stand-alone program, driver and/or stub software must be developed for each unit test.
- In most applications a driver is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results.
- A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.
- *Stubs* serve to replace modules that are subordinate the component to be tested.

Unit Test Procedures



Unit Test Environment

- Drivers and stubs represent overhead. That is, both are software that must be written but that is not delivered with the final software product.
- In such cases, complete testing can be postponed until the integration test step
- Unit testing is simplified when a component with high cohesion is designed.
- When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

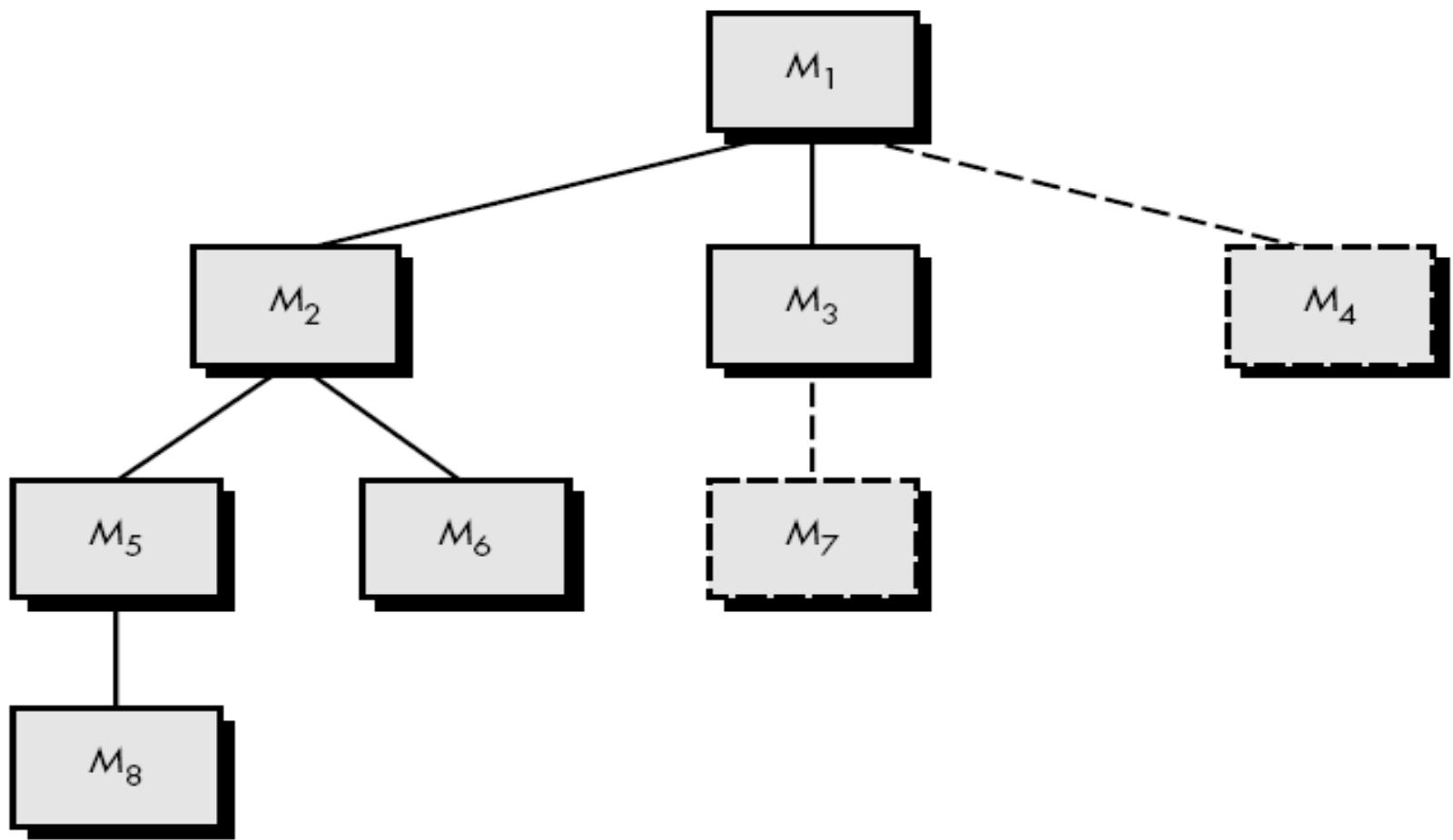
Integration testing

- Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing.
- The objective is to take unit tested components and build a program structure that has been dictated by design.
- There is often a tendency to attempt non-incremental integration; that is, to construct the program using a "big bang" approach.
- A set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program.
- Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.
- Incremental integration is the exact opposite of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct;

Top-down Integration

- *Top-down integration testing is an incremental approach to construction of program structure.*
- Modules subordinate to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.
- *Depth-first integration would integrate all components on a major control path of the structure.*
- Selection of *a major path is somewhat arbitrary and depends on application-specific characteristics.*
- For example, selecting the left hand path,
 - Components M1, M2 , M5 would be integrated first.
 - Next, M8 or M6 would be integrated
 - The *central and right hand control paths are built.*

Top down integration



- *Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally.*
- Step would be:
 - components M2, M3, and M4 would be integrated first
 - next control level, M5, M6, and so on follows.

Top-down Integration process five steps:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
3. Tests are conducted as each component is integrated
4. On completion of each set of tests, another stub is replaced with the real component.
5. Regression testing may be conducted to ensure that new errors have not been introduced.

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

Problem occur in top-down integration

- Logistic problems can arise
- most common problems occurs when processing at low levels in the hierarchy is required to adequately test upper levels.
- No significant data can flow upward in the program structure due to stubs replace low level modules at the beginning of top-down testing. In this case, Tester will have 3 choice
 - Delay many tests until stubs are replaced with actual modules
 - develop stubs that perform limited functions that simulate the actual module
 - integrate the software from the bottom of the hierarchy upward

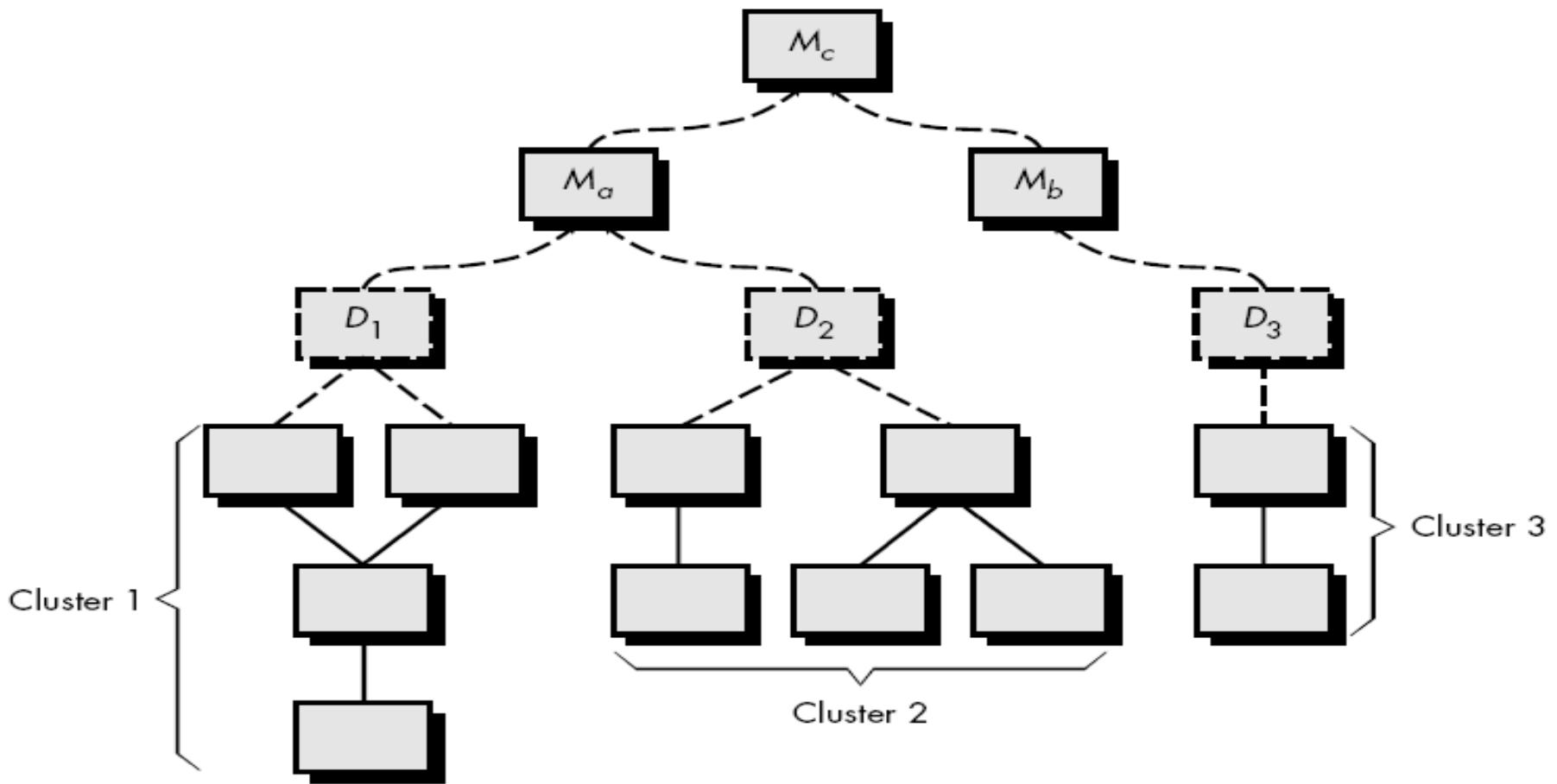
Bottom-up Integration

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

Bottom up integration process steps

- Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software sub function.
- A driver (a control program for testing) is written to coordinate test case input and output.
- The cluster is tested.
- Drivers are removed and clusters are combined moving upward in the program structure.

Bottom up integration



Example

- Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver.
- Components in clusters 1 and 2 are subordinate to Ma.
- Drivers D1 and D2 are removed and the clusters are interfaced directly to Ma. Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb.
- Both Ma and Mb will ultimately be integrated with component Mc, and so forth.

Regression Testing

- Each time a new module is added as part of integration testing
 - New data flow paths are established
 - New I/O may occur
 - New control logic is invoked
- Due to these changes, may cause problems with functions that previously worked flawlessly.
- *Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.*
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.

Contd.

- Regression testing is the activity that **helps to ensure that changes do not introduce unintended behavior or additional errors.**
- Regression testing may be conducted manually, by re-executing a subset of all test cases or using **automated capture/playback tools.**
- Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison.
- **Regression testing contains 3 diff. classes of test cases:**
 - A representative sample of tests that will **exercise all software functions**
 - **Additional tests that focus on software functions that are likely to be affected by the change.**
 - Tests that focus on the software components that have been changed.

Contd.

- As integration testing proceeds, the number of regression tests can grow quite large.
- Regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions.
- It is impractical and inefficient to re-execute every test for every program function once a change has occurred.

Smoke Testing

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

Smoke testing approach activities

- 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 is smoke tested daily.
 - The **Integration approach** may be top down or bottom up.

Smoke Testing benefits

- ***Integration risk is minimized.***
 - Smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early
- ***The quality of the end-product is improved.***
 - Smoke testing is likely to uncover both functional errors and architectural and component-level design defects. At the end, better product quality will result.
- ***Error diagnosis and correction are simplified.***
 - Software that has just been added to the build(s) is a probable cause of a newly discovered error.
- ***Progress is easier to assess.***
 - Frequent tests give both managers and practitioners a realistic assessment of integration testing progress.

What is a critical module and why should we identify it?

- As integration testing is conducted, the tester should identify *critical modules*.
- A **critical module** has one or more of the following characteristics:
 - Addresses several software requirements,
 - Has a high level of control (Program structure)
 - Is complex or error prone
 - Has definite performance requirements.
- Critical modules should be tested as early as is possible. In addition, regression tests should focus on critical module function.

Integration Test Documentation

- An overall plan for integration of the software and a description of specific tests are documented in a *Test Specification*
- It contains a test plan, and a test procedure, is a work product of the software process, and becomes part of the software configuration.
- The test plan describes the overall strategy for integration.
- Testing is divided into phases and builds that address specific functional and behavioral characteristics of the software.
- Integration testing might be divided into the following test phases:
 - User interaction
 - Data manipulation and analysis
 - Display processing and generation
 - Database management

Contd.

- Therefore, groups of modules are created to correspond to each phase.
- The following criteria and corresponding tests are applied for all test phases:
- **Interface integrity**- Internal and external interfaces are tested as each module
- **Functional validity** - Tests designed to uncover functional errors are conducted
- **Information content** - associated with local or global data structures are conducted
- **Performance** - to verify performance

Contd.

- A schedule for integration and related topics is also discussed as part of the test plan.
- Start and end dates for each phase are established
- A brief description of overhead software (stubs and drivers) concentrates on characteristics that might require special effort.
- Finally, test environment and resources are described.
- The order of integration and corresponding tests at each integration step are described.
- A listing of all test cases and expected results is also included.
- A history of actual test results, problems is recorded in the *Test Specification*.

Validation Testing

- *Validation testing* succeeds when software functions in a manner that can be *reasonably expected by the customer.*
- Like all other testing steps, validation tries to uncover errors, but the focus is at the requirements level— on things that will be immediately apparent to the end-user.
- Reasonable expectations are defined in the *Software Requirements Specification*— a document that describes all user-visible attributes of the software.
- Validation testing comprises of
 - Validation Test criteria
 - Configuration review
 - Alpha & Beta Testing

Validation Test criteria

- It is achieved through a series of tests that demonstrate agreement with requirements.
- A *test plan* outlines the classes of *tests to be conducted* and a *test procedure* defines specific test cases that will be used to demonstrate agreement with requirements.
- Both the plan and procedure are designed to ensure that
 - all functional requirements are satisfied,
 - all behavioral characteristics are achieved,
 - all performance requirements are attained,
 - documentation is correct,
 - other requirements are met
- After each validation test case has been conducted, one of two possible conditions exist:
 1. The function or performance characteristics conform to specification and are accepted
 2. A deviation from specification is uncovered and a deficiency list is created

Configuration Review

- The intent of the review is to ensure that all elements of the software configuration have been *properly developed, are cataloged*, and have the necessary detail to the support phase of the software life cycle.
- The configuration review, sometimes called an *audit*.

Alpha and Beta Testing

- When custom software is built for one customer, a series of *acceptance tests* are conducted to enable the customer to validate all requirements.
- Conducted by the end-user rather than software engineers, an acceptance test can range from an informal "test drive" to a planned and systematically executed series of tests.
- Most software product builders use a process called alpha and beta testing to uncover errors that only the end-user seems able to find.

Alpha testing

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

Beta testing

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

System Testing

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

Recovery Testing

- *Recovery testing* is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.
- If recovery is automatic (performed by the system itself), reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness.
- If recovery requires human intervention, that is mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

Security Testing

- *Security testing* attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper break through .
- During security testing, the tester plays the role(s) of the individual who desires to break through the system.
- Given enough time and resources, good security testing will ultimately penetrate a system.
- The role of the system designer is to make penetration cost more than the value of the information that will be obtained.
- The tester may attempt to acquire passwords through externally, may attack the system with custom software designed to breakdown any defenses that have been constructed; may browse through insecure data; may purposely cause system errors.

Stress Testing

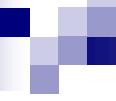
- *Stress testing* executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.

For example,

1. special tests may be designed that generate ten interrupts per second
 2. Input data rates may be increased by an order of magnitude to determine how input functions will respond
 3. test cases that require maximum memory or other resources are executed
 4. test cases that may cause excessive hunting for disk-resident data are created
- A variation of stress testing is a technique called *sensitivity testing*

Performance Testing

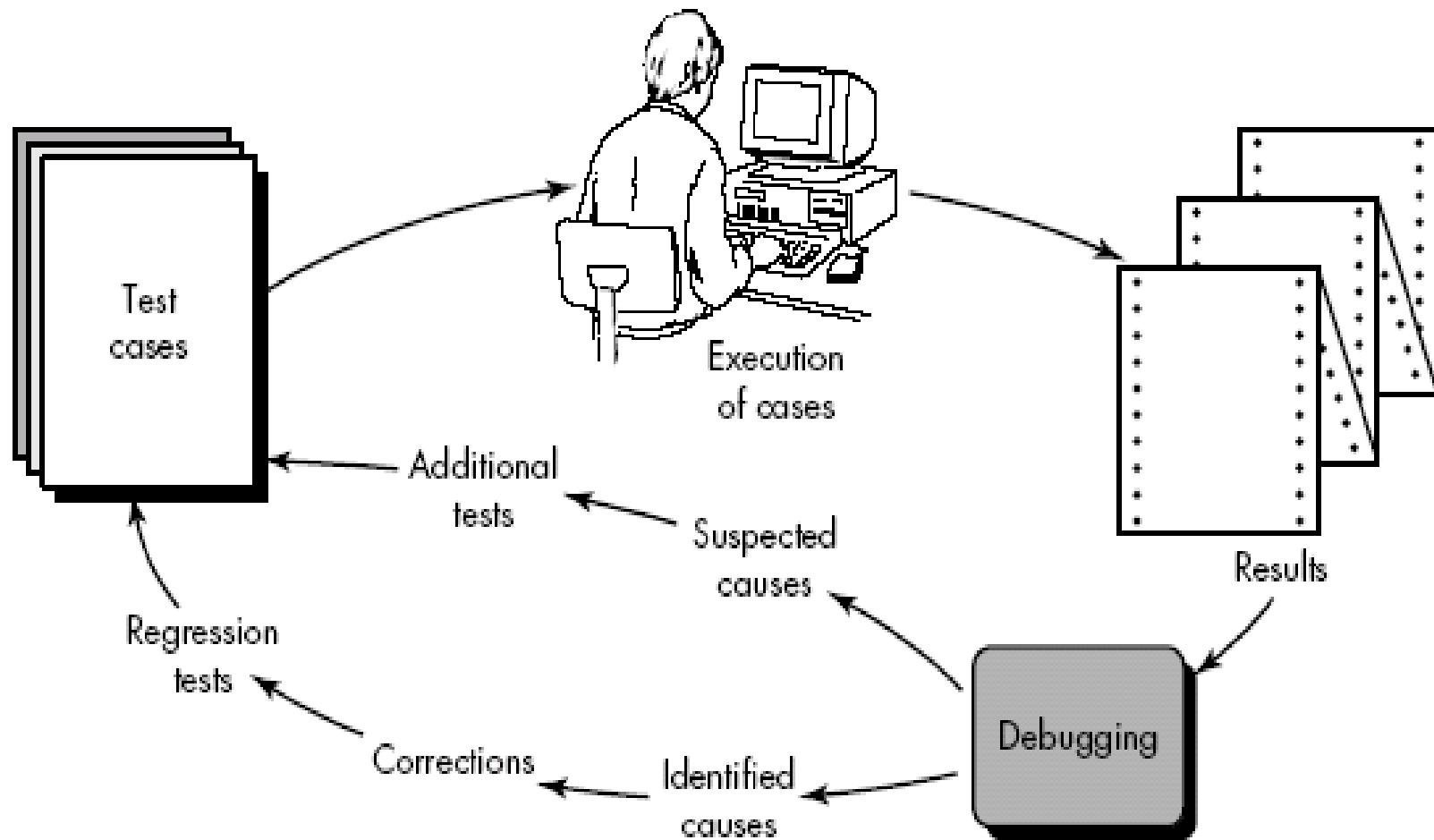
- Performance testing occurs throughout all steps in the testing process.
- Even at the unit level, the performance of an individual module may be assessed as white-box tests are conducted.
- Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation
- It is often necessary to measure resource utilization (e.g., processor cycles).



THE ART OF DEBUGGING

- Debugging is the process that results in the removal of the error.
- Although debugging can and should be an orderly process, it is still very much an art.
- Debugging is not testing but always occurs as a consequence of testing.

Debugging Process



Debugging Process

- Results are examined and a lack of correspondence between expected and actual performance is encountered (due to cause of error).
- Debugging process attempts to match symptom with cause, thereby leading to error correction.
- One of two outcomes always comes from debugging process:
 - The cause will be found and corrected,
 - The cause will not be found.
- The person performing debugging may suspect a cause, design a test case to help validate that doubt, and work toward error correction in an iterative fashion.

Why is debugging so difficult?

1. The symptom may disappear (temporarily) when another error is corrected.
2. The symptom may actually be caused by non-errors (e.g., round-off inaccuracies).
3. The symptom may be caused by human error that is not easily traced (e.g. wrong input, wrongly configure the system)
4. The symptom may be a result of timing problems, rather than processing problems.(e.g. taking so much time to display result).
5. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).

occurring at irregular intervals; not continuous or steady

6. The symptom may be **intermittent** (connection irregular or broken). This is particularly common in **embedded systems** that couple hardware and software
7. The symptom may be due to causes that are **distributed across a** number of tasks running on different processors

As the consequences of an error increase, the amount of pressure to find the cause also increases. Often, pressure sometimes forces a software developer to fix one error and at the same time introduce two more.

Debugging Approaches or strategies

- Debugging has one overriding objective: to find and correct the cause of a software error.
- Three categories for debugging approaches
 - Brute force
 - Backtracking
 - Cause elimination

Brute Force:

- probably the most common and least efficient method for isolating the cause of a software error.
- Apply brute force debugging methods when all else fails.
- Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with WRITE or PRINT statements
- It more frequently leads to wasted effort and time.

Backtracking:

- common debugging approach that can be used successfully in small programs.
- Beginning at the site where a symptom has been open, the source code is traced backward (manually) until the site of the cause is found.

Cause elimination

- Is cleared by induction or deduction and introduces the concept of binary partitioning (i.e. valid and invalid).
- A list of all possible causes is developed and tests are conducted to eliminate each.

Correcting the error

- The correction of a bug can introduce other errors and therefore do more harm than good.

Questions that every software engineer should ask before making the "correction" that removes the cause of a bug:

- **Is the cause of the bug reproduced in another part of the program?** (i.e. cause of bug is logical pattern)
- **What "next bug" might be introduced by the fix I'm about to make?** (i.e. cause of bug can be in logic or structure or design).
- **What could we have done to prevent this kind of bug previously?** (i.e. same kind of bug might generated early so developer can go through the steps)

Software Engineering

|

Lecture 11 **Risk Analysis and Management**

Reactive Risk Management

- Project team reacts to risks when they occur.
- More commonly, the software team does nothing about risks until something goes wrong.
- Then, the team involved into action in an attempt to correct the problem rapidly. This is often called a *fire fighting mode*.
- When this fails, “crisis management” takes over and the project is in real jeopardy.

Proactive Risk Management

- A proactive strategy begins long before technical work is initiated.
- Potential risks are identified, their probability and impact are assessed, and they are ranked by importance.
- Then, the software team establishes a plan for managing risk.
- The primary objective is to avoid risk, but because not all risks can be avoided, the team works to develop a contingency plan that will enable it to respond in a controlled and effective manner.

Software Risks

- Risk always involves two characteristics:
 - *Uncertainty* —the risk may or may not happen; that is, there are no 100% probable risks.
 - *Loss*—if the risk becomes a reality, unwanted consequences or losses will occur
- When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk.
- To accomplish this, different categories or types of risks are considered.

□ Project Risks	Please
□ Technical Risks	Take
□ Business Risks	Both
□ Known Risks.	Knowledge and
□ Predictable Risks	Power with
□ Unpredictable Risks	U

Project Risk

- ***Project risks*** make threats the project plan.
- That is, if project risks become real, it is likely that project schedule will slip and that costs will increase.
- Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, customer, and requirements problems and their impact on a software project.
- Project complexity, size, and the degree of structural uncertainty were also defined as project risk factors.

Technical risks

- *Technical risks* threaten the **quality and timeliness** of the software to be produced.
- If a technical risk becomes a reality, implementation may become **difficult or impossible**.
- Technical risks identify potential design, implementation, interface, verification, and maintenance problems.
- In addition, specification ambiguity, technical uncertainty, and "leading-edge" technology are also risk factors.

Business Risk

- *Business risks threaten the feasibility of the software to be built.*
- Business risks often jeopardize the project or the product.
- Top five business risks are
 - Building a excellent product or system that no one really wants (**market risk**),
 - Building a product that no longer fits into the overall business strategy for the company (**strategic risk**)
 - Building a product that the sales force doesn't understand how to sell.
 - **Losing the support of senior management due to a change in focus or a change in people (management risk)**
 - **Losing budgetary or personnel commitment (budget risks)**

Known Risk

- *Known risks* are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed,
- Other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment).

- *Predictable risks* are generalized from past project experience (e.g., staff turnover, poor communication with the customer, etc).
- *Unpredictable risks* are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

Risk Identification

- *Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.).*
- By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.
- Two distinct types of risks
 - *Generic risks are a potential threat to every software project*
 - *Product-specific risks can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the project at hand.*

Contd.

- To identify product-specific risks, the project Plan and the software statement of scope are examined.
- One method for identifying risks is to create a *risk item checklist*.
- The checklist can be used for risk identification and focuses on some subset of *known and predictable risks* in the following generic subcategories:
- ***Product size*** —risks associated with the overall size of the software to be built or modified.
- ***Business impact*** —risks associated with constraints imposed by management or the marketplace.

Risk Item Checklist contd.

- ***Customer characteristics*** —risks associated with the *sophistication of the customer* and the developer's ability to communicate with the customer in a timely manner.
- ***Process definition*** —risks associated with the degree to which the software process has been defined and is followed by the development organization.
- ***Development environment*** —risks associated with the *availability and quality* of the tools to be used to build the product.
- ***Technology to be built*** —risks associated with the *complexity of the system* to be built and the "newness" of the technology that is packaged by the system.
- ***Staff size and experience*** —risks associated with the overall *technical and project* experience of the software engineers who will do the work.

Risk Components

PM identify the risk drivers that affect software 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

Components		Performance	Support	Cost	Schedule
Category					
Catastrophic	1	Failure to meet the requirement would result in mission failure		Failure results in increased costs and schedule delays with expected values in excess of \$500K	
	2	Significant degradation to nonachievement of technical performance	Nonresponsive or unsupportable software	Significant financial shortages, budget overrun likely	Unachievable IOC
Critical	1	Failure to meet the requirement would degrade system performance to a point where mission success is questionable		Failure results in operational delays and/or increased costs with expected value of \$100K to \$500K	
	2	Some reduction in technical performance	Minor delays in software modifications	Some shortage of financial resources, possible overruns	Possible slippage in IOC
Marginal	1	Failure to meet the requirement would result in degradation of secondary mission		Costs, impacts, and/or recoverable schedule slips with expected value of \$1K to \$10K	
	2	Minimal to small reduction in technical performance	Responsive software support	Sufficient financial resources	Realistic, achievable schedule
Negligible	1	Failure to meet the requirement would create inconvenience or nonoperational impact		Error results in minor cost and/or schedule impact with expected value of less than \$1K	
	2	No reduction in technical performance	Easily supportable software	Possible budget underrun	Early achievable IOC

Risk Projection

- *Risk projection*, also called *risk estimation*, attempts to rate each risk in two ways:
 - The likelihood or probability that the risk is real.
 - the consequences (i.e. effect or result) of the problems associated with the risk, should it occur.
- The project planner, along with other managers and technical staff, performs four risk projection activities:
 - Establish a scale that reflects the supposed likelihood of a risk
 - Describe the consequences of the risk,
 - Estimate the impact of the risk on the project and the product,
 - Note the overall accuracy of the risk projection so that there will be no misunderstanding
- The intent of these steps is to consider risks in a manner that leads to prioritization. By prioritizing risks, the team can allocate resources where they will have the most impact.

Developing Risk Table

Risks	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End-users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet expectations	TE	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	
•				
•				
•				

Impact values:

- 1—catastrophic
- 2—critical
- 3—marginal
- 4—negligible

Procedure to build risk table

- Listing all risks in first column. This can be accomplished with the help of the risk item checklists
- Each risk is categorized in the second column
- The probability of occurrence of each risk is entered in the next column of the table. Which can be estimated by team members.
- Impact of each risk is assessed. Each *risk component* is assessed using the characterization and an impact categories like *catastrophic, critical, marginal and negligible* are determined.
- Once table is completed, manager will give order of prioritization to the risk. Therefore, the table is sorted by probability and by impact.

Contd.

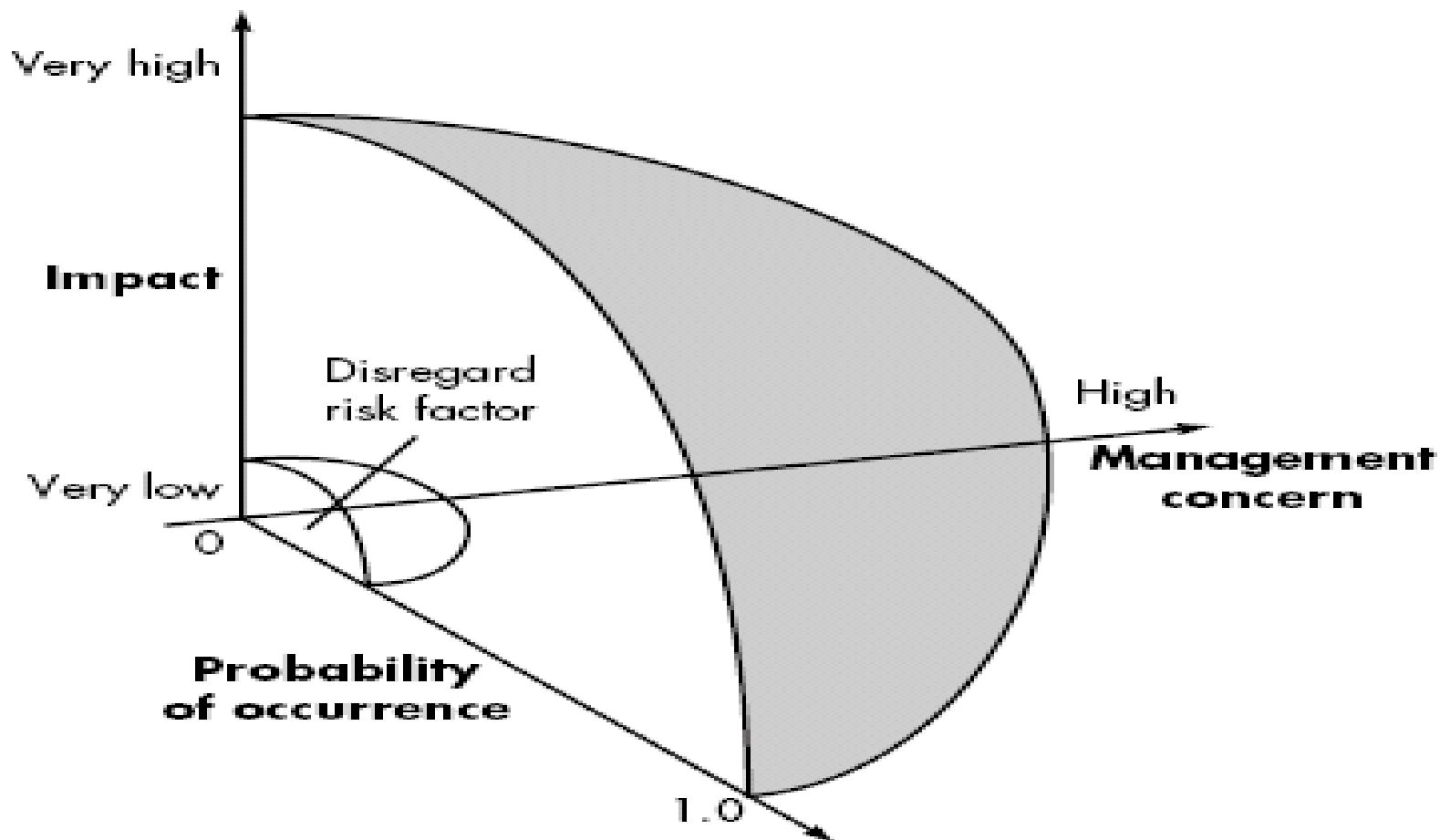
- High-probability, high-impact risks get into the top of the table, and low-probability risks drop to the bottom. (First order prioritization).
- The project manager studies the resultant sorted table and defines a cutoff line.
- The *cutoff line* implies that only risks that lie above the line will be given further attention.
- Risks that fall below the line are considered as second-order prioritization.

Contd.

- Risk impact and probability have a distinct influence on management concern.
- Risk factor that has a high impact but a very low probability of occurrence then management will give little attention or some time no attention.
- But if risk factor that has high impact and high probability of occurrence then management will give high attention.
- All risks that lie above the cutoff line must be managed and specify in last column of the table under RMMM column.

Risk Mitigation, Monitoring, and Management Plan

Contd.



Assessing Risk Impact

- Three factors affect the consequences that are likely if a risk does occur:
 - Nature,
 - Scope, and
 - Timing.
- The *nature* of the risk indicates the problems that are likely if it occurs.
For example, a technical risk, development environment change
- The *scope* of a risk combines the strictness with its overall distribution.
- For ex. how much of the project will be affected or how many customers are harmed?
- The *timing* of a risk considers when and for how long the impact will be felt.



To determine the overall consequences of a risk:

- Determine the average probability of occurrence value for each risk component.
- Determine the impact for each component based on the criteria.
- Complete the risk table and analyze the results as described

Now measure, Risk exposure (RE).

$$\text{RE} = P \times C$$

P is the probability of occurrence for a risk and C is the cost to the project.

Example- the software team defines a project risk in the following manner

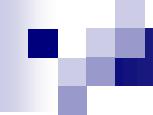
- **Risk Identification** - Only 70 percent of the software components scheduled for reuse and remaining functionality will have to be custom developed.
- **Risk probability.** 80% (likely).
- **Risk Impact** – Assume total no. of component is 60. If only 70 percent can be used, 18 components would have to be developed from scratch. $60 - 60 \times 0.7 = 18$
- Since the average component is 100 LOC and local data indicate that the software engineering cost for each LOC is \$14.00,
- the overall cost (impact) to develop the components would be
$$18 \times 100 \times 14 = \$25,200.$$
- **Risk exposure.** $RE = 0.80 \times 25,200 \sim \$20,200.$

Contd.

- once an estimate of the cost of the risk is derived, compute RE for each risk in risk table.
- The total risk exposure for all risks (above the cutoff in the risk table) can provide a means for adjusting the final cost estimate for a project.
- The project team should revisit the risk table at regular intervals, re-evaluating each risk to determine when new circumstances cause its probability and impact to change.
- As a consequence of this activity, it may be necessary to add new risks to the table, remove some risks that are no longer relevant, and change the relative positions of still others.
- *Compare RE for all risks to the cost estimate for the project. If RE is greater than 50 percent of project cost, the feasibility of the project must be evaluated.*

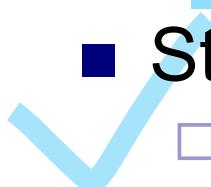
Risk Mitigation, Monitoring, and Management

- Risk analysis goal - to assist the project team in developing a strategy for dealing with risk. An effective strategy must consider three issues:
 - Risk avoidance or mitigation.
 - Risk monitoring
 - Risk management and contingency planning
- **Proactive approach** to risk, avoidance is always the best strategy. This is achieved by developing a plan for *risk mitigation*.
- For example, assume that high staff turnover (i.e. revenue) is noted as a project risk.



- To mitigate this risk, project management must develop a strategy for reducing turnover.

- Steps are:

- 
- Meet with current staff to determine causes for turnover (e.g., low pay, competitive job market).
 - Mitigate those causes that are under our control before the project starts.
 - Organize project teams so that information about each development activity is widely dispersed. scatter
 - Define documentation standards and establish mechanisms to be sure that documents are developed in a timely manner.
 - Conduct peer reviews of all work
 - Assign a backup staff member for every critical technologist.

- As the project proceeds, *risk monitoring* activities commence.
- The project manager monitors factors that may provide an indication of whether the risk is becoming more or less likely.
- In the case of high staff turnover, the following factors can be monitored:
 - General attitude of team members based on project pressures.
 - Potential problems with compensation and benefits.
 - The availability of jobs within the company and outside it.
- *Risk management and contingency planning* assumes that mitigation efforts have failed and that the risk has become a reality.

- The project is well underway and a number of people announce that they will be leaving. If the mitigation strategy has been followed, backup is available, information is documented, and knowledge has been dispersed across the team.
- In addition, the project manager may temporarily refocus resources (and readjust the project schedule) to those functions that are **fully staffed**, enabling newcomers who must be added to the team to “get up to speed.”
- Those individuals who are leaving are asked to stop all work and spend their last weeks in “knowledge transfer mode.”
- This might include video-based knowledge capture, the development of “commentary documents,” and/or meeting with other team members who will remain on the project.

RMMM steps incur additional project cost. For example, spending the time to “backup” every critical technologist costs money.

THE RMMM PLAN

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



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

Software Engineering

Lecture 12

Software Quality Assurance

Quality Concepts

QD: characteristics that designers specify for a product

QC: focuses on the degree to which the implementation follows design and the resulting system meets its requirements and performance goal

1. Quality

- Quality as “**a characteristic or attribute of something.**”
- Two kinds of quality may be encountered:
 - *Quality of design* of a product increases, if the product is manufactured according to specifications.
 - *Quality of conformance* is the degree to which the design specifications are followed during manufacturing.
- In software development,
 - Quality of design encompasses requirements, specifications, and the design of the system.
 - Quality of conformance is an issue focused primarily on implementation.

User satisfaction = compliant product + good quality + delivery within budget and schedule

2. Quality Control

- *Quality control* involves the series of inspections, reviews, and tests used throughout the software process.
- Quality control includes a feedback loop to the process.
- A key concept of quality control is that all work products have defined, measurable specifications to which we may compare the output of each process.
- The feedback loop is essential to minimize the defects produced.

3. Quality Assurance

- *Quality assurance* consists of the auditing and reporting functions of management.
- If the data provided through quality assurance identify problems, it is management's responsibility to address the problems and apply the necessary resources to resolve quality issues.

4. Cost of Quality

- The *cost of quality* includes all costs incurred in the pursuit of quality or in performing quality-related activities.
- *Quality costs* may be divided 3 mode of cost:
 - Prevention
 - Appraisal
 - Failure.

- Prevention costs include
 - Quality planning
 - Formal technical reviews
 - Test equipment
 - Training
- Appraisal costs include activities to gain insight into product
 - In-process and Inter-process inspection
 - Equipment calibration and maintenance
 - Testing
- Failure costs
 - Internal Failure Cost
 - rework
 - repair
 - failure mode analysis
 - External Failure Cost
 - complaint resolution
 - product return and replacement
 - help line support
 - warranty work

Software Quality Assurance (SQA)

■ Definition:

- Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

■ Definition serves to emphasize three important points:

- Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
- Specified standards define a set of development criteria, If the criteria are not followed, lack of quality will almost surely result.
- A set of implicit requirements often goes unmentioned. If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect

SQA group activities

- SQA group made up of software engineers, project managers, customers, salespeople, and the individuals members.
- Software quality assurance is composed of two different constituencies.
 - Software engineers who do technical work
 - SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting.
- Software engineers address quality activities by applying technical methods and measures, conducting formal technical reviews, and performing well-planned software testing.
- SQA group is to assist the software team in achieving a high quality end product.

Role of an SQA group

1. Prepares an SQA plan for a project.

- The plan is developed during project planning and is reviewed by all stakeholders.
- The plan identifies
 - Evaluations to be performed
 - Audits and reviews to be performed
 - Standards that are applicable to the project
 - Procedures for error reporting and tracking
 - Documents to be produced by the SQA group
 - Amount of feedback provided to the software project team

2. Participates in the development of the project's software process description.
 - The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.
3. Reviews software engineering activities to verify compliance with the defined software process.
 - The SQA group identifies, documents, and tracks deviations from the process and verifies that corrections have been made.
4. Audits designated software work products to verify compliance with those defined as part of the software process.
 - The SQA group reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made; and periodically reports the results of its work to the project manager.

5. **Ensures that deviations in software work and work products are documented and handled according to a documented procedure.**
 - Deviations may be encountered in the project plan, process description, applicable standards, or technical work products. failure to act in accordance with a wish or command
6. **Records any noncompliance and reports to senior management.**
 - Noncompliance items are tracked until they are resolved.

Software Review

- Software reviews are a "filter" for the software engineering process.
- That is, reviews are applied at various points during software development and serve to uncover errors and defects that can then be removed.
- Types of Review
 - Informal Review
 - Meeting around the coffee machine and discussing technical problems.
 - Formal Review
 - Formal presentation of software design to an audience of customers, management, and technical staff
- A FTR is the most effective filter from a quality assurance standpoint. [Formal Technical Review](#)

Cost Impact of Software Defects

- The primary objective of formal technical reviews is to find errors during the process so that they do not become defects after release of the software.
- Direct benefit is early discovery of error, do not propagate to the next step.
- Design activities introduce between 50 and 65 percent of all errors (and ultimately, all defects) during the software process.
- However, FTR have been shown to be up to 75 percent effective in uncovering design flaws.
- FTP substantially reduces the cost of subsequent steps in the development and support phases.

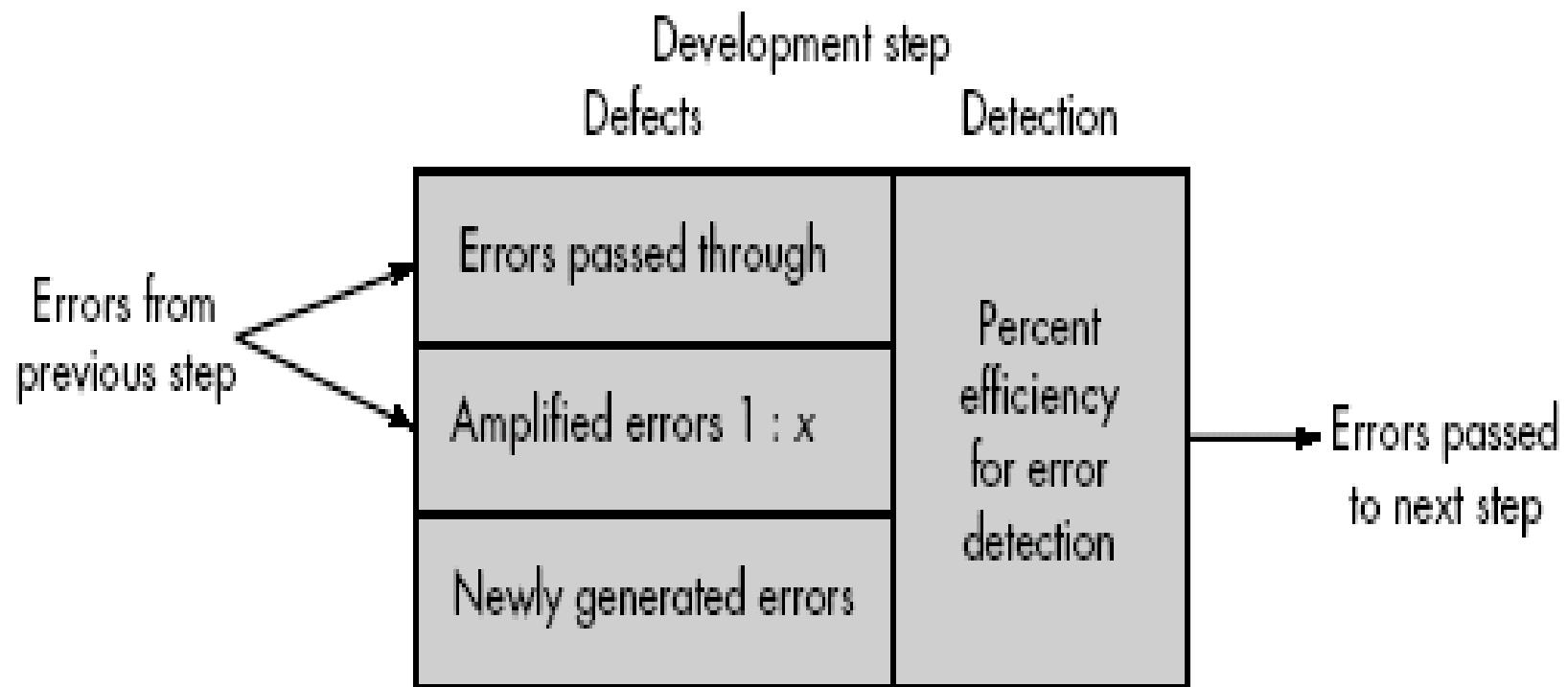
Contd.

- Assume that an error uncovered during design will cost 1.0 monetary unit to correct.
- Relative to this cost, the same error uncovered just before testing commences will cost 6.5 units; during testing, 15 units; and after release, between 60 and 100 units.

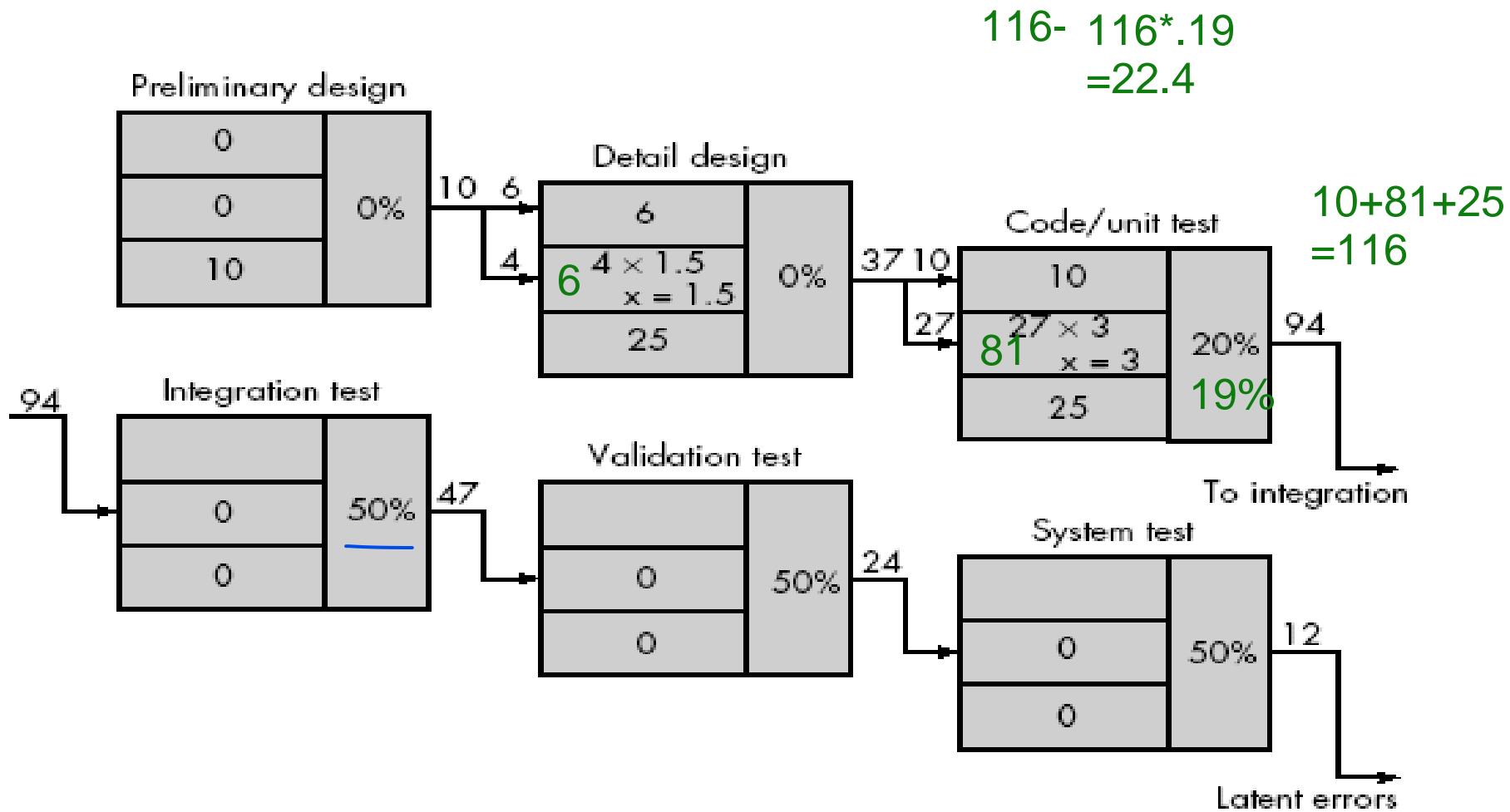
Defect Amplification and Removal

- A defect amplification model can be used to illustrate the generation and detection of errors during the preliminary design, detail design, and coding steps of the software engineering process.
- A box represents a software development step. During the step, errors may be by mistake generated.
- Review may fail to uncover newly generated errors and errors from previous steps, resulting in some number of errors that are passed through.
- In some cases, errors passed through from previous steps are amplified (amplification factor, x) by current work.
- The box subdivisions represent each of these characteristics and the percent of efficiency for detecting errors.

Defect Amplification model

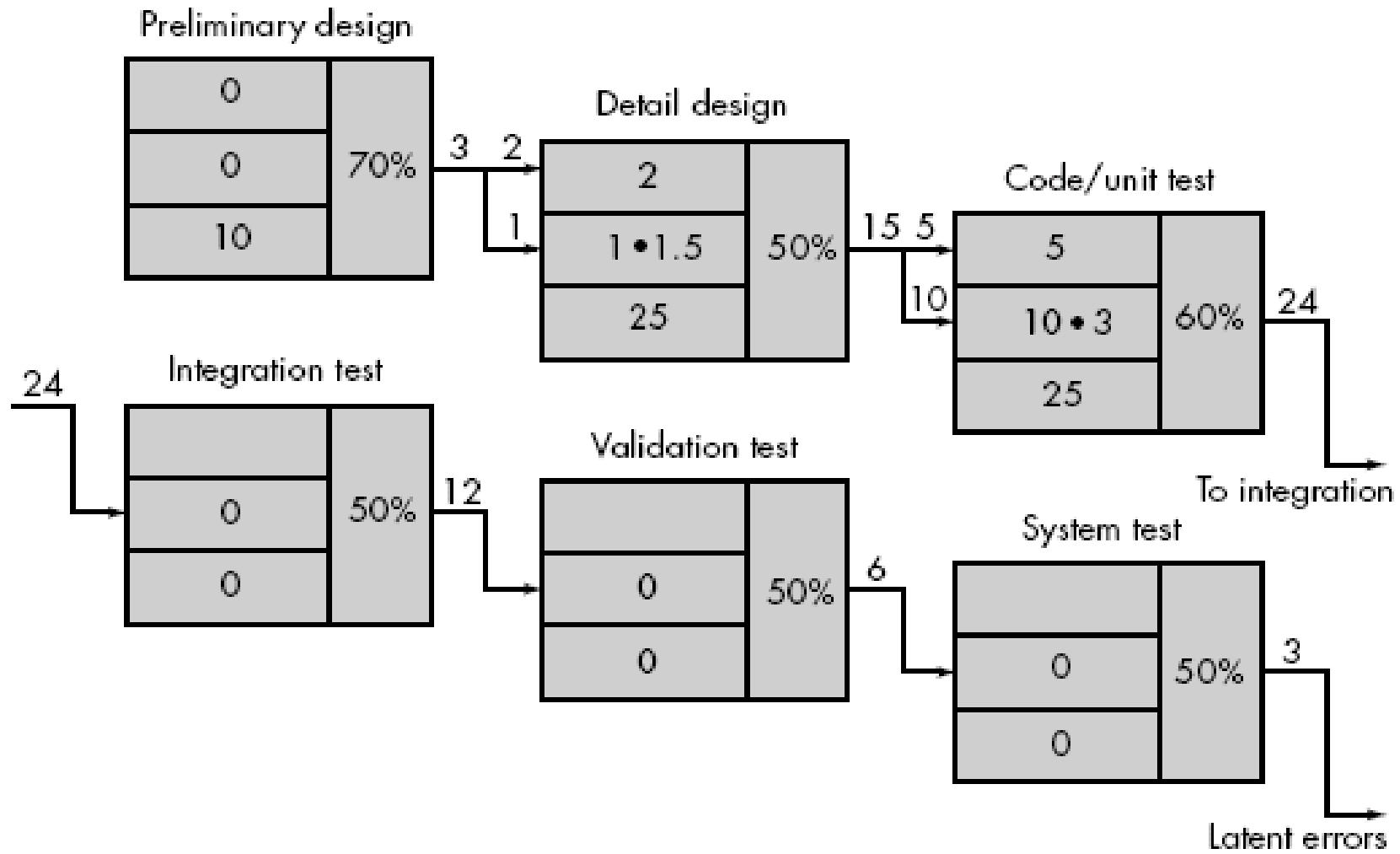


Defect Amplification – No Review



- Because of no review conducted, Ten preliminary design defects are amplified to 94 errors before testing commences.
- Each test step is assumed to correct 50 percent of all incoming errors without introducing any new errors.
- Twelve latent errors are released to the field or to the customers.

Defect Amplification –Review conducted



- Considers the same conditions except that design and code reviews are conducted as part of each development step.
- In this case, ten initial preliminary design errors are amplified to 24 errors before testing commences.
- Only three latent errors exist.
- Now we can estimate overall cost (with and without review for our hypothetical example)
- Using these data, the total cost for development and maintenance when reviews are conducted.
- To conduct reviews, a software engineer must expend time and effort and the development organization must spend money.

FORMAL TECHNICAL REVIEWS

- It is a software quality assurance activity performed by software engineers

Objectives of the FTR are

- To uncover errors in function, logic, or implementation for any representation of the software;
 - To verify that the software under review meets its requirements;
 - To ensure that the software has been represented according to predefined standards;
 - To achieve software that is developed in a uniform manner;
 - To make projects more manageable.
- The FTR is actually a class of reviews that includes walkthroughs, inspections, round-robin reviews and other small group technical assessments of software.

FTR- Review meeting

- Every review meeting should abide by the following constraints:
- Between three and five people (typically) should be involved in the review.
- Advance preparation should occur but should require no more than two hours of work for each person.
- The duration of the review meeting should be less than two hours.
- FTR focuses on a specific (and small) part of the overall software.
- e.g. rather than attempting to review an entire design, are conducted for each component or small group of components.

- The focus of the FTR is on a work product. (e.g. requirement, analysis, design, coding)
- *The producer*—informs the project leader that the work product is complete and that a review is required.
- The project leader contacts a *review leader*, who evaluates the product for readiness, generates copies of product materials, and distributes them to two or three reviewers for advance preparation.
- Each reviewer is expected to spend between one and two hours reviewing the product & making notes.
- Concurrently, the review leader also reviews the product and establishes an agenda for the review meeting, which is typically scheduled for the next day.
- The FTR begins with an introduction of the agenda and a brief introduction by the producer.
- The producer then proceeds to "walk through" the work product, explaining the material, while reviewers raise issues based on their advance preparation.

Contd.

- When valid problems or errors are discovered, the recorder notes each.
- At the end of the review, all attendees of the FTR must decide whether to
 - Accept the product without further modification,
 - Reject the product due to severe errors
 - Accept the product provisionally
- The decision made, all FTR attendees complete a sign-off, indicating their participation in the review and their concurrence with the review team's findings.

Review Reporting and Record Keeping

- During the FTR, a reviewer (the recorder) actively records all issues that have been raised.
- These are summarized at the end of the review meeting and a review issues list is produced.
- A *review summary report* answers three questions:
 1. What was reviewed?
 2. Who reviewed it?
 3. What were the findings and conclusions?
- *Review summary report* becomes part of the project historical record and may be distributed to the project leader and other interested parties.

- The *review issues list* serves two purposes:
 - To identify problem areas within the product
 - To serve as an action item checklist that guides the producer as corrections are made.
- An issues list is normally attached to the summary report.
- It is important to establish a follow-up procedure to ensure that items on the issues list have been properly corrected. Unless this is done, it is possible that issues raised can “fall between the cracks.”
- One approach is to assign the responsibility for follow-up to the review leader.

Review Guidelines

- *Review the product, not the producer.*
 - Don't point out errors harshly. One way to be gentle is to ask a question that enables the producer to discover his or her own error.
- *Set an agenda and maintain it.*
 - An FTR must be kept on track and on schedule.
- *Limit debate and rebuttal:*
 - Rather than spending time debating the question, the issue should be recorded for further discussion off-line
- *Enunciate problem areas, but don't attempt to solve every problem noted.*
 - Review only some small part of component.
- *Take written notes.*
 - make notes on a wall board, so that wording and priorities can be assessed by other reviewers

- *Limit the number of participants and insist upon advance preparation.*
 - Keep the number of people involved to the necessary minimum. However, all review team members must prepare in advance.
- *Develop a checklist for each product that is likely to be reviewed.*
 - helps the review leader to structure the FTR meeting and helps each reviewer to focus on important issues.
- *Allocate resources and schedule time for FTRs*
- *Conduct meaningful training for all reviewers.*
 - To be effective all review participants should receive some formal training
- *Review your early reviews.*

SOFTWARE RELIABILITY

- *Software reliability* is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time".
- What is meant by the term *failure*?
 - In the context of any discussion of software quality and reliability, failure is nonconformance to software requirements.
 - Correction of one failure may in fact result in the introduction of other errors that ultimately result in other failures.
- Software reliability can be measured directed and estimated using historical and developmental data.

Measures of Reliability and Availability

- A simple measure of reliability is *mean-time- between-failure* (MTBF), where

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

The acronyms MTTF and MTTR are mean-time-to-failure and mean-time-to-repair, respectively.

- MTBF is a far more useful measure than defects/KLOC or defects/FP.
- Stated simply, an end-user is concerned with failures, not with the total error count. Because each error contained within a program does not have the same failure rate, the total error count provides little indication of the reliability of a system.
- In addition to a reliability measure, we must develop a measure of availability.

- *Software availability* is the probability that a program is operating according to requirements at a given point in time and is defined as

$$\text{Availability} = [\text{MTTF}/(\text{MTTF} + \text{MTTR})] \\ 100\%$$

- The MTBF reliability measure is equally sensitive to MTTF and MTTR.
- The availability measure is somewhat more sensitive to MTTR, an indirect measure of the maintainability of software.