

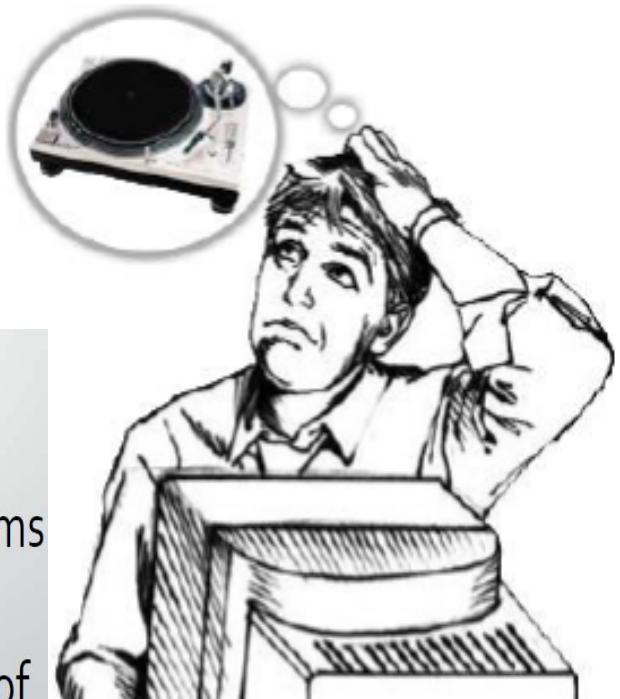
Software Engineering



What is Software Engineering?

Program vs. Software

- o Program is set of instructions written for a specific purpose.
- o Software is: (1) **instructions** (computer programs) that when executed provide desired features, function, and performance; (2) **data structures** that enable the programs to adequately manipulate information and (3) **documentation** that describes the operation and use of the programs.



What is Software?

- Software is **developed** or engineered, it is **not manufactured** in the classical sense.
- Software doesn't **wear out**, but **deterioration**.
- Although the industry is moving toward component-based construction, most software continues to be **custom-built**.

Programs vs Software Products

Characteristics	Program	SW product
Users	self	Others
Number of user	Self/few	Large number
Size	small	Large
Functionality	limited	Large
Interfaces	Ok	Well designed
Environment	One	Several
System	Used by itself	Works with other systems
User background	Similar	Varied
Presence of bugs	Not a major concern	Major concern
Documentation	Minimal	Exhaustive
Testing	Minimal	Exhaustive
Cost/user	High	low
Developers	One /few	Many
Use of standards, etc	Not essential	essential

What is Software Engineering?

Program vs. Software

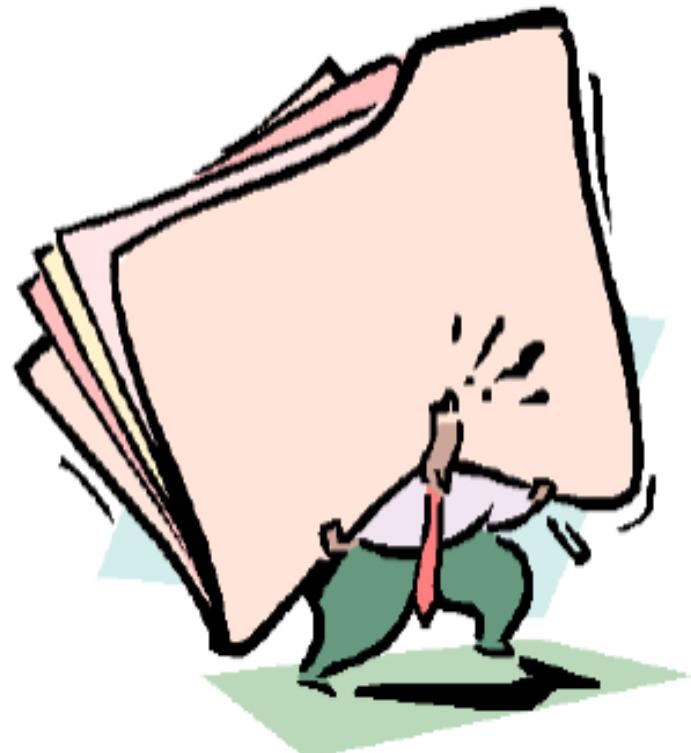
- o Software Requirements and Specification document
- o Software Design Document
- o Test plan document
- o Test suite document
- o Source code



What is Software Engineering?

Program vs. Software

- o Installation manual
- o System administration manual
- o Beginner's guide tutorial
- o System overview
- o Reference guide



7 Software Applications

- **System software:** such as compilers, editors, file management utilities
- **Application software:** stand-alone programs for specific needs.
- **Engineering/scientific software:** Characterized by “number crunching” algorithms, such as automotive stress analysis, molecular biology, orbital dynamics etc
- **Embedded software:** resides within a product or system, such as key pad control of a microwave oven, digital function of dashboard display in a car
- **Product-line software:** focus on a limited marketplace to address mass consumer market. Ex. word processing, graphics, database management
- **WebApps (Web applications):** network centric software. As web 2.0 emerges, more sophisticated computing environments is supported integrated with remote database and business applications.
- **AI software** uses non-numerical algorithm to solve complex problem. Robotics, expert system, pattern recognition game playing

Software Applications – New

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

Application Type



- Stand-alone applications
- Interactive transaction-based applications
- Embedded control systems
- Batch processing systems
- Entertainment systems
- Systems for modeling and simulation
- Data collection systems
- Systems of systems

Software + Engineering

- **Software** is considered to be a collection of executable programming code, associated libraries and documentations.
- Software, when made for a specific requirement is called **software product**.
- **Engineering** on the other hand, is all about developing products, using well-defined, scientific principles and methods.
- **Software engineering as an engineering branch associated with the development of software product using well-defined scientific principles, methods and procedures.**
- ***The outcome of software engineering is an efficient and reliable software product.***

Software Engineering Definition

The seminal definition:

[Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

The IEEE definition:

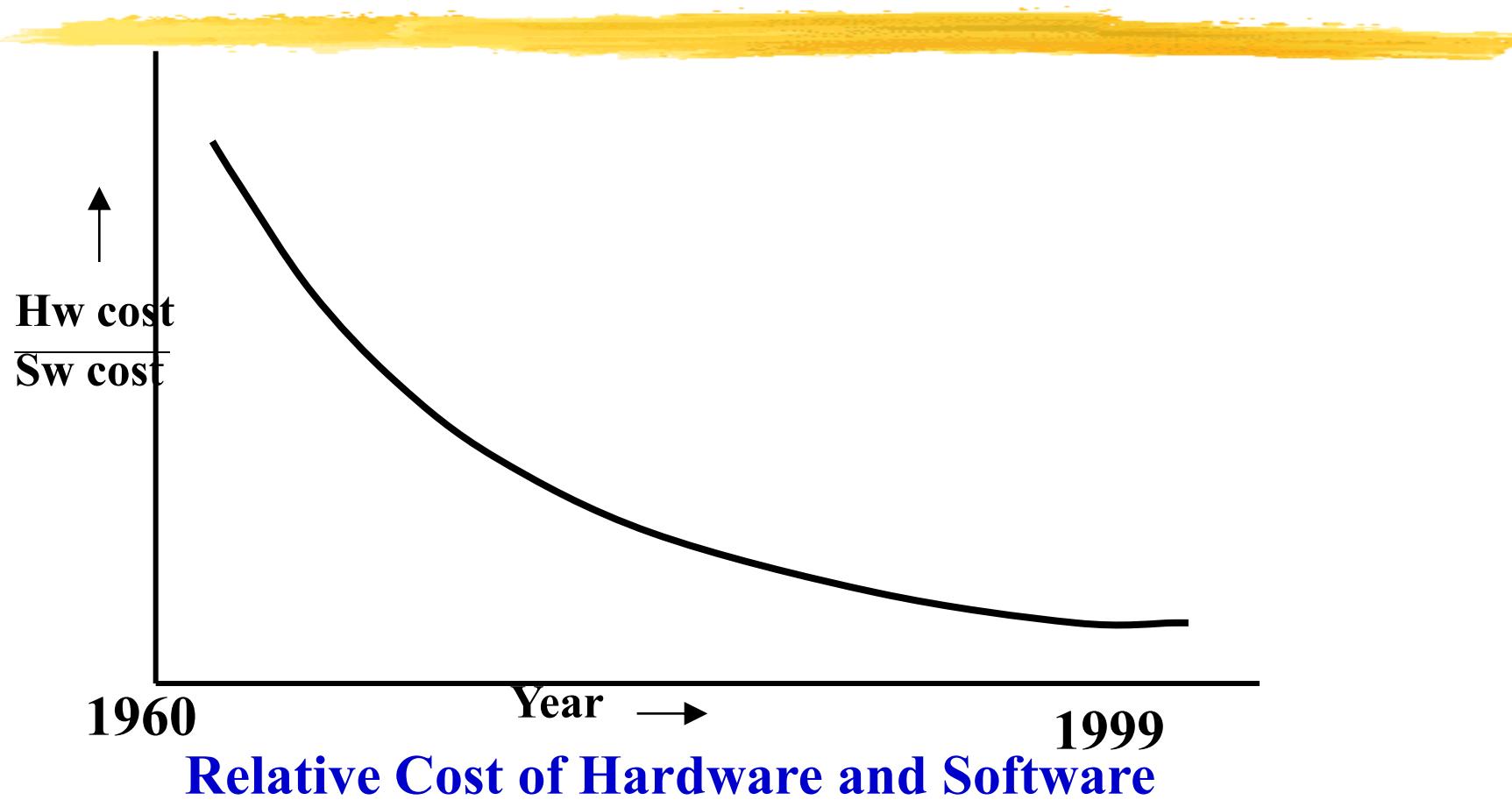
Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

Software Crisis



- Software products:
 - fail to meet user requirements.
 - frequently crash.
 - expensive.
 - difficult to alter, debug, and enhance.
 - often delivered late.
 - use resources non-optimally.

Software Crisis (cont.)



Factors contributing to the software crisis

The “Software Crisis”

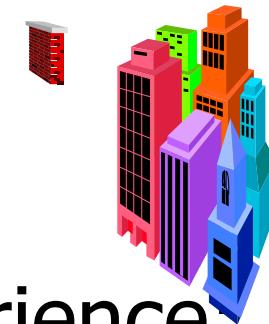
Consecutive revolutions are being achieved in many domains thanks to computer software.



However, software engineering is not seeing the same evolution rate as in the domains it addresses.

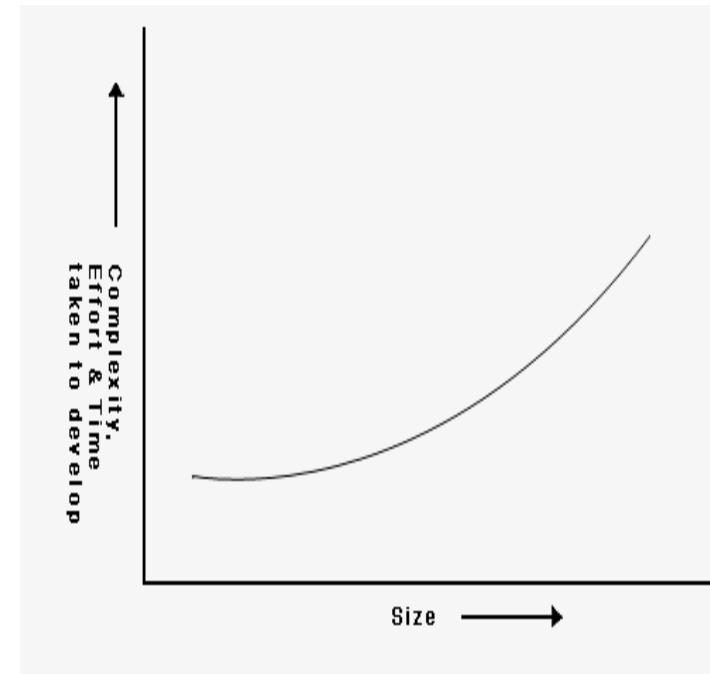
What is Software Engineering?

- Engineering approach to develop software.
 - Building Construction Analogy.
- Systematic collection of past experience:
 - techniques,
 - methodologies,
 - guidelines.



Scope and necessity of software engineering

- Ex- **Difference between building a Wall and a Multistoried Building.**
- In industry it is usually needed to develop large programs to accommodate multiple functions.
- Problem with developing such large commercial programs is that the **complexity and difficulty levels of the programs increase exponentially with their sizes**

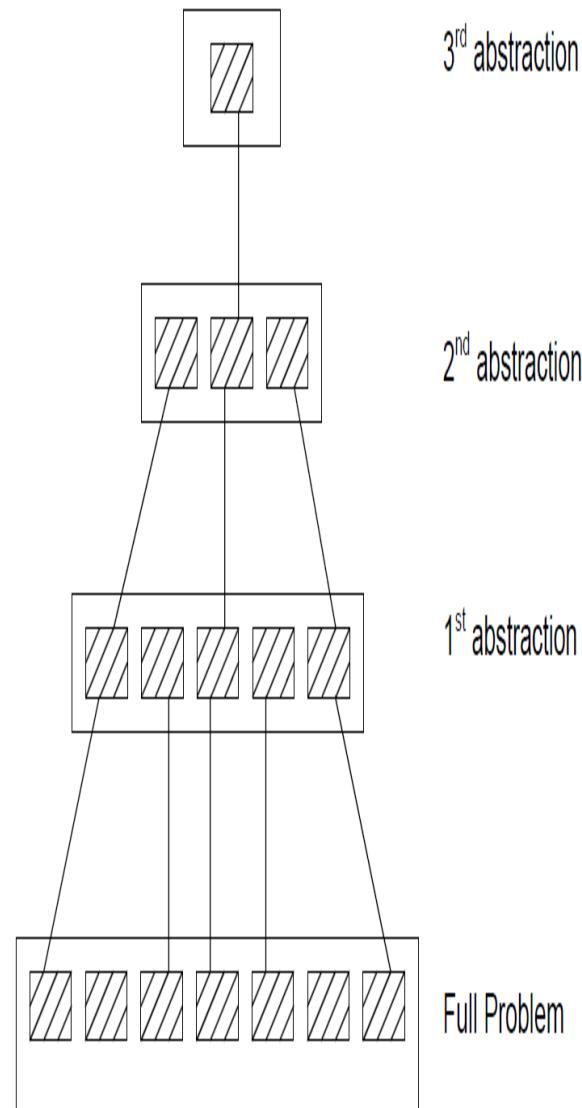


Increase in development time and effort with problem size

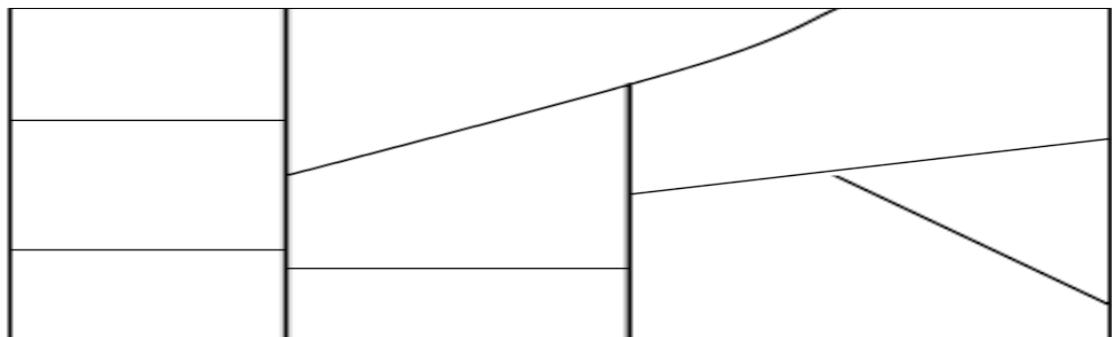
- For example, a program of size 1,000 lines of code has some complexity.
- 
- But a program with 10,000 LOC is not just 10 times more difficult to develop, but may as well turn out to be 100 times more difficult unless software engineering principles are used.
 - In such situations software engineering techniques come to rescue to reduce the programming complexity.
 - Software engineering principles use two important techniques to reduce problem complexity: **abstraction and decomposition**.

- The principle of abstraction implies that a **problem can be simplified by omitting irrelevant details.**

- Consider only those aspects of the problem that are relevant for certain purpose and suppress other aspects that are not relevant for the given purpose.
- Once the simpler problem is solved, then the omitted details can be taken into consideration to solve the next lower level abstraction, and so on.
- Abstraction is a powerful way of reducing the complexity of the problem.**



- The other approach to tackle problem complexity is **decomposition**. A complex problem is divided into several smaller problems and then the smaller problems are solved one by one.
- However, in this technique any random decomposition of a problem into smaller parts will not help.
- The problem has to be decomposed such that each component of the **decomposed problem can be solved independently** and then the solution of the different components can be combined to get the full solution.
- **Challenge:** A good decomposition of a problem as should minimize interactions among various components.



NEED OF SOFTWARE ENGINEERING

Because of higher rate of change in user requirements and environment on which the software is working.

Large software - It is easier to build a wall than to a house or building, likewise, as the size of software become large engineering has to step to give it a scientific process.

Scalability- If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.

Cost- As hardware industry has shown its skills and huge manufacturing has lower down the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.

NEED OF SOFTWARE ENGINEERING

Dynamic Nature- The always growing and adapting nature of software hugely depends upon the environment in which the user works.

If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.

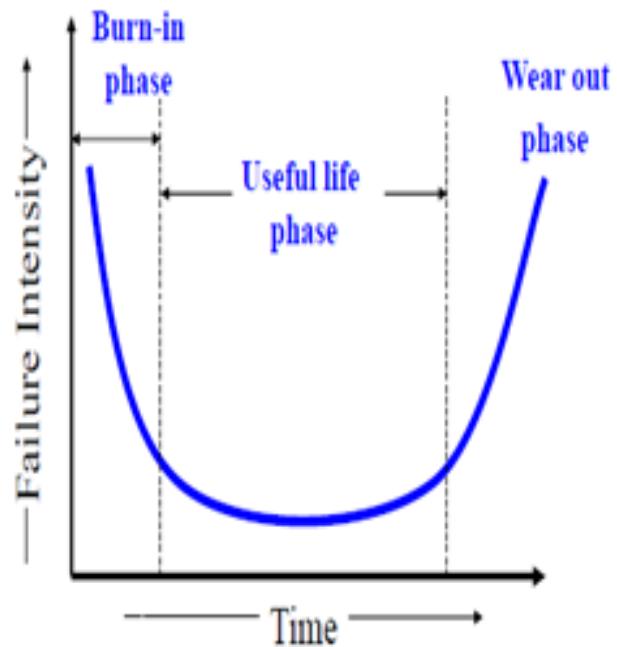
Quality Management- Better process of software development provides better and quality software product.

What is a Software Process

- A set of activities, methods, practices, and transformations that people use to develop and maintain software and the associated products (e.g., project plans, design documents, code, test cases, and user manuals)
- As an organization matures, the software process becomes better defined and more consistently implemented throughout the organization
- Software process maturity is the extent to which a specific process is explicitly defined, managed, measured, controlled, and effective

Hardware vs Software

- Depicts failure rate as a function of time for hardware. The relationship, often called the "**bathtub curve**," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects);
- Defects are corrected and the failure rate drops to a steady-state level (ideally, quite low) for some period of time.
- As time passes, the **failure rate rises again as hardware components suffer from the cumulative affects of dust, vibration, temperature extremes, and many other environmental conditions.** Hardware begins to wear out.

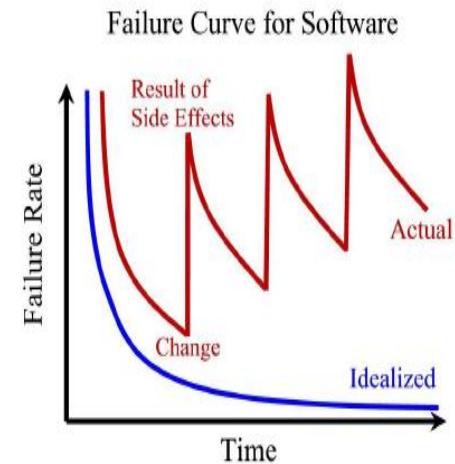


Software doesn't "wear out."

- Software is not influenced to the environmental conditions that cause hardware to wear out.
- The failure rate curve for software should take the form of the "idealized curve". Undiscovered defects will cause high failure rates early in the life of a program.
- However, these are corrected (ideally, without introducing other errors) and the curve flattens as shown.
- As changes are made, it is likely that some new defects will be introduced, causing the failure rate curve to spike.
- Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

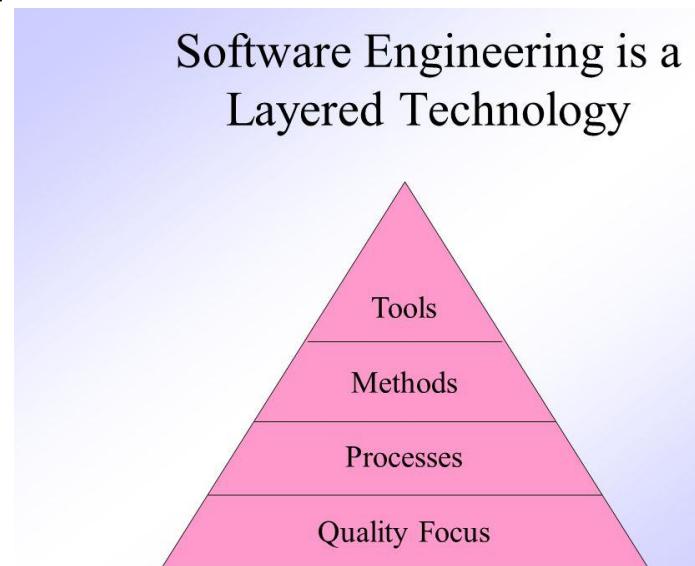
Software Characteristics

- ✓ Flexibility of Software
- ✓ Reusability of Software



Software Engineering as Layered Technology

- Software engineering is totally a layered technology.
 - It means that, to develop a software one will have to go from one layer to another layer.
 - The all layers are related and each layer fulfill the all requirements of the previous layer.



- **Quality layer:** An engineering approach must rest on a quality. A product should meet its specification. The "Bed Rock" that supports software engineering is a quality focus.
- **Process** defines a framework for a set of Key Process Areas (KPAs) that must be established for effective delivery of software engineering technology.
- This establishes the context in which technical methods are applied, work products such as models, documents, data, reports, forms, etc. are produced, milestones are established, quality is ensured, and change is properly managed.

- 
- Software engineering **methods** provide the technical how-to's for building software.
 - Methods will include requirements analysis, design, program construction, testing, and support. This relies on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.
 - Software engineering **tools** provide automated or semi-automated support for the process and the methods.
 - When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called **computer-aided software engineering**, is established.

CHARACTERISTICS OF GOOD SOFTWARE



- A software product can be judged by what it offers and how well it can be used.
- This software must satisfy on the following grounds:
 - **Operational**
 - **Transitional**
 - **Maintenance**

Operational

- This tells us how well software works in operations. It can be measured on:
 - **Budget** : in terms of cost, manpower
 - **Usability** : ease of use
 - **Efficiency** : minimum expenditure of time and effort
 - **Correctness**: with respect to a specification
 - **Functionality** : having a practical use
 - **Dependability** : extent to which a critical system is trusted by its users
 - **Security** : degree of resistance or protection from

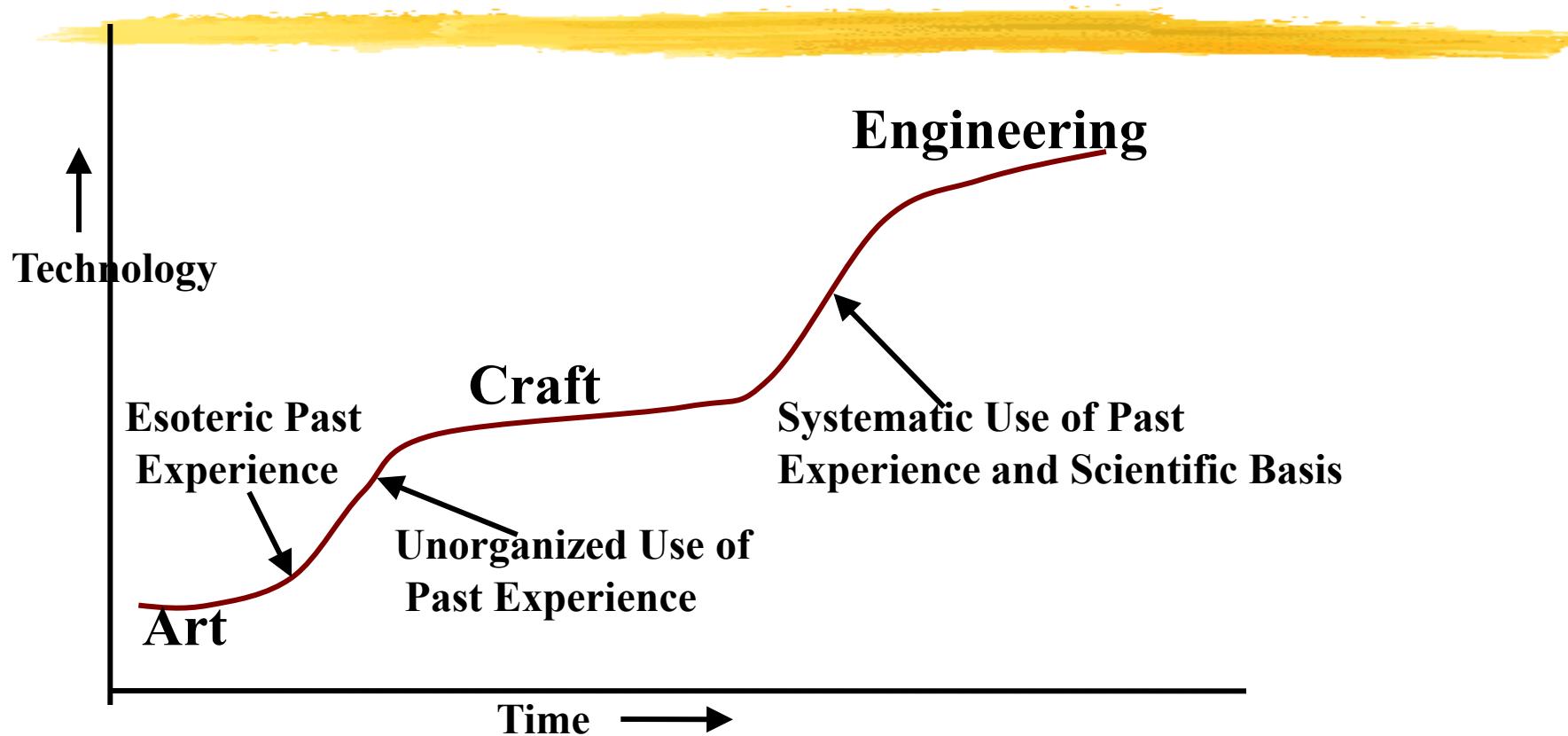
Transitional

- This aspect is important when the software is moved from one platform to another:
 - **Portability** : usability of the same software in different environments
 - **Interoperability** : ability of a system or a product to work with other systems
 - **Reusability** : use of existing assets in some form within the software product development process
 - **Adaptability** : able to change or be changed in order to fit or work better in some situation

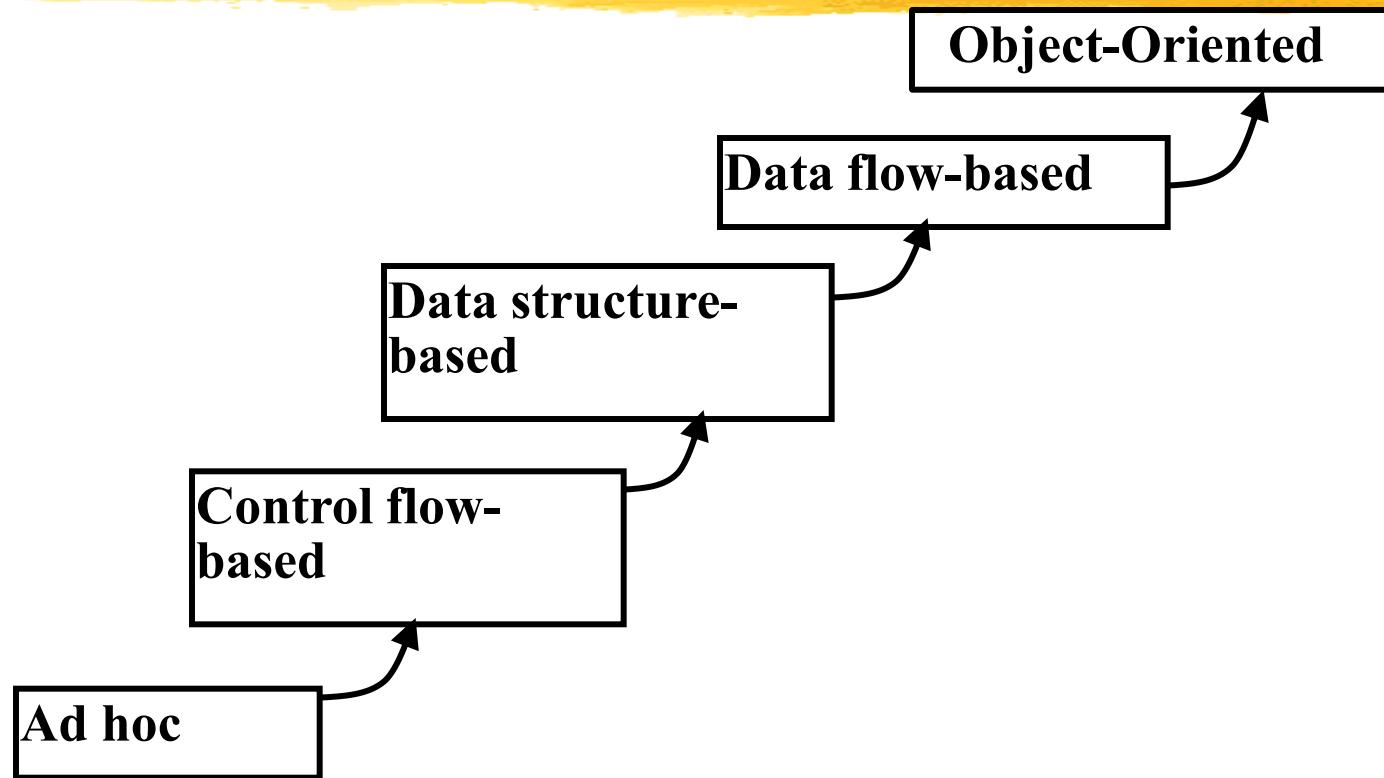
Maintenance

- This aspect briefs about how well a software has the capabilities to maintain itself in the ever-changing environment:
 - **Modularity:** degree to which a system's components may be separated and recombined
 - **Maintainability:** degree to which an application is understood, repaired, or enhanced
 - **Flexibility:** ability for the solution to adapt to possible or future changes in its requirements
 - **Scalability:** ability of a program to scale

Technology Development Pattern



Evolution of Design Techniques



Exploratory style vs. modern style of software development.

Exploratory style	Modern style
Emphasis on error correction	Emphasis on error prevention
Coding was synonymous with software development.	Coding is regarded as only a small part of the overall software development activities.
Believed in developing a working system as quickly as possible and then successively modifying it until it performed satisfactorily.	There are several development activities such as design and testing which typically require much more effort than coding.

Perceived Complexity	Computational Complexity
<p>Problem concerns the difficulty level experienced by the programmer while solving the problems using exploratory development style.</p>	<p>Complexity theory focuses on classifying computational problems according to their resource usage, and relating these classes to each other.</p>
<p>With human cognitive limitations perceived difficulties grow exponentially during implementation</p>	<p>With software engineering principles and high level object oriented programming style and by taking computational complexity into account the complexity can be reduced.</p>

Software Life Cycle

- Software life cycle (or software process):
 - series of identifiable stages that a software product undergoes during its life time:
 - * Feasibility study
 - * requirements analysis and specification,
 - * design,
 - * coding,
 - * testing
 - * maintenance.

Why Model Life Cycle ?

- A written description:
 - forms a common understanding of activities among the software developers.
 - helps in identifying inconsistencies, redundancies, and omissions in the development process.
 - Helps in tailoring a process model for specific projects.

Life Cycle Model (CONT.)

- A life cycle model:
 - defines entry and exit criteria for every phase.
 - A phase is considered to be complete:
 - * only when all its exit criteria are satisfied.
- The phase exit criteria for the software requirements specification phase:
 - Software Requirements Specification (SRS) document is complete, reviewed, and approved by the customer.
- A phase can start:
 - only if its phase-entry criteria have been satisfied.

Life Cycle Model (CONT.)

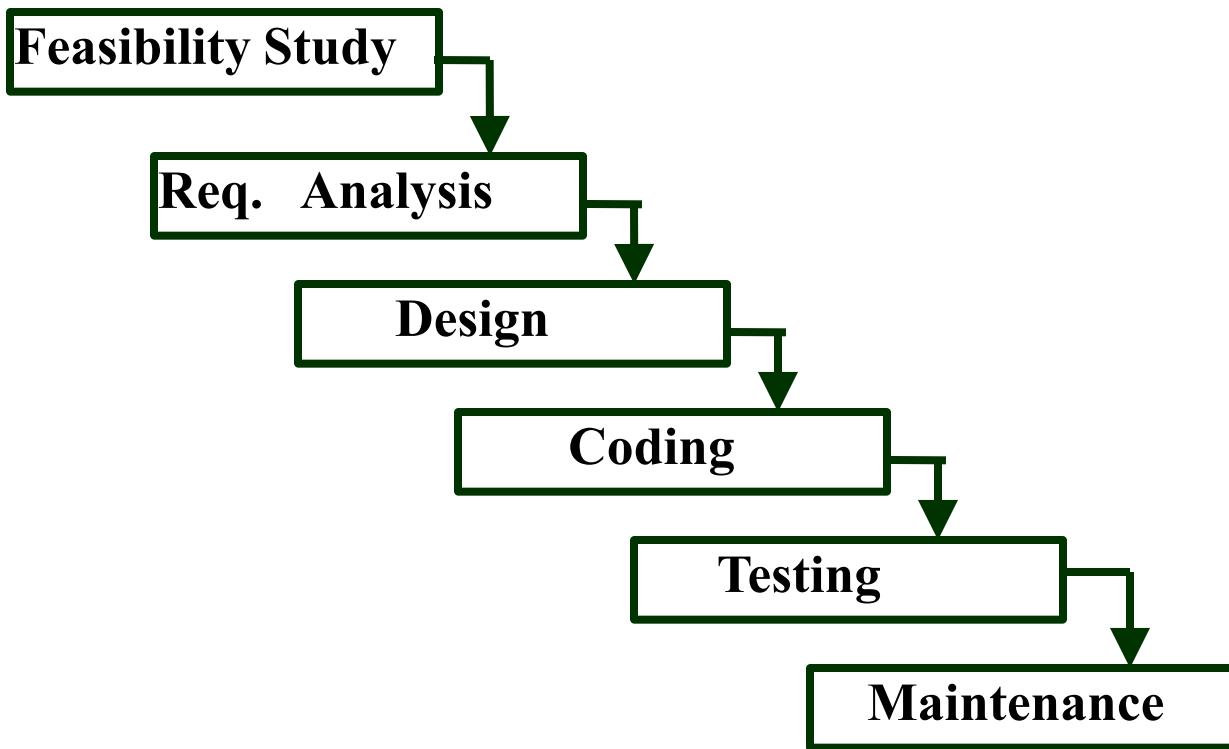


- Many life cycle models have been proposed.
- We will confine our attention to a few important and commonly used models.
 - classical waterfall model
 - iterative waterfall,
 - evolutionary,
 - prototyping, and
 - spiral model

Classical Waterfall Model

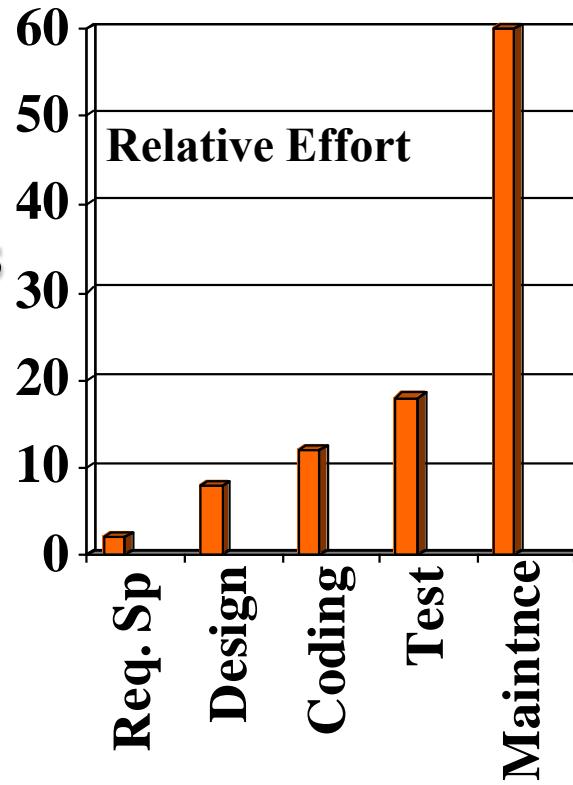
- The Waterfall Model was the first Process Model to be introduced.
- It is also referred to as a **linear-sequential life cycle model**.
- It is very simple to understand and use.
- In a waterfall model, each phase must be completed before the next phase can begin and there is no overlapping in the phases.
- All these phases are cascaded to each other in which progress is seen as flowing steadily downwards (like a waterfall) through the phases.

Classical Waterfall Model



Relative Effort for Phases

- Phases between feasibility study and testing
 - known as **development phases**.
- Among all life cycle phases
 - **maintenance phase consumes maximum effort.**
- Among development phases,
 - **testing phase consumes the maximum effort.**



Classical Waterfall Model

(CONT.)

- **Most organizations usually define:**
 - standards on the outputs (deliverables) produced at the end of every phase
 - entry and exit criteria for every phase.
- **They also prescribe specific methodologies for:**
 - specification,
 - design,
 - testing,
 - project management, etc.

Feasibility Study

- **Main aim of feasibility study: determine whether developing the product**
 - **financially worthwhile**
 - **technically feasible.**
- **First roughly understand what the customer wants by:**
 - study different input data to the system and output data to be produced by the system
 - After an overall understanding of the problem they investigate the different solutions that are possible.
 - Examine each of the solutions in terms of resources required, cost of development and development time for each solution.

Activities during Feasibility Study



- **Perform a cost/benefit analysis:**
 - Based on this analysis pick the best solution and determine whether the solution is feasible financially and technically.
 - Check whether the customer budget would meet the cost of the product and whether they have sufficient technical expertise in the area of development.

Requirements Analysis and Specification

- **Aim of this phase:**
 - understand the exact requirements of the customer,
 - **document them properly.**
- **Consists of two distinct activities:**
 - requirements gathering and analysis
 - requirements specification.

Requirements Gathering

- **Gathering relevant data:**
 - usually collected from the end-users through interviews and discussions.
 - For example, for a business accounting software:
 - * interview all the accountants of the organization to find out their requirements.

Requirements Analysis (CONT.)

- Ambiguities and contradictions:
 - must be identified
 - resolved by discussions with the customers.
- Next, requirements are organized:
 - into a Software Requirements Specification (SRS) document.
- Engineers doing requirements analysis and specification are designated as analysts.

Design

- Design phase transforms requirements specification:
 - into a form suitable for implementation in some programming language.
 - during design phase, software architecture is derived from the SRS document.
- Two design approaches:
 - traditional approach,
 - object oriented approach.

Traditional approach – Design Phase

- Consists of two activities:
 - **Structured analysis**
 - **Structured design**
- **Structured Analysis Activity**
 - Identify all the **functions** to be performed.
 - Identify **data flow** among the **functions**.
 - Decompose each function recursively into **sub-functions**.
 - * Identify data flow among the subfunctions as well.
- **Carried out using Data flow diagrams (DFDs).**

Structured Design

- After structured analysis, carry out **structured design**:
 - **architectural design** (or high-level design)
 - **detailed design** (or low-level design).
- **High-level design:**
 - decompose the system into ***modules***,
 - represent invocation relationships among the modules.
- **Detailed design:**
 - different modules designed in greater detail:
 - * data structures and algorithms for each module are designed.

Object Oriented Design

- First identify various objects (real world entities) occurring in the problem:
 - identify the relationships among the objects.
 - For example, the objects in a pay-roll software may be:
 - * employees,
 - * managers,
 - * pay-roll register,
 - * Departments, etc.
- Object structure
 - further refined to obtain the detailed design.
- OOD has several advantages:
 - lower development effort,
 - lower development time,
 - better maintainability.

Implementation

- **Purpose of implementation phase (aka **coding and unit testing phase**):**
 - **translate software design into source code.**
- During the implementation phase:
 - each module of the design is coded,
 - each module is **unit** tested
 - * tested independently as a stand alone unit, and debugged,
 - each module is documented.
- **The purpose of unit testing:**
 - test if individual modules work correctly.
- **The end product of implementation phase:**
 - **a set of program modules that have been tested individually.**

Integration and System Testing

- Different modules are integrated in a planned manner:
 - **modules are almost never integrated in one shot.**
 - Normally integration is carried out through a number of steps.
- During each integration step,
 - the partially integrated system is tested.



System Testing

- After all the modules have been successfully integrated and tested:
 - **system testing is carried out.**
- Goal of system testing:
 - ensure that the developed system functions according to its requirements as specified in the SRS document.

- System testing usually consists of three different kinds of testing activities:
 - α – testing: It is the system testing performed by the development team.
 - β – testing: It is the system testing performed by a friendly set of customers.
 - acceptance testing: It is the system testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered product.

Maintenance

- **Maintenance of any software product:**
 - requires much more effort than the effort to develop the product itself.
 - **development effort to maintenance effort is typically 40:60.**
- Corrective maintenance:
 - Correct errors which were not discovered during the product development phases.
- Perfective maintenance:
 - Improve implementation of the system
 - enhance functionalities of the system.
- Adaptive maintenance:
 - Port software to a new environment,
 - * e.g. to a new computer or to a new operating system.

Waterfall Model - Application

- Every software developed is different and requires a suitable SDLC approach to be followed based on the internal and external factors.
- Some situations where the use of Waterfall model is most appropriate are –
 - Requirements are very well documented, clear and fixed.
 - Product definition is stable.
 - Technology is understood and is not dynamic.
 - There are no ambiguous requirements.
 - Ample resources with required expertise are available to support the product.
 - The project is short.

Advantages of the Waterfall Model

- The advantages of waterfall development are that it allows for departmentalization and control.
 - Each phase has specific deliverables and a review process.
 - Phases are processed and completed one at a time.
 - Works well for smaller projects where requirements are very well understood.
 - Clearly defined stages.
 - Well understood milestones.
 - Process and results are well documented.

Waterfall Model - Disadvantages

- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Not suitable for the projects where requirements are at a moderate to high risk of changing. So, risk and uncertainty is high with this process model.
- Cannot accommodate changing requirements.
- Integration is done as a "big-bang." at the very end, which doesn't allow identifying any technological or business bottleneck or challenges early.

Software Life Cycle



Iterative Waterfall Model

(CONT.)

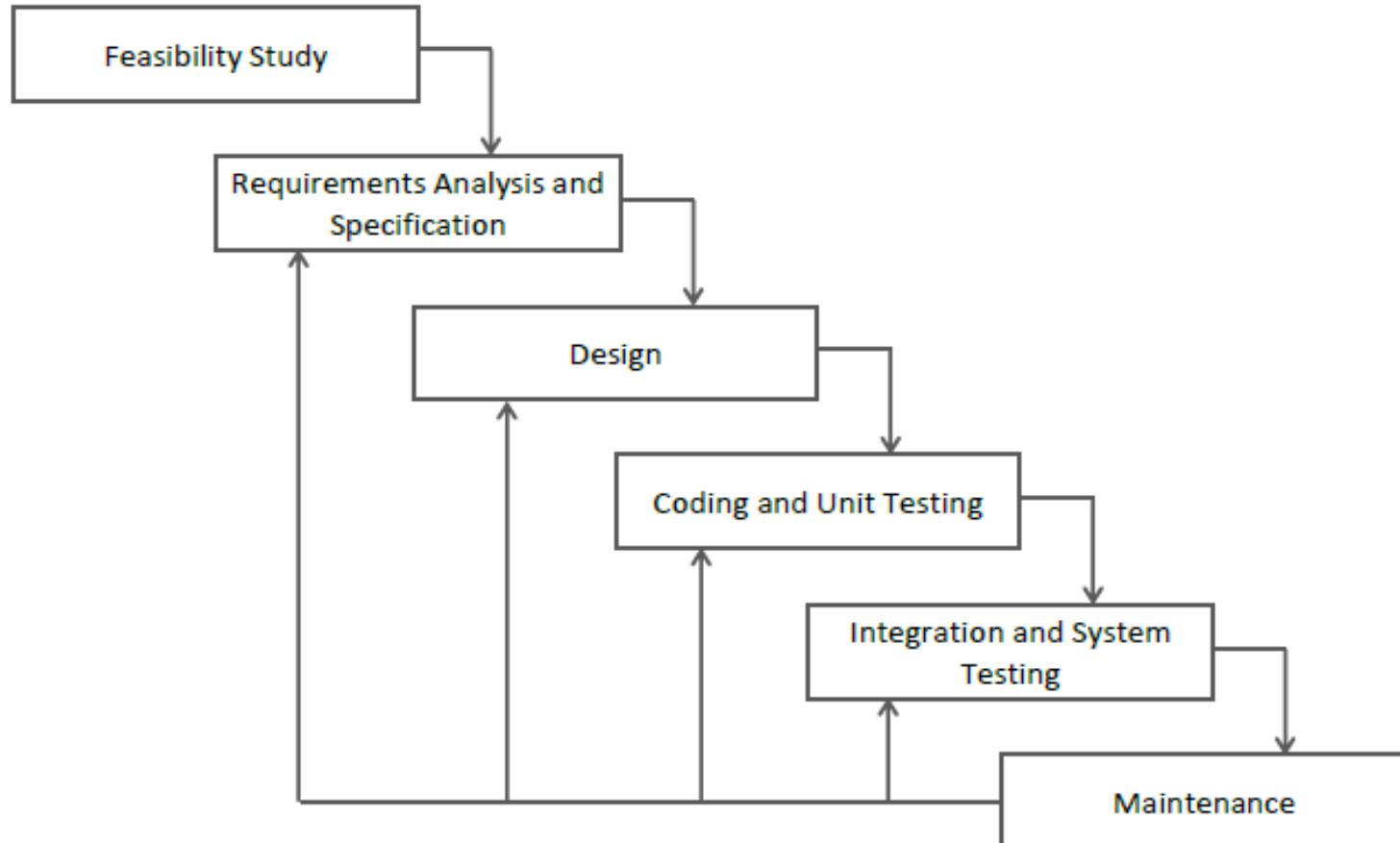


Figure 2: Iterative Waterfall Model

Iterative Waterfall Model

- **Errors should be detected**
 - in the same phase in which they are introduced.
- **For example:**
 - **if a design problem is detected in the design phase itself,**
 - the problem can be taken care of much more easily
 - than say if it is identified at the end of the integration and system testing phase.

Phase containment of errors

- **Reason:** rework must be carried out not only to the design but also to code and test phases.
- The principle of detecting errors as close to its point of introduction as possible:
 - is known as **phase containment of errors.**
- Iterative waterfall model is by far the most widely used model.
 - Almost every other model is derived from the waterfall model.

Prototyping Model

- **Before starting actual development,**
 - **a working prototype of the system should first be built.**
- **A prototype is a toy implementation of a system:**
 - **limited functional capabilities,**
 - **low reliability,**
 - **inefficient performance.**

Reasons for developing a prototype

- Illustrate to the customer:
 - input data formats, messages, reports, or interactive dialogs.
- Examine technical issues associated with product development:
 - Often major design decisions depend on issues like:
 - * response time of a hardware controller,
 - * efficiency of a sorting algorithm, etc.

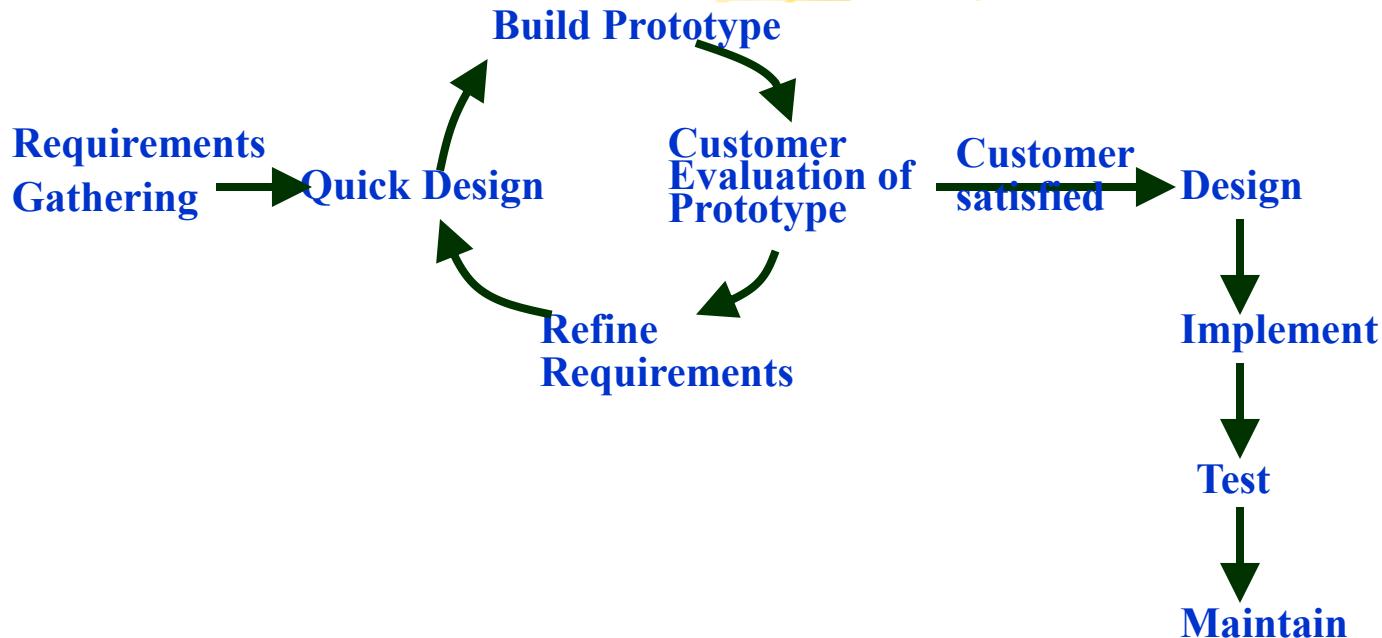
Prototyping Model (CONT.)

- **The third reason for developing a prototype is:**
 - it is impossible to ``get it right" the first time,
 - we must plan to throw away the first product
 - * if we want to develop a good product.
- **Carry out a quick design.**
- **Prototype model is built using several short-cuts:**
 - Short-cuts might involve using inefficient, inaccurate, or dummy functions.
 - * A function may use a table look-up rather than performing the actual computations.

Prototyping Model (CONT.)

- **The developed prototype is submitted to the customer for his evaluation:**
 - Based on the user feedback, requirements are refined.
 - This cycle continues until the user approves the prototype.
- **The actual system is developed using the classical waterfall approach.**

Prototyping Model (CONT.)



Prototyping Model (cont.)

- Requirements analysis and specification phase becomes redundant:
 - final working prototype (with all user feedbacks incorporated) serves as an animated requirements specification.
- Design and code for the prototype is usually thrown away:
 - However, the experience gathered from developing the prototype helps a great deal while developing the actual product.

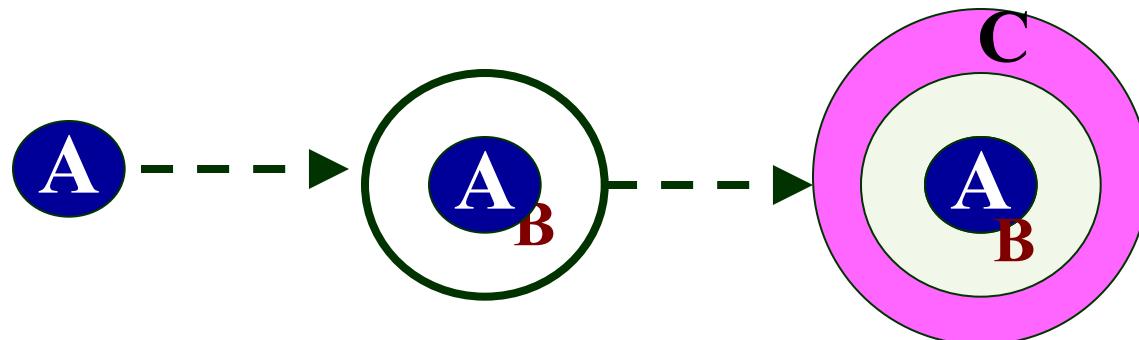
Prototyping Model (cont.)

- Even though construction of a working prototype model involves additional cost --- overall development cost might be lower for:
 - systems with unclear user requirements,
 - systems with unresolved technical issues.
- Many user requirements get properly defined and technical issues get resolved:
 - these would have appeared later as change requests and resulted in incurring massive redesign costs.

Evolutionary Model

- **Evolutionary model (aka successive versions or incremental model):**
 - The system is broken down into several modules which can be incrementally implemented and delivered.
- **First develop the core modules of the system.**
- **The initial product skeleton is refined into increasing levels of capability:**
 - by adding new functionalities in successive versions.
- **Successive version of the product:**
 - functioning systems capable of performing some useful work.
 - A new release may include new functionality:
 - * also existing functionality in the current release might have been enhanced.

Evolutionary Model (CONT.)



Advantages of Evolutionary Model

- **Users get a chance to experiment with a partially developed system:**
 - much before the full working version is released,
- **Helps finding exact user requirements:**
 - much before fully working system is developed.
- **Core modules get tested thoroughly:**
 - reduces chances of errors in final product.

Disadvantages of Evolutionary Model

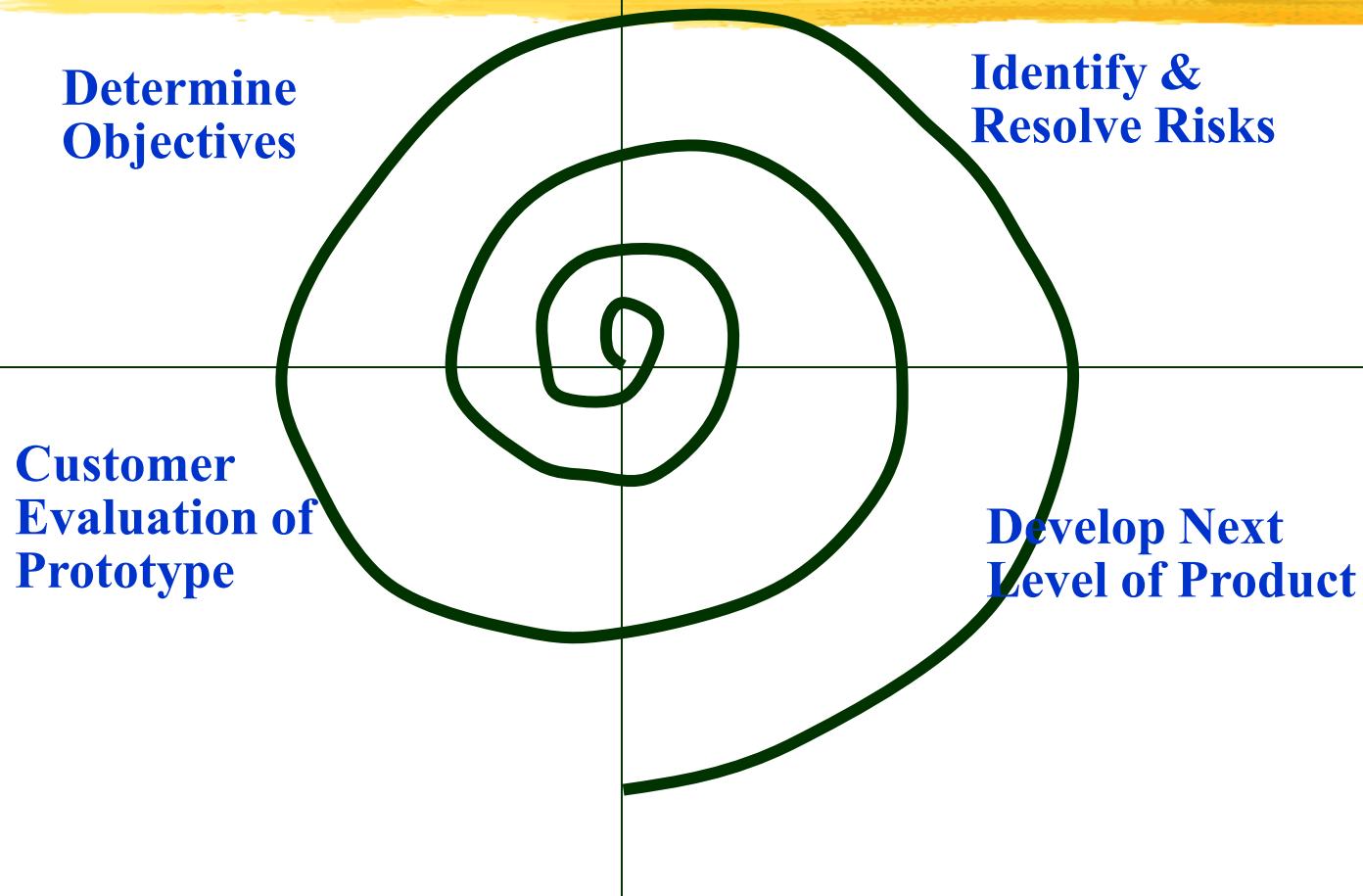
- Often, difficult to subdivide problems into functional units:
 - which can be incrementally implemented and delivered.
 - evolutionary model is useful for very large problems,
 - * where it is easier to find modules for incremental implementation.

Evolutionary Model with Iteration

- Many organizations use a combination of iterative and incremental development:
 - a new release may include new functionality
 - existing functionality from the current release may also have been modified.
- Several advantages:
 - Training can start on an earlier release
 - * customer feedback taken into account
 - Markets can be created:
 - * for functionality that has never been offered.
 - Frequent releases allow developers to fix unanticipated problems quickly.

Spiral Model

(CONT.)



Spiral Model

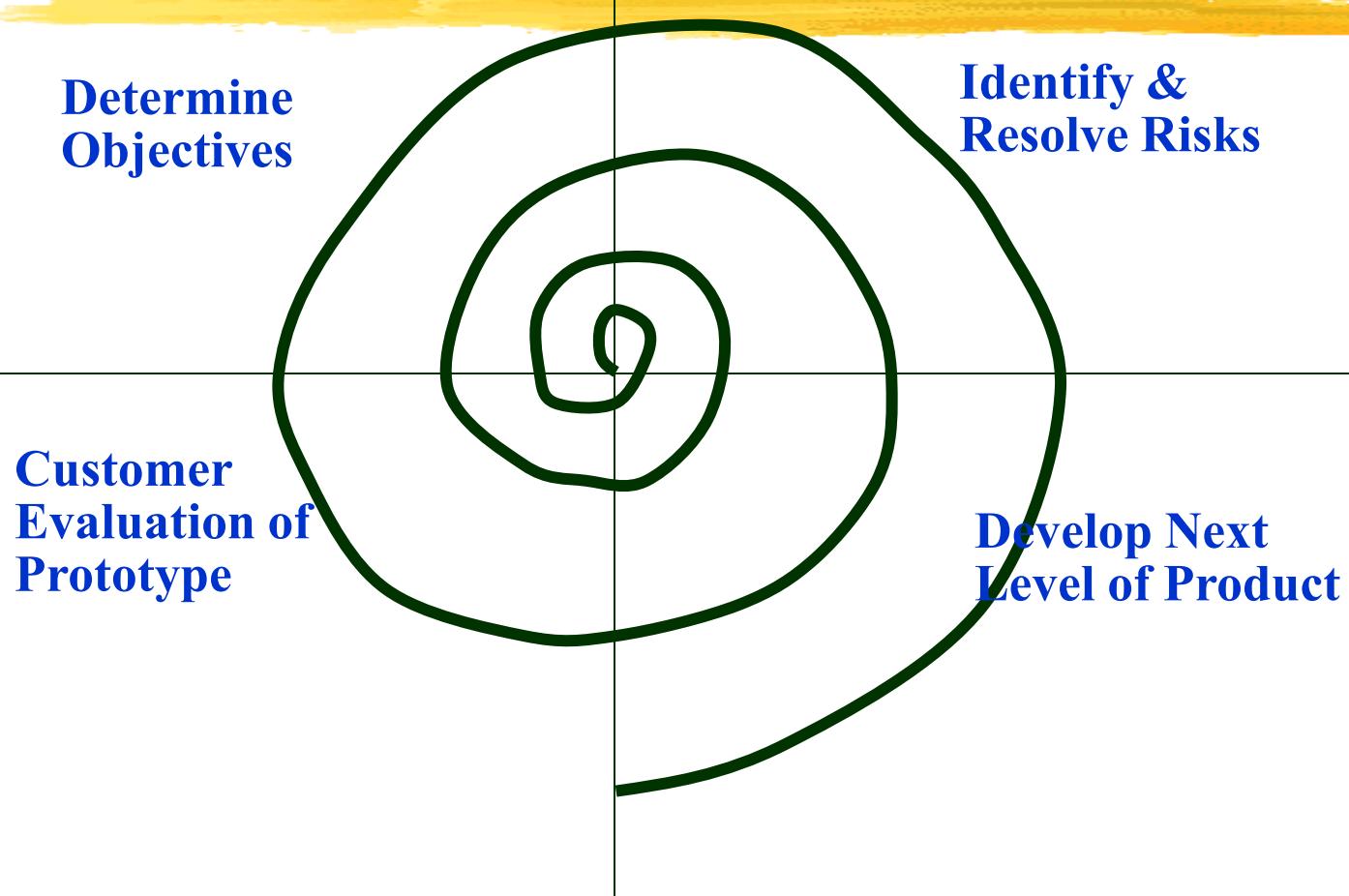
- **Each loop of the spiral represents a phase of the software process:**
 - the innermost loop might be concerned with system feasibility,
 - the next loop with system requirements definition,
 - the next one with system design, and so on.
- **The exact number of loops in the spiral is not fixed.**

Spiral Model (CONT.)

- The team must decide:
 - how to structure the project into phases.
- Start work using some generic model:
 - add extra phases
 - * for specific projects or when problems are identified during a project.
- Each loop in the spiral is split into four sectors (quadrants).

Spiral Model

(CONT.)



Objective Setting (First Quadrant)

- Identify objectives of the phase,
- Examine the **risks** associated with these objectives.
 - Risk:
 - * any adverse circumstance that might hamper successful completion of a software project.
- Find alternate solutions possible.

Risk Assessment and Reduction (Second Quadrant)

- **For each identified project risk,**
 - **a detailed analysis is carried out.**
- **Steps are taken to reduce the risk.**
- **For example, if there is a risk that the requirements are inappropriate:**
 - **a prototype system may be developed.**

Spiral Model (CONT.)

- **Development and Validation (Third quadrant):**
 - develop and validate the next level of the product.
- **Review and Planning (Fourth quadrant):**
 - review the results achieved so far with the customer and plan the next iteration around the spiral.
- **With each iteration around the spiral:**
 - progressively more complete version of the software gets built.

Spiral Model as a meta model

- **Subsumes all discussed models:**
 - a single loop spiral represents waterfall model.
 - uses an evolutionary approach --
 - * iterations through the spiral are evolutionary levels.
 - enables understanding and reacting to risks during each iteration along the spiral.
 - uses:
 - * prototyping as a risk reduction mechanism
 - * retains the step-wise approach of the waterfall model.

Advantages of Spiral Model

- High amount of risk analysis hence, avoidance of Risk is enhanced.
- Good for large and mission-critical projects.
- Strong approval and documentation control.
- Additional Functionality can be added at a later date.
- Software is produced early in the software life cycle.

Disadvantages

- Can be a costly model to use.
- Risk analysis requires highly specific expertise.
- Project's success is highly dependent on the risk analysis phase.
- Doesn't work well for smaller projects.

Circumstances to use spiral model

- The spiral model is called a meta model since it encompasses all other life cycle models.
- Risk handling is inherently built into this model.
- The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks.
- However, this model is much more complex than the other models – **this is probably a factor deterring its use in ordinary projects.**

Comparison of Different Life Cycle Models

- **Iterative waterfall model**
 - most widely used model.
 - But, suitable only for well-understood problems.
- **Prototype model is suitable for projects not well understood:**
 - user requirements
 - technical aspects

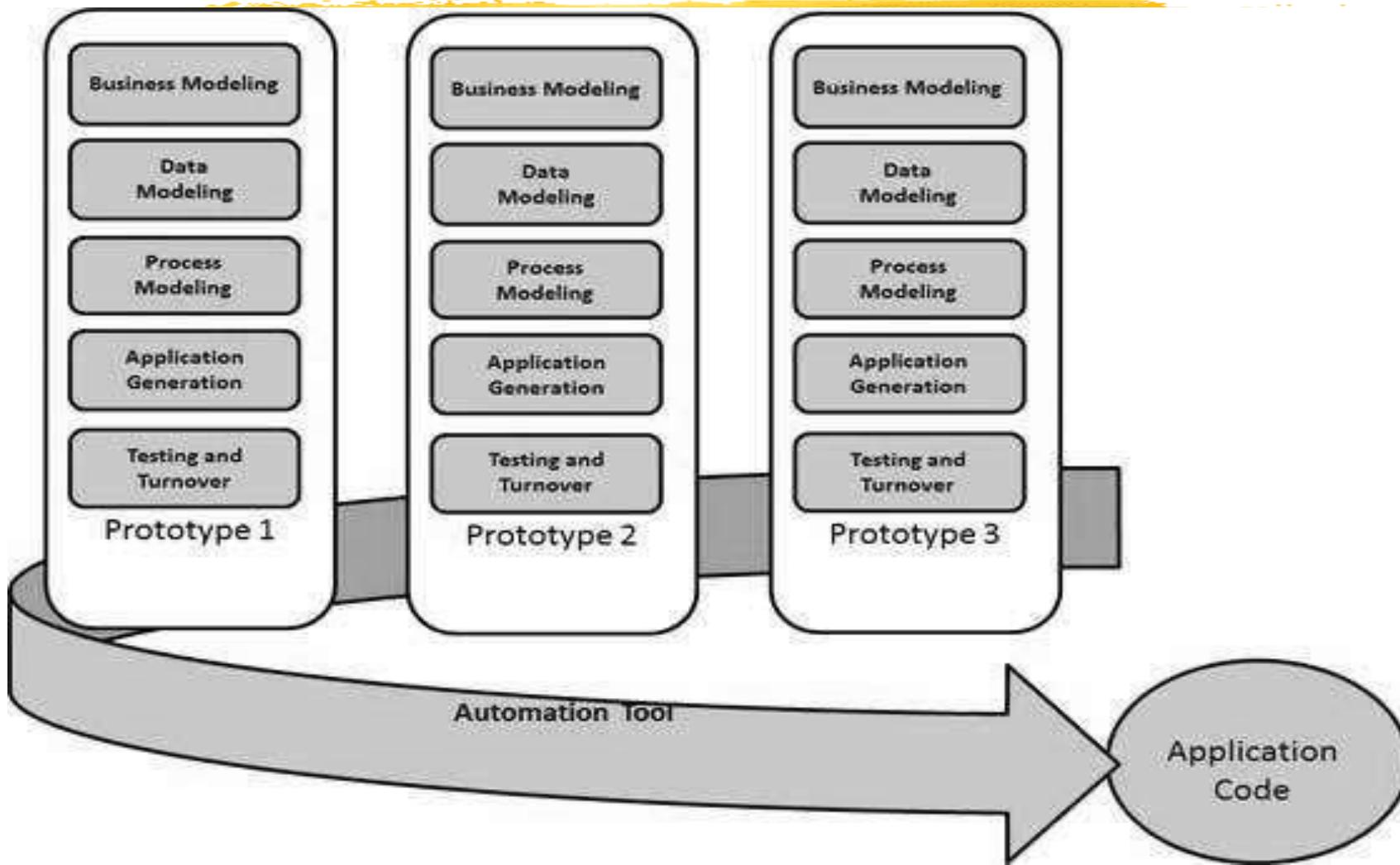
Comparison of Different Life Cycle Models (CONT.)

- **Evolutionary model is suitable for large problems:**
 - can be decomposed into a set of modules that can be incrementally implemented,
 - incremental delivery of the system is acceptable to the customer.
- **The spiral model:**
 - suitable for development of technically challenging software products that are subject to several kinds of risks.

RAD

- RAD model is Rapid Application Development model.
- Based on prototyping and iterative development with no specific planning involved.
- In RAD model the components or functions are developed in parallel as if they were mini projects.
- Focuses on gathering customer requirements through workshops or focus groups, early testing of the prototypes by the customer using iterative concept, reuse of the existing prototypes (components), continuous integration and rapid delivery.

The most important aspect for this model to be successful is to make sure that the prototypes developed are reusable.

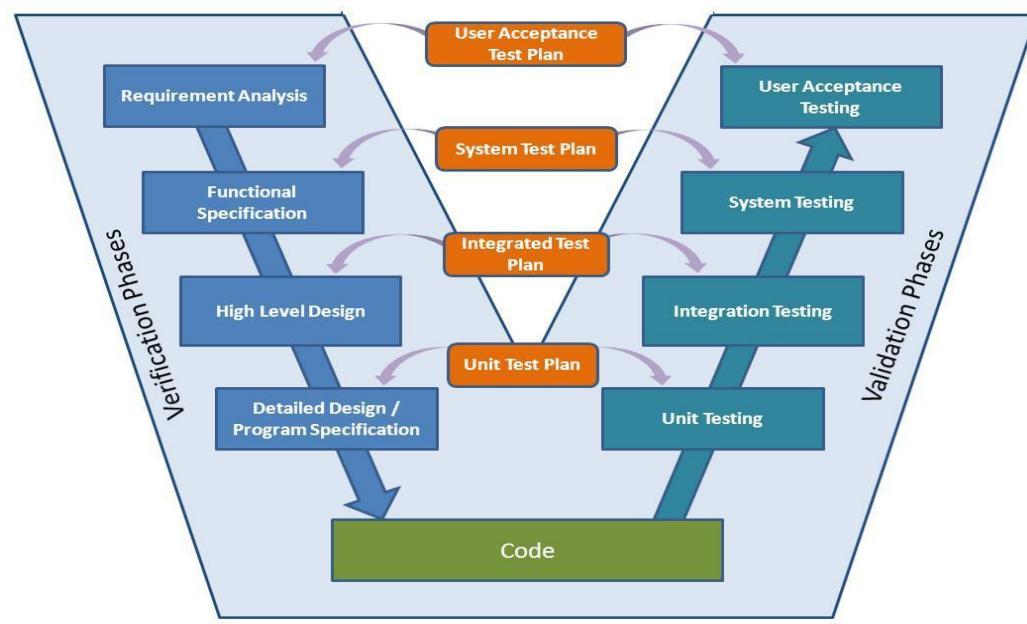


- The phases in the rapid application development (RAD) model are:
 - **Business Modeling:** Model should be designed based on the information available from different business activities.
 - **Data Modeling:** All the required and necessary data based on business analysis are identified in data modeling phase.
 - **Process Modeling:** In this phase all the data modification process is defined. Process descriptions for adding, deleting, retrieving or modifying a data object are given.
 - **Application Modeling:** The actual system is built and coding is done by using automation tools to convert process and data models into actual prototypes.
 - **Testing and turnover:** All the testing activates are performed to test the developed application.

- **Advantages of RAD Model:**
 - Fast application development and delivery.
 - Least testing activity required.
 - Visualization of progress.
 - Review by the client from the very beginning of development so very less chance to miss the requirements.
- **Disadvantages of RAD Model:**
 - High skilled resources required.
 - On each development phase client's feedback required.
 - Automated code generation is very costly.
 - Difficult to manage.

V-Shaped Model

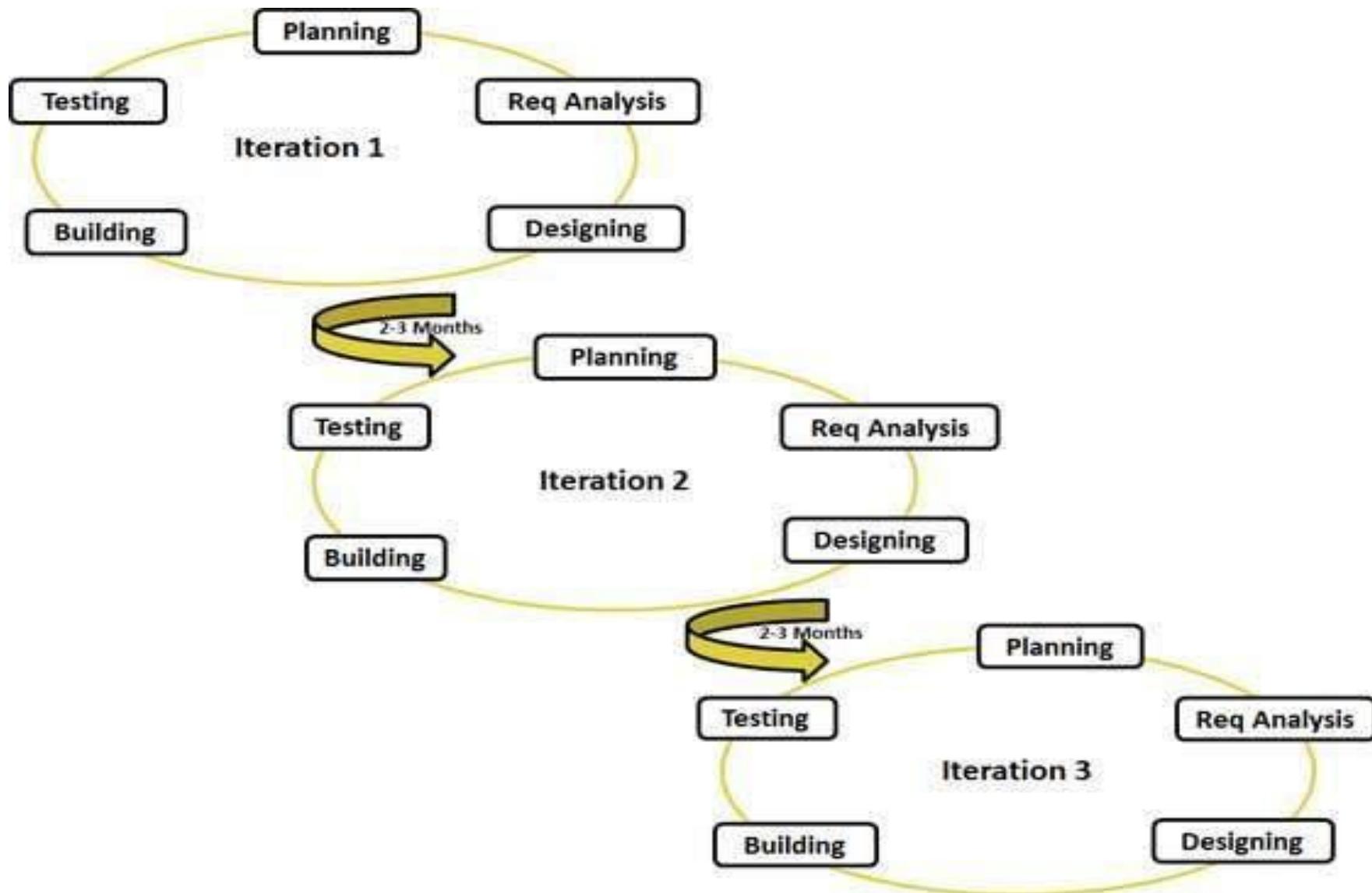
- The V-model is an SDLC model where execution of processes happens in a sequential manner in a V-shape. It is also known as **Verification and Validation model**.
- The V-Model is an extension of the waterfall model and is based on the association of a testing phase for each corresponding development stage.
- This means that for every single phase in the development cycle, there is a directly associated testing phase. This is a highly-disciplined model and the next phase starts only after completion of the previous phase.



- The usage
 - Software requirements clearly defined and known
 - Software development technologies and tools is well-known
- **Advantages**
 - Simple and easy to use.
 - Each phase has specific deliverables.
 - Higher chance of success over the waterfall model due to the development of test plans early on during the life cycle.
 - Verification and validation of the product in early stages of product development

Agile Model

- Combination of iterative and incremental process models with focus on process adaptability and customer satisfaction by rapid delivery of working software product.
- Agile Methods break the product into small incremental builds.
- These builds are provided in iterations. Each iteration typically lasts from about one to three weeks.
- Every iteration involves cross functional teams working simultaneously on various areas like planning, requirements analysis, design, coding, unit testing, and acceptance testing.
- At the end of the iteration a working product is displayed to the customer and important stakeholders.



Advantages

- Is a very realistic approach to software development
- Promotes teamwork and cross training.
- Functionality can be developed rapidly and demonstrated.
- Good model for environments that change steadily.
- Minimal rules, documentation easily employed.
- Enables concurrent development and delivery within an overall planned context.

Disadvantages

- Not suitable for handling complex dependencies.
- Strict delivery management dictates the scope, functionality to be delivered, and adjustments to meet the deadlines.
- Depends heavily on customer interaction, so if customer is not clear, team can be driven in the wrong direction.
- There is very high individual dependency, since there is minimum documentation generated.
- Transfer of technology to new team members may be quite challenging due to lack of documentation.

Software Development Methodology: Agile

Merits and limitations of Agile approach (MyPMExpert, 2009)

Merits	Limitations
<ul style="list-style-type: none">• Agile methodology is adaptive and hence it can adapt well with changing requirements.• Dedicated time and effort is not required because customer requirement may change• Customer can give continuous inputs and have face to face interactions with the team.• Agile method does not give room for guesswork, documentation is exhaustive but crisp• Customer satisfaction is ensured	<ul style="list-style-type: none">• Difficult to scale when projects are large where documentation is needed• Agile teams need to have experience and skills• In some deliverables it is problematic to evaluate as required at beginning of SDLC• Design is not much emphasized• The project can be easily distracted when the customer is not clear on outcomes• Testing and test construction is difficult and needs specialized skills

Comparison of Different Life Cycle Models

- **Iterative waterfall model**
 - most widely used model.
 - But, suitable only for well-understood problems.
- **Prototype model is suitable for projects not well understood:**
 - user requirements
 - technical aspects

Comparison of Different Life Cycle Models (CONT.)

- **Evolutionary model is suitable for large problems:**
 - can be decomposed into a set of modules that can be incrementally implemented,
 - incremental delivery of the system is acceptable to the customer.
- **The spiral model:**
 - suitable for development of technically challenging software products that are subject to several kinds of risks.

- Identify the definite stages through which a software product undergoes during its lifetime.
- 
- Explain the problems that might be faced by an organization if it does not follow any software life cycle model.
 - Identify two basic roles of a system analyst.
 - Differentiate between structured analysis and structured design.
 - Identify at least three activities undertaken in an object-oriented software design approach.

Software Life Cycle

- A software life cycle model (also called process model) is a **descriptive and diagrammatic representation** of the software life cycle.
- Represents all the **activities** required to make a software product transit through its life cycle phases.
- Captures the **order in which these activities** are to be undertaken.
- In other words maps the different activities performed on a software product from its inception to retirement.

- Different life cycle models may map the basic development activities to phases in different ways.
- Thus, no matter which life cycle model is followed, the basic activities are included in all life cycle models though the activities may be carried out in different orders in different life cycle models.
- During any life cycle phase, more than one activity may also be carried out.
- For example, the **design phase** might consist of the **structured analysis** activity followed by the **structured design activity**.

Need for a software life cycle model

- Development team must identify a suitable life cycle model for the particular project and then adhere to it.
- Without using of a particular life cycle model the development of a software product would not be in a systematic and disciplined manner.
- When a software product is being developed by a team there must be a clear understanding among team members about when and what to do.
- Otherwise it would lead to chaos and project failure.

- Suppose a software development problem is divided into several parts and assigned to the team members.
- From then on, suppose the team members are allowed the freedom to develop the parts assigned to them in whatever way they like.
- It is possible that one member might start writing the code for his part,
- Another might decide to prepare the test documents first,
- And some other engineer might begin with the design phase of the parts assigned to him.
- **This would be one of the perfect recipes for project failure.**

- A software life cycle model defines **entry and exit criteria for every phase.**
- 

- A phase can start only if its **phase-entry criteria** have been satisfied.
- So without software life cycle model the **entry and exit criteria for a phase cannot be recognized.**
- Without software life cycle models it becomes difficult for software project managers to monitor the progress of the project.

Requirements Analysis and Specification

Organization of this Lecture



- ⌘ Introduction
- ⌘ Requirements analysis
- ⌘ Requirements specification
- ⌘ SRS document
- ⌘ Decision table
- ⌘ Decision tree
- ⌘ Summary

Requirements Analysis and Specification

- ⌘ Goals of requirements analysis and specification phase:
 - ▢ fully understand the user requirements
 - ▢ remove inconsistencies, anomalies, etc. from requirements
 - ▢ document requirements properly in an SRS document
- ⌘ Consists of two distinct activities:
 - ▢ Requirements Gathering and Analysis
 - ▢ Specification

Requirements Analysis and Specification

- ⌘ The person who undertakes requirements analysis and specification:
 - ↳ known as **systems analyst**:
 - ↳ collects data pertaining to the product
 - ↳ analyzes collected data:
 - ☒ to understand what exactly needs to be done.
 - ↳ writes the **Software Requirements Specification (SRS)** document.
- ⌘ **Final output of this phase:**
 - ↳ **Software Requirements Specification (SRS) Document.**
- ⌘ The SRS document is reviewed by the customer.
 - ↳ **reviewed SRS document forms the basis of all future development activities.**

Requirements Analysis

- ⌘ Requirements analysis consists of two main activities:
 - ▢ Requirements gathering
 - ▢ Analysis of the gathered requirements
- ⌘ Analyst gathers requirements through:
 - ▢ observation of existing systems,
 - ▢ studying existing procedures,
 - ▢ discussion with the customer and end-users,
 - ▢ analysis of what needs to be done, etc.

Inconsistent requirement

⌘ Some part of the requirement:
 ☒ contradicts with some other part.

⌘ Example:

- ☒ One customer says turn off heater and open water shower when temperature > 100 C
- ☒ Another customer says turn off heater and turn ON cooler when temperature > 100 C

Incomplete requirement

⌘ Some requirements have been omitted:

◻ due to oversight.

⌘ Example:

◻ The analyst has not recorded:
when temperature falls below 90 C
 ☒ heater should be turned ON
 ☒ water shower turned OFF.

Analysis of the Gathered Requirements (CONT.)



⌘ Requirements analysis involves:

- └ obtaining a clear, in-depth understanding of the product to be developed,
- └ remove all ambiguities and inconsistencies from the initial customer perception of the problem.

Software Requirements Specification

⌘ Main aim of requirements specification:

- ↗ systematically organize the requirements arrived during requirements analysis
- ↗ document requirements properly.

⌘ The SRS document is useful in various contexts:

- ↗ statement of user needs
- ↗ contract document
- ↗ reference document
- ↗ definition for implementation

Software Requirements Specification: A Contract Document

- ⌘ Requirements document is a reference document.
- ⌘ SRS document is a contract between the development team and the customer.
 - ↗ Once the SRS document is approved by the customer,
 - ↙ any subsequent controversies are settled by referring the SRS document.
- ⌘ Once customer agrees to the SRS document:
 - ↗ development team starts to develop the product according to the requirements recorded in the SRS document.
- ⌘ The final product will be acceptable to the customer:
 - ↗ as long as it satisfies all the requirements recorded in the SRS document.

SRS Document (cont.)

- ⌘ The SRS document is known as black-box specification:
 - ↗ the system is considered as a black box whose internal details are not known.
 - ↗ only its visible external (i.e. input/output) behavior is documented.



- ⌘ SRS document concentrates on:
 - ↗ what needs to be done
 - ↗ carefully avoids the solution ("how to do") aspects.
- ⌘ The SRS document serves as a contract
 - ↗ between development team and the customer.
 - ↗ Should be carefully written

Properties of a good SRS document

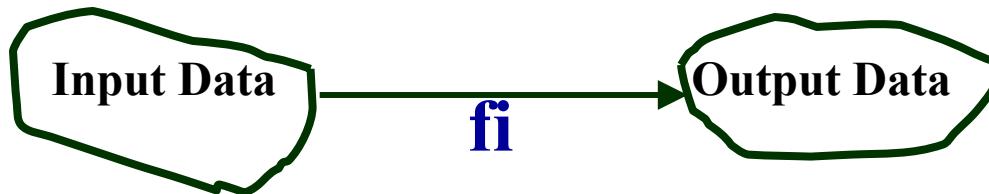
- # It should be concise
 - ↗ and at the same time should not be ambiguous.
- # It should specify what the system must do
 - ↗ and not say how to do it.
- # Easy to change.,
 - ↗ i.e. it should be well-structured.
- # It should be consistent.
- # It should be complete.
- # It should be traceable
 - ↗ you should be able to trace which part of the specification corresponds to which part of the design and code, etc and vice versa.
- # It should be verifiable
 - ↗ e.g. “system should be user friendly” is not verifiable

SRS Document (CONT.)

- ⌘ SRS document, normally contains three important parts:
 - ⌘ functional requirements,
 - ⌘ nonfunctional requirements,
 - ⌘ constraints on the system.

SRS Document (CONT.)

- ⌘ It is desirable to consider every system:
 - ⌘ performing a set of functions $\{f_i\}$.
 - ⌘ Each function f_i considered as:
 - ⌘ transforming a set of input data to corresponding output data.



Example: Functional Requirement

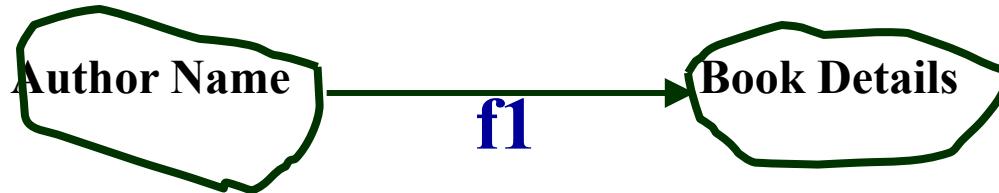
⌘F1: Search Book

↗ Input:

- ☒ an author's name:

↗ Output:

- ☒ details of the author's books and the locations of these books in the library.



Functional Requirements

- ⌘ Functional requirements describe:
 - ℳ A set of high-level requirements
 - ℳ Each high-level requirement:
 - ℳ takes in some data from the user
 - ℳ outputs some data to the user
 - ℳ Each high-level requirement:
 - ℳ might consist of a set of identifiable functions
- ⌘ For each high-level requirement:
 - ℳ every function is described in terms of
 - ℳ input data set
 - ℳ output data set
 - ℳ processing required to obtain the output data set from the input data set



Nonfunctional Requirements

- ⌘ Characteristics of the system which can not be expressed as functions:
 - ☒ maintainability,
 - ☒ portability,
 - ☒ usability, etc.

Nonfunctional Requirements



- ⌘ Nonfunctional requirements include:
 - ⌘ reliability issues,
 - ⌘ performance issues,
 - ⌘ human-computer interface issues,
 - ⌘ Interface with other external systems,
 - ⌘ security, maintainability, etc.

Constraints

⌘ Constraints describe things that the system should or should not do.

↗ For example,

 ☒ standards compliance

 ☒ how fast the system can produce results

- so that it does not overload another system to which it supplies data, etc.

Examples of constraints



- ⌘ Hardware to be used,
- ⌘ Operating system
 - ↗ or DBMS to be used
- ⌘ Capabilities of I/O devices
- ⌘ Standards compliance
- ⌘ Data representations
 - ↗ by the interfaced system

Organization of the SRS Document

⌘ Introduction.

⌘ Functional Requirements

⌘ Nonfunctional Requirements

- ↗ External interface requirements

- ↗ Performance requirements

⌘ Constraints

Example Functional Requirements

⌘ List all functional requirements
 └ with proper numbering.

- ⌘ Req. 1:
 - └ Once the user selects the “search” option,
 - └ he is asked to enter the key words.
 - └ The system should output details of all books
 - └ whose title or author name matches any of the key words entered.
 - └ Details include: Title, Author Name, Publisher name, Year of Publication, ISBN Number, Catalog Number, Location in the Library.

Example Functional Requirements

⌘ Req. 2:

- ◻ When the “renew” option is selected,
 - ☒ the user is asked to enter his membership number and password.
- ◻ After password validation,
 - ☒ the list of the books borrowed by him are displayed.
- ◻ The user can renew any of the books:
 - ☒ by clicking in the corresponding renew box.

Req. 1:

⌘ R.1.1:

↗**Input:** “search” option,

↗**Output:** user prompted to enter the key words.

⌘ R1.2:

↗**Input:** key words

↗**Output:** Details of all books whose title or author name matches any of the key words.

☒**Details include:** Title, Author Name, Publisher name, Year of Publication, ISBN Number, Catalog Number, Location in the Library.

↗**Processing:** Search the book list for the keywords

Req. 2:

- ⌘ R2.1:
 - └─ Input: “renew” option selected,
 - └─ Output: user prompted to enter his membership number and password.
- ⌘ R2.2:
 - └─ Input: membership number and password
 - └─ Output:
 - ✗ list of the books borrowed by user are displayed.
User prompted to enter books to be renewed or
 - ✗ user informed about bad password
 - └─ Processing: Password validation, search books issued to the user from borrower list and display.

Req. 2:

⌘ R2.3:

- ☒ **Input:** user choice for renewal of the books issued to him through mouse clicks in the corresponding renew box.
- ☒ **Output:** Confirmation of the books renewed
- ☒ **Processing:** Renew the books selected by the in the borrower list.

Examples of Bad SRS Documents

⌘ Unstructured Specifications:

- Narrative essay --- one of the worst types of specification document:
 - ☒ Difficult to change,
 - ☒ difficult to be precise,
 - ☒ difficult to be unambiguous,
 - ☒ scope for contradictions, etc.

⌘ Noise:

- Presence of text containing information irrelevant to the problem.

⌘ Silence:

- aspects important to proper solution of the problem are omitted.

Examples of Bad SRS Documents

⌘ Overspecification:

- ☒ Addressing “how to” aspects
- ☒ For example, “Library member names should be stored in a sorted descending order”
- ☒ Overspecification restricts the solution space for the designer.

⌘ Contradictions:

- ☒ Contradictions might arise
 - ☒ if the same thing described at several places in different ways.

⌘ Ambiguity:

- ☒ Literary expressions
- ☒ Unquantifiable aspects, e.g. “good user interface”

⌘ Forward References:

- ☒ References to aspects of problem
 - ☒ defined only later on in the text.

⌘ Wishful Thinking:

- ☒ Descriptions of aspects
 - ☒ for which realistic solutions will be hard to find.

Using Diagrams

Graphical representation of the analysis can present the information better using:

⌘ Decision Tables

⌘ Decision Trees

Representation of complex processing logic:



⌘ Decision trees

⌘ Decision tables

Decision Trees and Decision Tables



- ⌘ Often our problem solutions require decisions to be made according to two or more conditions or combinations of conditions
- ⌘ Decision trees represent such decision as a **sequence of steps**
- ⌘ Decision tables describe **all possible combinations of conditions** and the decision appropriate to each combination
- ⌘ **Levels of uncertainty** can also be built into decision trees to account for the relative probabilities of the various outcomes

Decision Trees

- Decision trees are useful when multiple branching occurs in a structured decision process, although they can be quite effective when only two decision paths are called for.
- They are helpful when necessary to maintain a certain order for a series of decisions.

Decision Trees

⌘ Decision trees:

- edges of a decision tree represent conditions
- leaf nodes represent actions to be performed.

⌘ A decision tree gives a graphic view of:

- logic involved in decision making
- corresponding actions taken.

Drawing Decision Trees

- Drawn horizontally
- Root of tree is to the left side
- Square Nodes indicate actions
- Circle Nodes represent possible conditions
- Circle is analogous to the condition part of an IF statement
- Square is analogous to the consequent of an IF statement (the 'THEN' part)
- “IF Circle THEN Square”

Decision Trees

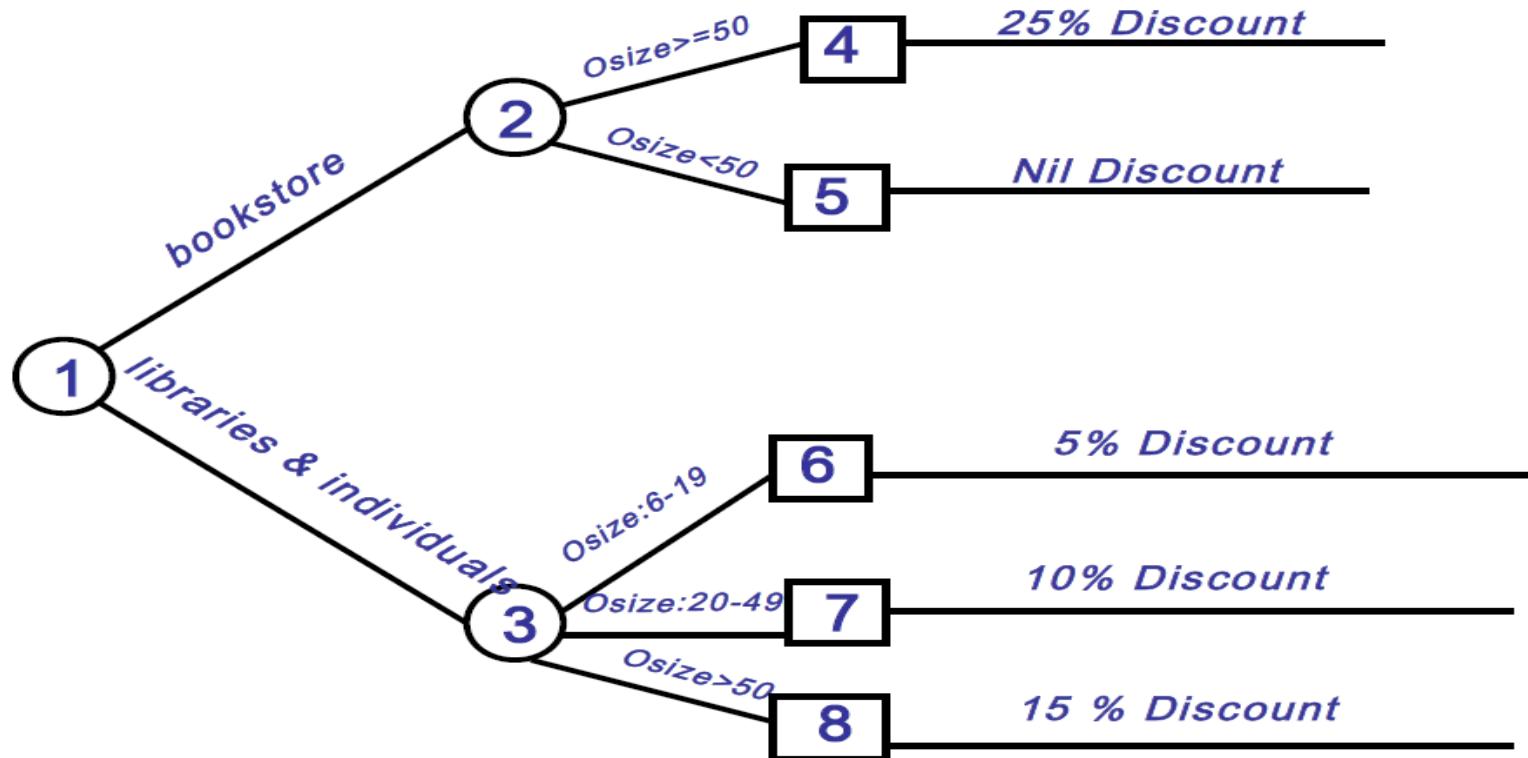
Assume the following discount policy:

“ Bookstores get a trade discount of 25%; for orders from libraries and individuals, 5% allowed on orders of 6-19 copies per book title; 10% on orders for 20-49 copies per book title; 15% on orders for 50 copies or more per book title.”

Using the Decision Table describe the above mentioned discount policy.

***Bookstores get a discount of 25% on orders for 50 copies or more per book title**

Decision Tree Example



Decision Table

- # A decision table shows in a tabular form:
 - ↗ processing logic and corresponding actions
- # Upper rows of the table specify:
 - ↗ the variables or conditions to be evaluated
- # Lower rows specify:
 - ↗ the actions to be taken when the corresponding conditions are satisfied.
- # In technical terminology,
 - ↗ a column of the table is called a rule:
 - ↗ A rule implies:
 - ☒ if a condition is true, then execute the corresponding action.

Decision Tables

- A **Decision table** is a table of rows and columns, separated into four quadrants and is designed to illustrate complex decision rules
 - **Condition Stub** – upper left quadrant
 - **Rules Stub** – upper right quadrant
 - **Action Stub** – bottom left quadrant
 - **Entries Stub** - bottom right quadrant
- Standard format used for presenting decision tables.

<i>Decision Stub</i>	<i>Rules Stub</i>
<i>Action Stub</i>	<i>Entries Stub</i>

Structure of Decision Table



	Decision rules			
	Rule 1	Rule 2	Rule 3	Rule 4
(condition stub)				
			(Condition entries)	
(action stub)			(Action entries)	

Decision Table Example

	1	2	3	4
Like Boss?	Y	Y	N	N
20% Pay Raise?	Y	N	Y	N
Stay Another Year	X	X		
Quit after 2 Months			X	
Quit next Week				X

Developing Decision Tables

- Process requires the determination of the number of **conditions** (inputs) that affect the decision.
- The set of possible **actions** (outputs) must likewise be determined
- The **number of rules** is computed
- Each rule must specify one or more **actions**

Number of Rules

- Each condition generally has two possible alternatives (outcomes): **Yes** or **No**
 - In more advanced tables, multiple outcomes for each condition are permitted
- The total number of rules is equal to
 $2^{\text{no. of conditions}}$
- Thus, if there are **four conditions**, there will be **sixteen possible rules**

Building the Table

- For each rule, select the appropriate action and indicate with an 'X'
- Identify rules that produce the same actions and attempt to combine those rules; for example:

<i>Condition 1</i>	<i>Y</i>	<i>Y</i>		<i>Condition 1</i>
<i>Y</i>				
<i>Condition 2</i>	<i>Y</i>	<i>N</i>		<i>Condition 2</i>
-				
<i>Action 1</i>		<i>X X</i>		<i>Action 1</i>
				<i>X</i>

Cleaning Things Up

- Check the table for any impossible situations, contradictions, and redundancies and eliminate such rules
- Rewrite the decision table with the most reduced set of rules; rearranging the rule order is permissible if it improves user understanding

Decision Table example: combine and reduce

Conditions and Actions	1	2	3	4	5	6	7	8
Order from Fall Catalog	Y	Y	Y	Y	N	N	N	N
Order from Christmas Catalog	Y	Y	N	N	Y	Y	N	N
Order from Special Catalog	Y	N	Y	N	Y	N	Y	N
Mail Christmas Catalog		X		X		X		X
Mail Special Catalog			X				X	
Mail Both Catalogs	X			X				

The four gray columns can be combined into a single rule. Note that for each, there was NO order placed from the Special Catalog.

In addition, Rules 1&5 and Rules 3&7 can be combined. Each pair produces the same action and each pair shares two common conditions.

Decision Table example ~ Final Version

Conditions and Actions	1	2	3
Order from Fall Catalog	--	--	--
Order from Christmas Catalog	Y	-	N
Order from Special Catalog	Y	N	Y
Mail Christmas Catalog		X	
Mail Special Catalog			X
Mail Both Catalogs	X		

Eliminates the need to check for every possible case.

Decision Table example: checking for completeness and accuracy

Conditions and Actions	1	2	3	4
Salary > \$50,000 per year	Y	Y	N	N
Salary < \$2,000 per month	Y	N	Y	N
Award Double Bonus		X		
Award Regular Bonus			X	
Award no Bonus				X

Although the Y-N Combinations suggest a rule, in this case, it is impossible for conditions 1 & 2 to exist simultaneously. They are in complete contradiction with each other. In the final version of the table, Rule 1 will disappear and Rules -4 will become Rules 1-3.



A Garment House announces its trade discount policy as follows:

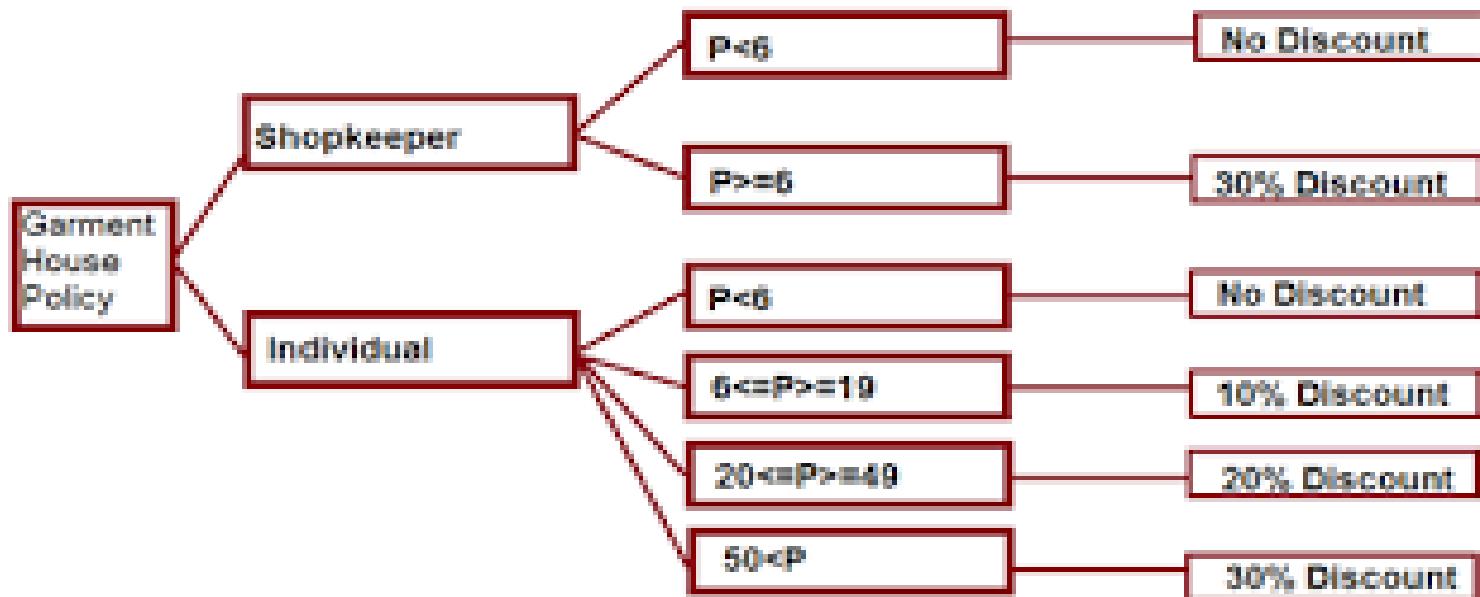
If a customer is from shop and does purchase garments more than 6 pair a flat discount of 30% would be provided.

If a customer is individual and does purchase garment of 6-19 pair 10% discount would be provided, 20% on discount for 20-49 pair and 30% discount on more than equal to 50 pair.

**In no other case, a discount would be provided.
Draw a decision tree and table table for above policies.**

- ⌘ A Garment House announces its trade discount policy as follows:-
- ☒ if a customer is from shop and does purchase garments more than 6 pair a flat discount of 30% would be provided.
 - ☒ If a customer is individual and does purchase garment of 6-19 pair 10% discount would be provided, 20% on discount for 20-49 pair and 30% discount on more than equal to 50 pair.
In no other case, a discount would be provided.
Draw a decision table for above policies.

Dicision Tree



Condition	R-1	R-2	R-3	R-4	Else
Shopkeeper	Y	N	N	N	
$p \geq 6$	Y				
$6 \leq p \leq 19$		Y			
$20 \leq p \leq 49$			Y		
$p \geq 50$				Y	

ACTION:

30%	X			X	
10%		X			
20%			X		
No Discount					X

Importance of Decision Tables

- Aids in the analysis of structured decisions
- Ensures completeness
- Checks for possible errors (impossible situations, contradictions, and redundancies, etc.)
- Reduces the amount of condition testing that must be done

Comparison



- ⌘ Both decision tables and decision trees
 - ↗ can represent complex program logic.
- ⌘ Decision trees are easier to read and understand
 - ↗ when the number of conditions are small.
- ⌘ Decision tables help to look at every possible combination of conditions.

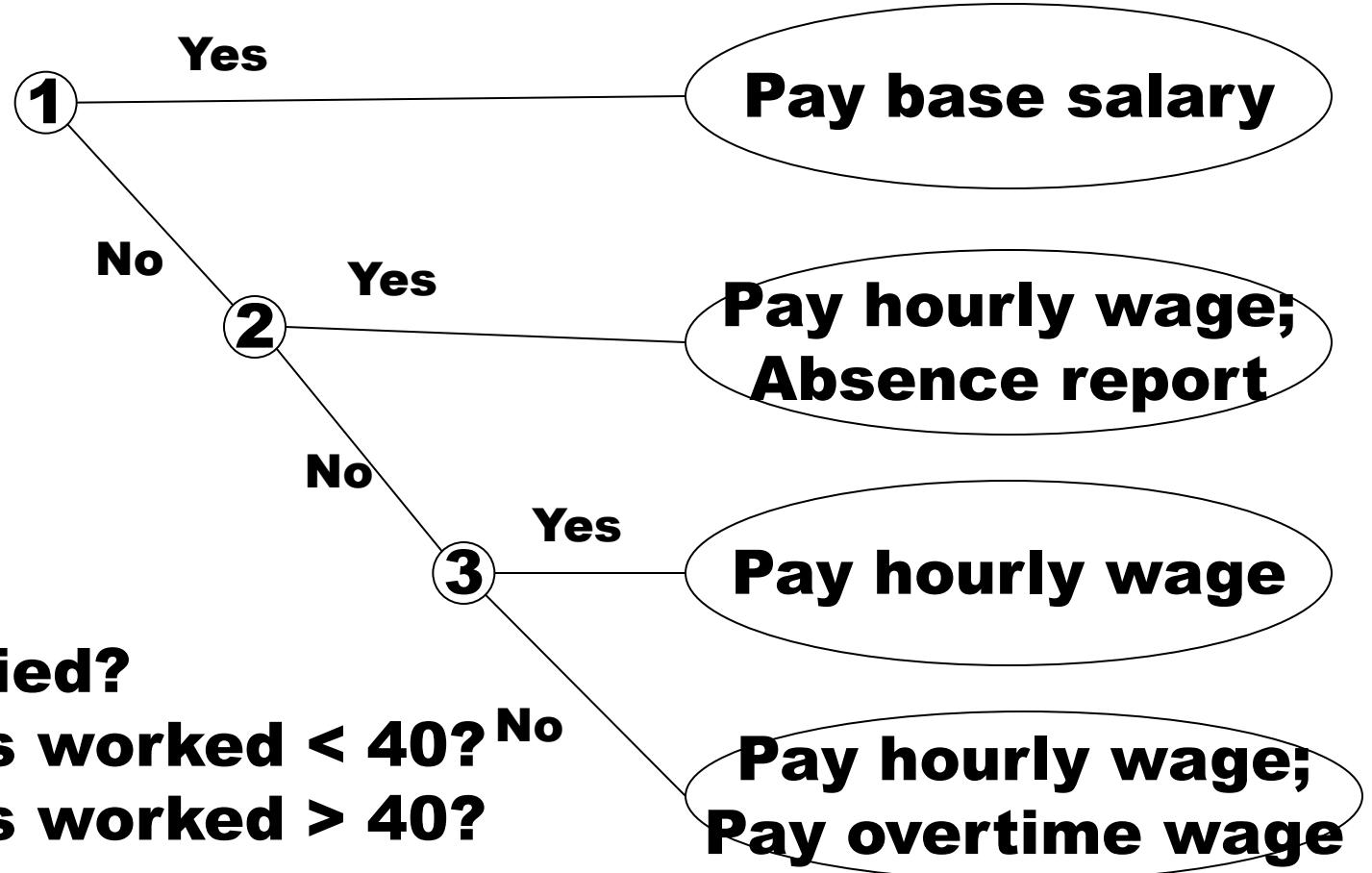


Constructing Decision Table



- ⌘ Name the conditions and the values each condition can assume
- ⌘ Name all possible actions that can occur
- ⌘ List all possible rules
- ⌘ Define the actions for each rule
- ⌘ Simplify the decision table

Example of Decision Tree



Example of Decision Table

Conditions/ Actions	R	u	I	e	s	
	1	2	3	4	5	6
Employee Type	S	H	S	H	S	H
Hours worked	<40	<40	40	40	>40	>40
Pay base salary	X		X		X	
Calculate Hourly wage		X		X		X
Calculate Overtime						X
Produce Absence Report		X				

Comparison



- ⌘ Both decision tables and decision trees
 - ↗ can represent complex program logic.
- ⌘ Decision trees are easier to read and understand
 - ↗ when the number of conditions are small.
- ⌘ Decision tables help to look at every possible combination of conditions.

Example: LMS

- ⌘ When the new member option is selected,
 - └ the software asks details about the member:
 - └ name,
 - └ address,
 - └ phone number, etc.
- ⌘ If proper information is entered,
 - └ a membership record for the member is created
 - └ a bill is printed for the annual membership charge plus the security deposit payable.

Example_(cont.)

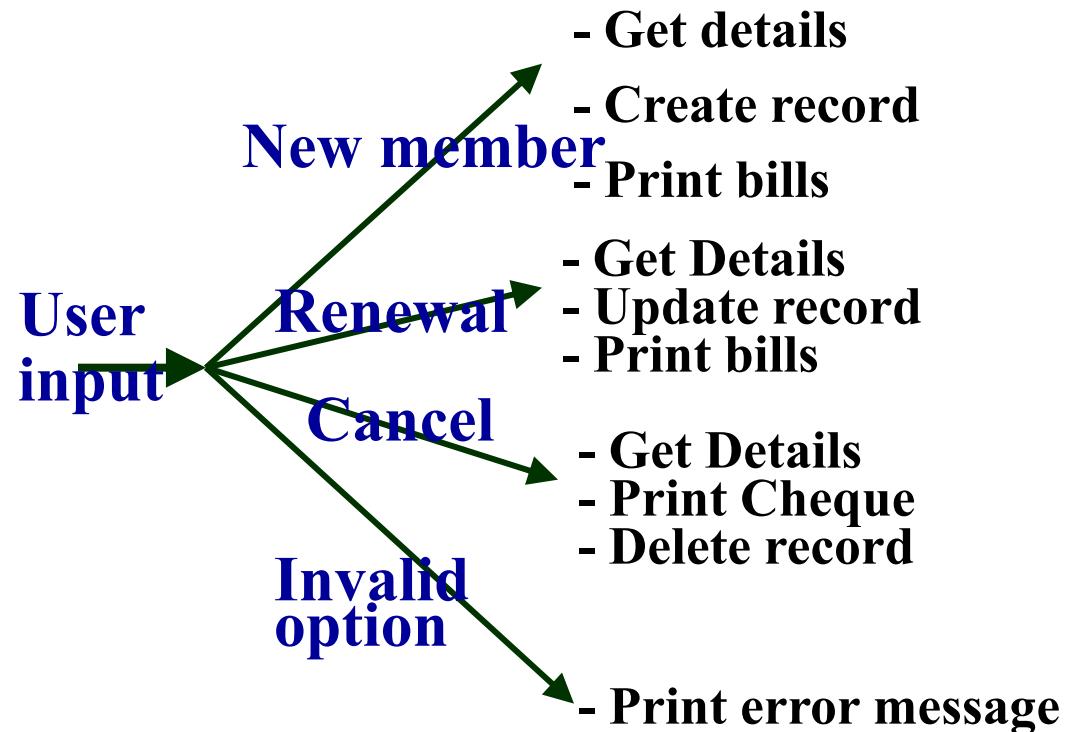


- ❖ If the renewal option is chosen,
 - ❖ LMS asks the member's name and his membership number
 - ❖ checks whether he is a valid member.
 - ❖ If the name represents a valid member,
 - ❖ the membership expiry date is updated and the annual membership bill is printed,
 - ❖ otherwise an error message is displayed.

Example_(cont.)

- ⌘ If the cancel membership option is selected and the name of a valid member is entered,
 - ℳ the membership is cancelled,
 - ℳ a cheque for the balance amount due to the member is printed
 - ℳ the membership record is deleted.

Decision Tree



Decision Table

❖ Decision tables specify:

- ❑ which variables are to be tested
- ❑ what actions are to be taken if the conditions are true,
- ❑ the order in which decision making is performed.

Decision Table

- # A decision table shows in a tabular form:
 - ↗ processing logic and corresponding actions
- # Upper rows of the table specify:
 - ↗ the variables or conditions to be evaluated
- # Lower rows specify:
 - ↗ the actions to be taken when the corresponding conditions are satisfied.
- # In technical terminology,
 - ↗ a column of the table is called a rule:
 - ↗ A rule implies:
 - ☒ if a condition is true, then execute the corresponding action.

Example:

⌘ Conditions

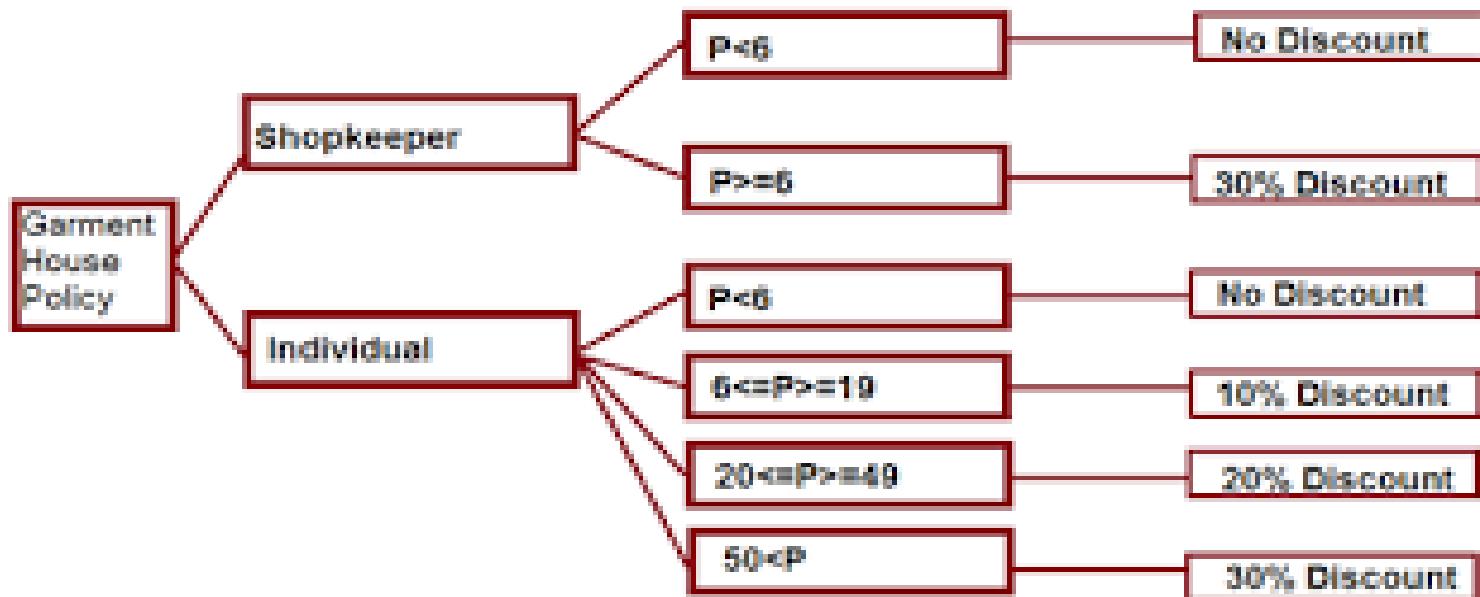
Valid selection		NO	YES	YES	YES
New member	--	YES	NO	NO	
Renewal	--	NO	YES	NO	
Cancellation		--	NO	NO	YES

⌘ Actions

Display error message	-	-	-	-
Ask member's name etc.				
Build customer record	--		--	--
Generate bill	--		--	
Ask membership details	--			
Update expiry date	--	--	--	
Print cheque	--	--	--	
Delete record	--	--	--	

- ⌘ A Garment House announces its trade discount policy as follows:-
- ☒ if a customer is from shop and does purchase garments more than 6 pair a flat discount of 30% would be provided.
 - ☒ If a customer is individual and does purchase garment of 6-19 pair 10% discount would be provided, 20% on discount for 20-49 pair and 30% discount on more than equal to 50 pair.
In no other case, a discount would be provided.
Draw a decision table for above policies.

Dicision Tree



Condition	R-1	R-2	R-3	R-4	Else
Shopkeeper	Y	N	N	N	
p>=6	Y				
6<=p>=19		Y			
20<=p<=49			Y		
p>=50				Y	
ACTION:					
30%	X			X	
10%		X			
20%			X		
No Discount					X

Summary

- ⌘ Requirements analysis and specification
 - ─ an important phase of software development:
 - ─ any error in this phase would affect all subsequent phases of development.

- ⌘ Consists of two different activities:
 - ─ Requirements gathering and analysis
 - ─ Requirements specification

Summary

- ⌘ The aims of requirements analysis:
 - ▢ Gather all user requirements
 - ▢ Clearly understand exact user requirements
 - ▢ Remove inconsistencies and incompleteness.

- ⌘ The goal of specification:
 - ▢ systematically organize requirements
 - ▢ document the requirements in an SRS document.

Summary

- ⌘ Main components of SRS document:
 - ─ functional requirements
 - ─ nonfunctional requirements
 - ─ constraints
- ⌘ Techniques to express complex logic:
 - ─ Decision tree
 - ─ Decision table

Summary

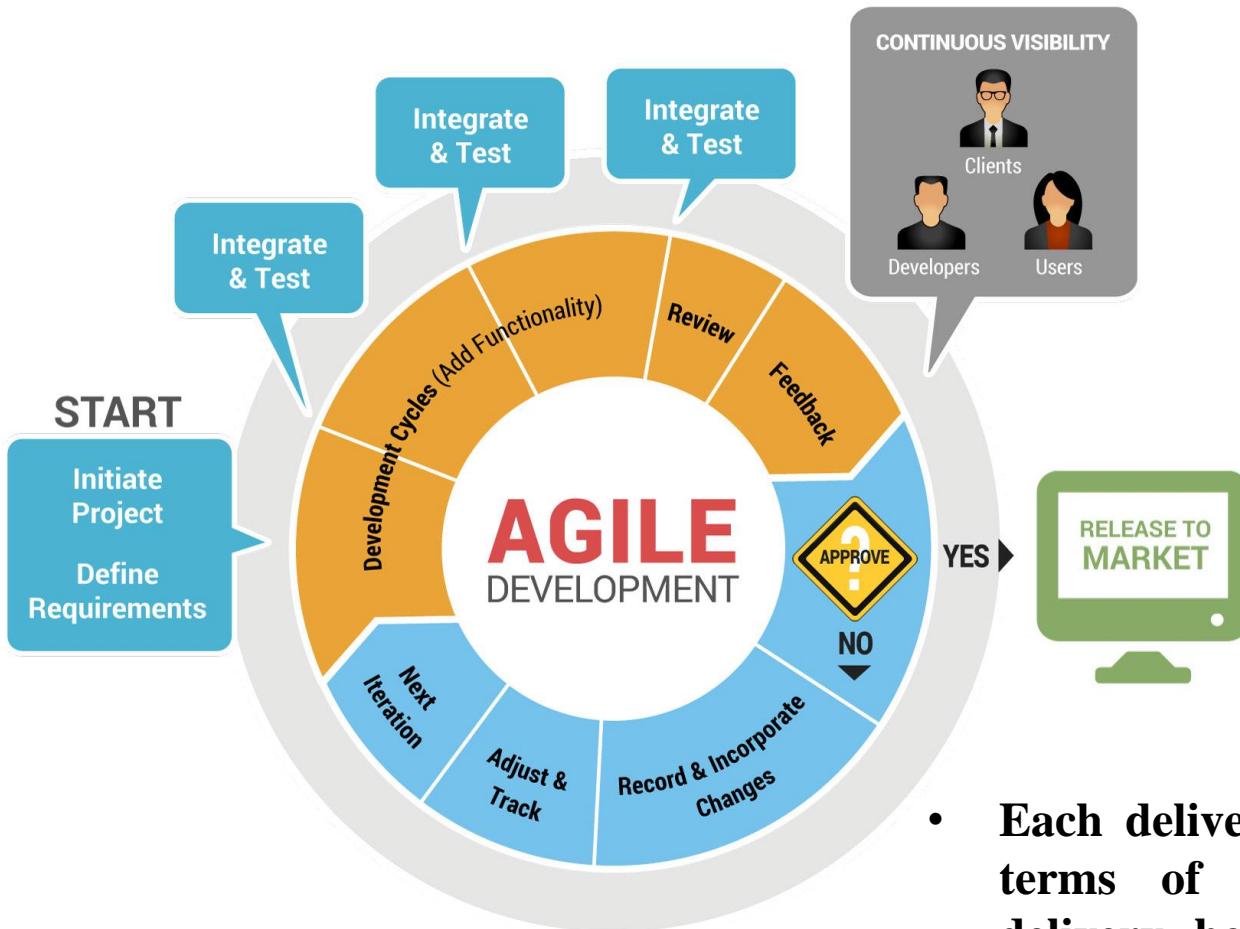
- ❖ Formal requirements specifications have several advantages.
- ❑ But the major shortcoming is that these are hard to use.

The word ‘agile’ means –
Able to move your body quickly and
easily.
Able to think quickly and clearly.

What is a Methodology?

- A *methodology* is a formalized process or set of practices for creating software
 - A set of rules you have to follow
 - A set of conventions the organization decides to follow
 - A systematical, engineering approach for organizing software projects

Agile Methodology



- Iterative approach is taken and working software is delivered after each iteration

- People collaborating together to build high quality products that meet customer needs at a sustainable pace
- The tasks are divided to time boxes (small time frames) to deliver specific features for a release
- Each delivery is incremental in terms of features; the final delivery holds all the features required by the customer

Agile v/s Waterfall

Agile Development

Allows interim change in customer development

Iterative and Incremental

Produces a workable software with every milestone

Requires less cumbersome documentation

Waterfall Model

Adheres to the plan developed at the beginning of project

Phased and Sequential

Only a final product is delivered at the end

Emphasis on clear and elaborate documentation

- Extreme Programming (XP) is an agile software development framework that aims to produce higher quality software, and higher quality of life for the development team.
- XP is the most specific of the agile frameworks regarding appropriate engineering practices for software development.

- The general characteristics where XP is appropriate :
 - Dynamically changing software requirements
 - Risks caused by fixed time projects using new technology
 - Small, co-located extended development team
 - The technology you are using allows for automated unit and functional tests

XP Values

- Basic values of Xp
 - Communication
 - Simplicity
 - Feedback
 - Courage

XP Value: Communication

- Poor communication in software teams is one of the root causes of failure of a project
- Stress on good communication between all stakeholders--customers, team members, project managers
- XP emphasizes value of communication in many of its practices:
 - On-site customer, user stories, pair programming, collective ownership (popular with open source developers), daily standup meetings, etc

XP Value: Simplicity

- ‘Do the Simplest Thing That Could Possibly Work’
 - Implement a new capability in the simplest possible way
 - Refactor the system to be the simplest possible code with the current feature set
- ‘You Aren’t Going to Need It’
 - Never implement a feature you don’t need now

XP Value: Feedback

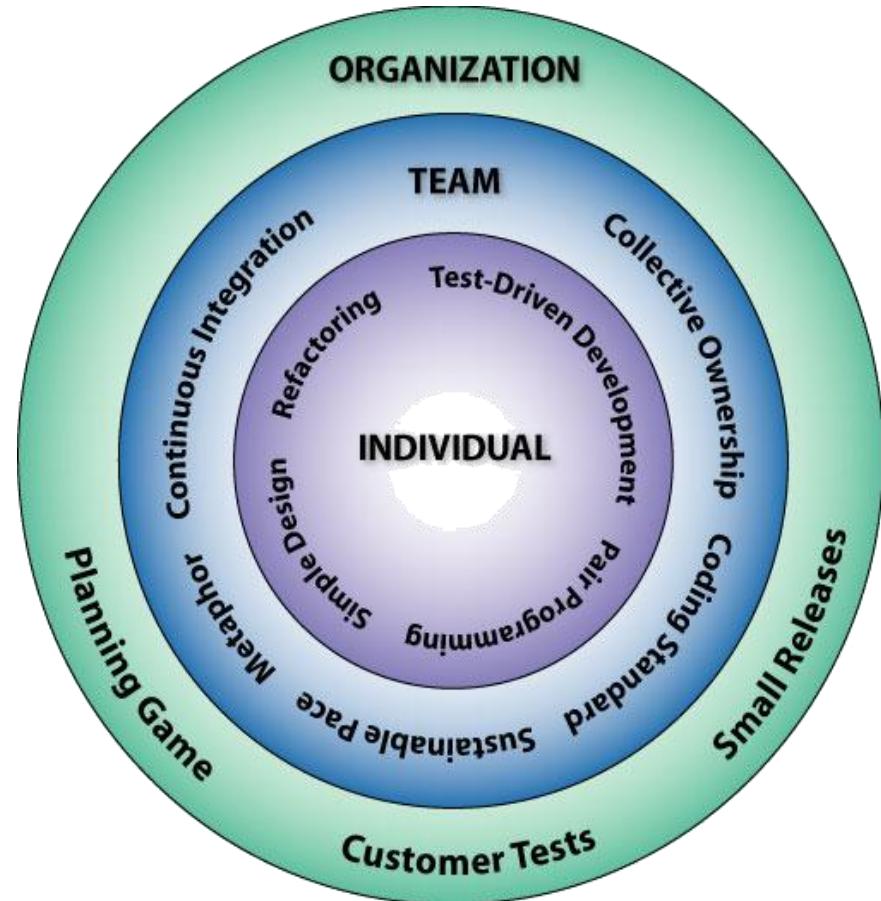
- Programmers produce new releases every 2-3 weeks for customers to review
- Unit tests tell programmers status of the system
- Small iteration and pair programming help a great deal to give a proper understanding of where they stand
- Hence, Feedback is repetitive and frequent in XP

XP Value: Courage

- The courage to communicate and accept feedback
- The courage to throw code away (prototypes)
- The courage to refactor the architecture of a system

The 12 Key Practices

- The Planning Game
- Small Releases
- Metaphor
- Simple Design
- Test-Driven Development
- Refactoring
- Pair Programming
- Collective Ownership
- Continuous Integration
- 40-Hour Workweek
- On-site Customer
- Coding Standards



Basic XP Practices

- XP is based on 12 key practices:
 - **The Planning process**
At the start of each iteration customers, developers meet to estimate the requirements(stories) for the next release.
 - **Small Releases**
Start with the smallest useful feature set. Release early and often, adding new features each time.
 - **System Metaphor**
Is a naming concept, that should make it easy for a team member to guess the functionality of a particular class/method, from its name only.
 - **Simple Design**
Go for the simplest possible design that gets the job done. Do what's needed to meet today's needs.

Basic XP Practices....contd

- **Continuous Testing**

Before programmers add a feature they write a test for it.

- **Refactoring**

Refactor out any duplicate code generated in a coding session.

- **Pair Programming**

All production code is written by two programmers setting at one machine.

Driver: types the code

Navigator or Observer: reviews the code.

The two programmers switch roles frequently.

- **Collective Ownership**

No single person owns a module.

Basic XP Practices....contd

- **Continuous Integration**

All changes are integrated into the code-base at least daily. The tests have to run 100% both before and after integration.

- **40-Hour Week**

Requirements should be selected for each iteration such that developers need not put in overtime.

- **On-Site Customer**

Development team has continuous access to a real live customer , that is , someone who will actually be using the system

- **Coding Standards**

Everyone codes to the same standards.

How XP Solve Some SE Problems

Problem	Solution
Slipped schedule	Short development cycles
Cancelled project	Intensive customer presence
Cost of changes	Extensive, ongoing testing, system always running
Defect rates	Unit tests, customer tests
Misunderstand the business	Customer part of the team
Business changes	Changes are welcome
Staff turnover	Intensive teamwork

Scrum

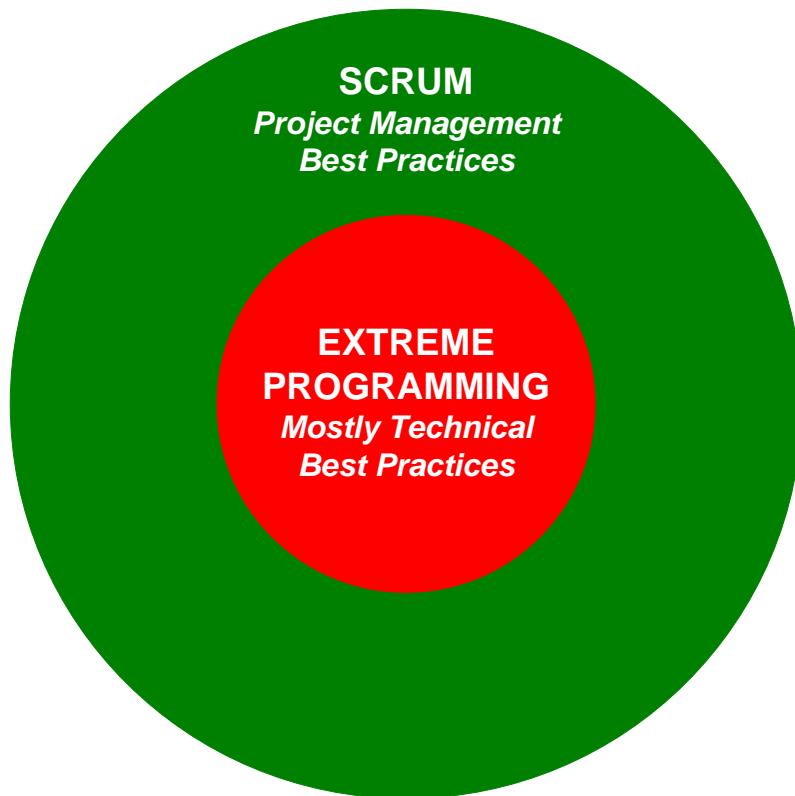
It is the most popular agile framework, which concentrates particularly on how to manage tasks within a team-based development environment.

Scrum uses iterative and incremental development model, with shorter duration of iterations.

Scrum is relatively simple to implement and focuses on quick and frequent deliveries.

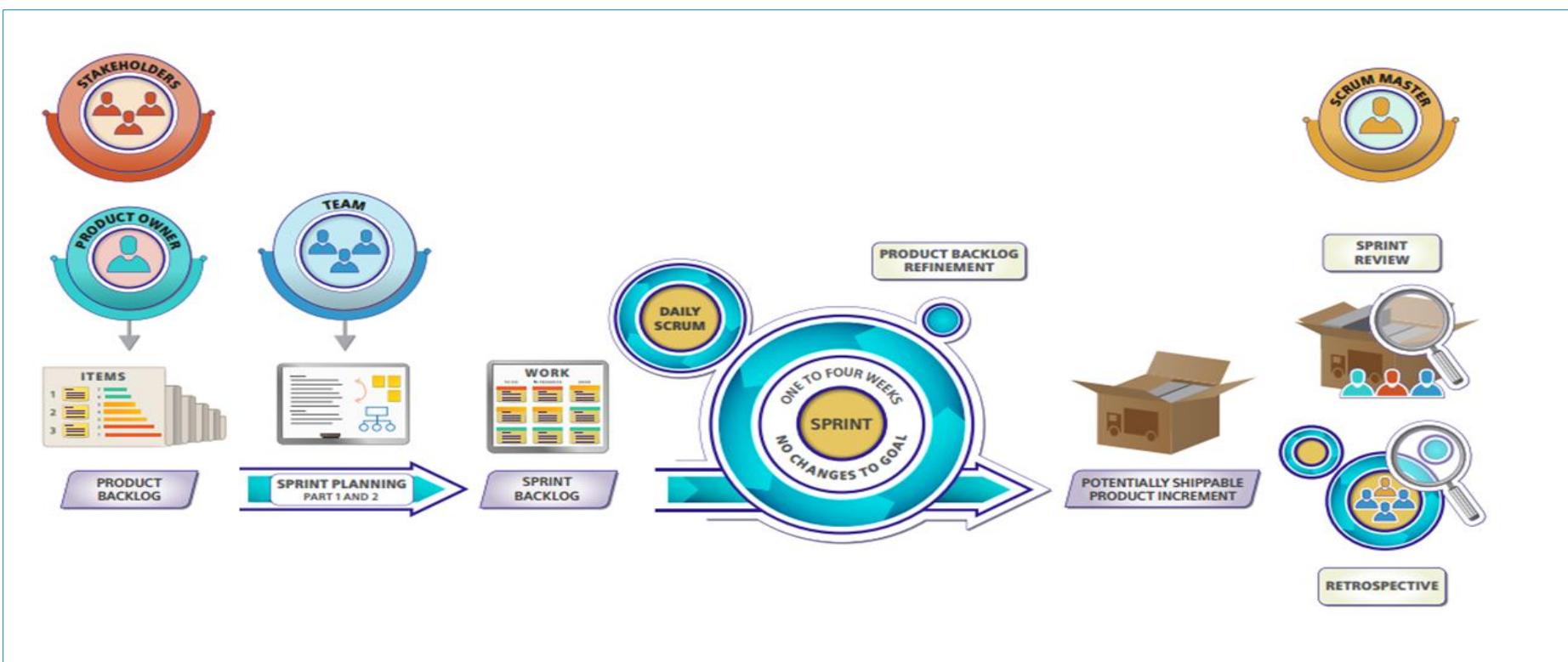
Agile using SCRUM

Scrum with Extreme Programming



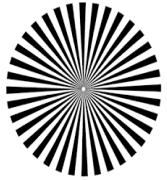
Scrum works well as a *wrapper* around Extreme Programming

What is SCRUM?



- **Scrum is an agile lightweight process framework for development**
- **Can manage and control software/ development**
- **Uses iterative, incremental practices**
- **Has a simple implementation and focuses on quick and frequent deliveries.**

SCRUM Values



Focus

- Focus only on few things at a time
- Work well together and produce excellent work

Respect

- Respect and trust each other within the team
- Sharing success and failure

Openness

- Every thing about the project is transparent to every one
- Create a culture of openness

Commitment

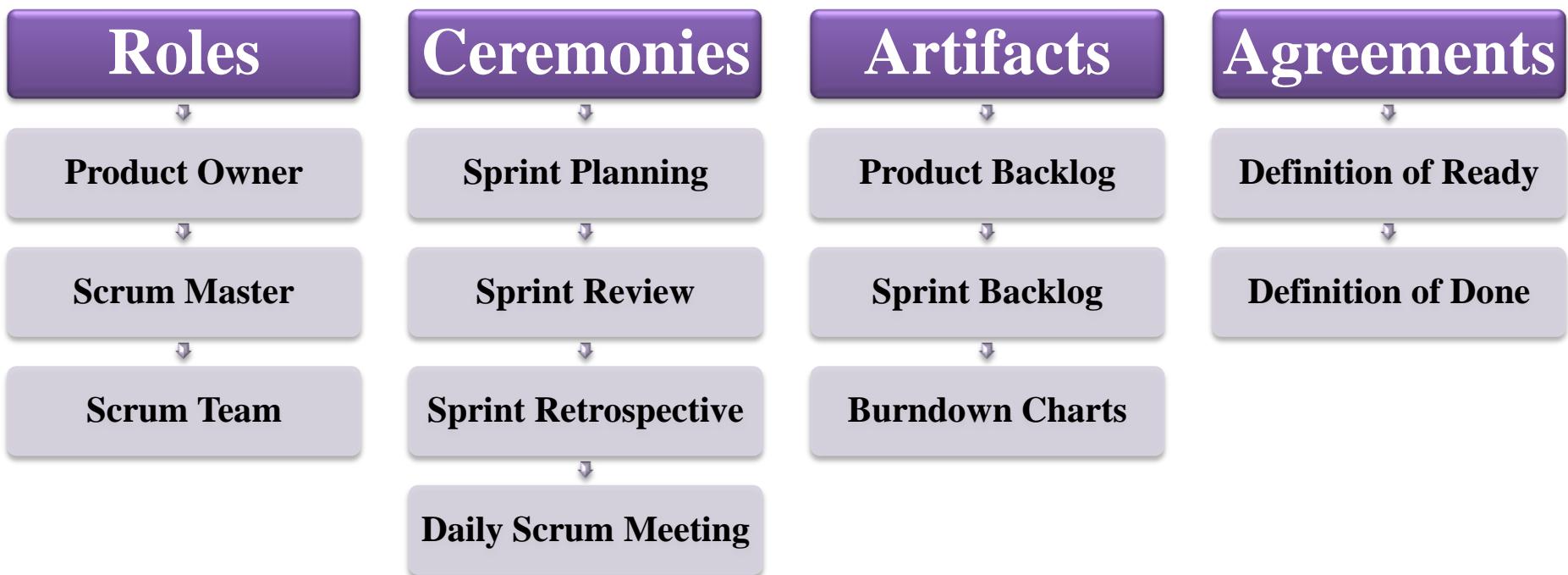
- Greater control over what we do and how we do
- We become more committed for success

Courage

- As we work together and feel supported we are courageous to be open and challenge ourselves to go beyond our capabilities

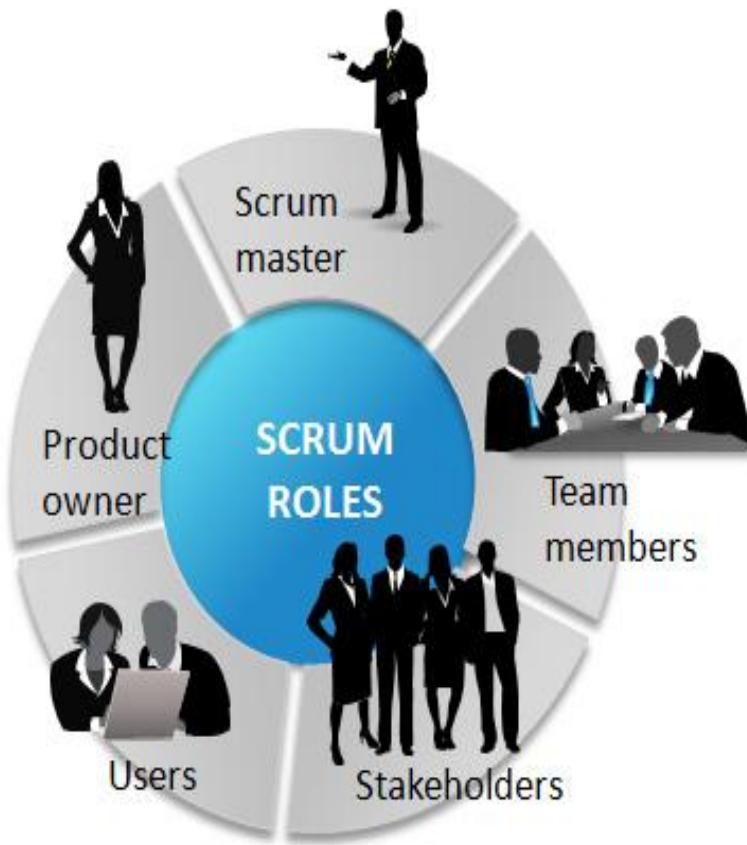
SCRUM Framework

- The Scrum framework consists of Scrum Teams and their associated roles, events, artifacts, and rules. Each component within the framework serves a specific purpose and is essential to Scrum's success and usage



SCRUM Roles

SCRUM Roles



Product Owner

- Represents the Business side of the Product / Software Development

Scrum Master

- Represents the People side of the Product/Software Development

Development Team

- Represents the Technology side of the Product/Software Development

Product Owner

Represents the business side of the Product/Software Development

The Product Owner is responsible for maximizing the value of the product and the work of the Development Team

The Product Owner is the sole person responsible for managing the Product Backlog

Creating Product Vision

Stakeholder Management

Scope Management

Release Management



Development Team

The Development Team is responsible for delivering a potentially shippable increment of working software

Develop high quality solution to fulfil the product/software vision

The Development Team is Self-organized

The Development Team is Cross functional

The Development Team is Empowered & autonomous

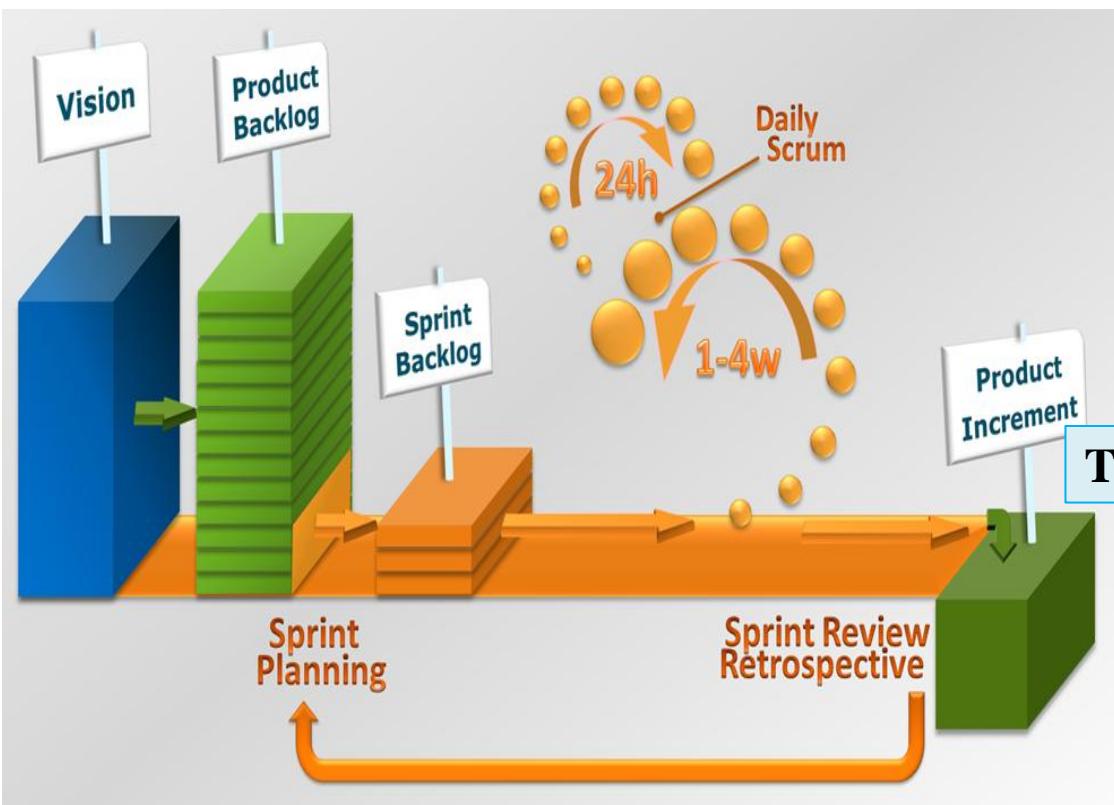
Collectively responsible for delivery

Size will be between 6 to 7 (recommended)



SCRUM Sprint

The Sprint



During which a “Done”, useable, and potentially shippable product Increment is created

A new Sprint starts immediately after the conclusion of the previous Sprint

Scope may be clarified and re-negotiated between the Product Owner and Development Team as more is learned.

The heart of Scrum is a Sprint

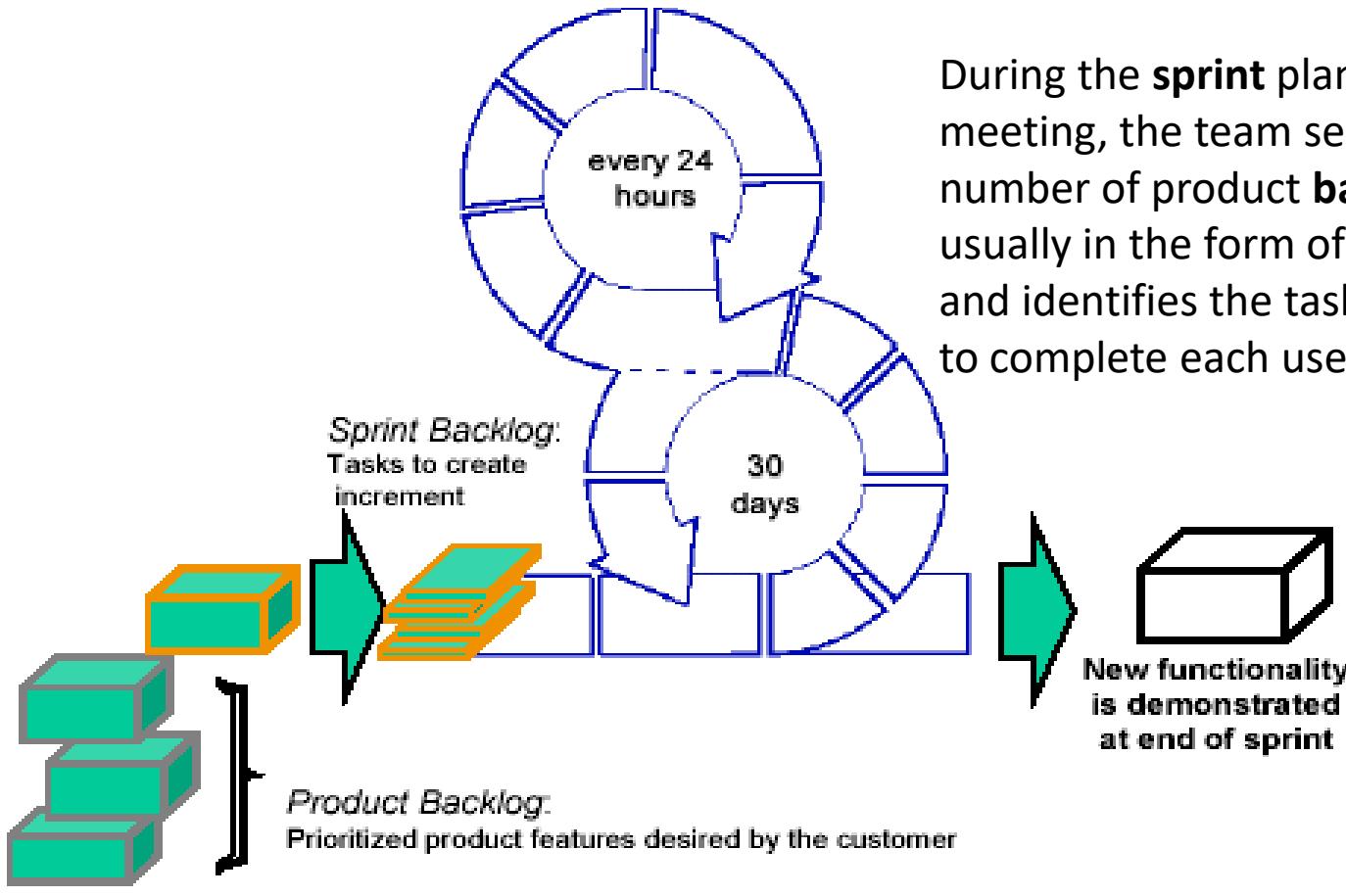
A sprint is a set period of time during which specific work has to be completed and made ready for review

Time-box of one month or less

Sprints consists of the Sprint Planning, Daily Scrums, the development work, the Sprint Review, and the Sprint Retrospective

No changes are made that would compromise the Sprint Goal

Scrum Scheduling and Tracking



During the **sprint** planning meeting, the team selects some number of product **backlog** items, usually in the form of user stories, and identifies the tasks necessary to complete each user story.

Product backlog in Scrum is a prioritized features list, containing short descriptions of all functionality desired in the **product**.

The **sprint backlog** is a list of tasks identified by the **Scrum** team to be completed during the **Scrum sprint**.

Timeboxing

Its fixed span of time called as Sprint in Scrum which has:

Fixed Goal, Fixed Team, Fixed Length, Product Increment at the end



Maximum length of the Sprint

-Recommended by Scrum framework is 4 weeks

Why Timeboxing?

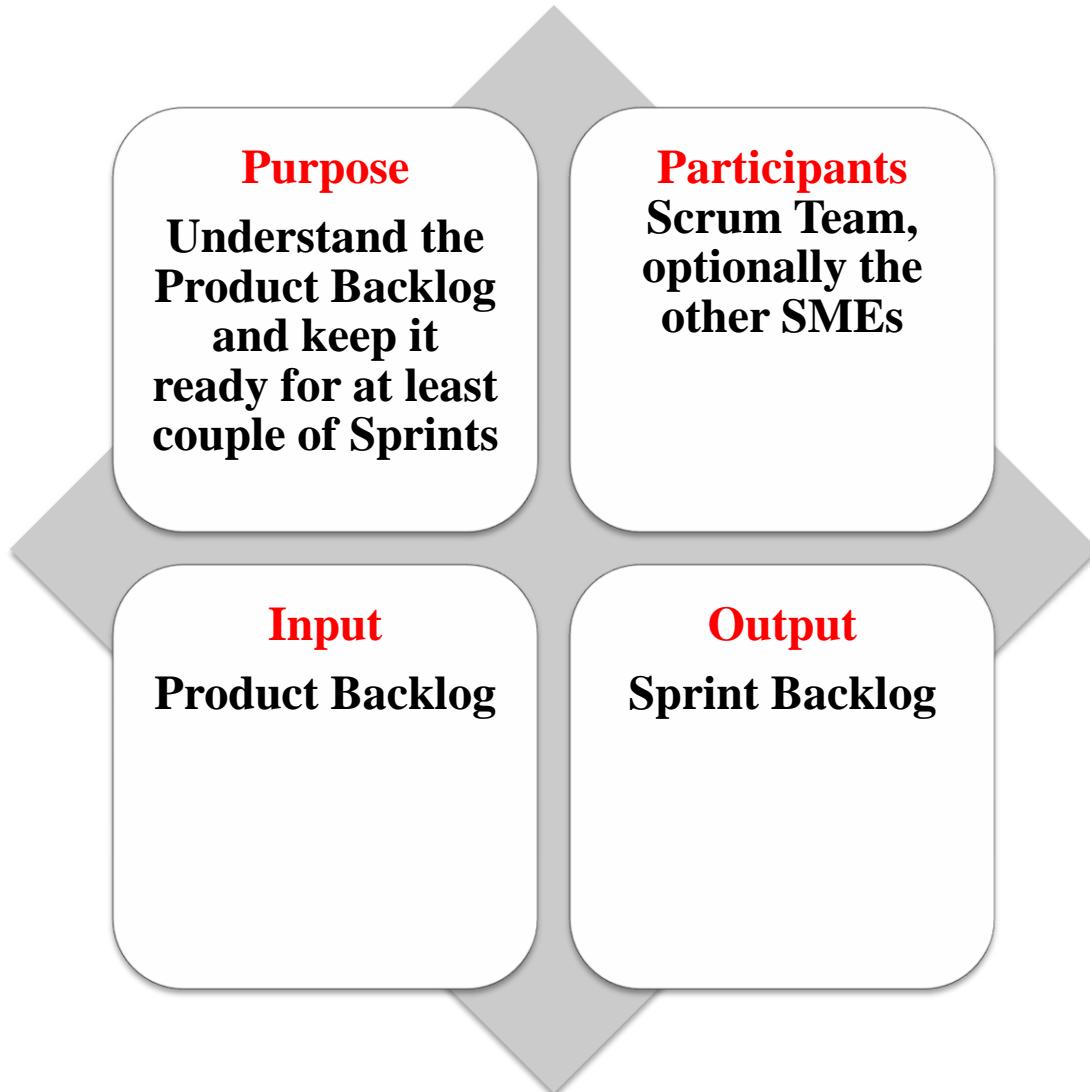
- Helps stay focused on the Goal**
- If we don't decide the time box of any ceremony or events, then some people will keep on discussing and others will lose interest**
- Moreover, the team will get less time to work on sprint tasks. So having the timeboxes is very important to keep people motivated**

Activities/Events/Ceremonies in SCRUM

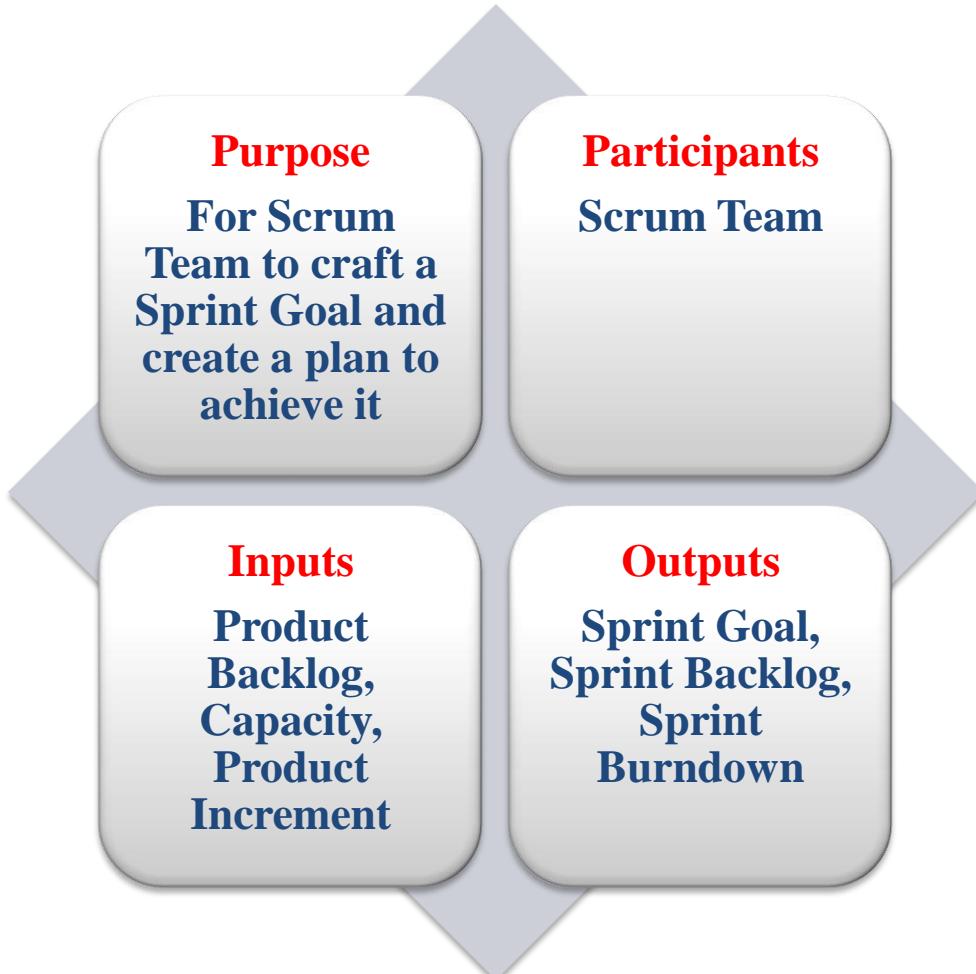


Product Backlog Refinement-

Timebox [Up to 10% of Sprint Capacity]



Sprint Planning- Timebox [Max 2-8 hours]



Activities:

- Craft a Goal:** Answer why are you running this sprint
- Create a Plan:** Identify PBIs (Product Backlog Items) required to achieve the Sprint Goal and commit based on Teams Velocity.

Sprint Planning- Timebox [Max 2-8 hours]

PRODUCT BACKLOG

[View Grades, current semester](#)
As a student, I can see my grades online so that I don't have to wait until I get to school to know whether I'm passing.
Acceptance Criteria: Columns align neatly on FingerFly 4.1 and iPhone.
EFFORT: SMALL.

[Update Grades, current semester](#)
As a teacher, I can update grades online so I don't depend on administrators to do it for me.
EFFORT: MEDIUM.

[View Grades, previous semester](#)
EFFORT: SMALL.

[Attendance](#)
EFFORT: MEDIUM.

[GPA](#)
EFFORT: SMALL.

[Report Cards](#)
EFFORT: EXTRA LARGE.

[Event Calendar](#)

[Alumni Archives](#)

SPRINT BACKLOG

COMMITTED BACKLOG ITEMS

NOT STARTED	IN PROGRESS	COMPLETED

PRODUCT OWNER

DEVELOPMENT TEAM



SPRINT TIMEBOX

Two weeks

M	T	W	Th	F	M	T	W	Th	F
5:00	6:00	7:00	8:00	9:00	10:00	11:00	12:00	1:00	2:00

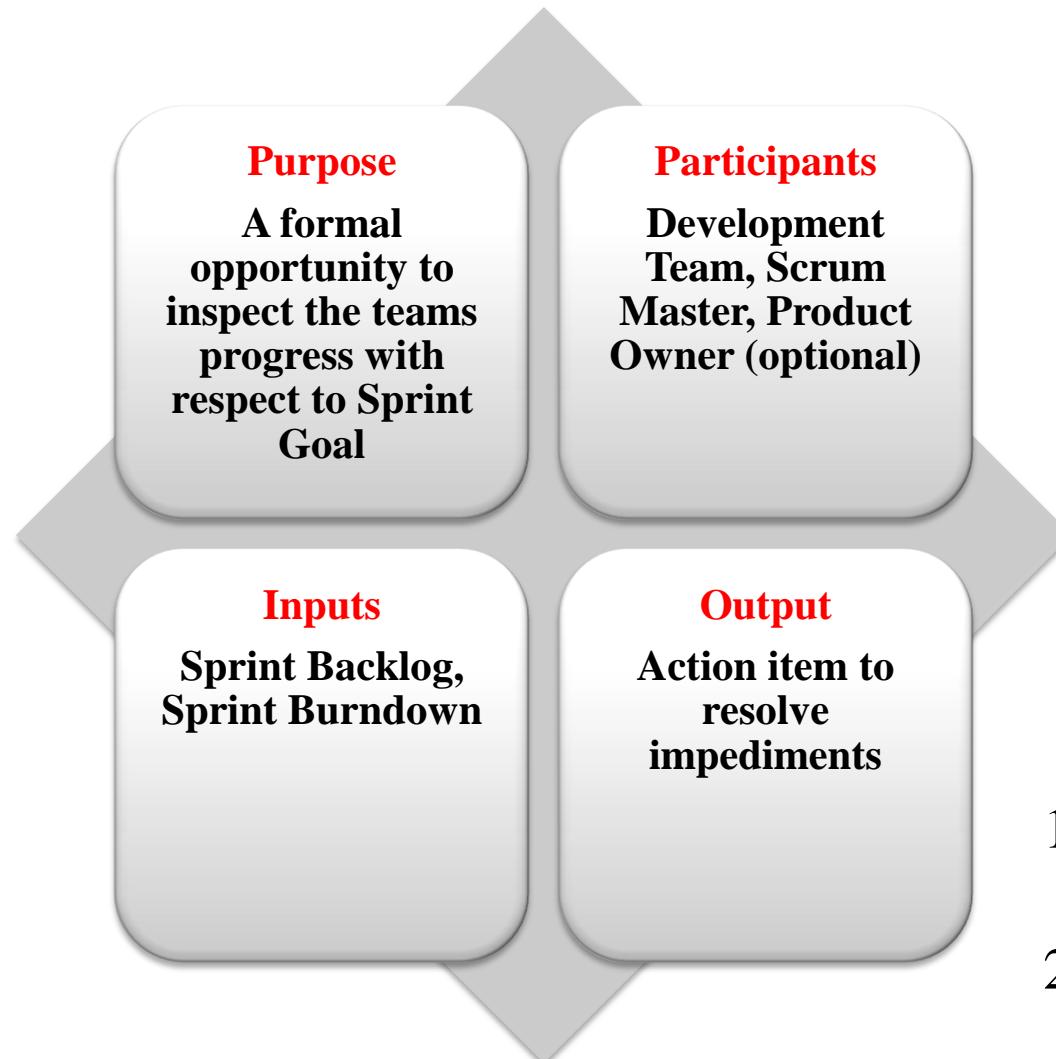
ScrumMaster

Meeting Timebox



Sprint Planning Phase

• Daily Scrum- Timebox [Max 15 minutes]



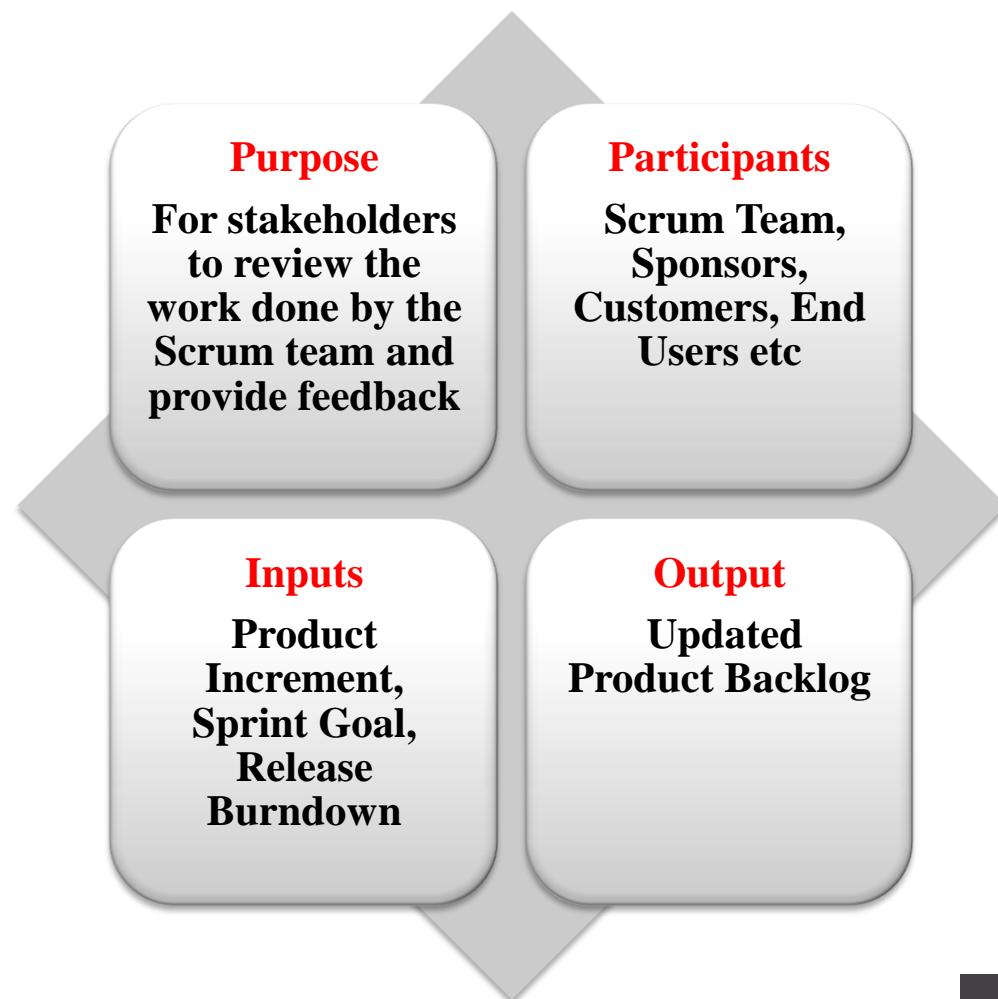
Activities:

1. ***Inspect***: Where do we stand with respect to the Sprint Goal?
2. ***Identify***: Are there any impediments to making progress?
3. ***Adapt***: What should we do to achieve Sprint Goal?

3 Important Questions

1. What work did you complete yesterday?
2. What have you planned for today?
3. Are you facing any problems or issues?

Sprint Review- Timebox [Max 1-4 hours]

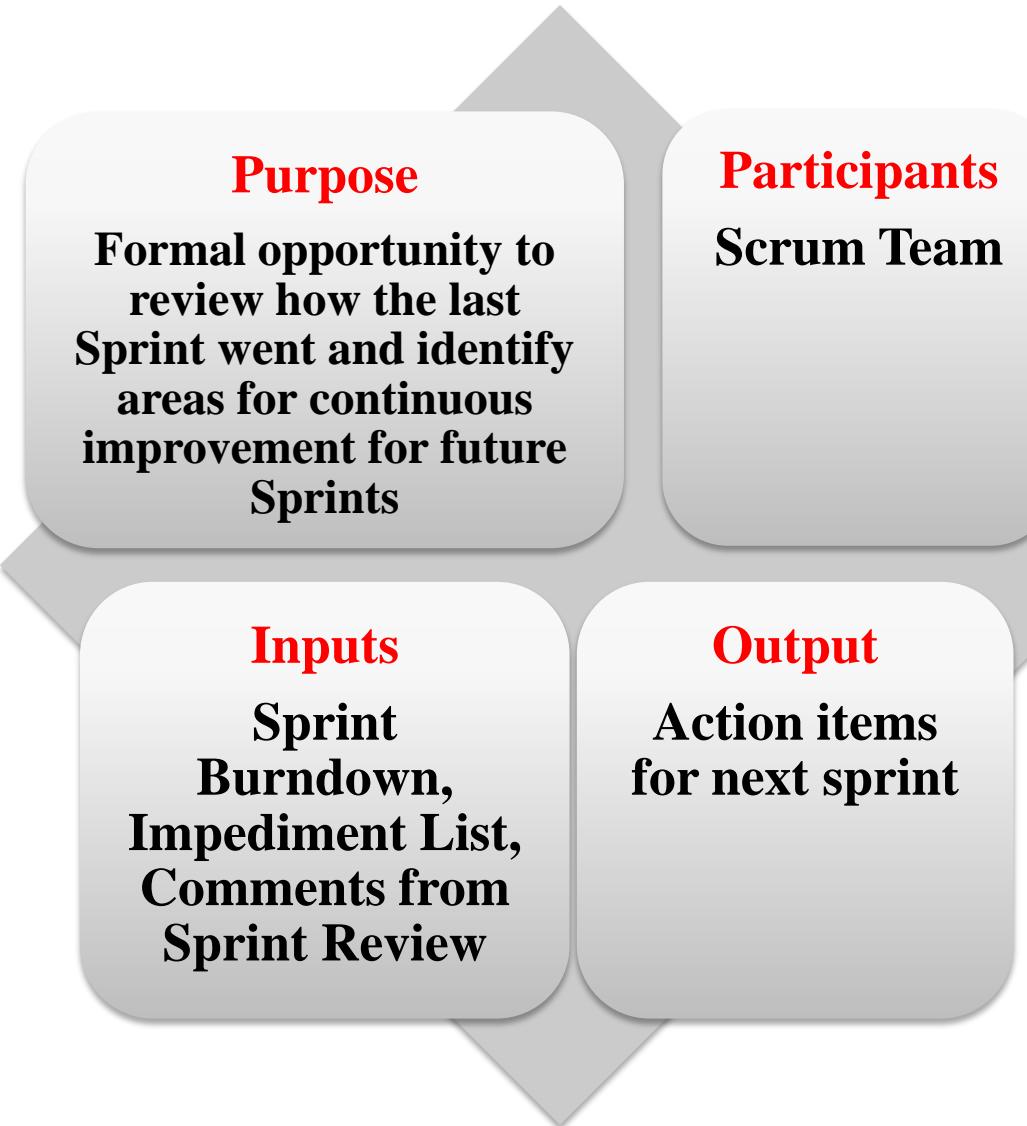


Activities:

- Report:** Product Owner explains the Sprint Goal and what is done and not done
- Demo:** Development Team demos the Product Increment and Stakeholders provide feedback
- Forecast:** Product Owner discusses the Product Backlog as it stands today and projected completion date

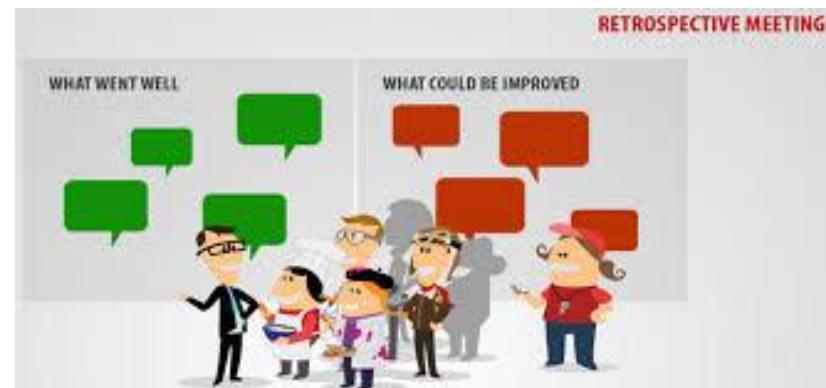


Sprint Retrospective - Timebox [Max 1-4 hours]

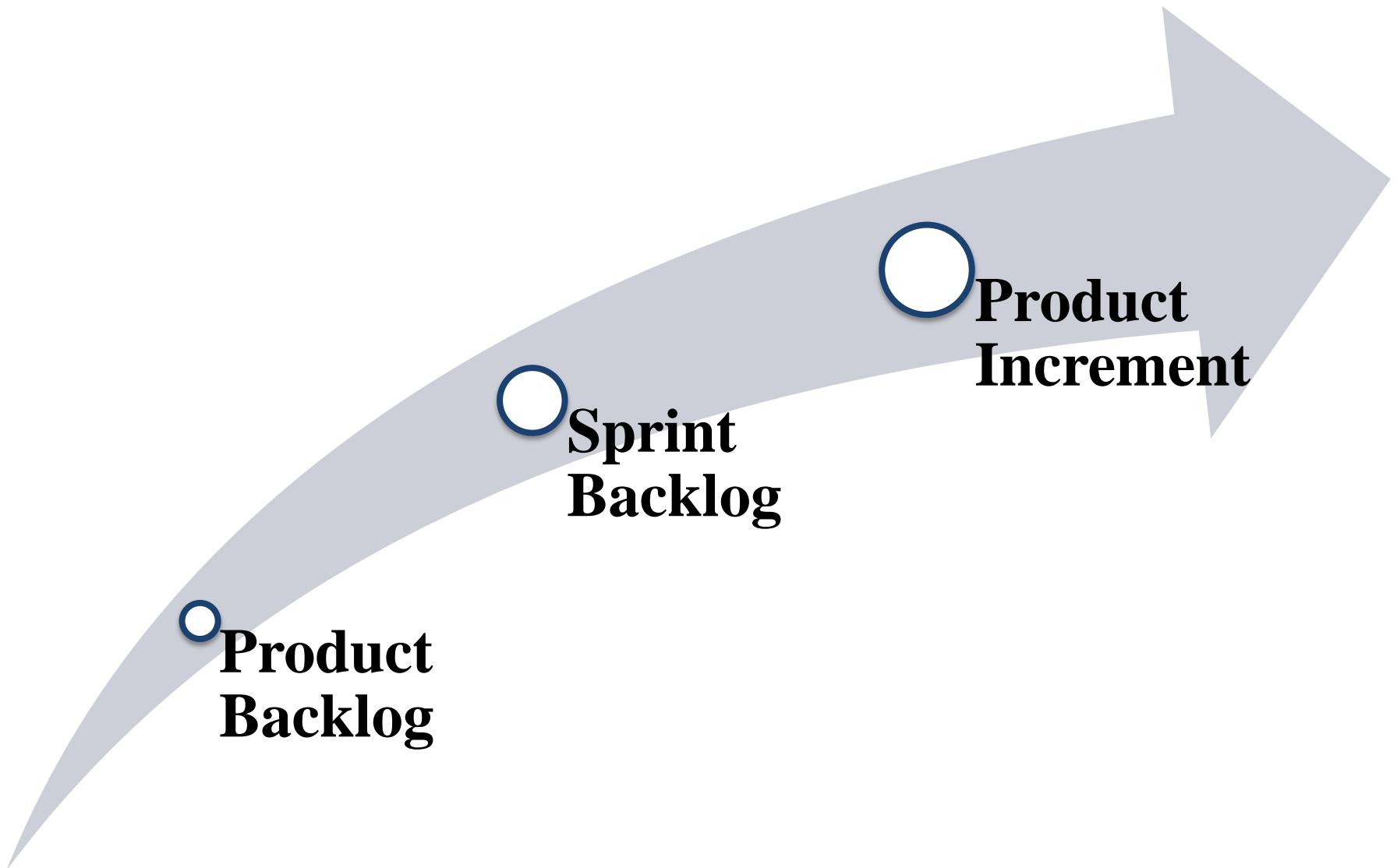


Activities:

- Reflect Back:** Scrum Team discusses how the Sprint went and identify root causes and action items for improvement
- Look Forward:** Identify improvement ideas for continuous improvement



Artifacts of SCRUM



Product Backlog

Anything and everything required to fulfill the product vision contains

- **Functional Requirements**
- **Non-functional Requirements**
- **Change Requests**
- **Enhancements**
- **Defects**
- **POC (Proof of Concept)**
- **Should be detailed enough**

Sprint Backlog

It's a plan which helps Development Team to self-organize to achieve the Sprint Goal

- The PBI's that team plans to deliver to achieve Sprint Goal
- Tasks required to accomplish planned PBI's

Product Increment

It's an integrated increment of final product released iteratively

- **Software Package**
- **Release Notes**
- **User Manual**
- **Support Documents etc.**

Scrum Summary

Roles



PO

Product Owner:
Set priorities



SM

ScrumMaster:
Manage process,
remove blocks



T

Team: Develop
product



SH

Stakeholders:
observe & advise

Key Artifacts

Product Backlog

- List of requirements & issues
- Owned by Product Owner
- Anybody can add to it

Sprint Goal

- One-sentence summary
- Declared by Product Owner

Sprint Backlog

- List of tasks
- Owned by team

Blocks List

- List of blocks & unmade decisions
- Owned by ScrumMaster

Increment

- Version of the product
- Shippable functionality (tested)

Key Meetings

Sprint Planning Meeting

- Hosted by ScrumMaster; ½-1 day
 - In: Product Backlog, existing product, business & technology conditions
1. Select highest priority items in Product Backlog; declare Sprint Goal
 2. Team turns selected items into

Daily Scrum

- Hosted by ScrumMaster
- Attended by all, but Stakeholders don't speak
- Same time every day
- Answer: 1) What did you do yesterday? 2) What will you do today? 3) What's in your way?
- Team updates Sprint Backlog;

Sprint Review Meeting

- Hosted by ScrumMaster
- Attended by all
- Informal, 4-hour, informational
- Team demos Increment
- All discuss
- Hold retrospective
- Announce next Sprint Planning

Development Process

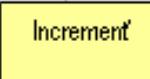


Sprint Planning Meeting

30 days each

Daily Scrum

Daily Work



Product Backlog'

Pros & Cons of SCRUM

Pros

- Is a very realistic approach to software development
- Promotes teamwork and cross training
- Functionality can be developed rapidly & demonstrated
- Resource requirements are minimum
- Suitable for fixed or changing requirements
- Delivers early partial working solutions
- Good model for environments that change steadily
- Minimal rules, documentation easily employed
- Enables concurrent development and delivery within an overall planned context
- Little or no planning required
- Easy to manage
- Gives flexibility to developers

Cons

- Not suitable for handling complex dependencies
- More risk of sustainability, maintainability and extensibility
- An overall plan, an agile leader and agile PM practice is a must without which it will not work
- Depends heavily on customer interaction, so if customer is not clear, team can be driven in the wrong direction
- There is very high individual dependency, since there is minimum documentation generated
- Transfer of technology to new team members may be quite challenging due to lack of documentation

Software Project Management



Organization of this Lecture:



- ⌘ Introduction to Project Planning
- ⌘ Software Cost Estimation
 - └ Cost Estimation Models
 - └ Software Size Metrics
 - └ Empirical Estimation
 - └ Heuristic Estimation
 - └ COCOMO
- ⌘ Staffing Level Estimation
- ⌘ Effect of Schedule Compression on Cost
- ⌘ Summary

Introduction



- ⌘ Many software projects fail:
 - ↗ due to faulty project management practices:
 - ☒ It is important to learn different aspects of software project management.

- ⌘ Goal of software project management:
 - ↗ enable a group of engineers to work efficiently towards successful completion of a software project.

Responsibility of project managers



- ⌘ Project proposal writing,
- ⌘ Project cost estimation,
- ⌘ Scheduling,
- ⌘ Project staffing,
- ⌘ Project monitoring and control,
- ⌘ Software configuration management,
- ⌘ Risk management,
- ⌘ Managerial report writing and presentations, etc.

Project Planning Activities

- # Estimating the following attributes of the project:
 - ▣ **Project size:** What will be problem complexity in terms of the effort and time required to develop the product?
 - ▣ **Cost**
 - ▣ **Duration**
 - ▣ **Effort**
- # The effectiveness of the subsequent planning activities is based on the accuracy of these estimations.
 - ▣ **Scheduling** manpower and other resources
 - ▣ **Staff organization** and staffing plans
 - ▣ **Risk identification**, analysis
 - ▣ **Miscellaneous plans** such as quality assurance plan, configuration, management plan, etc.

Project planning

- ⌘ Requires utmost care and attention --- commitments to unrealistic time and resource estimates result in:
 - ↗ irritating delays.
 - ↗ customer dissatisfaction
 - ↗ adverse affect on team morale
 - ☒ poor quality work
 - ↗ project failure.

Sliding Window Planning

- ⌘ **Planning a project over a number of stages** protects managers from making big commitments too early. This technique of staggered planning is known as **Sliding Window Planning**.
- ⌘ In the sliding window technique, starting with an initial plan, the project is **planned more accurately in successive development stages**.

Organization of SPMP Document

⌘ After planning is complete:

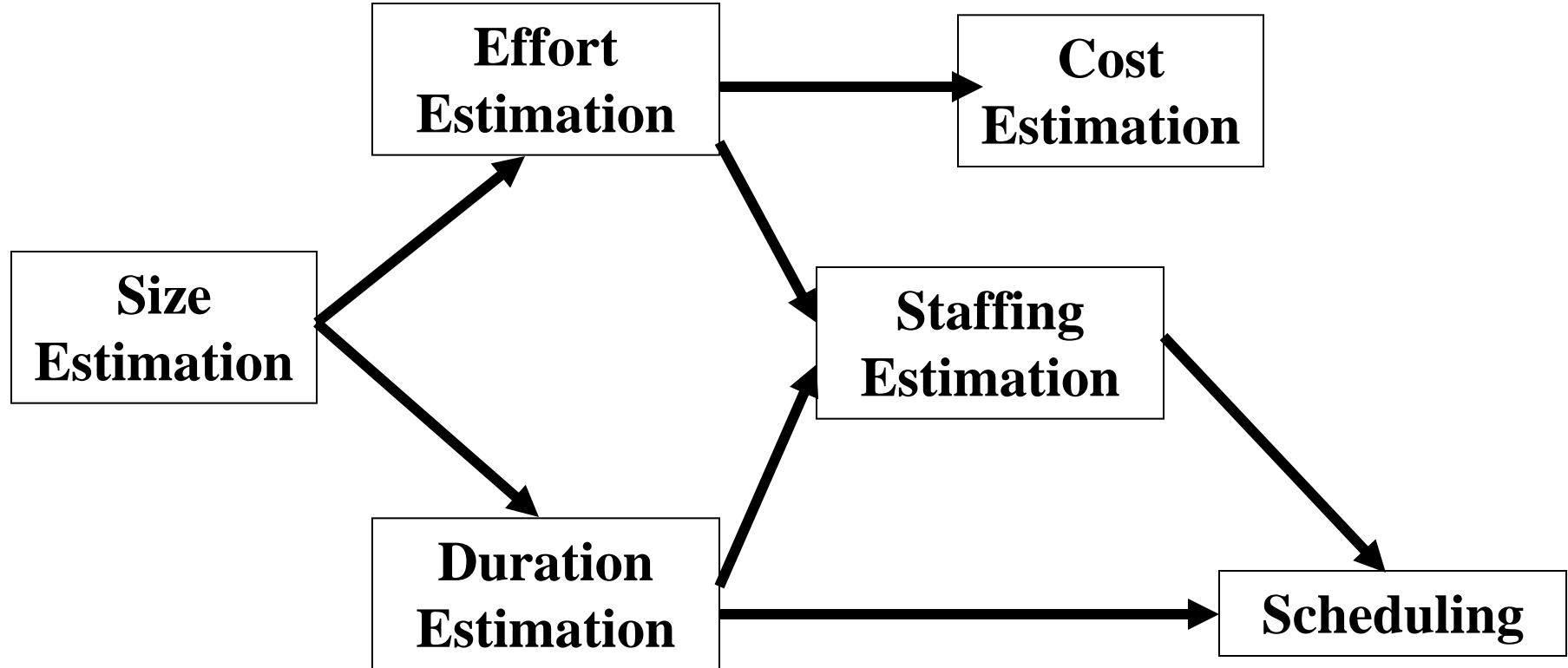
- └ Document the plans in a **Software Project Management Plan(SPMP) document.**
 - └ **Introduction** (Objectives, Major Functions,)
 - └ **Project Estimates** (Historical Data, Estimation Techniques, Effort, Cost,)
 - └ **Project Resources Plan** (People, Hardware and Software,)
 - └ **Schedules** (Work Breakdown Structure, Gantt Chart Represent...)
 - └ **Risk Management Plan** (Risk Analysis, Risk Identification,)
 - └ **Project Tracking and Control Plan**
 - └ **Miscellaneous Plans**(Process Tailoring, Quality Assurance)

Software Project Size Estimation

- ⌘ The project size is a measure of the problem complexity in terms of the **effort and time** required to develop the product.

- ⌘ Two metrics widely used to estimate size:
 - ↗ lines of code (LOC)
 - ↗ function point (FP).

Software Project Size Estimation



Software Size Metrics

⌘ LOC (Lines of Code):

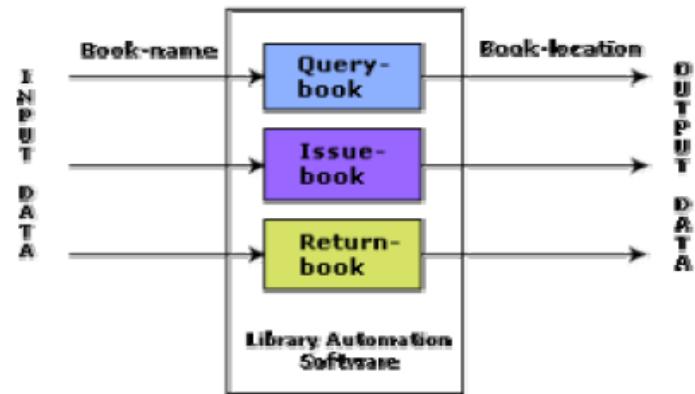
- ▣ Using this metric, the project size is estimated by **counting** the **number of source instructions** in the developed program.
- ▣ While counting the number of source instructions, lines used for commenting the code and the header lines should be ignored.
- ▣ Accurate estimation of the LOC count at the beginning of a project is very difficult.
- ▣ To estimate project managers usually divide the problem into modules, submodules until the sizes of the different leaf-level modules can be approximately predicted.

Disadvantages of Using LOC

- # Size can vary with coding style.
- # Focuses on coding activity alone.
- # Correlates poorly with quality and efficiency of code.
- # Penalizes higher level programming languages, code reuse, etc.

Function Point Metric

- ⌘ Estimate the size of a software product directly from the problem specification.
- ⌘ Size of a software product is directly dependent on the number of different functions or features it supports.
- ⌘ For example, the **issue book feature** of a Library Automation Software takes the **name of the book** as **input** and **displays** its **location and the number of copies available**.
- ⌘ In addition to the number of basic functions that a software performs, **the size is also dependent on the number of files and the number of interfaces.**



Function Point Metric

- ⌘ Proposed by Albrecht in early 80's:
 - ↗ **Total Count =** 4 #inputs + 5 #Outputs + 4 #inquiries + 10 #files + 7 #interfaces
- ⌘ **Number of inputs (EI):** Each data item input by the user is counted.
 - ↗ Example: employee pay roll software; the data items **name, age, address, phone number**, etc. are together considered as a **single input**.
- ⌘ **Number of outputs (EO) :** The outputs considered refer to **reports printed, screen outputs, error messages etc.**
- ⌘ **Number of inquiries (EQ) :** user commands which require specific action by the system. Ex: **print account balance, print student grades etc.**
- ⌘ **Number of files (ILF) :** Each logical file is counted. Ex. File for storing **customer details**, file for **daily purchase record** in a supermarket.
- ⌘ **Number of interfaces (EIF) :** Used to exchange information with other systems. Examples of such interfaces are data files on disks, communication links with other systems etc.

1. FPs of an application is found out by counting the number and types of functions used in the applications. Various functions used in an application can be put under five types as shown in Table 2.1:

Table 2.1 Types of FP Attributes

<i>Measurement Parameter</i>	<i>Examples</i>
1. Number of external inputs (EI)	Input screen and tables.
2. Number of external outputs (EO)	Output screens and reports.
3. Number of external inquiries (EQ)	Prompts and interrupts.
4. Number of internal files (ILF)	Databases and directories.
5. Number of external interfaces (EIF)	Shared databases and shared routines.

All these parameters are then individually assessed for complexity.

- FP characterizes the complexity of the software system and hence can be used to depict the project time and the manpower requirement.
- The effort required to develop the project depends on what the software does.
- FP is programming language independent.
- FP method is used for data processing systems, business systems like information systems.
- The 5 parameters mentioned above are also known as information domain characteristics.
- All the above-mentioned parameters are assigned some weights that have been experimentally determined and are shown in Table 2.2.

<i>Measurement Parameter</i>	<i>Count</i>		<i>Weighing factor</i>			
			<i>Simple</i>	<i>Average</i>	<i>Complex</i>	
1. Number of external inputs (EI)	—	*	3	4	6	=
2. Number of external outputs (EO)	—	*	4	5	7	=
3. Number of external inquiries (EQ)	—	*	3	4	6	=
4. Number of internal files (ILF)	—	*	7	10	15	=
5. Number of external interfaces (EIF)	—	*	5	7	10	=
Count-total →						—

Note here that weighing factor will be simple, average or complex for a measurement parameter type.

CAF: Complexity Adjustment Factor

The Function Point (FP) is thus calculated with the following formula

$$\begin{aligned} \text{FP} &= \text{Count-total} * [0.65 + 0.01 * \sum(F_i)] \\ &= \text{Count-total} * \text{CAF} \end{aligned}$$

where Count-total is obtained from the above Table.

$$\text{CAF} = [0.65 + 0.01 * \sum(F_i)]$$

and $\sum(F_i)$ is the sum of all 14 questionnaires and show the complexity adjustment value/factor-CAF (where i ranges from 1 to 14). Usually, a student is provided with the value of $\sum(F_i)$. Also note that $\sum(F_i)$ ranges from 0 to 70, i.e.,

$$0 \leq \sum(F_i) \leq 70$$

and CAF ranges from 0.65 to 1.35 because

- When $\sum(F_i) = 0$ then $\text{CAF} = 0.65$
- When $\sum(F_i) = 70$ then $\text{CAF} = 0.65 + (0.01 * 70) = 0.65 + 0.7 = 1.35$

Relative complexity adjustment factor (RCAF) – form

No	Subject	Grade
1	Requirement for reliable backup and recovery	0 1 2 3 4 5
2	Requirement for data communication	0 1 2 3 4 5
3	Extent of distributed processing	0 1 2 3 4 5
4	Performance requirements	0 1 2 3 4 5
5	Expected operational environment	0 1 2 3 4 5
6	Extent of online data entries	0 1 2 3 4 5
7	Extent of multi-screen or multi-operation online data input	0 1 2 3 4 5
8	Extent of online updating of master files	0 1 2 3 4 5
9	Extent of complex inputs, outputs, online queries and files	0 1 2 3 4 5
10	Extent of complex data processing	0 1 2 3 4 5
11	Extent that currently developed code can be designed for reuse	0 1 2 3 4 5
12	Extent of conversion and installation included in the design	0 1 2 3 4 5
13	Extent of multiple installations in an organization and variety of customer organizations	0 1 2 3 4 5
14	Extent of change and focus on ease of use	0 1 2 3 4 5
	Total = RCAF	

- 0 = No Influence
- 1 = Incidental
- 2 = Moderate
- 3 = Average
- 4 = Significant
- 5 = Essential

VALUE ADJUSTMENT FACTOR (VAF)

AWFAP VRAASIIHAI IVAIAK AAI

Value	Characteristic
0	Not Present, No influence
1	Incidental influence
2	Moderate influence
3	Average influence
4	Significant influence
5	Strong influence throughout

Sl. No	Degree of Influence	Value (0-5)	Comments
1	Data Communications	0	
2	Distributed Data Processing	0	
3	Performance	0	
4	Heavily used configuration	0	
5	Transaction rate	0	
6	Online data entry	0	
7	End-user efficiency	0	
8	Online update	0	
9	Complex processing	0	
10	Reusability	0	
11	Installation ease	0	
12	Operational ease	0	
13	Multiple sites	0	
14	Facilitate change	0	
Total		0	

	Value Adjustment Factor (VAF)	0.65
--	--------------------------------------	-------------

Q. Given the following values, compute function point when all complexity adjustment factor (CAF) and weighting factors are average.

User Input = 50, User Output = 40, User Inquiries = 35, User Files = 6

External Interface = 4

Step-1: As complexity adjustment factor is average (given in question), hence, scale = 3. $F = 14 * 3 = 42$

Step-2: CAF = $0.65 + (0.01 * 42) = 1.07$

Step-3: As weighting factors are also average (given in question) hence we will multiply each individual function point to corresponding values in TABLE.

$$UFP = (50*4) + (40*5) + (35*4) + (6*10) + (4*7) = 628$$

Step-4: Function Point = $628 * 1.07 = 671.96$

Example 4 Compute the function point value for a project with the following information domain characteristics:

- (1) No. of user inputs = 24
- (2) No. of user outputs = 65
- (3) No. of user inquiries = 12
- (4) No. of files = 12
- (5) No. of external interfaces = 4

Various processing complexity factors are: 4, 1, 0, 3, 3, 5, 4, 4, 3, 3, 2, 2, 4, 5.

Compute the function point value for an Complex condition.

Function Type	Estimated Count
EI	24
EO	16
EQ	22
ILF	4
ELF	2

General System Characteristics (GSCs)	Degree of Influence (DI)
1. Data Communications	2
2. Distributed Data Processing	0
3. Performance	5
4. Heavily Used Configuration	5
5. Transaction Rate	2
6. Online Data Entry	4
7. End-User Efficiency	3
8. Online Update	5
9. Complex Processing	4
10. Reusability	5
11. Installation Ease	4
12. Operational Ease	3
13. Multiple Sites	4
14. Facilitate Change	5

Function Point Metric (CONT.)

- # Suffers from a major drawback:
 - ☒ the size of a function is considered to be independent of its complexity.
- # Extend function point metric:
 - ☒ **Feature Point metric:**
 - ☒ considers an extra parameter:
 - ☒ **Algorithm Complexity:** Greater is the effort required to develop it and therefore its size should be larger compared to simpler functions.
- # Opponents claim:
 - ☒ it is subjective --- Different people can come up with different estimates for the same problem.
- # Proponents claim:
 - ☒ FP is language independent. Size can be easily derived from problem description

Software Project Size/Cost Estimation

Empirical techniques:

 an educated guess based on past experience.

Heuristic techniques:

 assume that the characteristics to be estimated can be expressed in terms of some mathematical expression.

Analytical techniques:

 derive the required results starting from certain simple assumptions.

Empirical Size Estimation Techniques

- ⌘ **Expert Judgement:** Estimates the cost of the different components (i.e. modules or subsystems) of the system and then combines them to arrive at the overall estimate. **Suffers from individual bias.**

- ☒ Experts divide a software product into component units:
 - ☒ e.g. GUI, database module, billing module, etc.
 - ☒ Add up the guesses for each of the components.

- ⌘ **Delphi Estimation: Team of Experts and a coordinator.**

- ⌘ Experts carry out estimation independently:
 - ☒ mention the unusual characteristic of the product which has influenced his estimation.
 - ☒ coordinator notes down any extraordinary rationale:
 - ☒ circulates among experts. Experts re-estimate.
- ⌘ Experts never meet each other to discuss their viewpoints as many estimators may easily get Influenced.

Heuristic Estimation Techniques

⌘ Single Variable Model:

⌘ Models provide a means to estimate the desired characteristics of a problem, **using some previously estimated basic (independent) characteristic** of the software product such as its size, staff etc.

☒ Parameter to be Estimated = $C_1(\text{Estimated Characteristic})d_1$

⌘ Multivariable Model:

☒ Assumes that the parameter to be estimated depends on more than one characteristic.

☒ Parameter to be Estimated = $C_1(\text{Estimated Characteristic})d_1 + C_2(\text{Estimated Characteristic})d_2 + \dots$

☒ Usually more accurate than single variable models.

COCOMO Model

- ⌘ COCOMO (COnstructive COst MOdel) is a Constructive Cost Estimation Model proposed by Boehm.
- ⌘ COCOMO Product classes correspond to:
 - ▣ **application**, **utility** and **system** programs respectively.
 - ▣ Data processing and scientific programs are considered to be **application programs**.
 - ▣ Compilers, linkers, editors, etc., are **utility programs**.
 - ▣ Operating systems and real-time system programs, etc. are **system programs**.

Divides software product developments into 3 categories:

- ⌘ **Organic**: Relatively small groups. Working to develop well-understood applications. **Examples** of this type of projects are simple business systems, simple inventory management systems, and data processing systems.
- ⌘ **Semidetached**: Project team consists of a mixture of experienced and inexperienced staff. **Example** of Semidetached system includes developing a new operating system (OS), a Database Management System (DBMS), and complex inventory management system.
- ⌘ **Embedded**: The software is strongly coupled to complex hardware, or real-time systems. **For Example**: ATM, Air Traffic control.

The Modes



- Organic
 - 2-50 KLOC, small, stable, little innovation
- Semi-detached
 - 50-300 KLOC, medium-sized, average abilities, medium time-constraints
- Embedded
 - > 300 KLOC, large project team, complex, innovative, severe constraints

COCOMO Model (CONT.)

⌘ For each of the three product categories:

- ☒ From size estimation (in KLOC), Boehm provides equations to predict:

- ☒ project duration in months
 - ☒ effort in programmer-months

⌘ Gives only an approximate estimation:

- ☒ **Effort = $a_1 \text{ (KLOC)}^{a_2}$**
- ☒ **Tdev = $b_1 \text{ (Effort)}^{b_2}$**
 - ☒ KLOC is the estimated kilo lines of source code,
 - ☒ a_1, a_2, b_1, b_2 are constants for different categories of software products,
 - ☒ Tdev is the estimated time to develop the software in months,
 - ☒ Effort estimation is obtained in terms of person months (PMs).

Development Effort Estimation

- # Organic : **Effort** = 2.4 (KLOC)1.05 PM
- # Semi-detached: **Effort** = 3.0(KLOC)1.12 PM
- # Embedded: **Effort** = 3.6 (KLOC)1.20PM

Development Time Estimation

- # Organic: **Tdev** = 2.5 (Effort)0.38 Months
 - # Semi-detached: **Tdev** = 2.5 (Effort)0.35 Months
 - # Embedded: **Tdev** = 2.5 (Effort)0.32 Months
-
- # **Software cost estimation is done through three stages:**
 - ▣ **Basic COCOMO,**
 - ▣ **Intermediate COCOMO,**
 - ▣ **Complete COCOMO.**

Average Staff Size

$$SS = \frac{E}{TDEV} = \frac{[\text{staff} - \cancel{\text{months}}]}{\cancel{[\text{months}]}} = [\text{staff}]$$



Productivity

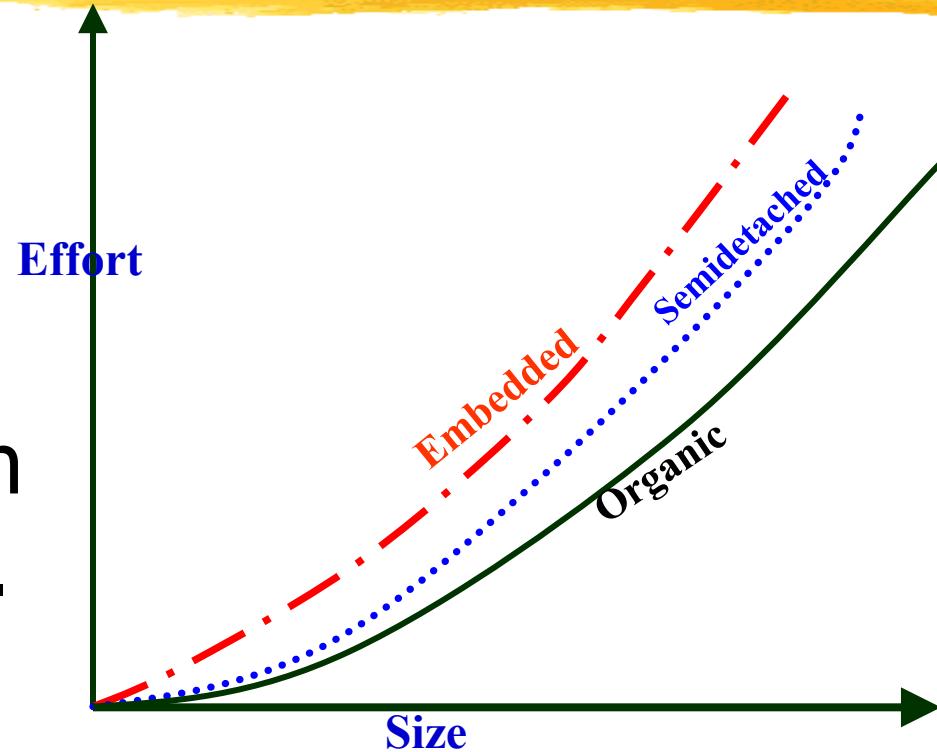
$$P = \frac{Size}{E} = \frac{[\text{KLOC}]}{[\text{staff} - \text{months}]} = \text{KLOC} / \text{staff} - \text{month}$$

Suppose an organic project has 7.5 KLOC,

- Suppose an organic project has 7.5 KLOC,
 - Effort $2.4(7.5)^{1.05} = 20$ staff-months
 - Development time $2.5(20)^{0.38} = 8$ months
 - Average staff $20 / 8 = 2.5$ staff
 - Productivity $7,500 \text{ LOC} / 20 \text{ staff-months} = 375 \text{ LOC} / \text{staff-month}$

Basic COCOMO Model (CONT.)

⌘ Effort is somewhat super-linear in problem size.



Basic COCOMO Model (CONT.)

⌘ Development time

- ⌘ sublinear function of product size.

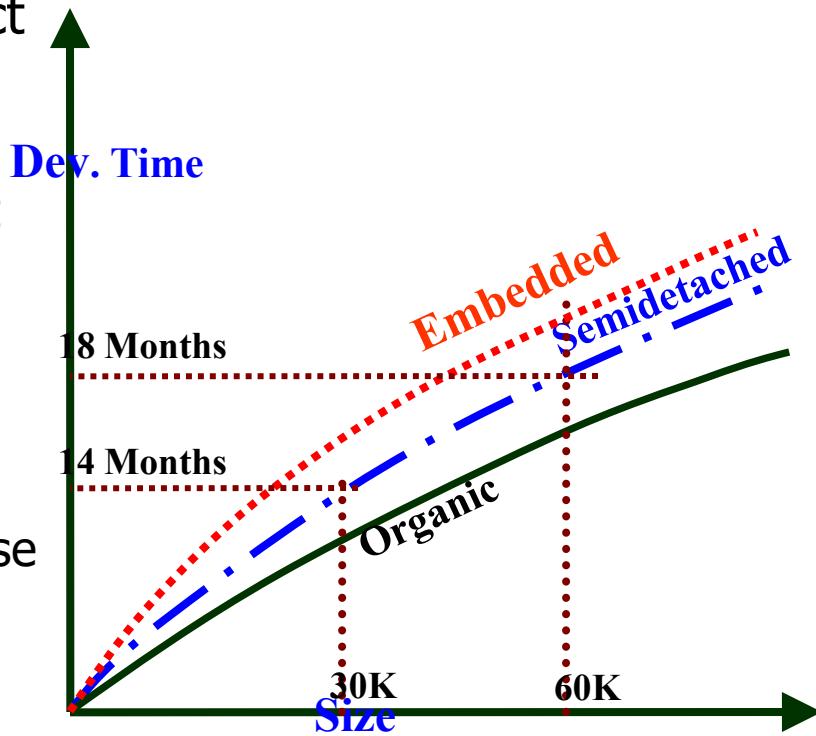
⌘ When product size increases two times, development time does not double.

⌘ Time taken:

- ⌘ almost same for all the three product categories.

⌘ Development time does not increase linearly with product size:

- ⌘ For larger products more parallel activities can be identified and performed by engineers



Example



- ⌘ The size of an organic software product has been estimated to be 32,000 lines of source code. Assume that the average salary of software engineers be Rs. 15,000/- per month. Determine the **effort** required to develop the software product and the **nominal development time**.
- 

Example



⌘ The size of an organic software product has been estimated to be 32,000 lines of source code. Assume that the average salary of software engineers be Rs. 15,000/- per month. Determine the **effort** required to develop the software product and the **nominal development time**.

⌘ Effort = $2.4 * (32)1.05 = 91 \text{ PM}$

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

⌘ Cost required to develop the product = **14 x 15,000**
= Rs. 210,000/-

Intermediate COCOMO

- # Basic COCOMO model assumes
 - ☒ effort and development time depend on product size alone.
- # However, several parameters affect effort and development time:
 - ☒ Reliability requirements
 - ☒ Availability of CASE tools and modern facilities to the developers
 - ☒ Size of data to be handled
- # For accurate estimation,
 - ☒ the effect of all relevant parameters must be considered:
 - ☒ Intermediate COCOMO model recognizes this fact:
 - ☒ refines the initial estimate obtained by the basic COCOMO by using a set of 15 cost drivers (multipliers).
 - ☒ Rate different parameters on a scale of **one to three** and multiply cost driver values with the estimate obtained using the basic COCOMO.
 - ☒ These 15 values are then multiplied to calculate the EAF (**Effort Adjustment Factor**).

Effort Adjustment Factor

- ⌘ The Effort Adjustment Factor in the effort equation is simply the product of the effort multipliers corresponding to each of the cost drivers for your project.
- ⌘ For example, if your project is rated Very High for Complexity (effort multiplier of 1.34), and Low for Language & Tools Experience (effort multiplier of 1.09), and all of the other cost drivers are rated to be Nominal (effort multiplier of 1.00), the EAF is the product of 1.34 and 1.09.

$$\text{Effort Adjustment Factor} = \text{EAF} = 1.34 * 1.09 = 1.46$$

⌘ Intermediate COCOMO equation:

$$E = a_i (KLOC) b_i * EAF$$

$$D = c_i (E) d_i$$

Coefficients for intermediate COCOMO

Project	a_i	b_i	c_i	d_i
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Intermediate COCOMO (CONT.)

⌘ Cost driver classes:

- └ **Product**: characteristics of the product that include inherent complexity of the product, reliability requirements of the product, etc.
- └ **Computer**: Execution time, storage requirements, etc.
- └ **Personnel**: Experience of personnel, programming capability, analysis capability, etc.
- └ **Development Environment**: Sophistication of the tools used for software development.

Complete COCOMO

- ⌘ Both models:
 - ℳ consider a software product as a single homogeneous entity:
 - ℳ Most large systems are made up of several smaller sub-systems.
 - ℳ Some sub-systems may be considered as organic type, some may be considered embedded, etc.
- ⌘ Cost of each sub-system is estimated separately.
- ⌘ Costs of the sub-systems are added to obtain total cost.
- ⌘ Reduces the margin of error in the final estimate.
- ℳ **Example:** A Management Information System (MIS) for an organization having offices at several places across the country:
 - ℳ Database part (**semi-detached**)
 - ℳ Graphical User Interface (GUI) part (**organic**)
 - ℳ Communication part (**embedded**)
- ℳ Costs of the components are estimated separately:
 - ℳ summed up to give the overall cost of the system.

Analytical Estimation Techniques

⌘ **Halstead's software** analytical technique to estimate:

- └ size,
- └ development effort,
- └ development time.

⌘ **Halstead used a few primitive program parameters**

- └ number of operators and operands

⌘ Derived expressions for:

- └ over all program length,
- └ potential minimum volume
- └ actual volume,
- └ language level,
- └ effort, and
- └ development time.

Staffing Level Estimation

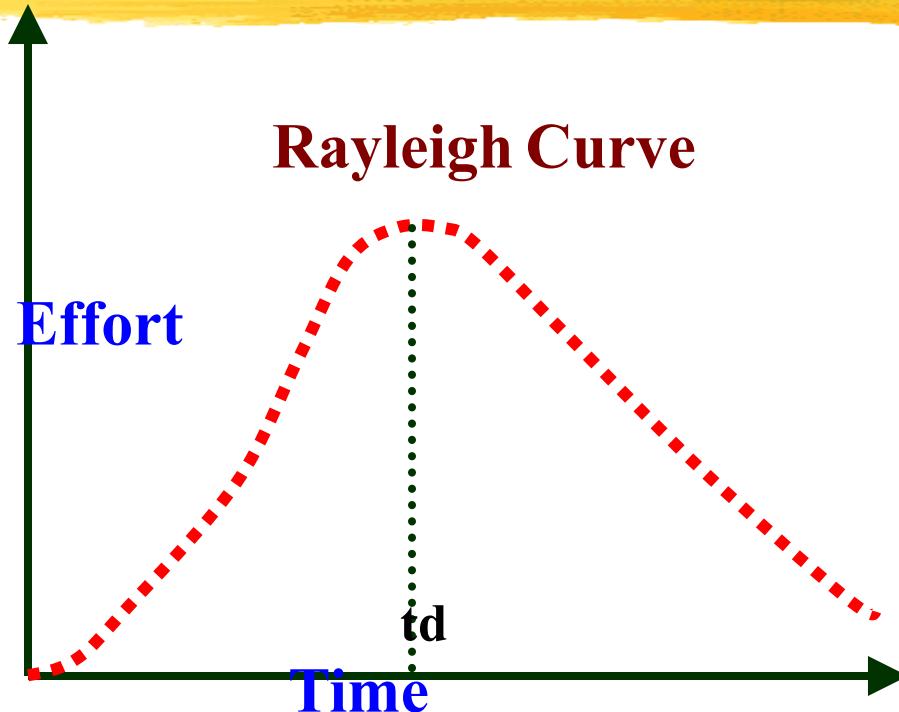
- ⌘ Once the **effort** required to develop a software has been determined, it is necessary to determine the **staffing requirement for the project**.
- ⌘ Norden in 1958 analyzed many R&D projects, and observed:
 - ▣ **Rayleigh curve** represents the number of full-time personnel required at any time.
- ⌘ Very small number of engineers are needed at the beginning of a project to carry out planning and specification.
- ⌘ As the project progresses:
 - ▣ more detailed work is required for which number of engineers slowly increases and reaches a peak.
 - ▣ This is the time at which the Rayleigh curve reaches its **maximum value** corresponds to system testing and product release.
 - ▣ After system testing,
 - ▣ the number of project staff falls till product installation and delivery.

Rayleigh Curve

⌘ Rayleigh curve is specified by two parameters:

- ─ td the time at which the curve reaches its maximum
- ─ K the total area under the curve.

⌘ $L = f(K, td)$



Project Scheduling

- ⌘ It involves deciding which tasks would be taken up when. In order to schedule the project activities, a software project manager needs to do the following task:
 - ↗ Identify all the tasks needed to complete the project.
 - ↗ Break down large tasks into small activities.
 - ↗ Determine the dependency among different activities.
 - ↗ Establish the most likely estimates for the time durations necessary to complete the activities.
 - ↗ Plan the starting and ending dates for various activities.
 - ↗ Determine the critical path. A critical path is the chain of activities that determines the duration of the project.

Work breakdown structure

- # Decompose a given task set recursively into small activities.
- # Root of the tree is labelled by the problem name.
- # Each node of the tree is broken down into smaller activities that are made the children of the node.
- # Each activity is recursively decomposed into smaller sub-activities until at the leaf level.

Activity networks and critical path method

- ⌘ Activity network shows the different activities making up a project, their estimated durations, and interdependencies.
- ⌘ Each activity is represented by a rectangular node and the duration of the activity is shown alongside each task.

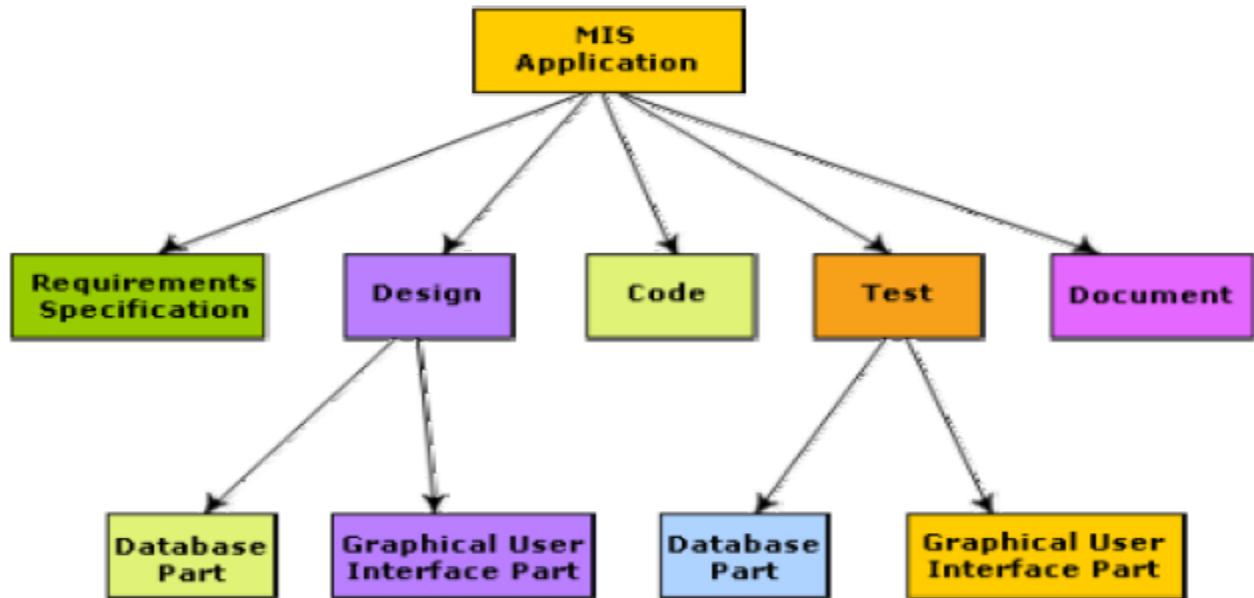


Fig. 11.7: Work breakdown structure of an MIS problem

Critical Path Method (CPM)

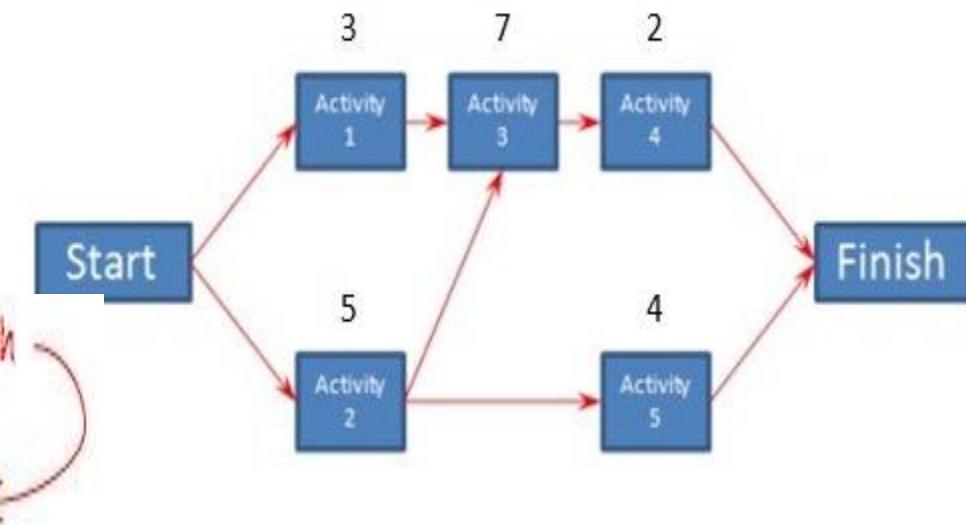
- The **Critical Path Method** (CPM) can help you keep your projects on track. *Critical path schedules...*
- Identify the activities that must be completed on time in order to complete the whole project on time.
- Show you which tasks can be delayed and for how long without impacting the overall project schedule.
- Calculate the minimum amount of time it will take to complete the project.
- The CPM has four key elements...

- Critical Path Analysis
- Float Determination
- Early Start & Early Finish Calculation
- Late Start & Late Finish Calculation

Start → Activity 1 → Activity 3 → Activity 4 → Finish $3 + 7 + 2 = 12$

Start → Activity 2 → Activity 3 → Activity 4 → Finish $5 + 7 + 2 = 14$

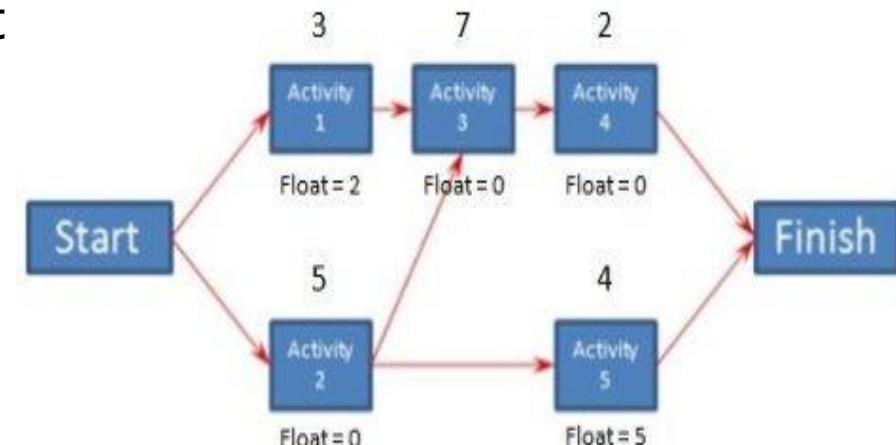
Start → Activity 2 → Activity 5 → Finish $5 + 4 = 9$



Float Determination/Slack Time

- ⌘ **Float (slack time) is the amount of time an activity can slip before it causes your project to be delayed.**
- ⌘ Activities 2, 3, and 4 are on the critical path so they have a float of zero.
- ⌘ The next longest path is Activities 1, 3, and 4. Since Activities 3 and 4 are also on the critical path, their float will remain as zero.
- ⌘ For any remaining activities, in this case Activity 1, the float will be the **duration of the critical path minus the duration of this path.** $14 - 12 = 2$. So Activity 1 has a float

- ⌘ **Next longest path is Activities 2 and 5.**
Activity 2 is on the critical path so it will have a float of zero. Activity 5 has a float of $14 - 9$, which is 5.



⌘ Early Start & Early Finish Calculation

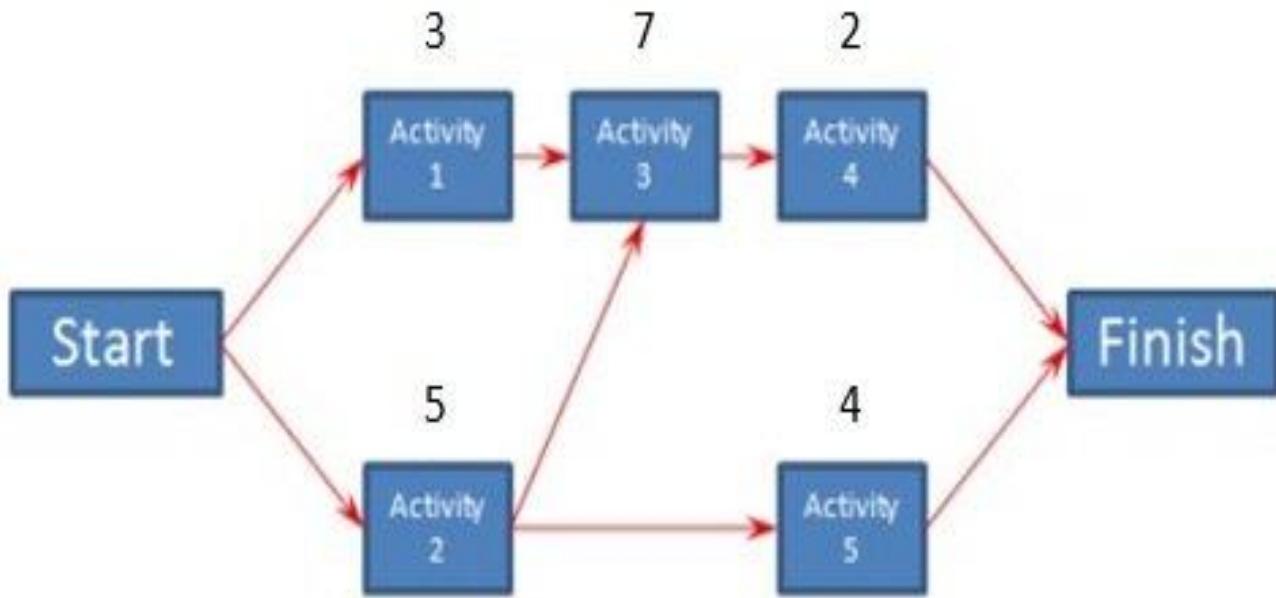
- ▣ The Critical Path Method includes a technique called the **Forward Pass** which is used to determine the earliest date an activity can start and the earliest date it can finish.
- ▣ These dates are valid as long as all prior activities in that path started on their earliest start date and didn't slip.

- ⌘ Earliest start time (ES) - The earliest time an activity can start once the previous dependent activities are over.
- ⌘ Earliest finish time (EF) - ES + activity duration.

⌘ Late Start & Late Finish Calculation

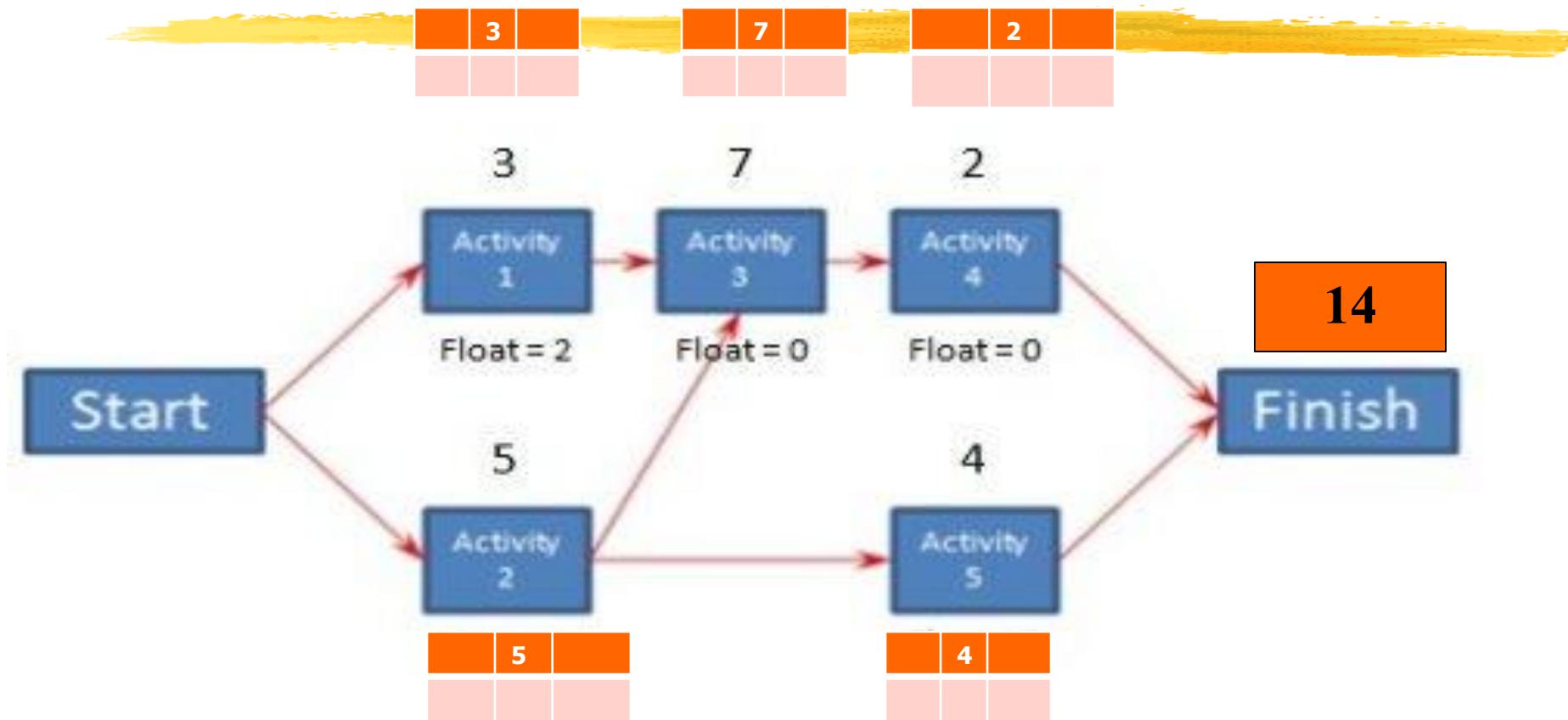
- ▣ The **Backward Pass** is a Critical Path Method technique you can use to determine the latest date an activity can start and the latest date it can finish before it delays the project.
 - ▣ You'll start once again with the critical path, but this time it will begin from the last activity in the path.
-
- ⌘ Latest finish time (LF) - The latest time an activity can finish without delaying the project.
 - ⌘ Latest start time (LS) - LF - activity duration.

ES	Duaration	EF
LS		LF

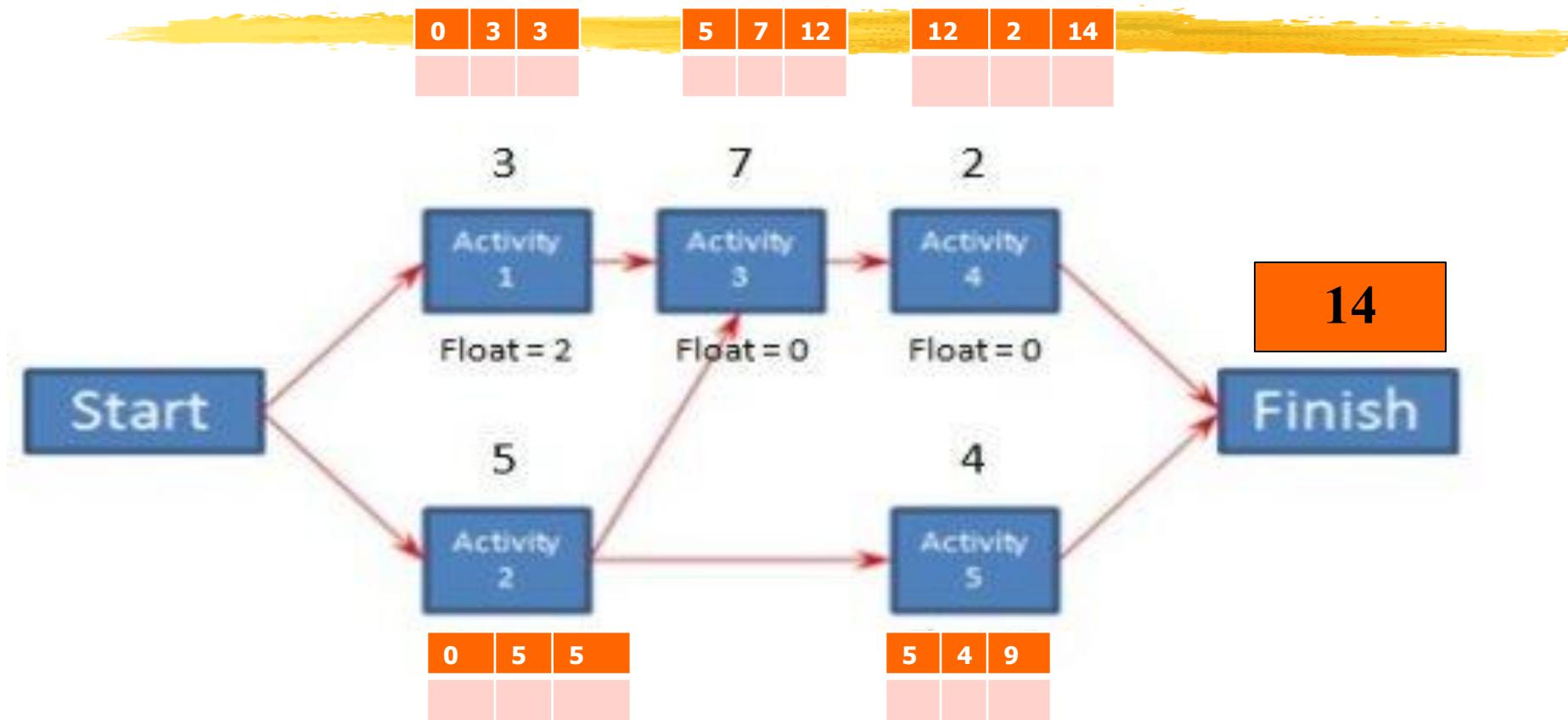


- * Determine the dependency among different activities.

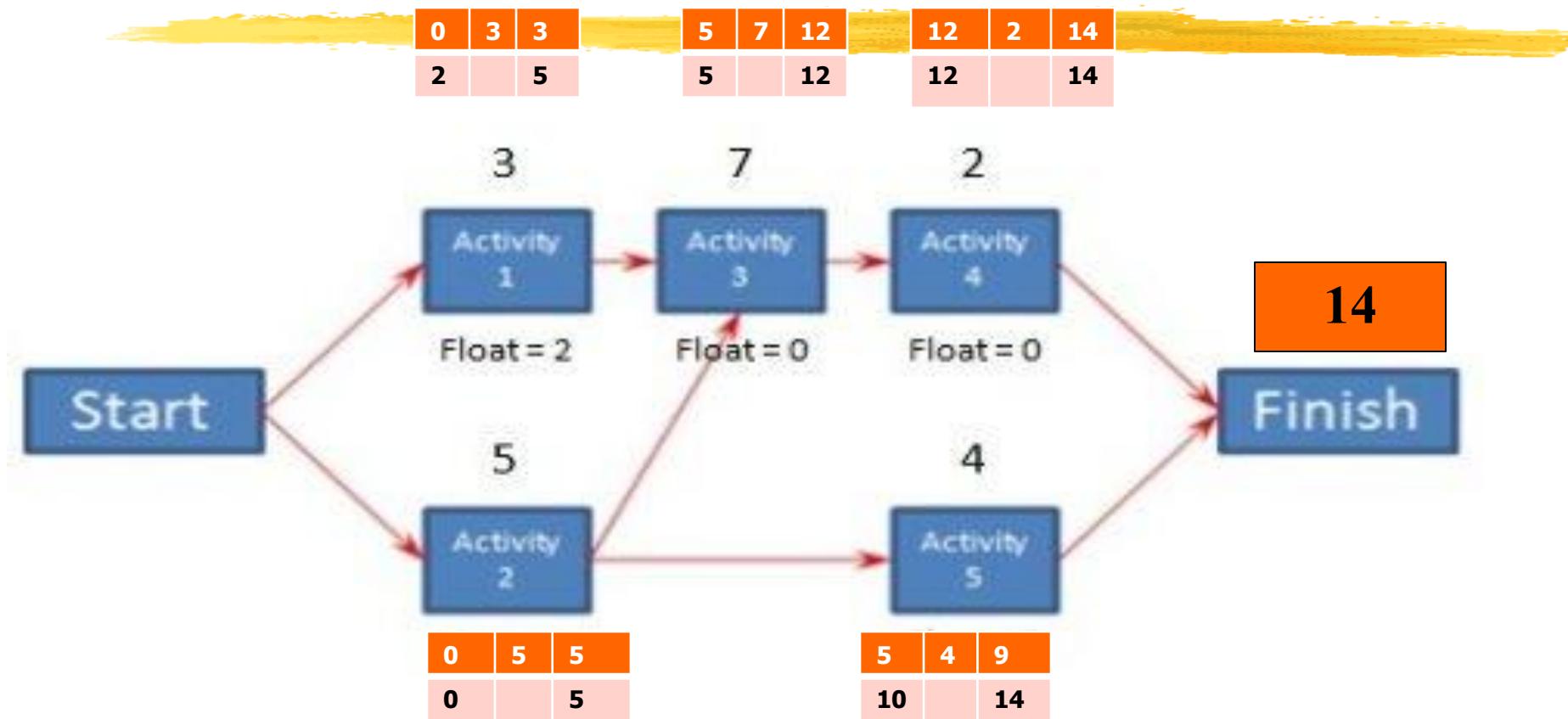
ES	Duaration	EF
LS		LF



ES	Duaration	EF
LS		LF



ES	Duaration	EF
LS		LF



Using the table below, draw the network diagram and answer the questions. When you have completed answering the questions, do a forward/backward pass.

Activity	Predecessor	Estimate in weeks
A	-	5
E	A	7
C	A	3
D	E	1
B	E, C	3
F	D, B	2

1. How many paths are in the network, and what are they?

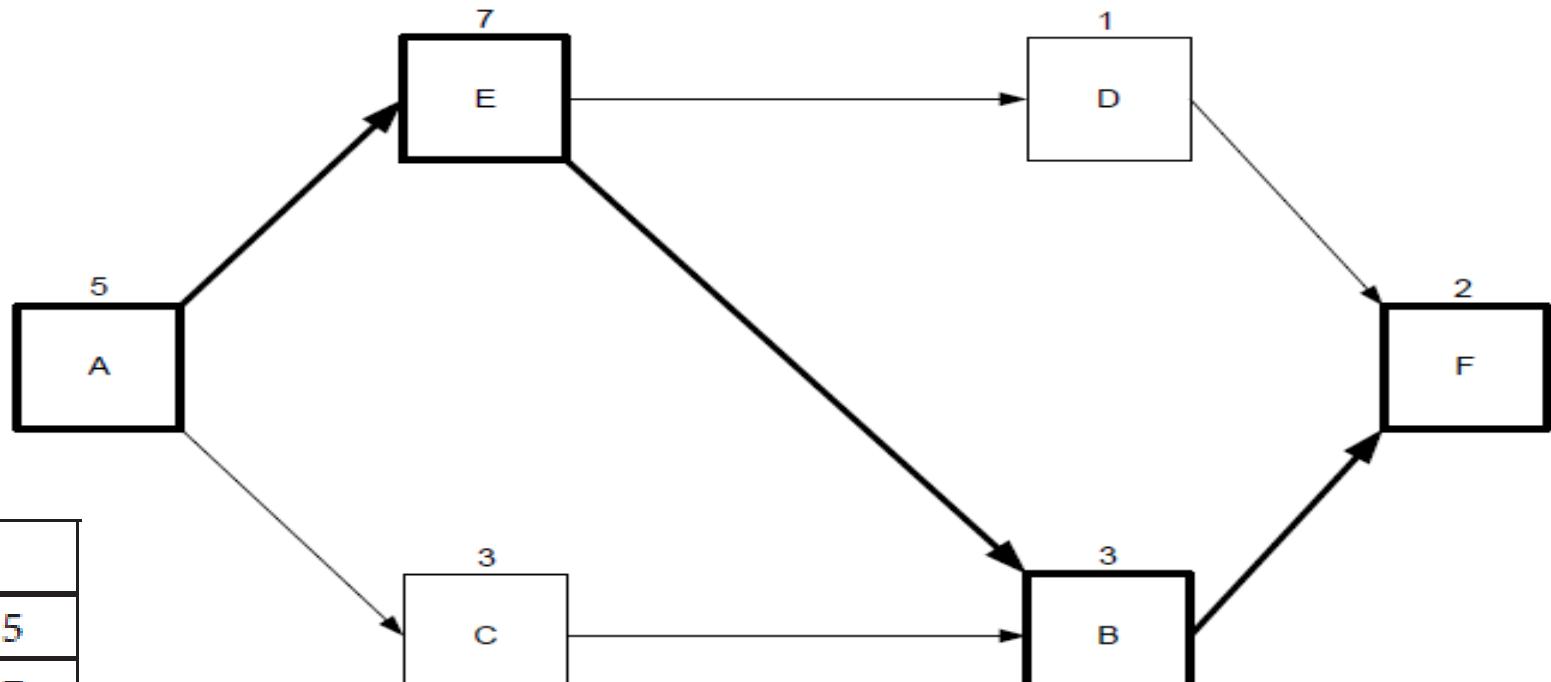
2. What is the critical path and its duration?

3. What is the float on activity B?

4. What is the impact to the project if activity C takes three weeks longer than planned?

Using the table below, draw the network diagram and answer the questions. When you have completed answering the questions, do a forward/backward pass.

Activity	Predecessor	Estimate in weeks
A	-	5
E	A	7
C	A	3
D	E	1
B	E, C	3
F	D, B	2



Paths	
A, E, D, F	15
A, E, B, F	17
A, C, B, F	13

- 
1. How many paths are in the network, and what are they?

There are three paths. See the details on the next page.

2. What is the critical path and its duration?

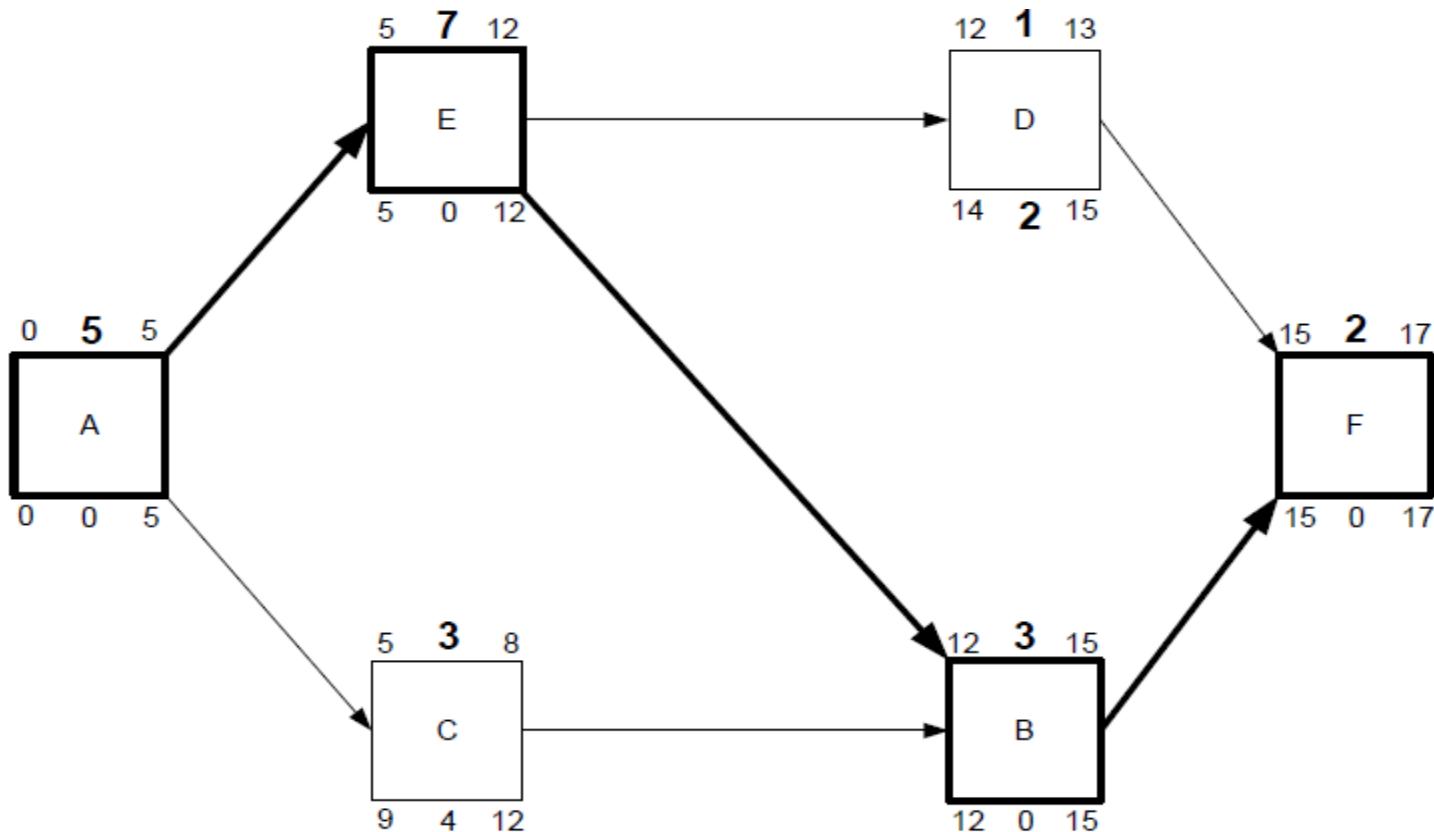
A, E, B, F *17 weeks*

3. What is the float on activity B?

Zero. It is on the critical path.

4. What is the impact to the project if activity C takes three weeks longer than planned?

There will be no impact to the critical path or its duration.



Legend			
ES	D	EF	
ID			
LS	F	LF	

ID = Activity name
D = Duration
F = Float
ES = Early start
EF = Early finish
LS = Late start
LF = Late finish

- 
- ⌘ There are two methods to calculate the float.
 - ▣ In the first, you subtract the duration of the non-critical path from the critical path.
 - ▣ In the second method, you find the total float for any activity by subtracting the Early Start date from the Late Start date ($LS - ES$) or
 - ▣ subtracting the Early Finish date from the Late Finish date ($LF - EF$) on any activity.

Activity Network

Activity	Predecessor	Duration
A	-	5
B	A	4
C	A	5
D	B	6
E	C	3
F	D,E	4

	Activity	
	Duaration	

Activity Network

Activity	Predecessor	Duration
A	-	7
B	-	9
C	A	12
D	A, B	8
E	D	9
F	C,E	6
G	E	5

	Activity	
	Duaration	

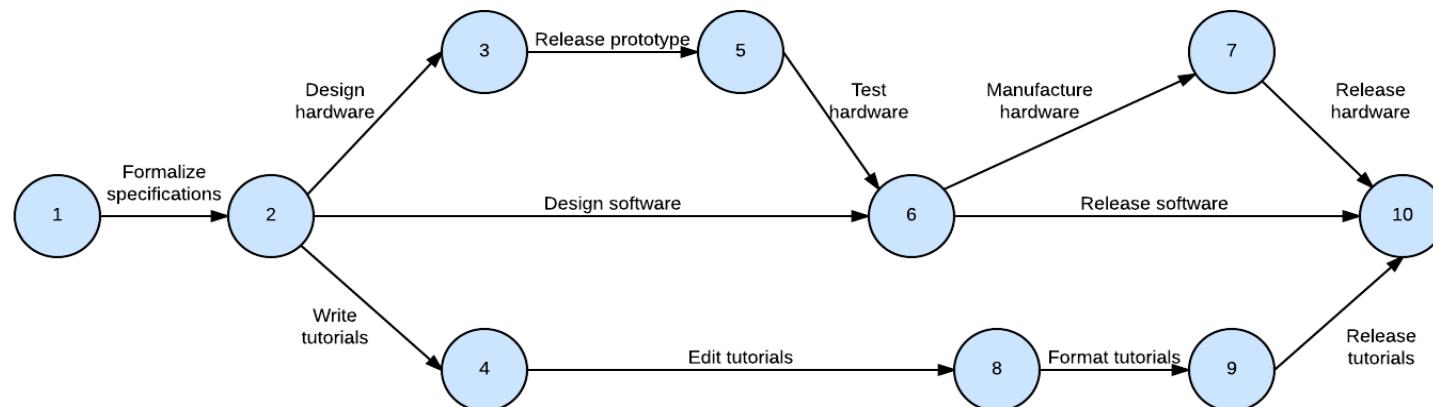
Gantt chart

- ⌘ Gantt charts are mainly used to allocate resources to activities.
- ⌘ The resources allocated to activities include staff, hardware, and software. a special type of bar chart where each bar represents an activity.
- ⌘ The bars are drawn along a time line. The length of each bar is proportional to the duration of time planned for the corresponding activity.

Task Name	Q1 2019			Q2 2019		Q3 2019	
	Jan 19	Feb 19	Mar 19	Apr 19	Jun 19	Jul 19	
Planning							
Research							
Design							
Implementation							
Follow up							

PERT chart

- ⌘ PERT (Project Evaluation and Review Technique) charts consist of a network of boxes and arrows. The boxes represent activities and the arrows represent task dependencies. A critical path in a PERT chart is shown by using thicker arrows.
- ⌘ PERT method can help to analyze the tasks involved in completing a project identifying the minimum time needed to complete the total project, so in fact it combines the Critical Path method.
- ⌘ Gantt chart representation of a project schedule is helpful in planning the utilization of resources, while PERT chart is useful for monitoring the timely progress of activities.



Gantt Chart	Pert Chart
Represented with Bar Chart	Represented with Flowchart (or Network Diagram)
Used for Small Projects	Used for Large and Complex Projects
Provide Accurate Time Duration and Percent Complete	Need to Predict the Time
Cannot Display Interconnecting Tasks that Depend on Each Other	Has Numerous Interconnecting Networks of Independent Tasks

Risk management

Three main categories of risks:

- ⌘ **Project risks:** risks concern various forms of budgetary, schedule, personnel, resource, and customer-related problems. An important project risk is schedule slippage.
- ⌘ **Technical risks:** risks concern potential design, implementation, interfacing, testing, and maintenance problems. Most technical risks occur due to the development team's insufficient knowledge about the project.
- ⌘ **Business risks:** risks include risks of building an excellent product that no one wants, losing budgetary or personnel commitments, etc.
- ⌘ **Risk assessment:** to rank the risks in terms of their damage causing potential.
$$p = r * s \quad \text{where}$$

p is the priority with which the risk must be handled,
r is the probability of the risk becoming true,
s is the severity of damage caused due to the risk becoming true.

Risk containment

- # Risks of a project are assessed, plans must be made to contain the most damaging and the most likely risks.
- # Different risks require different containment procedures. Three main strategies to plan for risk containment:
 - ▣ **Avoid the risk:** discussing with the customer to change the requirements to reduce the scope of the work, giving incentives to the engineers to avoid the risk of manpower turnover, etc.
 - ▣ **Transfer the risk:** Involves getting the risky component developed by a third party, buying insurance cover, etc.
 - ▣ **Risk reduction:** Planning ways to contain the damage due to a risk. For example, if there is risk that some key personnel might leave, new recruitment may be planned.

Risk leverage

- # Different strategies must be considered for handling a risk, the project manager must consider the cost of handling the risk and the corresponding reduction of risk.

Risk leverage = (risk exposure before reduction – risk exposure after reduction) / (cost of reduction)

Software Configuration Management

- ⌘ The deliverables of a SW product consist of a number of objects, e.g. source code, design document, SRS document, test document, user's manual, etc.
 - ⌘ These objects are modified by many software engineers through out development cycle.
 - ⌘ The state of each object changes as bugs are detected and fixed during development .
-
- ⌘ **The configuration of the software is the state of all project deliverables at any point of time.**
 - ⌘ **SCM deals with effectively tracking and controlling the configuration of a software during its life cycle.**

Release vs. Version vs. Revision of SW

- ⌘ A new **version** results from a significant change in functionality, technology, or the platform
 - ▣ Example: one version of a SW might be Unix-based, another Windows based.
- ⌘ **revision** refers to minor bug fix .
- ⌘ A **release** results from bug fix, minor enhancements to the functionality, usability
- ⌘ A new release of SW is an improved system replacing the old one.
- ⌘ Systems are described as Version m, Release n; or simple m.n.

Necessity of SCM

- ⌘ To control access to deliverable objects with a view to avoiding the following problems
 - ▣ **Inconsistency problem when the objects are replicated.**
 - ▣ **Problems associated with concurrent access.**
 - ▣ **Providing a stable development environment.**
 - ▣ **System accounting and status information.**
 - ▣ **Handling variants.** If a bug is found in one of the variants, it has to be fixed in all variants

SCM activities

- ▣ **Configuration identification :**
 - ▣ deciding which objects (configuration items) are to be kept track of.
- ▣ **Configuration control :**
 - ▣ ensuring that changes to a system happen without ambiguity.

⌘ Baseline

- ▣ When an effective SCM is in place, the manager freezes the objects to form a **baseline**.
- ▣ A **baseline** is the status of all the objects under configuration control. When any of the objects under configuration control is changed , a new baseline is formed.

Configuration item identification

⌘ Categories of objects:

- ▣ **Controlled objects:** are under Configuration Control (CC) . Formal procedures followed to change them.
- ▣ **Precontrolled objects:** are not yet under CC, but will eventually be under CC.
- ▣ **Uncontrolled objects:** are not subjected to CC
- ▣ **Controllable objects** include both **controlled and precontrolled objects**; examples:

SRS document, Design documents, Source code , Test cases

The SCM plan is written during the project planning phase and it lists all controlled objects.

⌘ Configuration Control (CC):

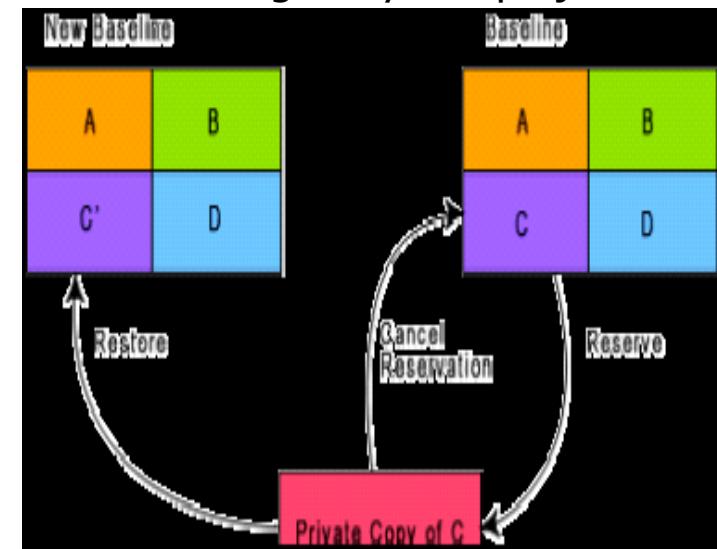
process of managing changes to controlled objects. Allows authorized changes to the controlled objects and prevents unauthorized changes .

Changing the Baseline

- # When one needs to change an object under configuration control, he is provided with a copy of the base line item.
- # The requester makes changes to his private copy.
- # After modifications, updated item replaces the old item and a new base line gets formed .

Reserve and restore operation in configuration control

- # obtains a private copy of the module through a reserve operation.
- # carries out all changes on this private copy.
- # restoring the changed module to the baseline requires the permission of a change control board(CCB).
- # Except for very large projects, the functions of the CCB are discharged by the project manager himself.



Organization Structure

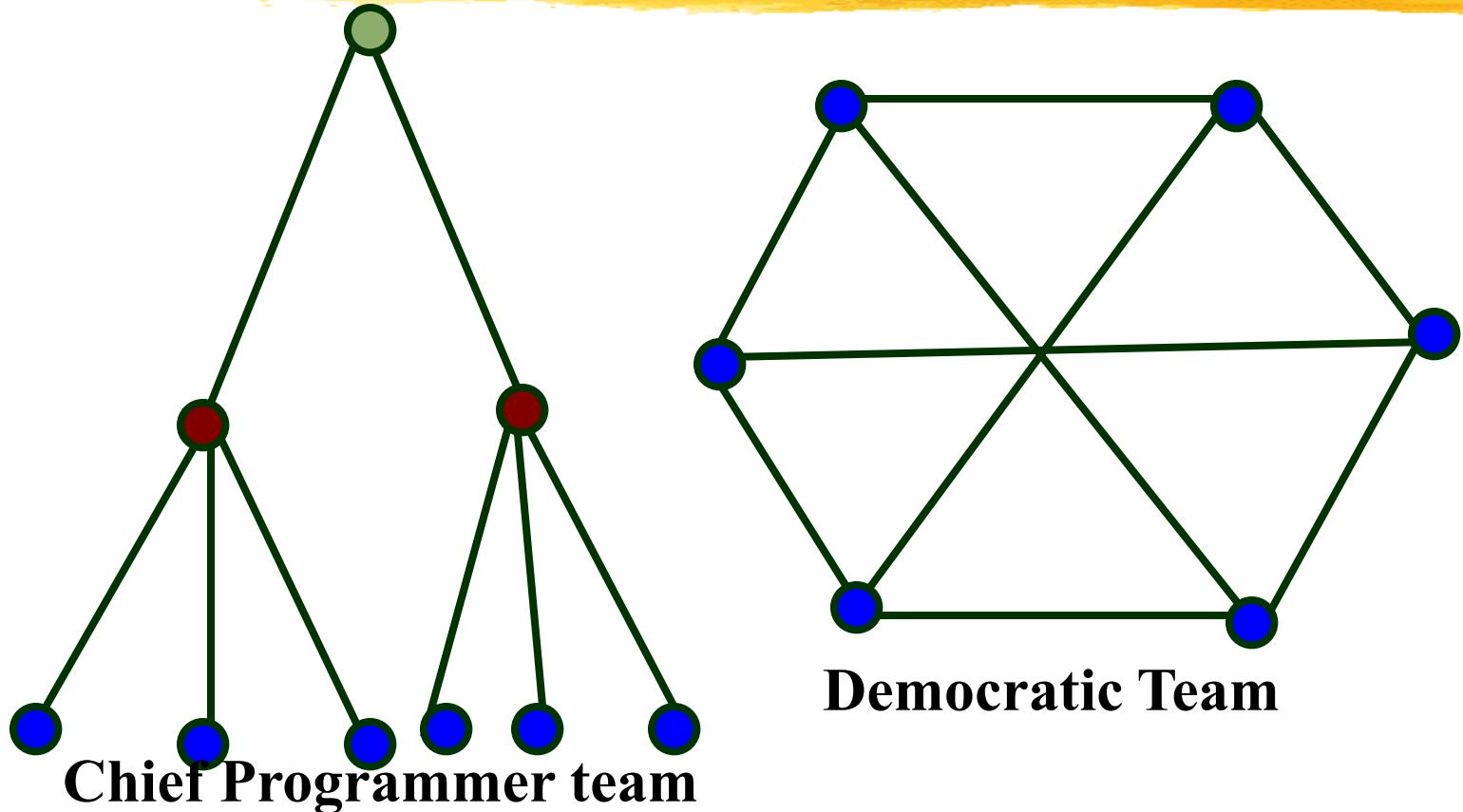
⌘ Functional Organization:

- ↳ Engineers are organized into functional groups, e.g.
 - ☒ specification, design, coding, testing, maintenance, etc.
- ↳ Engineers from functional groups get assigned to different projects

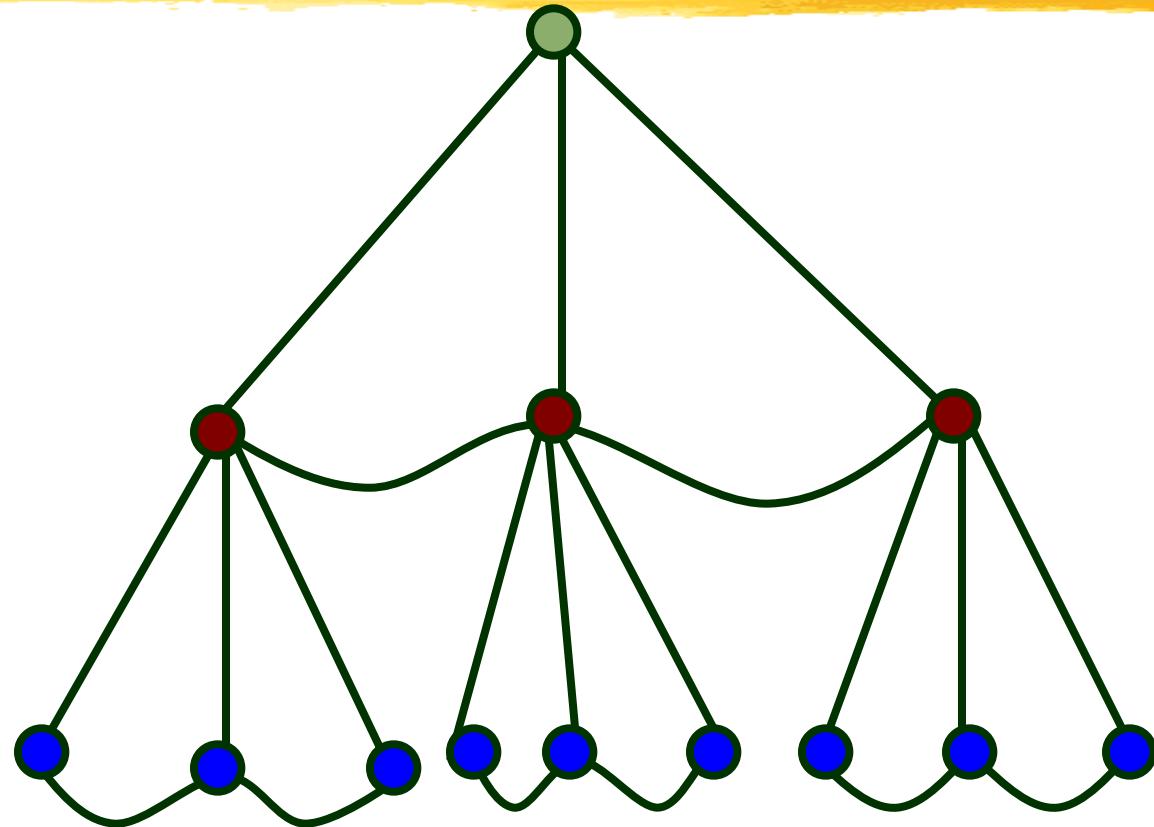
⌘ Problems of different complexities and sizes require different team structures:

- ↳ Chief-programmer team
- ↳ Democratic team
- ↳ Mixed organization

Team Organization



Mixed team organization



Chief Programmer Team

- # A senior engineer provides technical leadership:
 - ▣ partitions the task among the team members.
 - ▣ verifies and integrates the products developed by the members.
- # Works well when
 - ▣ the task is well understood
 - ☒ also within the intellectual grasp of a single individual,
 - ▣ importance of early completion outweighs other factors
 - ☒ team morale, personal development, etc.
- # Chief programmer team is subject to **single point failure**:
 - ▣ too much responsibility and authority is assigned to the chief programmer.

Democratic Teams

- # Suitable for:
 - ☒ small projects requiring less than five or six engineers
 - ☒ research-oriented projects
- # A manager provides administrative leadership:
 - ☒ at different times different members of the group provide technical leadership.
- # Democratic organization provides
 - ☒ higher morale and job satisfaction to the engineers
 - ☒ therefore leads to less employee turnover.
- # Disadvantage:
 - ☒ team members may waste a lot time arguing about trivial points:
 - ☒ absence of any authority in the team.

Mixed Control Team Organization



- # Draws upon ideas from both:
 - ↗ democratic organization and
 - ↗ chief-programmer team organization.
- # Communication is limited
 - ↗ to a small group that is most likely to benefit from it.
- # Suitable for large organizations.



Software Design



Organization of this Lecture



- ~ Brief review of previous lectures
- ~ Introduction to software design
- ~ Goodness of a design
- ~ Functional Independence
- ~ Cohesion and Coupling
- ~ Function-oriented design vs. Object-oriented design
- ~ Summary

Software design

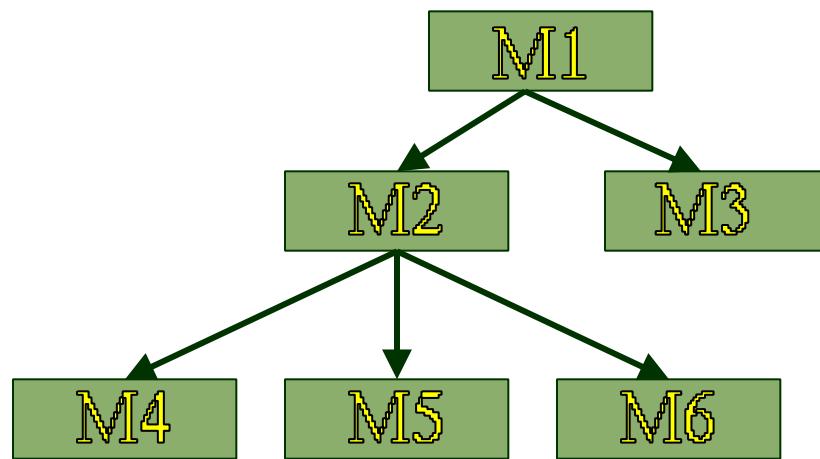
- Ñ Software design deals with transforming the customer requirements, as described in the SRS document, into a form (a set of documents) that is suitable for implementation in a programming language.
- Ñ A good software design is seldom arrived by using a single step procedure but rather through several iterations through a series of steps.
- Ñ Design activities can be broadly classified into two important parts:
 - y Preliminary (or high-level) design and
 - y Detailed design

Items Designed During Design Phase



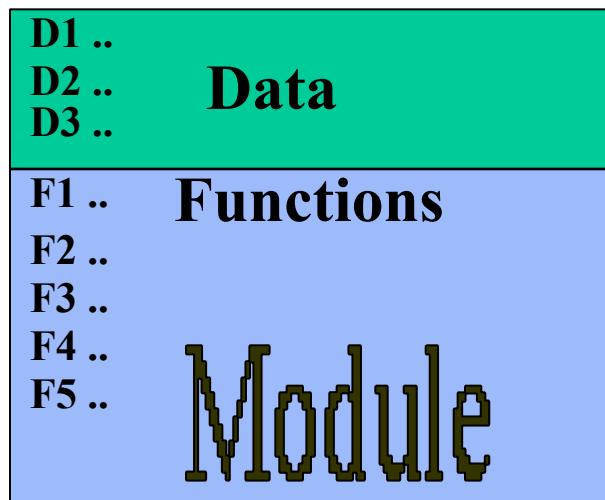
- Ñ module structure,
- Ñ control relationship among the modules
 - y call relationship or invocation relationship
- Ñ interface among different modules,
 - y data items exchanged among different modules,
- Ñ data structures of individual modules,
- Ñ algorithms for individual modules.

Module Structure



Introduction

~ A module consists of:
y several functions
y associated data structures.



High Level and Detailed Design

- Ñ High-level design means identification of different modules and the control relationships among them and the definition of the interfaces among these modules.
- Ñ The outcome of high-level design is called the **program structure or software architecture**. Ex: Tree-like structure.
- Ñ Detailed design, the data structure and the algorithms of the different modules are designed.
- Ñ The outcome of the detailed design stage is usually known as the module-specification document.

What Is Good Software Design?

- Ñ Should implement all functionalities of the system correctly.
- Ñ Should be easily understandable.
- Ñ Should be efficient.
- Ñ Should be easily amenable to change,
 - y i.e. easily maintainable.
- Ñ Understandability of a design is a major issue:
 - y determines goodness of design:
 - y a design that is easy to understand:
 - x also easy to maintain and change.

Understandability

- Ñ Use consistent and meaningful names
 - y for various design components,
- Ñ Design solution should consist of:
 - y a cleanly decomposed set of modules (modularity),
- Ñ Different modules should be neatly arranged in a hierarchy:
 - y in a neat tree-like diagram.

Modularity

ÑModularity is a fundamental attributes of any good design.

- y Decomposition of a problem cleanly into modules:
- y Modules are almost independent of each other
- y divide and conquer principle.

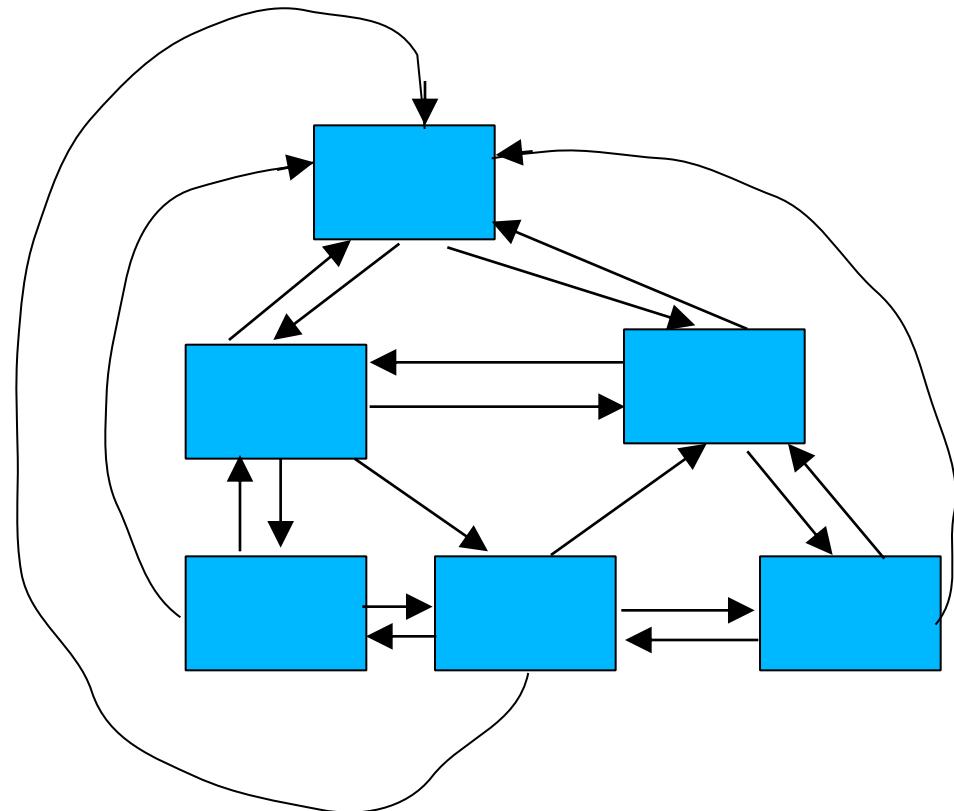
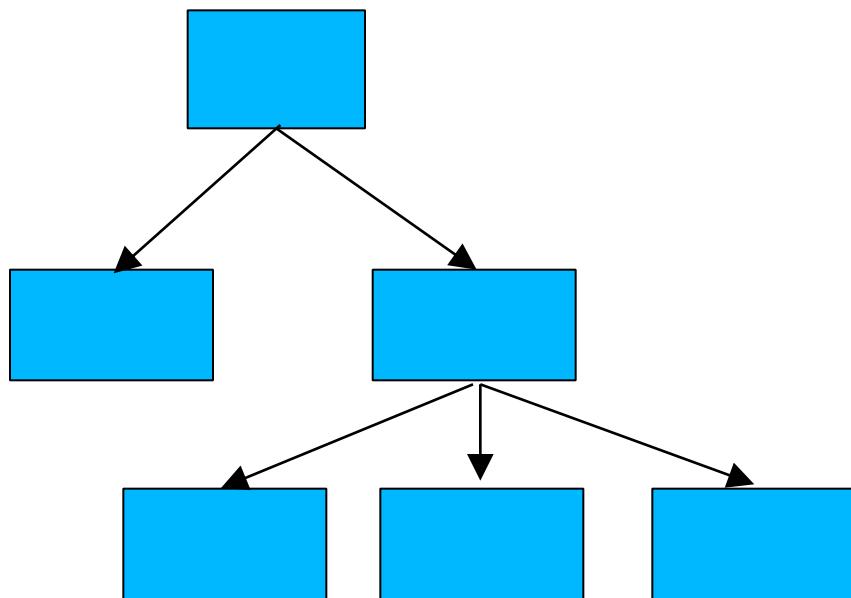
Modularity

Ñ If modules are independent:

- y modules can be understood separately,
- x reduces the complexity greatly.

- y To understand why this is so,
 - x remember that it is very difficult to break a bunch of sticks but very easy to break the sticks individually.

Example of Cleanly and Non-cleanly Decomposed Modules



Modularity

ÑIn technical terms, modules should display:

- y high cohesion
- y low coupling.

Cohesion and Coupling

- ~ Cohesion is a measure of:
 - y functional strength of a module.
 - y A cohesive module performs a single task or function.
- ~ Coupling between two modules:
 - y a measure of the degree of interdependence or interaction between the two modules.

Cohesion and Coupling

~ A module having high cohesion and low coupling:

y functionally independent of other modules:

x A functionally independent module has minimal interaction with other modules.

Advantages of Functional Independence

- Ñ Better understandability and good design:
- Ñ Complexity of design is reduced,
- Ñ Different modules easily understood in isolation:
 - y modules are independent

Advantages of Functional Independence

~ **N**Functional independence reduces error propagation.

y degree of interaction between modules is low.

y an error existing in one module does not directly affect other modules.

~ **N**Reuse of modules is possible.

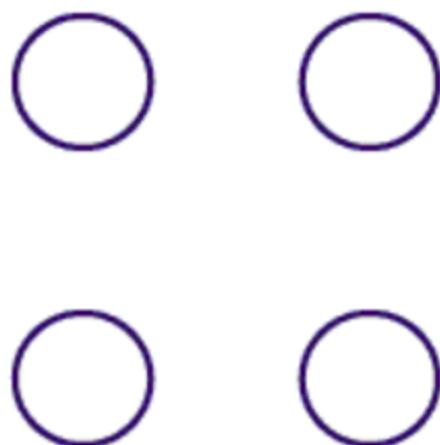
Advantages of Functional Independence

ÑA functionally independent module:

- y can be easily taken out and reused in a different program.
- x each module does some well-defined and precise function
- x the interfaces of a module with other modules is simple and minimal.

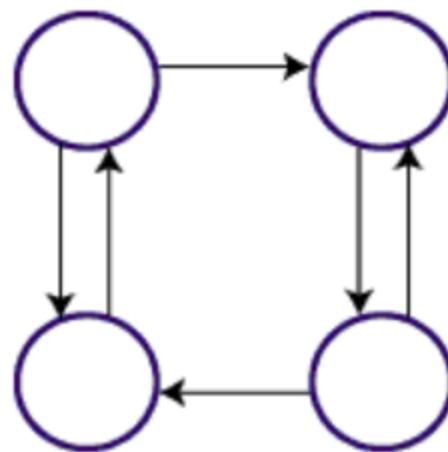
The various types of coupling techniques are shown in fig:

Module Coupling



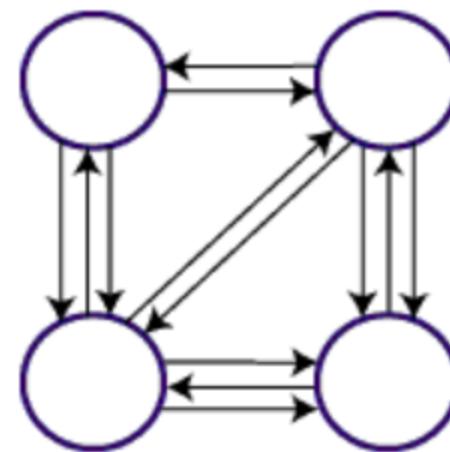
Uncoupled: no
dependencies

(a)



Loosely Coupled:
Some dependencies

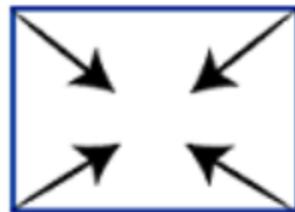
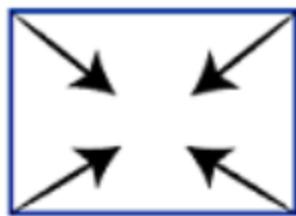
(b)



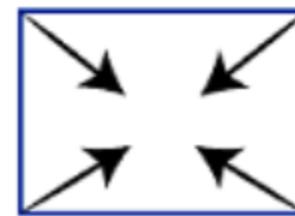
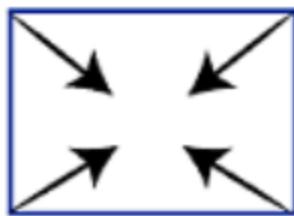
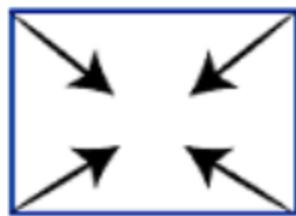
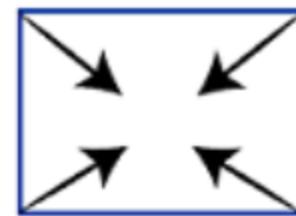
Highly Coupled:
Many dependencies

(c)

Cohesion

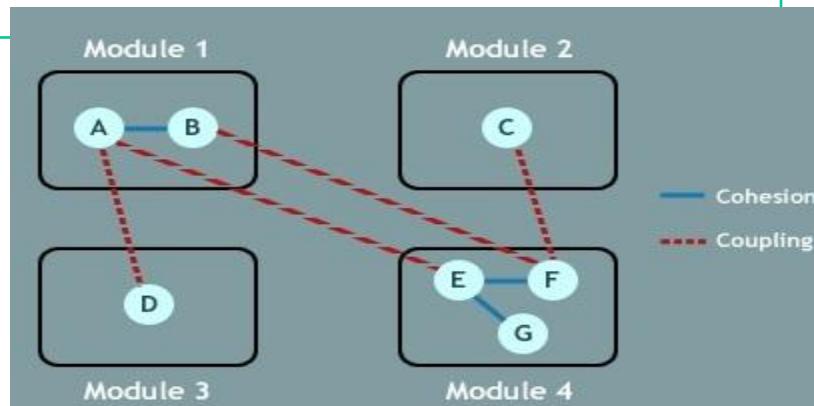


Module
Strength

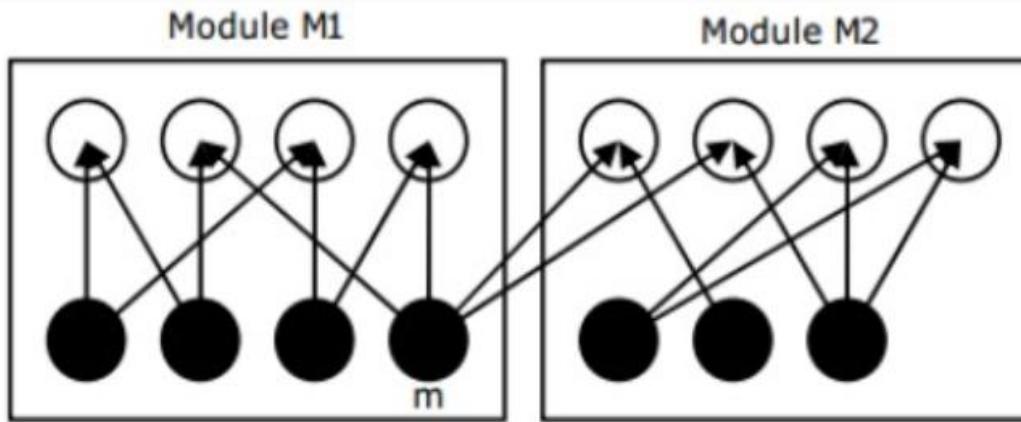


Cohesion= Strength of relations within Modules

Cohesion	Coupling
Cohesion is the indication of the relationship within module.	Coupling is the indication of the relationships between modules.
Cohesion shows the module's relative functional strength .	Coupling shows the relative independence among the modules .
Cohesion is a degree (quality) to which a component / module focuses on the single thing .	Coupling is a degree to which a component / module is connected to the other modules .
While designing you should strive for high cohesion i.e. a cohesive component/ module focus on a single task.	While designing you should strive for low coupling i.e. Dependency between modules should be less.
Cohesion is Intra – Module Concept.	Coupling is Inter -Module Concept.



Filled circles represents Methods Unfilled circle represent Attributes

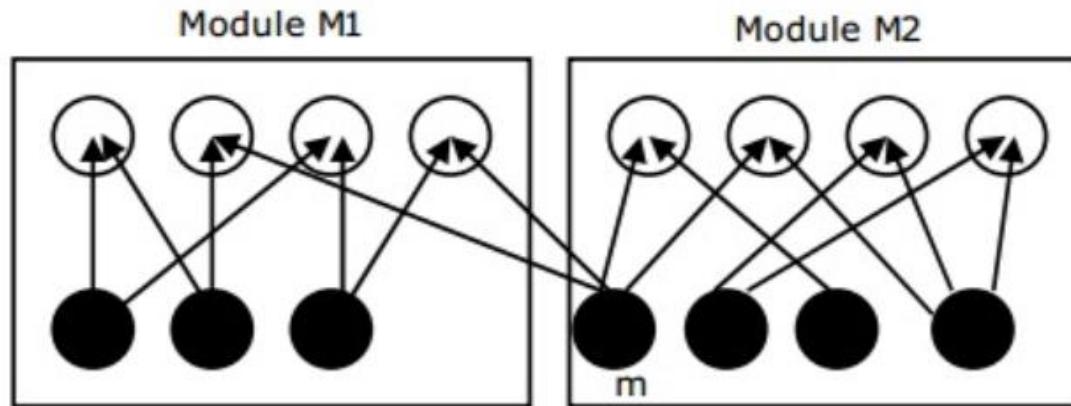


$$\text{Coupling} = \frac{\text{number of external links}}{\text{number of modules}} = \frac{2}{2}$$

$$\text{Cohesion of a module} = \frac{\text{number of internal links}}{\text{number of methods}}$$

$$\text{Cohesion of } M_1 = \frac{8}{4}; \text{ Cohesion of } M_2 = \frac{6}{3}; \text{ Average cohesion} = 2$$

After moving method m to M2, graph will become

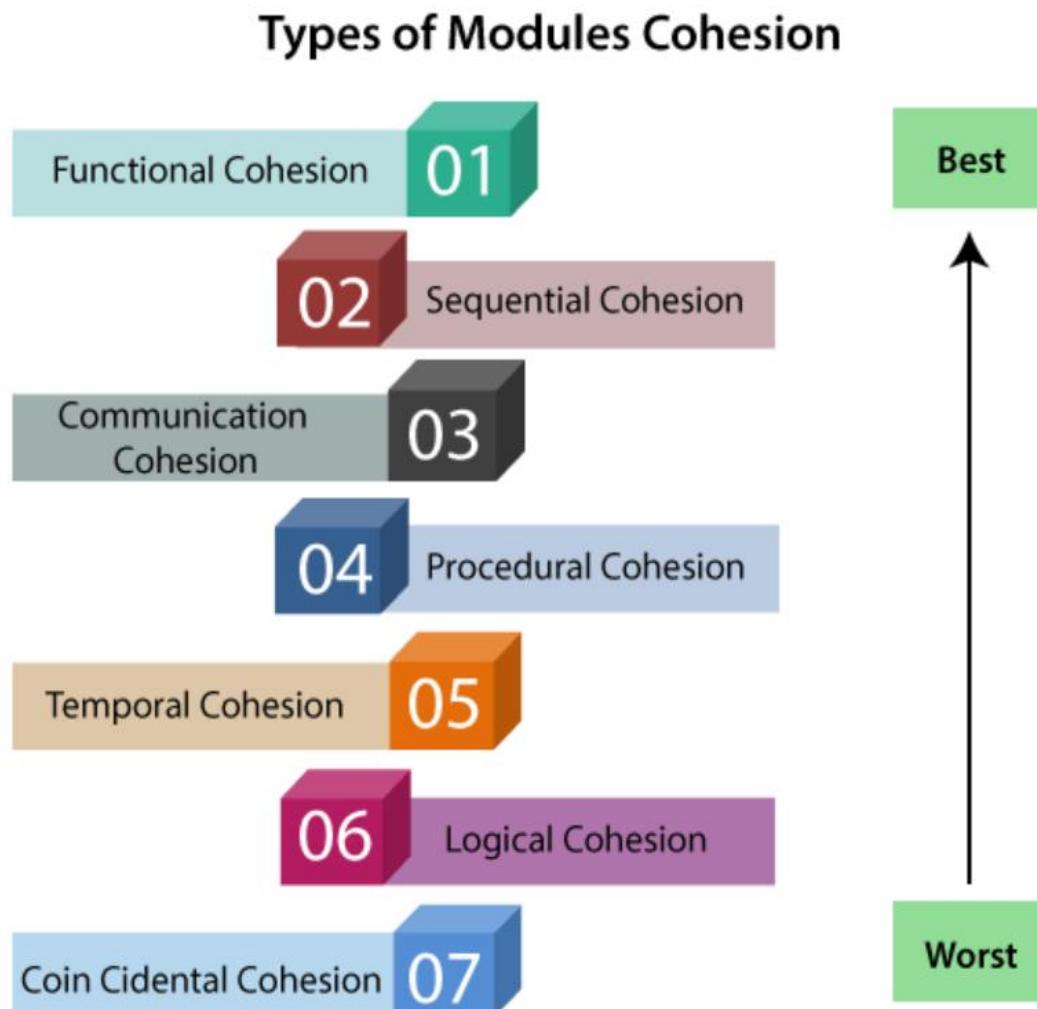


$$\text{Coupling} = \frac{2}{2}$$

$$\text{Cohesion of } M_1 = \frac{6}{3}; \text{ Cohesion of } M_2 = \frac{8}{4}; \quad \text{Average cohesion} = 2$$

∴ answer is no change

Types of Modules Cohesion



Classification of Cohesiveness

Classification is often subjective:

y yet gives us some idea about cohesiveness of a module.

By examining the type of cohesion exhibited by a module:

y we can roughly tell whether it displays high cohesion or low cohesion.

- 
- 1. Functional Cohesion:** Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.
 - 2. Sequential Cohesion:** A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output from one component of the sequence is input to the next.
 - 3. Communicational Cohesion:** A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack.

- 
4. **Procedural Cohesion:** A module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.
 5. **Temporal Cohesion:** When a module includes functions that are associated by the fact that all the methods must be executed in the same time, the module is said to exhibit temporal cohesion.
 6. **Logical Cohesion:** A module is said to be logically cohesive if all the elements of the module perform a similar operation. For example Error handling, data input and data output, etc.
 7. **Coincidental Cohesion:** A module is said to have coincidental cohesion if it performs a set of tasks that are associated with each other very loosely, if at all.

Coupling

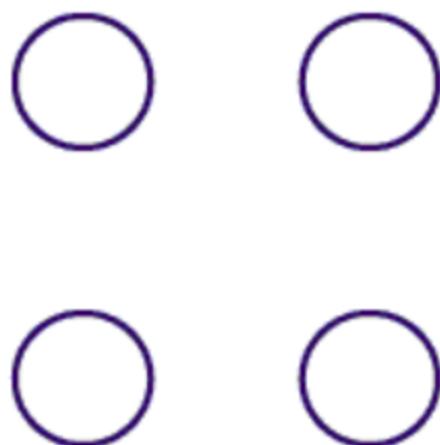


ÑCoupling indicates:

- y how closely two modules interact or how interdependent they are.
- y The degree of coupling between two modules depends on their interface complexity.

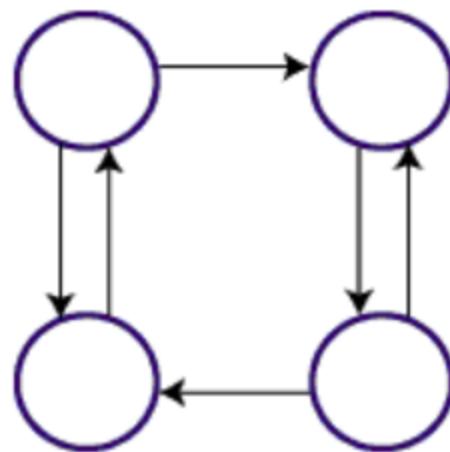
The various types of coupling techniques are shown in fig:

Module Coupling



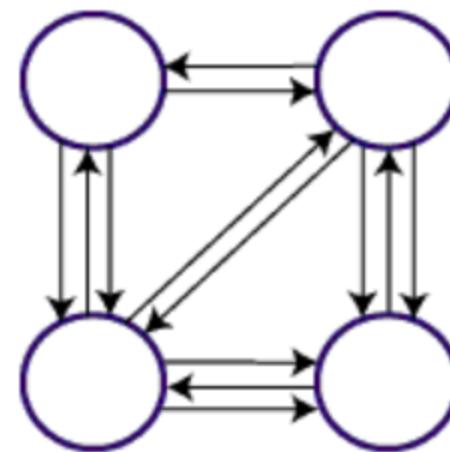
Uncoupled: no
dependencies

(a)



Loosely Coupled:
Some dependencies

(b)



Highly Coupled:
Many dependencies

(c)

Coupling

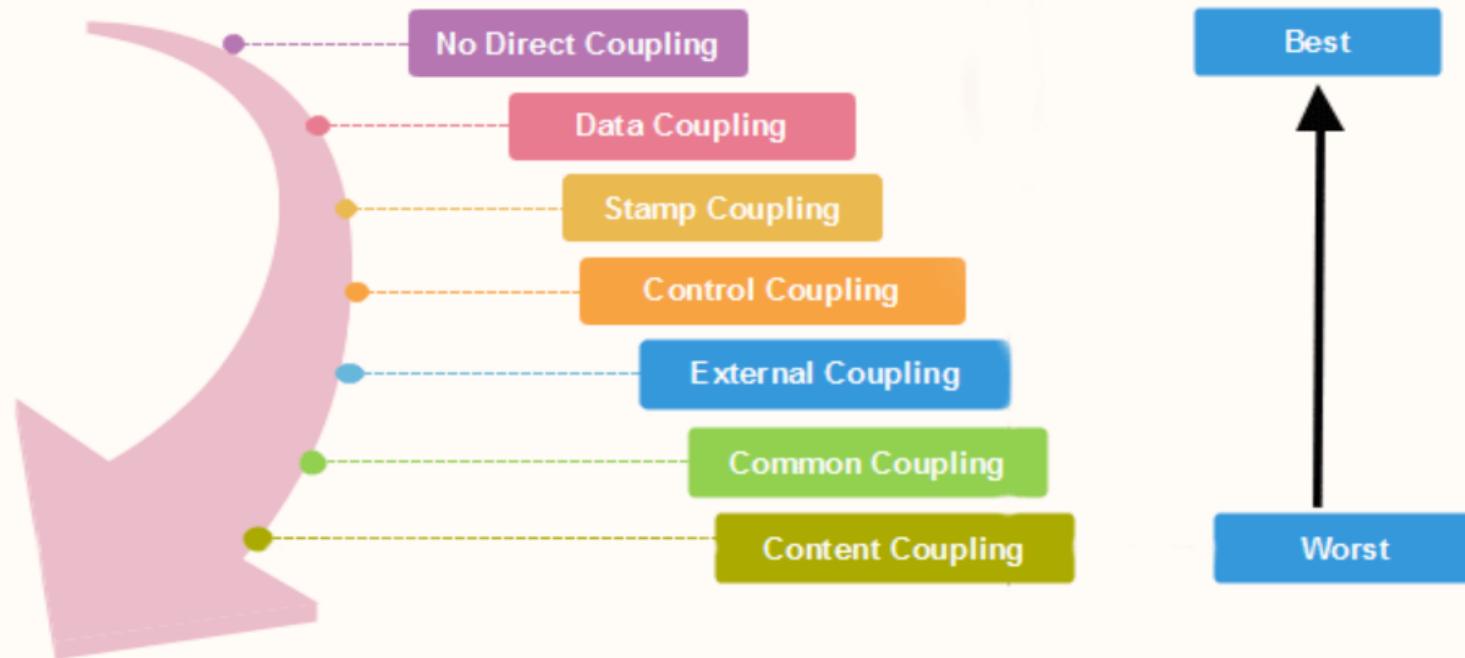


- Ñ There are no ways to precisely determine coupling between two modules:
 - y classification of different types of coupling will help us to approximately estimate the degree of coupling between two modules.
- Ñ Five types of coupling can exist between any two modules.

Types of Module Coupling

Types of Modules Coupling

There are various types of module Coupling are as follows:

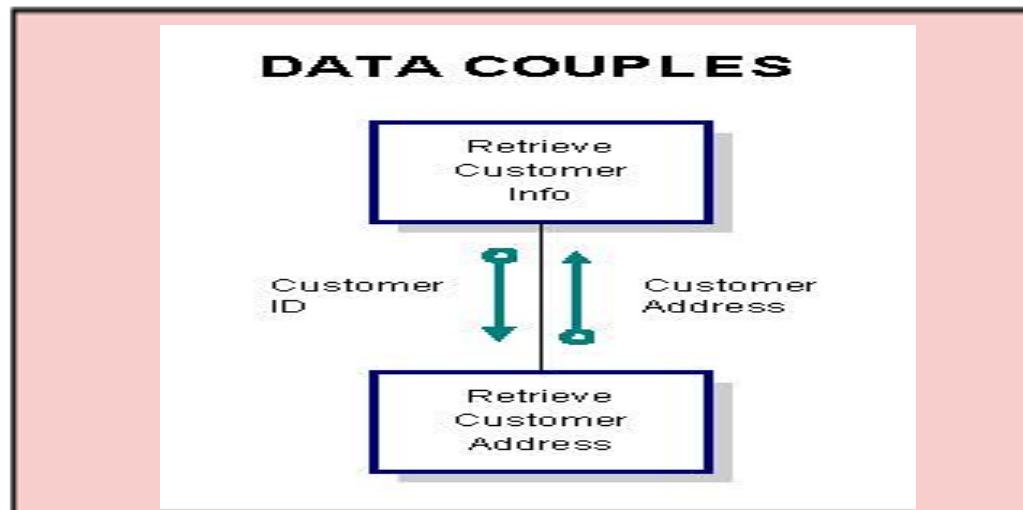


1. No Direct Coupling: There is no direct coupling between M1 and M2.



In this case, modules are subordinates to different modules. Therefore, no direct coupling.

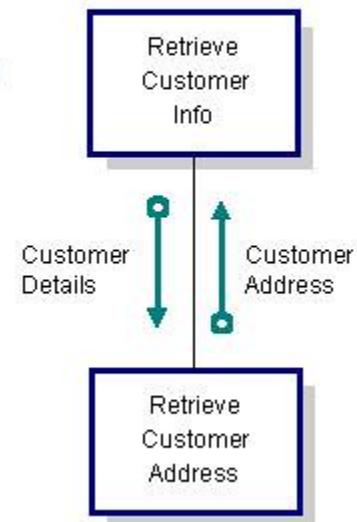
2. Data Coupling: When data of one module is passed to another module, this is called data coupling.



Data coupling occurs between two modules when data are passed by parameters using a simple argument list and every item in the list is used.

An example of data coupling is a module which retrieves customer address using customer id.

STAMP COUPLE



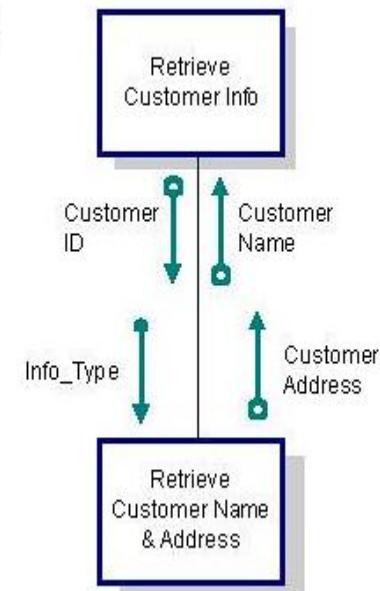
3. Stamp Coupling: Two modules are stamp coupled if they communicate using composite data items such as structure, objects, etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled. For example, passing structure variable in C or object in C++ language to a module.

An example of stamp coupling where a module that retrieves customer address using only customer id which is extracted from a parameter named customer details.

CONTROL COUPLE

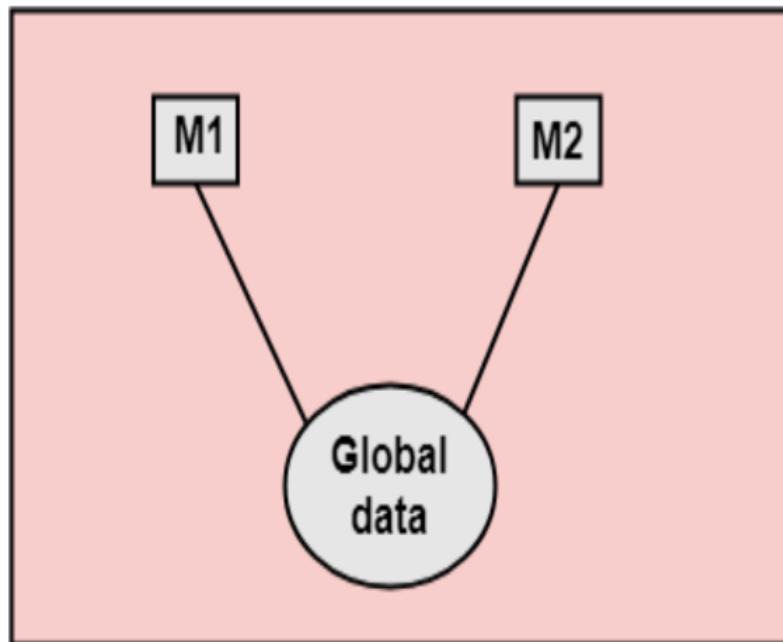
4. Control Coupling: Control Coupling exists among two modules if data from one module is used to direct the structure of instruction execution in another.

example of control coupling, a module that retrieves either a customer name or an address depending on the value of a flag is illustrated.



5. External Coupling: External Coupling arises when two modules share an externally imposed data format, communication protocols, or device interface. This is related to communication to external tools and devices.

6. Common Coupling: Two modules are common coupled if they share information through some global data items.



7. Content Coupling: Content Coupling exists among two modules if they share code, e.g., a branch from one module into another module.

Neat Hierarchy

~ Control hierarchy represents:

y organization of modules.

y control hierarchy is also called
program structure.

~ Most common notation:

y a tree-like diagram called structure chart.

Neat Arrangement of modules



~Essentially means:

yellow fan-out

yellow abstraction

Characteristics of Module Structure

Ñ Depth:

- y number of levels of control

Ñ Width:

- y overall span of control.

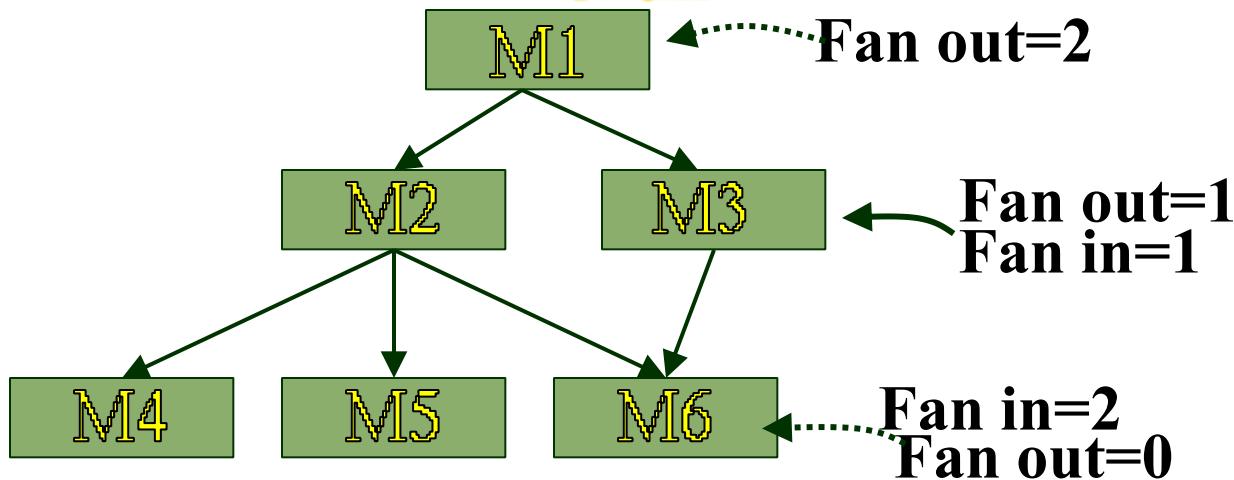
Ñ Fan-out:

- y a measure of the number of modules directly controlled by given module.

Ñ Fan-in:

- y indicates how many modules directly invoke a given module.
- y High fan-in represents code reuse and is in general encouraged.

Module Structure



Goodness of Design

Ñ A design having modules:

- y with high fan-out numbers is not a good design:

- y a module having high fan-out lacks cohesion.

Ñ A module that invokes a large number of other modules:

- y likely to implement several different functions:

- y not likely to perform a single cohesive function.

Control Relationships

- Ñ A module that controls another module:
y said to be superordinate to it.

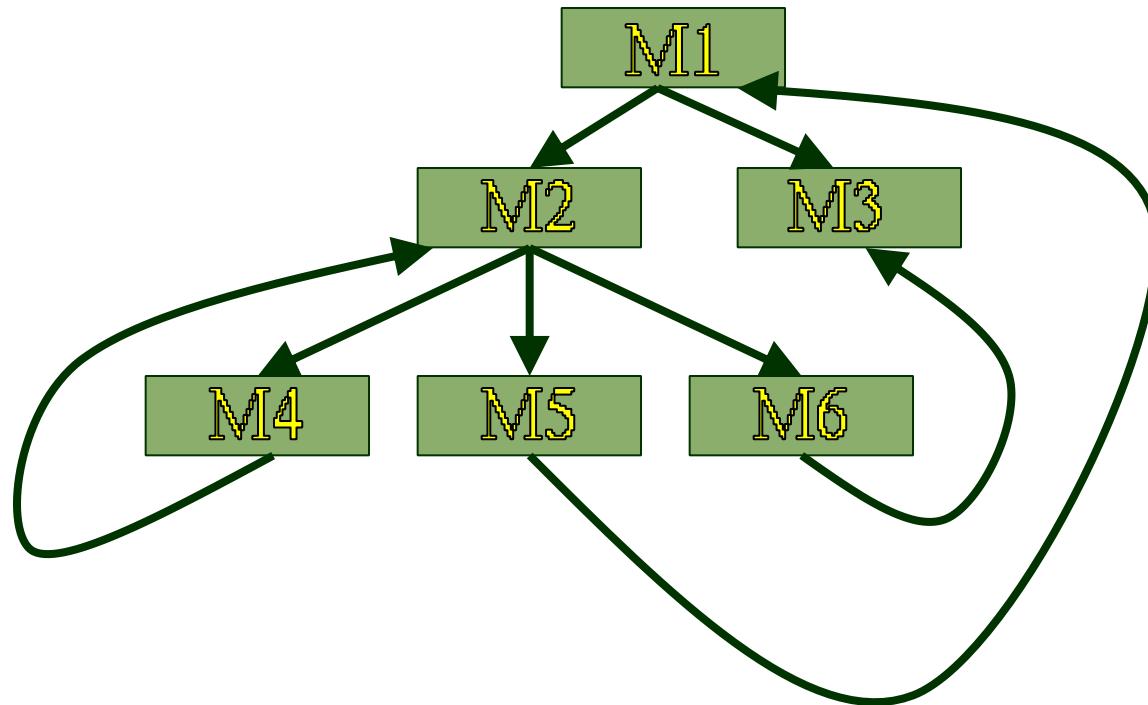
- Ñ Conversely, a module controlled by another module:
y said to be subordinate to it.

Visibility and Layering

Ñ A module A is said to be visible by another module B,
y if A directly or indirectly calls B (Embedding).

Ñ The layering principle requires
y modules at a layer can call only the modules immediately below it (Sequence).

Bad Design



Abstraction

Ñ Lower-level modules:

y do input/output and other low-level functions.

Ñ Upper-level modules:

y do more managerial functions.

Ñ The principle of abstraction requires:

y lower-level modules do not invoke functions of higher level modules.

y Also known as layered design.

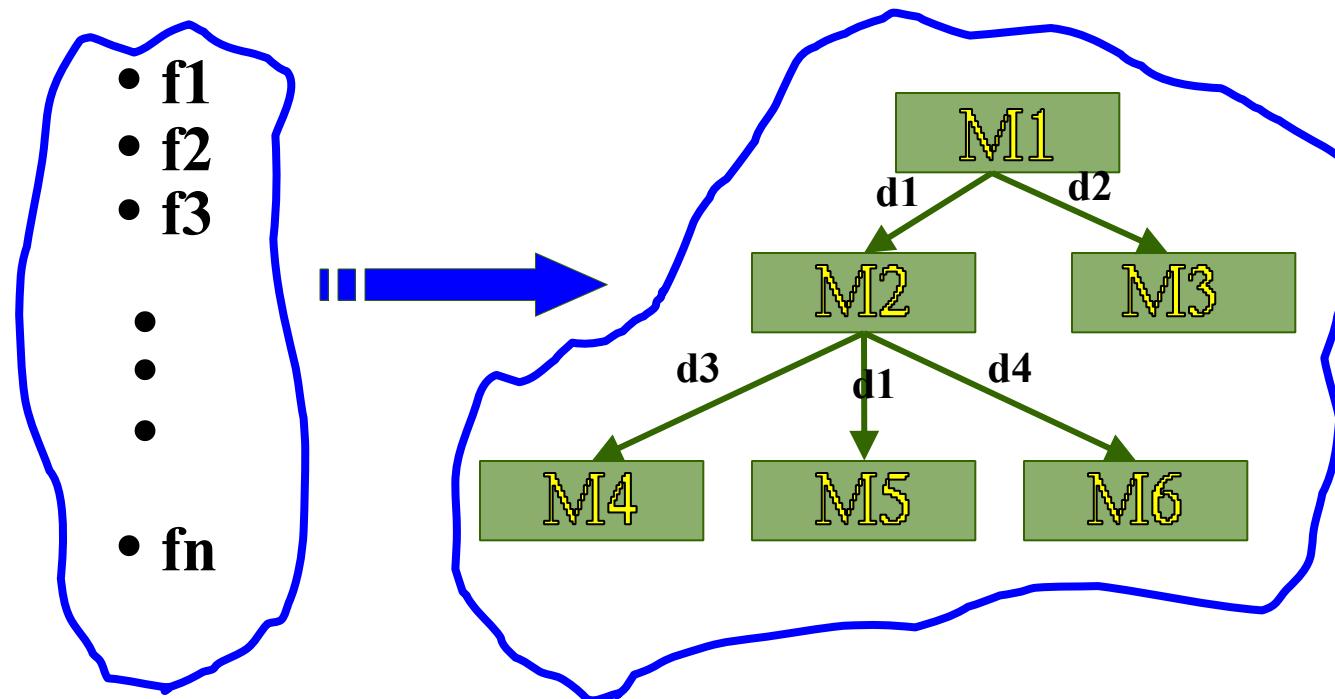
High-level Design



~ High-level design maps functions into modules $\{f_i\}$ $\{m_j\}$ such that:

- y Each module has high cohesion
- y Coupling among modules is as low as possible
- y Modules are organized in a neat hierarchy

High-level Design



Design Approaches

Ñ Two fundamentally different software design approaches:

- y Function-oriented design
- y Object-oriented design

Ñ These two design approaches are radically different.

- y However, are complementary
 - x rather than competing techniques.
- y Each technique is applicable at
 - x different stages of the design process.

Function-Oriented Design

- Ñ A system is looked upon as something
 - y that performs a set of functions.
- Ñ Starting at this high-level view of the system:
 - y each function is successively refined into more detailed functions.
 - y Functions are mapped to a module structure.
- Ñ Example : The function **create-new-library-member**:
 - y **creates** the record for a new member,
 - y **assigns** a unique membership number
 - y **prints** a bill towards the membership

Example



~Create-library-member function
consists of the following sub-
functions:

- y assign-membership-number
- y create-member-record
- y print-bill

Function-Oriented Design

- ~ N Each subfunction:
 - y split into more detailed subfunctions and so on.
- ~ N The system state is centralized:
 - y accessible to different functions,
 - y member-records:
 - x available for reference and updation to several functions:
 - create-new-member
 - delete-member
 - update-member-record

Function-Oriented Design

Several function-oriented design approaches have been developed:

- y Structured design (Constantine and Yourdon, 1979)
- y Jackson's structured design (Jackson, 1975)
- y Warnier-Orr methodology
- y Wirth's step-wise refinement
- y Hatley and Pirbhai's Methodology

Object-Oriented Design

- Ñ System is viewed as a collection of objects (i.e. entities).
- Ñ System state is decentralized among the objects:
 - y each object manages its own state information.

Example:

- Ñ Library Automation Software:
 - y each library member is a separate object
 - x with its own data and functions.
 - y Functions defined for one object:
 - x cannot directly refer to or change data of other objects.

Object-Oriented Design

- ~ Objects have their own internal data:
y defines their state.
- ~ Similar objects constitute a class.
y each object is a member of some class.
- ~ Classes may inherit features
y from a super class.
- ~ Conceptually, objects communicate by message passing.

Object-Oriented versus Function-Oriented Design

- Ñ Unlike function-oriented design,
 - y in OOD the basic abstraction is not functions such as "sort", "display", "track", etc.,
 - y but real-world entities such as "employee", "picture", "machine", "radar system", etc.

- Ñ In OOD:
 - y software is not developed by designing functions such as:
 - x update-employee-record,
 - x get-employee-address, etc.
 - y but by designing objects such as:
 - x employees,
 - x departments, etc.

Example:

Ñ In an employee pay-roll system, the following can be global data:

- y names of the employees,
- y their code numbers,
- y basic salaries, etc.

Ñ Whereas, in object oriented systems:

- y data is distributed among different employee objects of the system.

Object-Oriented versus Function-Oriented Design

- Ñ Function-oriented techniques group functions together if:
 - y as a group, they constitute a higher level function.
- Ñ On the other hand, object-oriented techniques group functions together:
 - y on the basis of the data they operate on.
- Ñ To illustrate the differences between object-oriented and function-oriented design approaches,
 - y let us consider an example ---
 - y An automated fire-alarm system for a large building.

Fire-Alarm System:

- Ñ We need to develop a computerized fire alarm system for a large multi-storied building:
 - y There are 80 floors and 1000 rooms in the building.
- Ñ Different rooms of the building:
 - y fitted with smoke detectors and fire alarms.
- Ñ The fire alarm system would monitor:
 - y status of the smoke detectors.
- Ñ Whenever a fire condition is reported by any smoke detector:
 - y the fire alarm system should:
 - x determine the location from which the fire condition was reported
 - x sound the alarms in the neighboring locations.

Fire-Alarm System

- Ñ The fire alarm system should:
 - y flash an alarm message on the computer console:
 - x fire fighting personnel man the console round the clock.

- Ñ After a fire condition has been successfully handled,
 - y the fire alarm system should let fire fighting personnel reset the alarms.

Function-Oriented Approach:

Ñ /* Global data (system state) accessible by various functions */

```
BOOL detector_status[1000];
int detector_locs[1000];
BOOL alarm_status[1000]; /* alarm activated when status set */
int alarm_locs[1000]; /* room number where alarm is located */
int neighbor_alarms[1000][10]; /* each detector has at most
                                 * 10 neighboring alarm locations */
```

Ñ The functions which operate on the system state:

```
interrogate_detectors();
get_detector_location();
determine_neighbor();
ring_alarm();
reset_alarm();
report_fire_location();
```

Object-Oriented Approach:

- Ñ class detector
 - attributes: status, location, neighbors
 - operations: create, sense-status, get-location, find-neighbors
- Ñ class alarm
 - attributes: location, status
 - operations: create, ring-alarm, get_location, reset-alarm
- Ñ In the object oriented program,
appropriate number of instances of the class detector and
alarm should be created.

Summary

~ We started with an overview of:
y activities undertaken during the software design phase.

~ We identified:
y the information need to be produced at the end of the design phase:
x so that the design can be easily implemented using a programming language.

Summary



ÑWe characterized the features of a good software design by introducing the concepts of:

- y fan-in, fan-out,
- y cohesion, coupling,
- y abstraction, etc.

Summary



ÑWe classified different types of cohesion and coupling:

yenables us to approximately determine the cohesion and coupling existing in a design.

Summary



Two fundamentally different approaches to software design:

- y function-oriented approach
- y object-oriented approach

Summary



ÑWe looked at the essential philosophy behind these two approaches

ythese two approaches are not competing but complementary approaches.

Function-Oriented Software Design

(lecture 5)

Organization of this Lecture



- # Brief review of last lecture
- # Introduction to function-oriented design
- # Structured Analysis and Structured Design
- # Data flow diagrams (DFDs)
 - ↗ A major objective of this lecture is that you should be able to develop DFD model for any problem.
- # Examples
- # Summary

Review of last lecture

⌘ Last lecture we started

└ with an overview of activities carried out during the software design phase.

⌘ We identified different information that must be produced at the end of the design phase:

└ so that the design can be easily implemented using a programming language.

Review of last lecture

- ⌘ We characterized the features of a good software design by introducing the concepts:
 - ↗ cohesion, coupling,
 - ↗ fan-in, fan-out,
 - ↗ abstraction, etc.
- ⌘ We classified different types of cohesion and coupling:
 - ↗ enables us to approximately determine the cohesion and coupling existing in a design.

Review of last lecture

⌘ There are two fundamentally different approaches to software design:

- ─ function-oriented approach
- ─ object-oriented approach

⌘ We looked at the essential philosophy of these two approaches:

- ─ the approaches are not competing but complementary approaches.

Introduction

- ⌘ **Function-oriented design techniques are very popular:**
 - ↗ currently in use in many software development organizations.
- ⌘ **Function-oriented design techniques:**
 - ↗ start with the functional requirements specified in the SRS document.

Introduction

- ⌘ During the design process:
 - ↗ high-level functions are successively decomposed:
 - ☒ into more detailed functions.
 - ↗ finally the detailed functions are mapped to a module structure.

Introduction

⌘ Successive decomposition
of high-level functions:

- ⌘ into more detailed functions.
- ⌘ Technically known as **top-down decomposition**.

Introduction

⌘ SA/SD methodology:

❑ has essential features of several important function-oriented design methodologies ---

SA/SD (Structured Analysis/Structured Design)

⌘ SA/SD technique draws heavily from the following methodologies:

- ◻ Constantine and Yourdon's methodology
- ◻ Hatley and Pirbhai's methodology
- ◻ Gane and Sarson's methodology
- ◻ DeMarco and Yourdon's methodology

⌘ SA/SD technique can be used to perform

- ◻ high-level design.

Overview of SA/SD Methodology

⌘ SA/SD methodology consists of two distinct activities:

- ⌘ Structured Analysis (SA)

- ⌘ Structured Design (SD)

⌘ During structured analysis:

- ⌘ functional decomposition takes place.

⌘ During structured design:

- ⌘ module structure is formalized.

Functional decomposition

❖ Each function is analyzed:

- ❑ hierarchically decomposed into more detailed functions.
- ❑ simultaneous decomposition of high-level data
- ❑ into more detailed data.

Structured analysis

- ❖ Transforms a textual problem description into a graphic model.
- ❖ done using data flow diagrams (DFDs).
- ❖ DFDs graphically represent the results of structured analysis.

Structured design

⌘ All the functions represented in the DFD:

 mapped to a **module structure**.

⌘ The module structure:

 also called as the **software architecture**:

Detailed Design

⌘ Software architecture:

- ▢ refined through detailed design.
- ▢ Detailed design can be directly implemented:
 - ☒ using a conventional programming language.

Structured Analysis vs. Structured Design

⌘ Purpose of structured analysis:

 **capture the detailed structure of
the system as the user views it.**

⌘ Purpose of structured design:

 **arrive at a form that is suitable
for implementation in some
programming language.**

Structured Analysis vs. Structured Design

- ⌘ **The results of structured analysis can be easily understood even by ordinary customers:**
 - ▢ **does not require computer knowledge**
 - ▢ **directly represents customer's perception of the problem**
 - ▢ **uses customer's terminology for naming different functions and data.**
- ⌘ **The results of structured analysis can be reviewed by customers:**
 - ▢ **to check whether it captures all their requirements.**

Structured Analysis

- ⌘ Based on principles of:
 - ↗ Top-down decomposition approach.
 - ↗ Divide and conquer principle:
 - ☒ each function is considered individually
(i.e. isolated from other functions)
 - ☒ decompose functions totally disregarding
what happens in other functions.
 - ↗ Graphical representation of results
using
 - ☒ data flow diagrams (or bubble charts).

Data flow diagrams

- ⌘ DFD is an elegant modelling technique:
 - ◻ useful not only to represent the results of structured analysis
 - ◻ applicable to other areas also:
 - ☒ e.g. for showing the flow of documents or items in an organization,
- ⌘ DFD technique is very popular because
 - ◻ it is simple to understand and use.

Data flow diagram

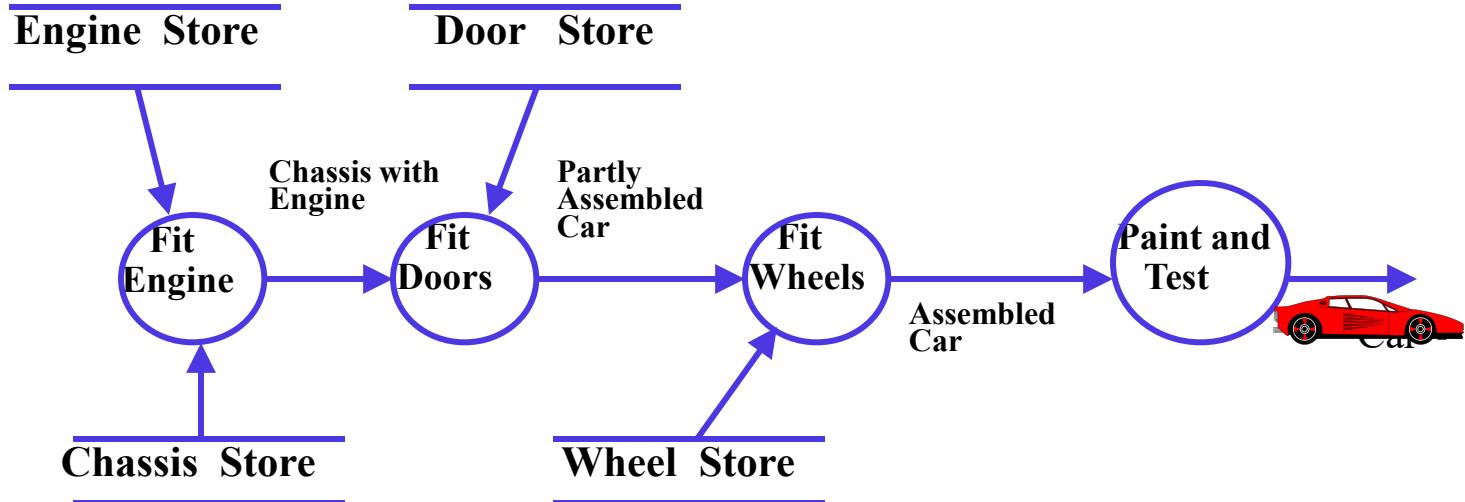
- ❖ DFD is a hierarchical graphical model:
 - ❖ shows the different functions (or processes) of the system and
 - ❖ data interchange among the processes.

DFD Concepts

⌘ It is useful to consider each function as a processing station:

- ❑ each function consumes some input data and
- ❑ produces some output data.

Data Flow Model of a Car Assembly Unit



Data Flow Diagrams (DFDs)

⌘ A DFD model:

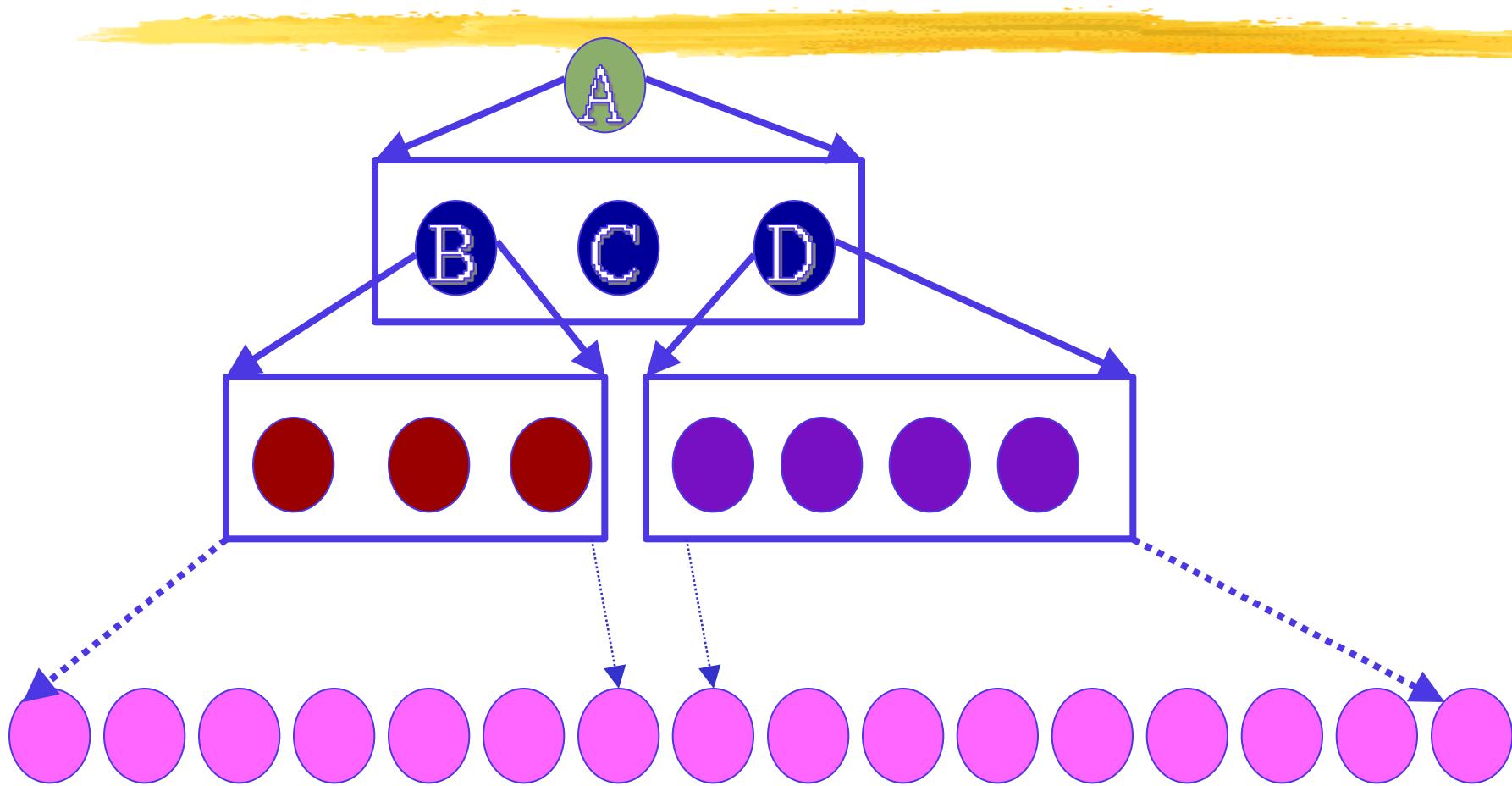
- ─ uses limited types of symbols.
- ─ simple set of rules
- ─ easy to understand:
 - ─ it is a hierarchical model.

Hierarchical model

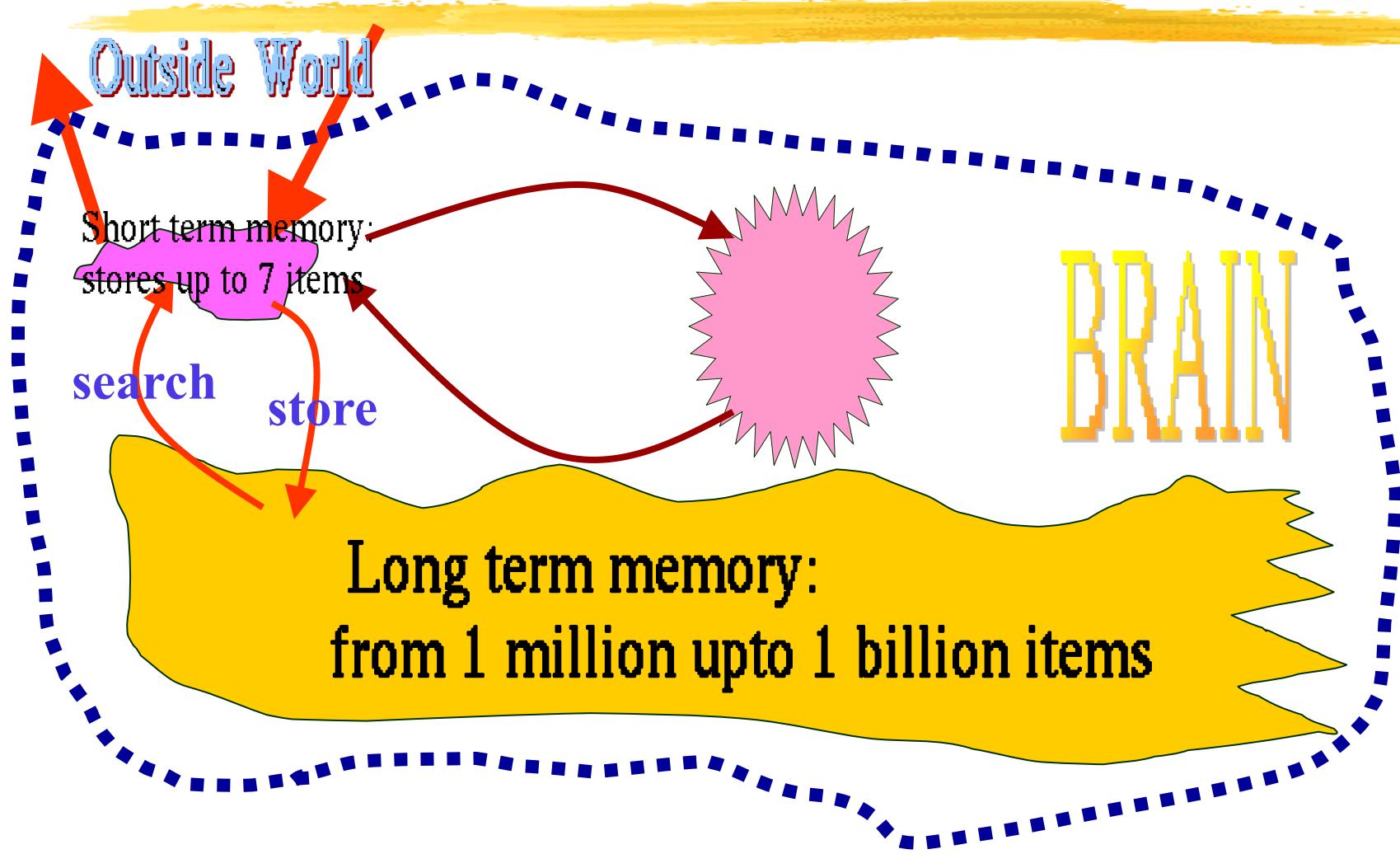
⌘ Human mind can easily understand any hierarchical model:

- ◻ in a hierarchical model:
 - ◻ we start with a very simple and abstract model of a system,
 - ◻ details are slowly introduced through the hierarchies.

Hierarchical Model



How does the human mind work? (Digression)



How does the human mind work? (Digression)

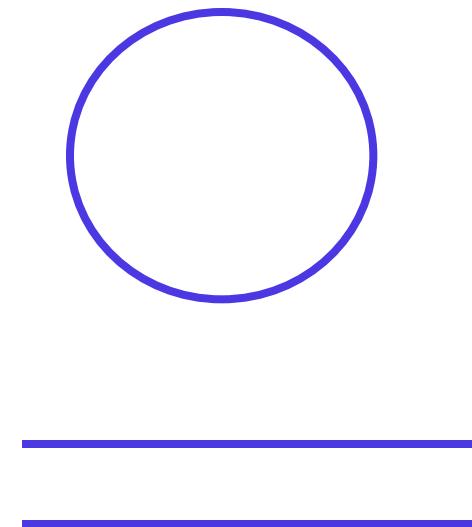
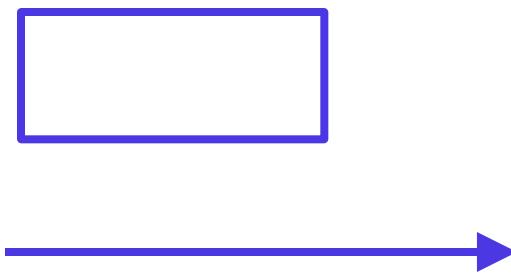
- ⌘ **Short term memory can hold upto 7 items:**
 - ◻ In Software Engineering the number 7 is called as the magic number.
- ⌘ **An item is any set of related information (called a chunk):**
 - ◻ an integer
 - ◻ a character
 - ◻ a word
 - ◻ a story
 - ◻ a picture, etc

How does the human mind work? (Digression)

- ⌘ To store 1,9,6,5 requires 4 item spaces:
 - ↗ but requires only one storage space when I recognize it as my year of birth.
- ⌘ It is not surprising that large numbers::
 - ↗ usually broken down into several 3 or 4 digit numbers
 - ↗ e.g. 61-9266-2948

Data Flow Diagrams (DFDs)

⌘ Primitive Symbols Used for Constructing DFDs:



External Entity Symbol

⌘ Represented by a rectangle

⌘ External entities are real physical entities:

input data to the system or

consume data produced by the system.

Sometimes external entities are called **terminator, source, or sink.**

Librarian

Function Symbol

⌘ A function such as “search-book” is represented using a circle:

- ◻ This symbol is called a **process** or **bubble** or **transform**.
- ◻ Bubbles are annotated with corresponding function names.
- ◻ Functions represent some activity:
 - ☒ **function names should be verbs.**



Data Flow Symbol

⌘ A directed arc or line.

⌘ represents data flow in the direction of the arrow.

⌘ Data flow symbols are annotated with names of data they carry.

Data Store Symbol

❖ Represents a logical file:

❖ A logical file can be:

☒ a data structure

☒ a physical file on disk.

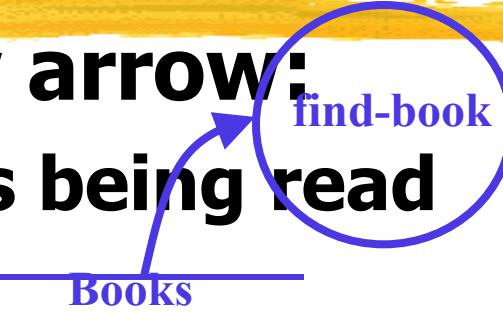
❖ Each data store is connected to a process:

☒ by means of a data flow symbol.

Data Store Symbol

⌘ **Direction of data flow arrow:**

─ shows whether data is being read from or written into it.



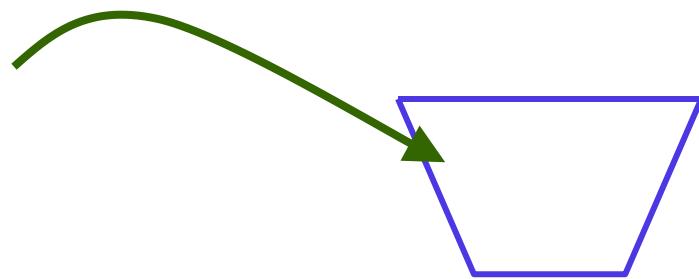
⌘ **An arrow into or out of a data store:**

─ implicitly represents the entire data of the data store

─ arrows connecting to a data store need not be annotated with any data name.

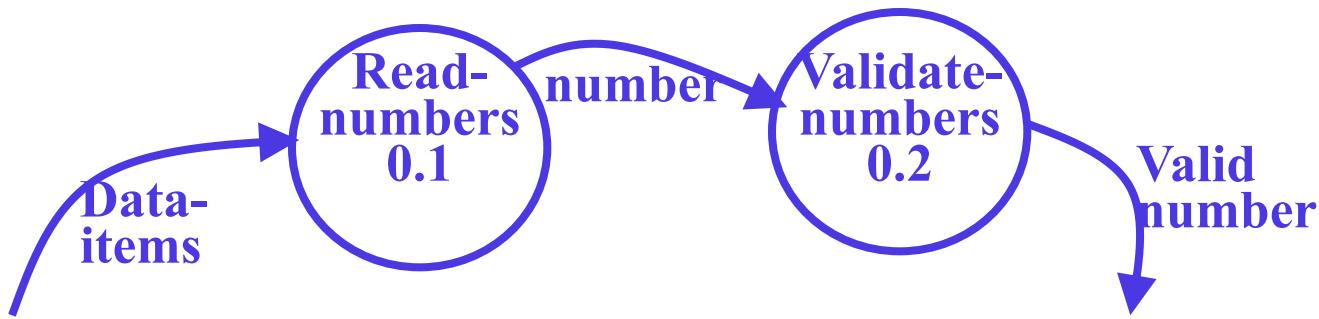
Output Symbol

⌘ Output produced by the system



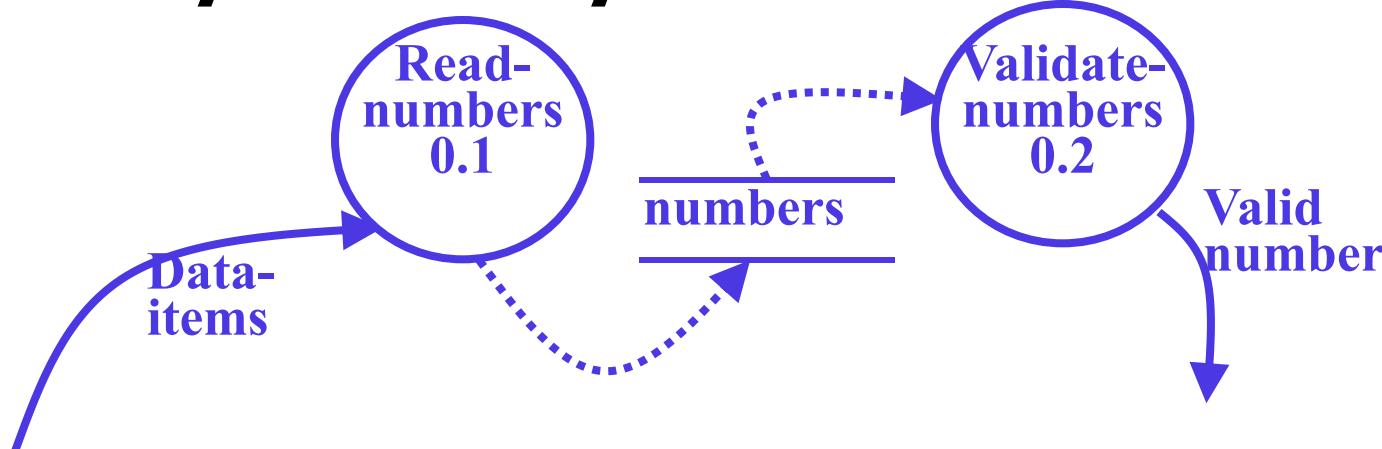
Synchronous operation

- ⌘ If two bubbles are directly connected by a data flow arrow:
 - they are synchronous



Asynchronous operation

- ⌘ If two bubbles are connected via a data store:
 - they are not synchronous.



Yourdon's vs. Gane Sarson Notations



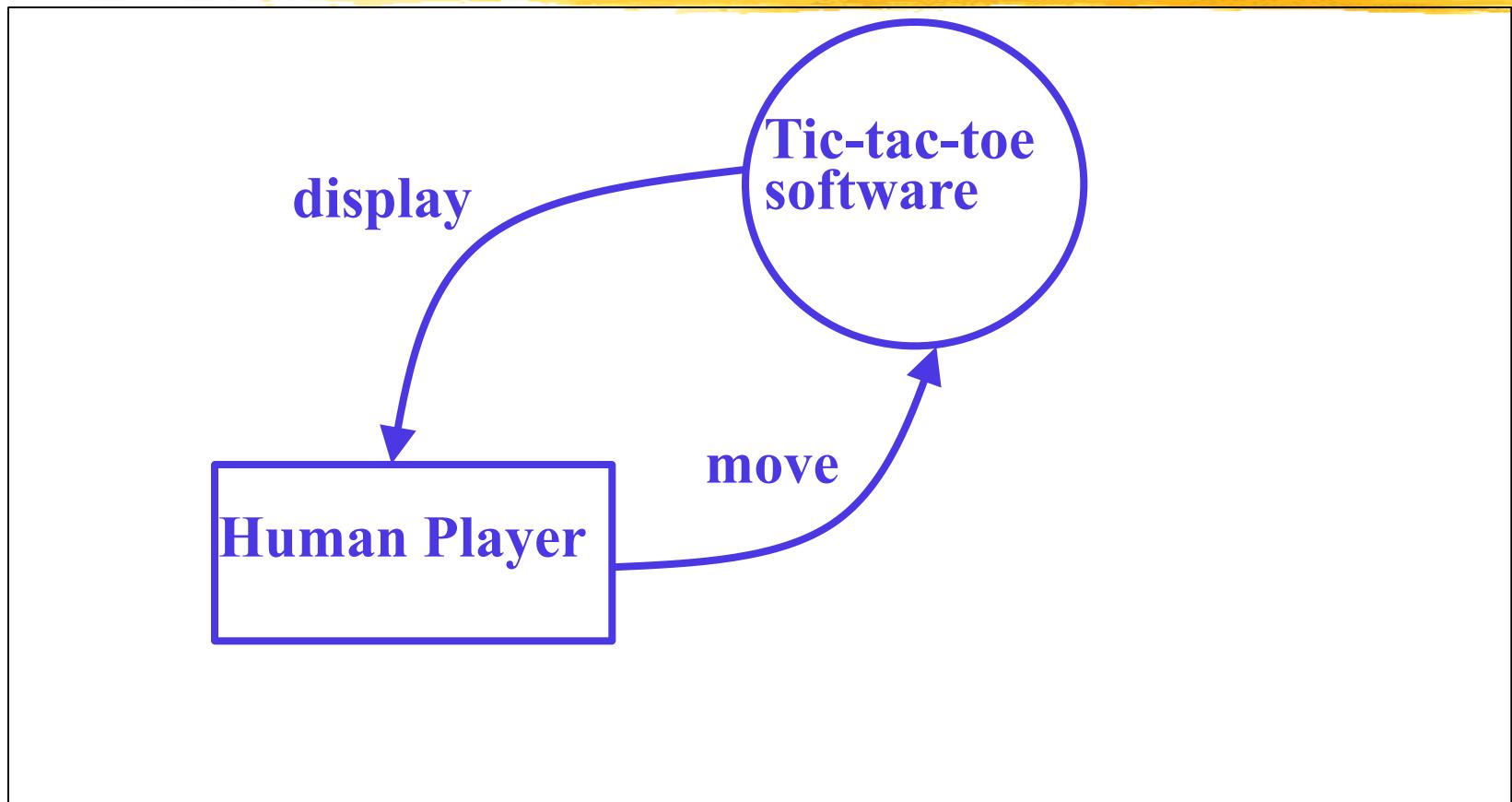
- ⌘ The notations that we would be following are closer to the Yourdon's notations
- ⌘ You may sometimes find notations in books that are slightly different
 - ↗ For example, the data store may look like a box with one end closed



How is Structured Analysis Performed?

- ❖ Initially represent the software at the most abstract level:
 - ❖ called the context diagram.
 - ❖ the entire system is represented as a single bubble,
 - ❖ this bubble is labelled according to the main function of the system.

Tic-tac-toe: Context Diagram



Context Diagram

⌘ A context diagram shows:

- ❑ data input to the system,
- ❑ output data generated by the system,
- ❑ external entities.

Context Diagram

❖ **Context diagram captures:**

- ❖ **various entities external to the system and interacting with it.**
- ❖ **data flow occurring between the system and the external entities.**

❖ **The context diagram is also called as the level 0 DFD.**

Context Diagram

Context diagram

 establishes the context of the system, i.e.

 represents:

 Data sources

 Data sinks.

Level 1 DFD

 Examine the SRS document:

-  Represent each high-level function as a bubble.
-  Represent data input to every high-level function.
-  Represent data output from every high-level function.

Higher level DFDs

- ⌘ Each high-level function is separately decomposed into subfunctions:
 - ⌘ identify the subfunctions of the function
 - ⌘ identify the data input to each subfunction
 - ⌘ identify the data output from each subfunction
- ⌘ These are represented as DFDs.

Decomposition

❖ Decomposition of a bubble:

❖ also called **factoring** or **exploding**.

❖ Each bubble is decomposed to

❖ between 3 to 7 bubbles.

Decomposition



- ⌘ Too few bubbles make decomposition superfluous:
 - ↗ if a bubble is decomposed to just one or two bubbles:
 - ✗ then this decomposition is redundant.

Decomposition



⌘ Too many bubbles:

- more than 7 bubbles at any level of a DFD
- make the DFD model hard to understand.

Decompose how long?

 Decomposition of a bubble should be carried on until:

 a level at which the function of the bubble can be described using a simple algorithm.

Example 1: RMS Calculating Software



⌘ Consider a software called RMS calculating software:

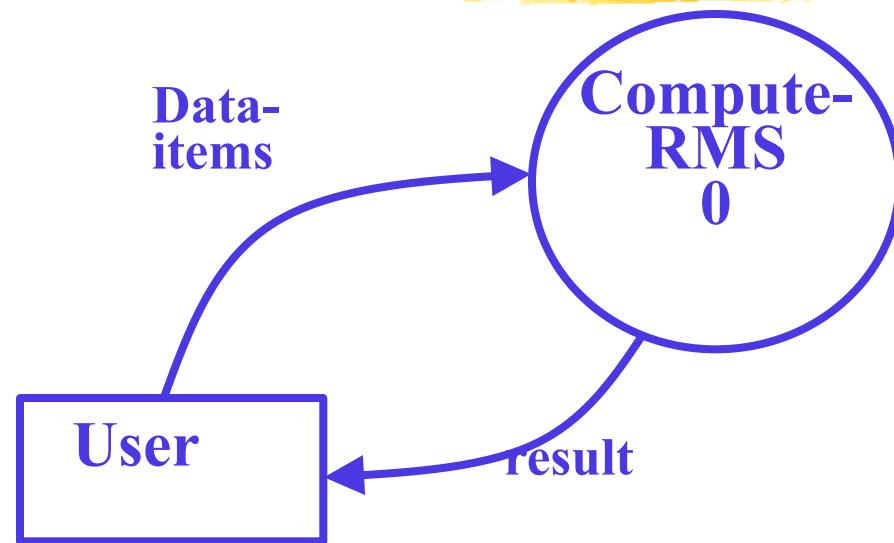
- ─ reads three integers in the range of -1000 and +1000
- ─ finds out the root mean square (rms) of the three input numbers
- ─ displays the result.

Example 1: RMS Calculating Software



- ⌘ The context diagram is simple to develop:
 - ↗ The system accepts 3 integers from the user
 - ↗ returns the result to him.

Example 1: RMS Calculating Software



Context Diagram

Example 1: RMS Calculating Software

 From a cursory analysis of the problem description:

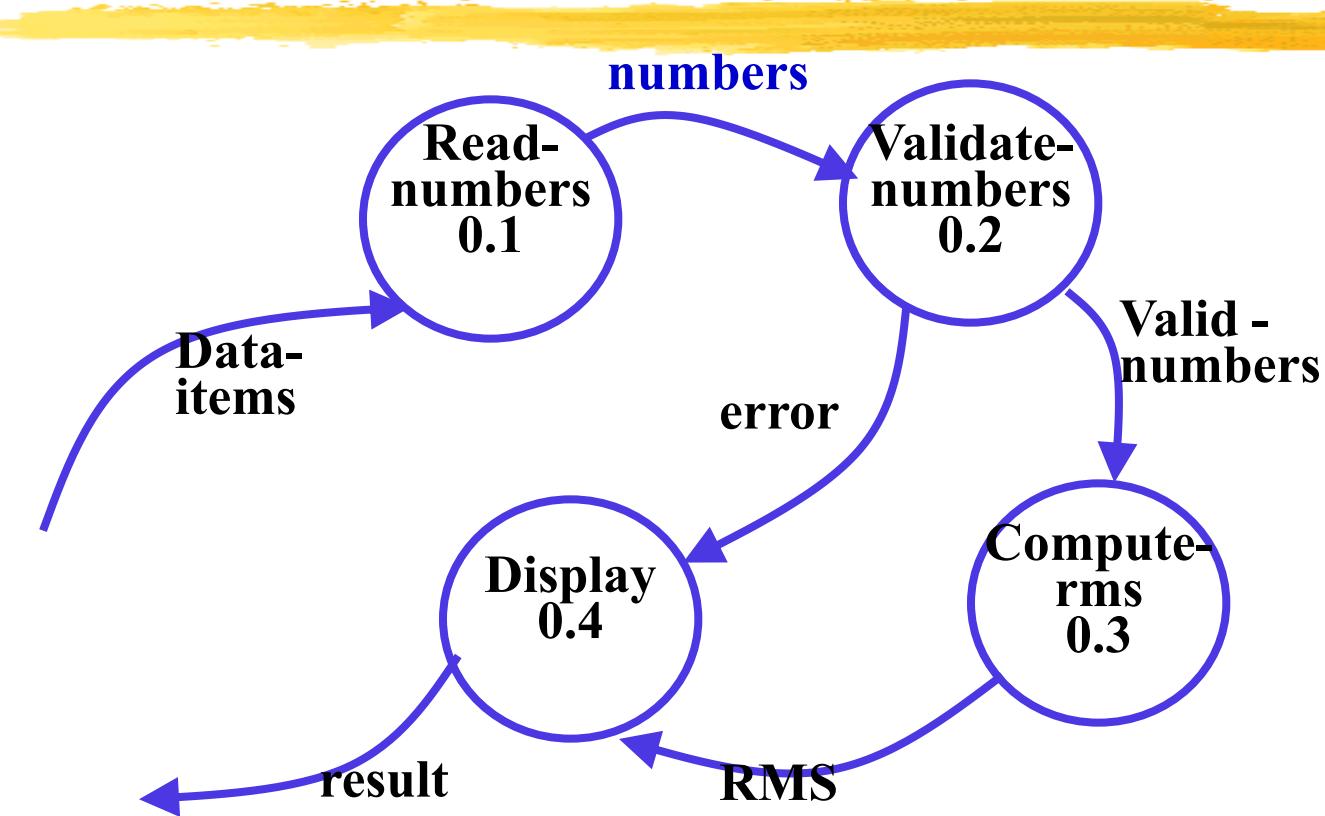
 we can see that the system needs to perform several things.

Example 1: RMS Calculating Software

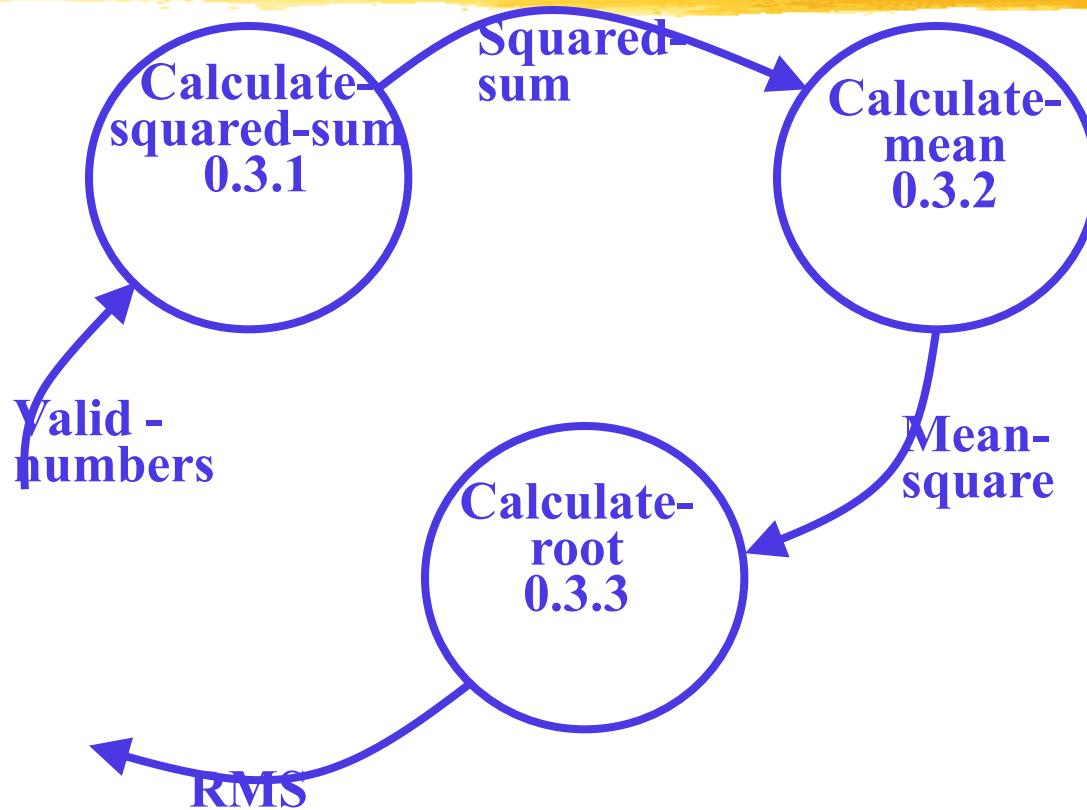
⌘ Accept input numbers from the user:

- ─ validate the numbers,
- ─ calculate the root mean square of the input numbers
- ─ display the result.

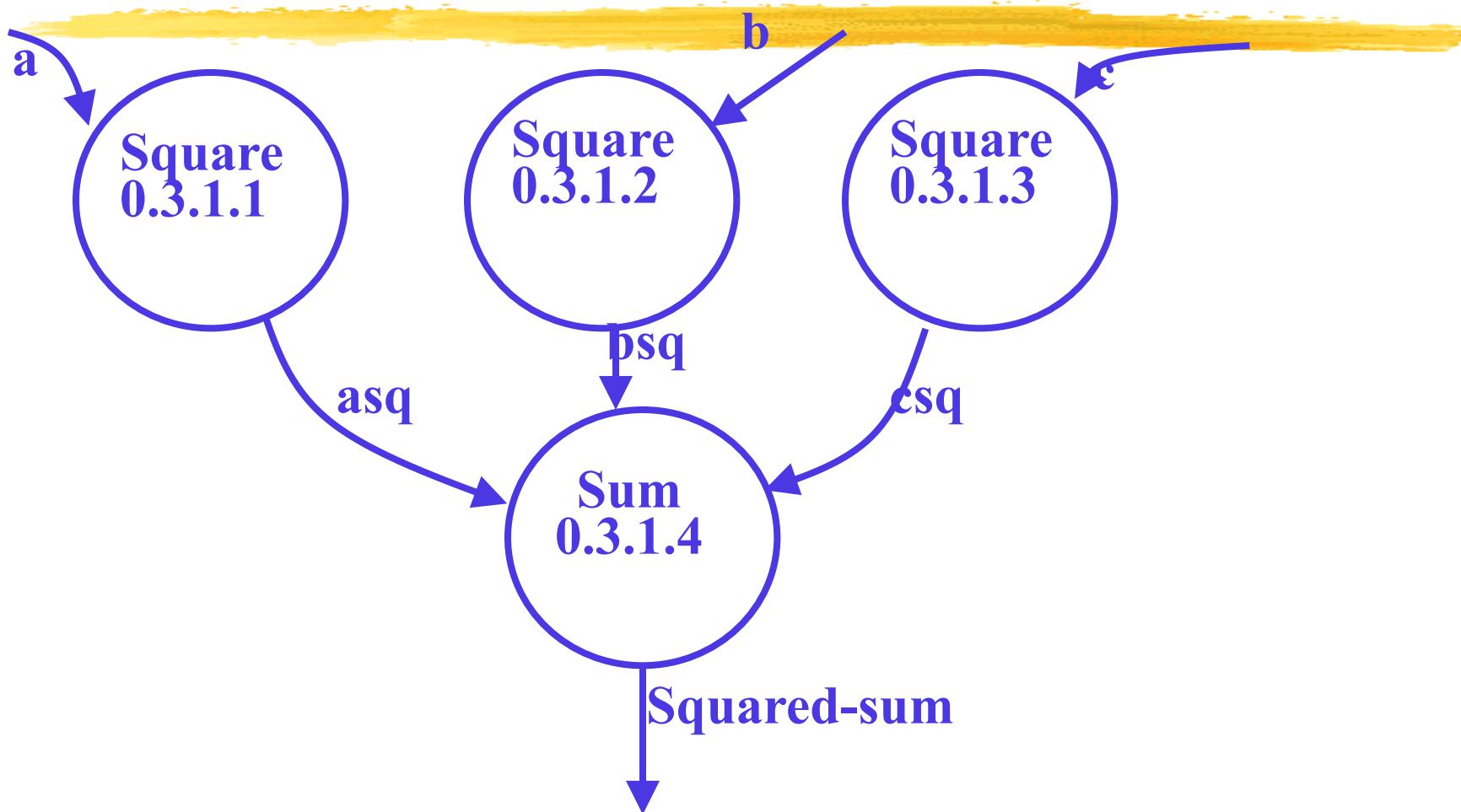
Example 1: RMS Calculating Software



Example 1: RMS Calculating Software



Example: RMS Calculating Software



Example: RMS Calculating Software

❖ Decomposition is never carried on up to basic instruction level:

- ◻ a bubble is not decomposed any further:
- ☒ if it can be represented by a simple set of instructions.

Data Dictionary

- ⌘ A DFD is always accompanied by a data dictionary.
- ⌘ A data dictionary lists all data items appearing in a DFD:
 - ↗ definition of all composite data items in terms of their component data items.
 - ↗ all data names along with the purpose of data items.
- ⌘ For example, a data dictionary entry may be:
 - ↗ **grossPay = regularPay+overtimePay**

Importance of Data Dictionary

- ⌘ Provides all engineers in a project with standard terminology for all data:
 - ↗ A consistent vocabulary for data is very important
 - ↗ different engineers tend to use different terms to refer to the same data,
 - ✖ causes unnecessary confusion.

Importance of Data Dictionary

- ⌘ **Data dictionary provides the definition of different data:**
 - ↗ **in terms of their component elements.**
- ⌘ **For large systems,**
 - ↗ **the data dictionary grows rapidly in size and complexity.**
 - ↗ **Typical projects can have thousands of data dictionary entries.**
 - ↗ **It is extremely difficult to maintain such a dictionary manually.**

Data Dictionary

⌘ CASE (Computer Aided Software Engineering) tools come handy:

↗ CASE tools capture the data items appearing in a DFD automatically to generate the data dictionary.

Data Dictionary

- ⌘ CASE tools support queries:
 - ↳ about definition and usage of data items.
- ⌘ For example, queries may be made to find:
 - ↳ which data item affects which processes,
 - ↳ a process affects which data items,
 - ↳ the definition and usage of specific data items, etc.
- ⌘ Query handling is facilitated:
 - ↳ if data dictionary is stored in a relational database management system (RDBMS).

Data Definition

⌘ Composite data are defined in terms of primitive data items using following operators:

⌘ +: denotes composition of data items,
e.g

└ a+b represents data a and b.

⌘ [,,,]: represents selection,

└ i.e. any one of the data items listed inside the square bracket can occur.

└ For example, [a,b] represents either a occurs or b occurs.

Data Definition

- ⌘(): contents inside the bracket represent optional data
 - ↗ which may or may not appear.
 - ↗ a+(b) represents either a or a+b occurs.
- ⌘ {}: represents iterative data definition,
 - ↗ e.g. {name}5 represents five name data.

Data Definition

⌘ {name}* represents

─ zero or more instances of name data.

⌘ = represents equivalence,

─ e.g. a=b+c means that a represents b and c.

⌘ * *: Anything appearing within * * is considered as comment.

Data dictionary for RMS Software

- # numbers=valid-numbers=a+b+c
- # a:integer * input number *
- # b:integer * input number *
- # c:integer * input number *
- # asq:integer
- # bsq:integer
- # csq:integer
- # squared-sum: integer
- # Result=[RMS,error]
- # RMS: integer * root mean square value*
- # error:string * error message*

Balancing a DFD

- ⌘ Data flowing into or out of a bubble:
 - ↗ must match the data flows at the next level of DFD.
 - ↗ This is known as balancing a DFD
- ⌘ In the level 1 of the DFD,
 - ↗ data item c flows into the bubble P3 and the data item d and e flow out.
- ⌘ In the next level, bubble P3 is decomposed.
 - ↗ The decomposition is balanced as data item c flows into the level 2 diagram and d and e flow out.

Balancing a DFD

⌘ Data flowing into or out of a bubble:

must match the data flows at the next level of DFD.

This is known as balancing a DFD

⌘ In the level 1 of the DFD,

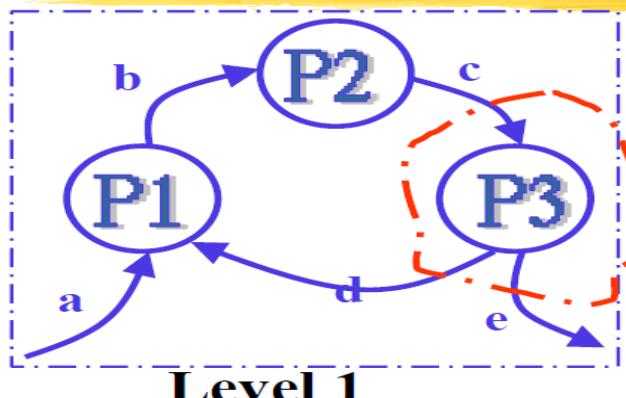
data item c flows into the bubble P3 and the data item d and e flow out.

⌘ In the next level, bubble P3 is decomposed.

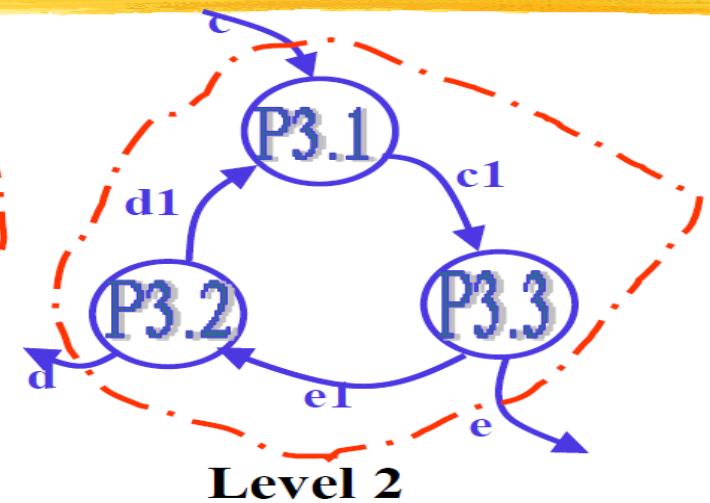
The decomposition is balanced as data item c flows into the level 2 diagram and d and e flow out.

6

Balancing a DFD

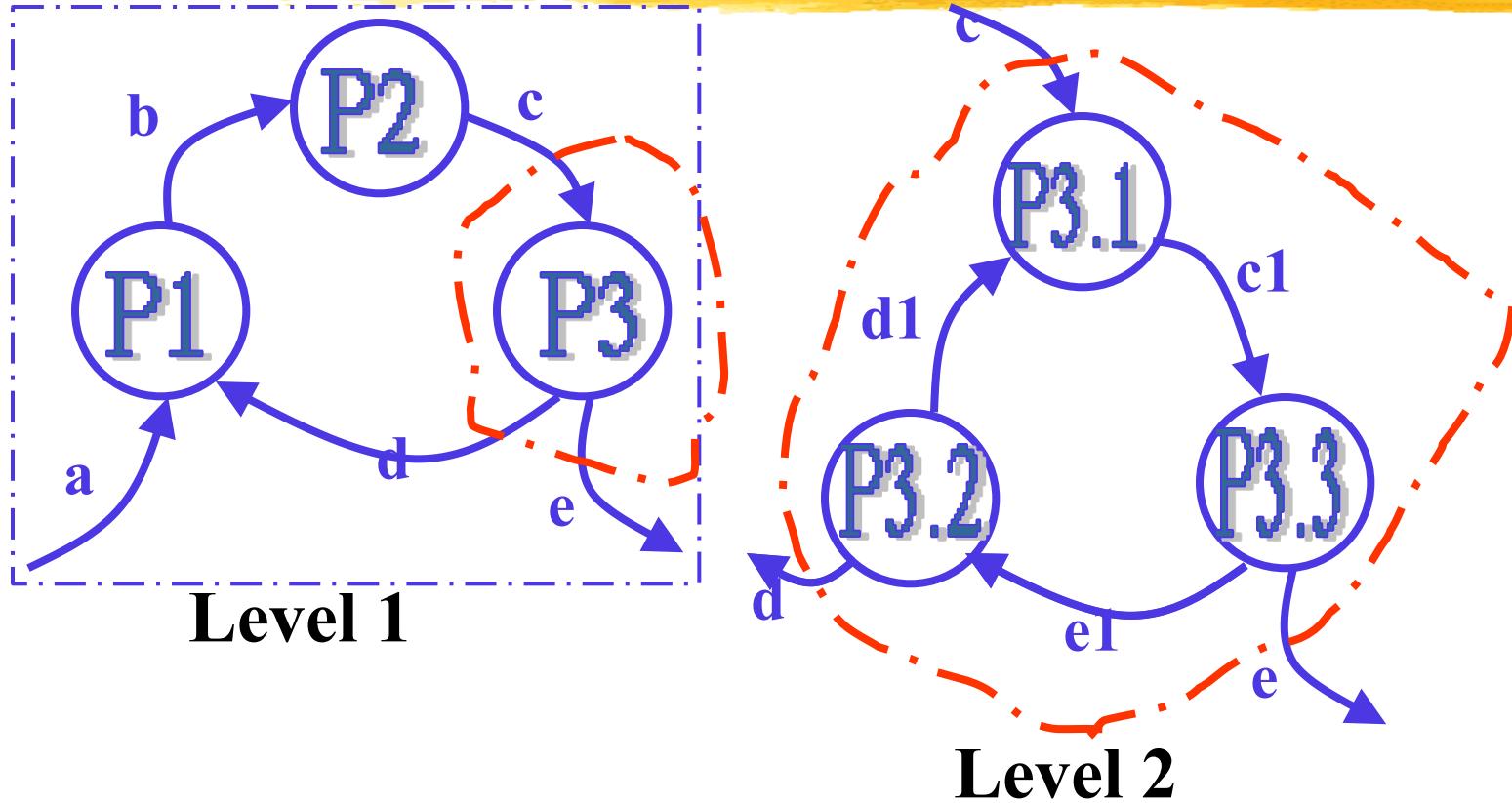


Level 1



Level 2

Balancing a DFD



Numbering of Bubbles:

- ⌘ **Number the bubbles in a DFD:**
 - ↳ numbers help in uniquely identifying any bubble from its bubble number.
- ⌘ **The bubble at context level:**
 - ↳ assigned number 0.
- ⌘ **Bubbles at level 1:**
 - ↳ numbered 0.1, 0.2, 0.3, etc
- ⌘ **When a bubble numbered x is decomposed,**
 - ↳ its children bubble are numbered x.1, x.2, x.3, etc.

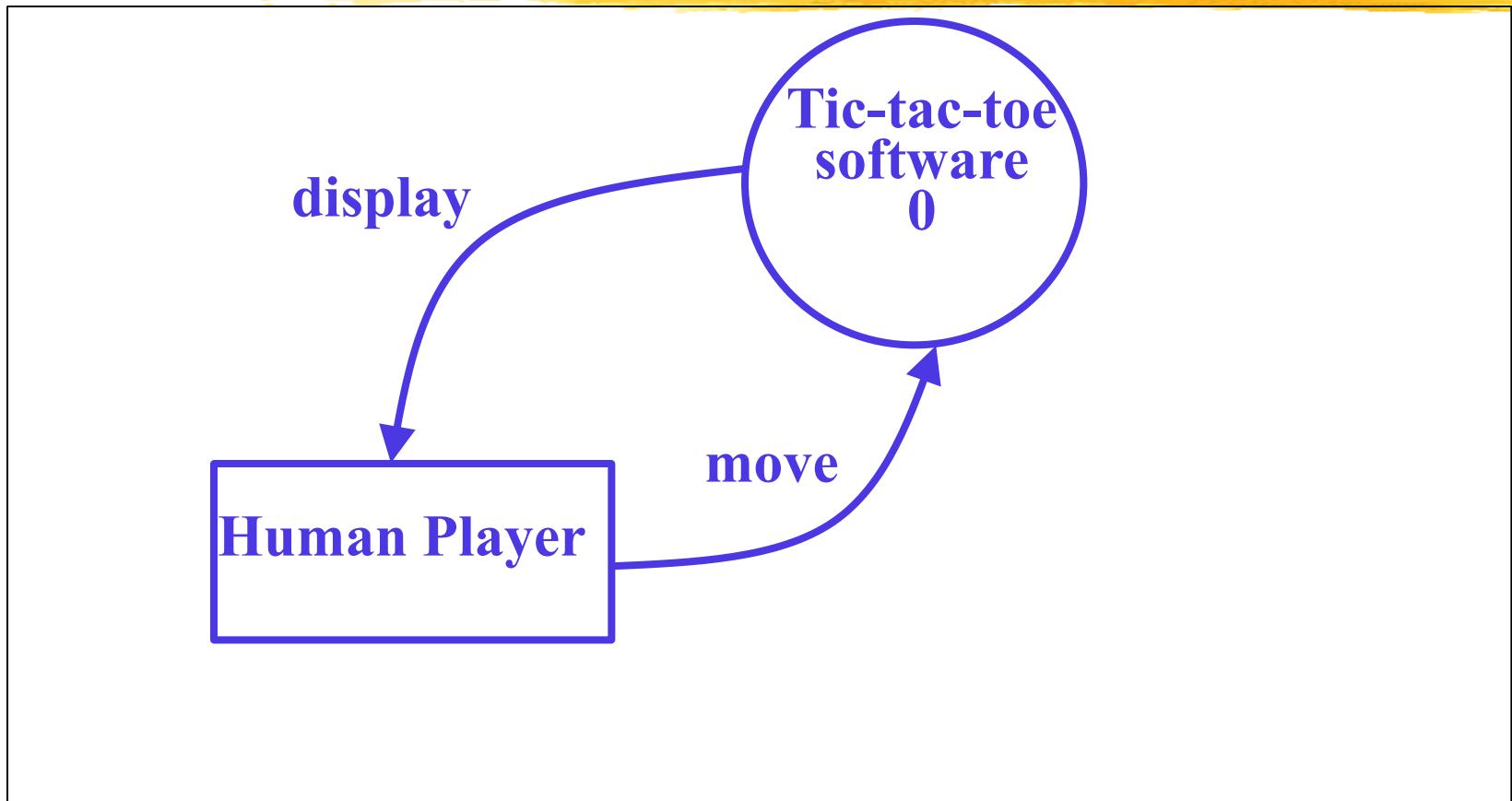
Example 2: Tic-Tac-Toe Computer Game

- # A human player and the computer make alternate moves on a 3 3 square.
- # A move consists of marking a previously unmarked square.
- # The user inputs a number between 1 and 9 to mark a square
- # Whoever is first to place three consecutive marks along a straight line (i.e., along a row, column, or diagonal) on the square wins.

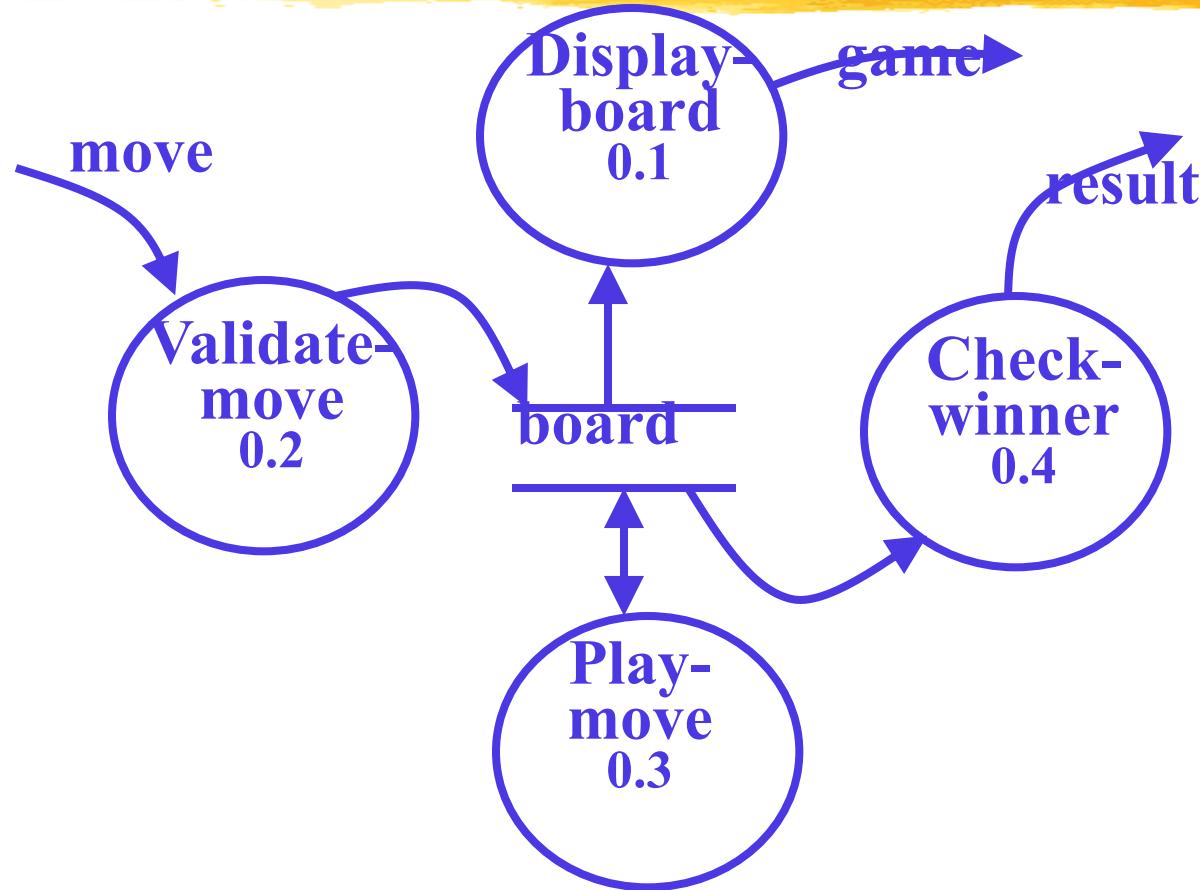
Example: Tic-Tac-Toe Computer Game

- ⌘ As soon as either of the human player or the computer wins,
 - ─ a message announcing the winner should be displayed.
- ⌘ If neither player manages to get three consecutive marks along a straight line,
 - ─ and all the squares on the board are filled up,
 - ─ then the game is drawn.
- ⌘ The computer always tries to win a game.

Context Diagram for Example



Level 1 DFD



Data dictionary



⌘ **Display=game + result**

⌘ **move = integer**

⌘ **board = {integer}9**

⌘ **game = {integer}9**

⌘ **result=string**

Summary

⌘ We discussed a sample function-oriented software design methodology:

- ⌘ Structured Analysis/Structured Design(SA/SD)

- ⌘ incorporates features from some important design methodologies.

⌘ SA/SD consists of two parts:

- ⌘ structured analysis

- ⌘ structured design.

Summary

- ⌘ The goal of structured analysis:
 - ↗ functional decomposition of the system.
- ⌘ Results of structured analysis:
 - ↗ represented using Data Flow Diagrams (DFDs).
- ⌘ We examined why any hierarchical model is easy to understand.
 - ↗ Number 7 is called the magic number.

Summary

- ⌘ During structured design,
 - ↗ the DFD representation is transformed to a structure chart representation.
- ⌘ DFDs are very popular:
 - ↗ because it is a very simple technique.

Summary

- ❖ A DFD model:
 - ❖ difficult to implement using a programming language:
 - ❖ structure chart representation can be easily implemented using a programming language.

Summary



⌘ We discussed structured analysis of two small examples:

- ❑ RMS calculating software
- ❑ tic-tac-toe computer game software

Summary

⌘ Several CASE tools are available:

- ─ support structured analysis and design.
- ─ maintain the data dictionary,
- ─ check whether DFDs are balanced or not.

Function-Oriented Software Design (continued):

Types of DFD

❖ Data Flow Diagrams are either Logical or Physical.

❖ **Logical DFD** - This type of DFD concentrates on the system process, and flow of data in the system.

❖ For example in a Banking software system, how data is moved between different entities.

❖ **Physical DFD** - This type of DFD shows how the data flow is actually implemented in the system.

❖ It is more specific and close to the implementation.

Commonly made errors

- ⌘ **Unbalanced DFDs**
- ⌘ **Forgetting to mention the names of the data flows**
- ⌘ **Unrepresented functions or data**
- ⌘ **External entities appearing at higher level DFDs**
- ⌘ **Trying to represent control aspects**
- ⌘ **Context diagram having more than one bubble**
- ⌘ **A bubble decomposed into too many bubbles in the next level**
- ⌘ **Terminating decomposition too early**
- ⌘ **Nouns used in naming bubbles**

Shortcomings of the DFD Model

- ⌘ DFD models suffer from several shortcomings:
- ⌘ DFDs leave ample scope to be imprecise.
 - ◻ In a DFD model, we infer about the function performed by a bubble from its label.
 - ◻ A label may not capture all the functionality of a bubble.

Shortcomings of the DFD Model

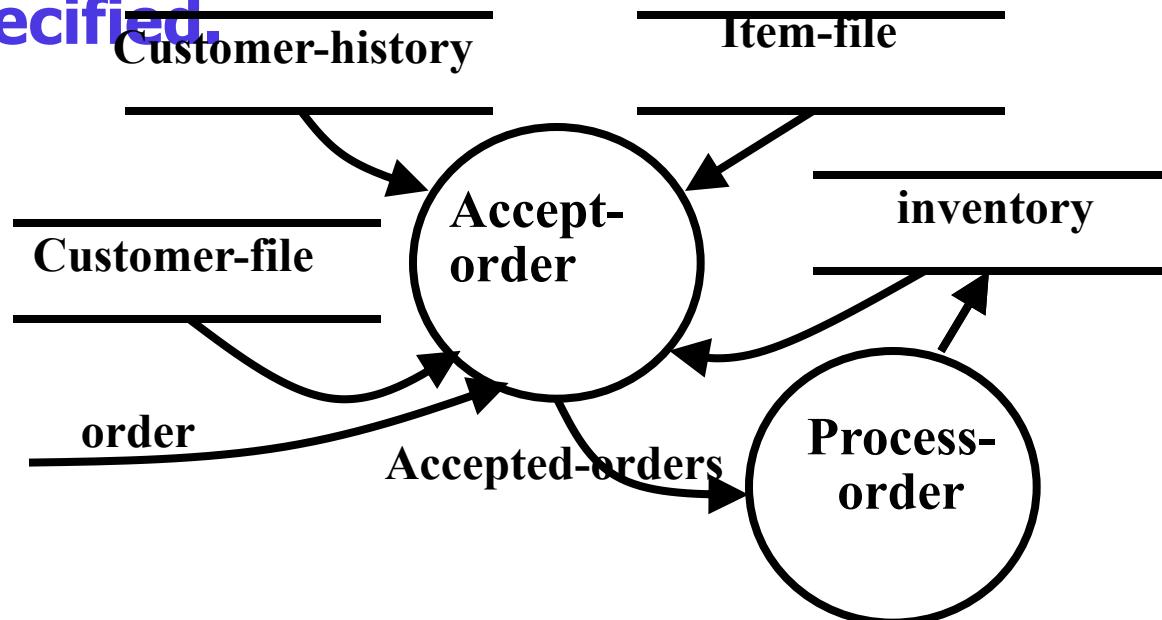
⌘ For example, a bubble named find-book-position has only intuitive meaning:

- ◻ does not specify several things:
 - ☒ what happens when some input information is missing or is incorrect.
 - ☒ Does not convey anything regarding what happens when book is not found
 - ☒ or what happens if there are books by different authors with the same book title.

Shortcomings of the DFD Model

⌘ Control information is not represented:

⌘ For instance, order in which inputs are consumed and outputs are produced is not specified.



Structured Design

- ⌘ **The aim of structured design**
 - ↗ transform the results of structured analysis (i.e., a DFD representation) into a structure chart.
- ⌘ **A structure chart represents the software architecture:**
 - ↗ various modules making up the system,
 - ↗ module dependency (i.e. which module calls which other modules),
 - ↗ parameters passed among different modules.

Structure Chart

⌘ Structure chart representation

⌘ easily implementable using programming languages.

⌘ Main focus of a structure chart:

⌘ define the module structure of a software,

⌘ interaction among different modules,

⌘ procedural aspects (e.g, how a particular functionality is achieved) are not represented.

Basic building blocks of structure chart

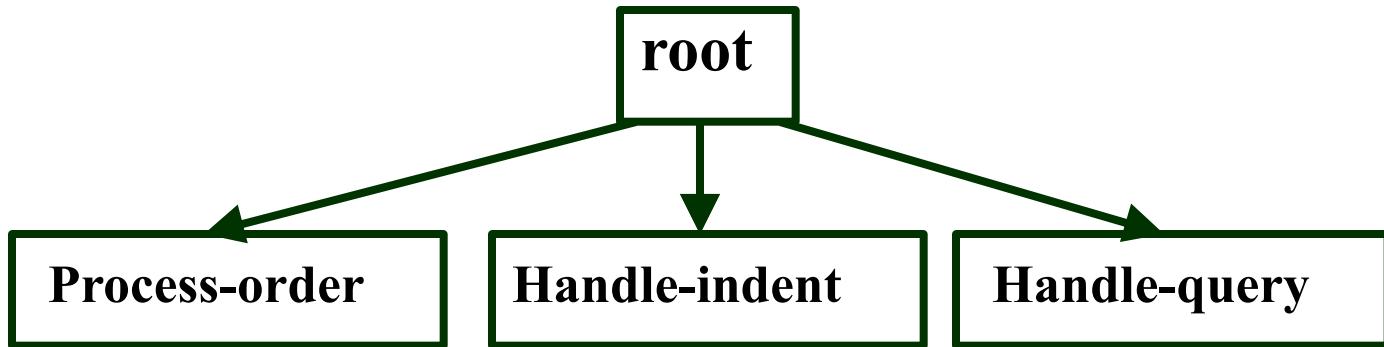
❖ Rectangular box:

- ❖ A rectangular box represents a module.
- ❖ annotated with the name of the module it represents.

Process-order

Arrows

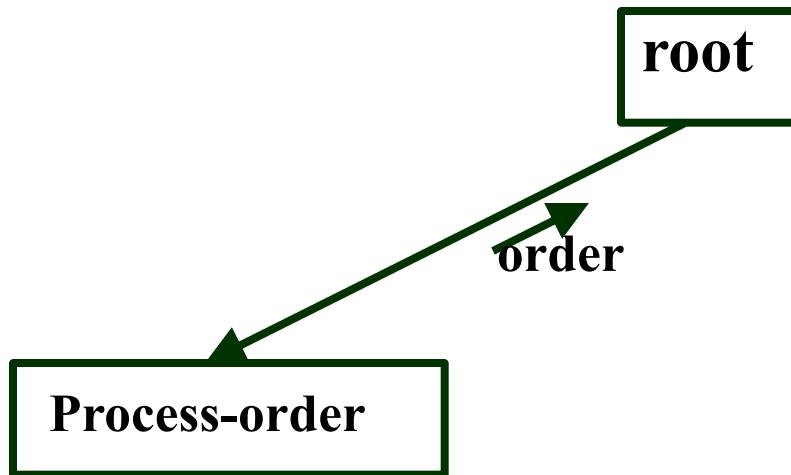
- ⌘ An arrow between two modules implies:
 - during execution control is passed from one module to the other in the direction of the arrow.



Data flow Arrows

⌘ Data flow arrows represent:

⌘ data passing from one module to another in the direction of the arrow.



Library modules

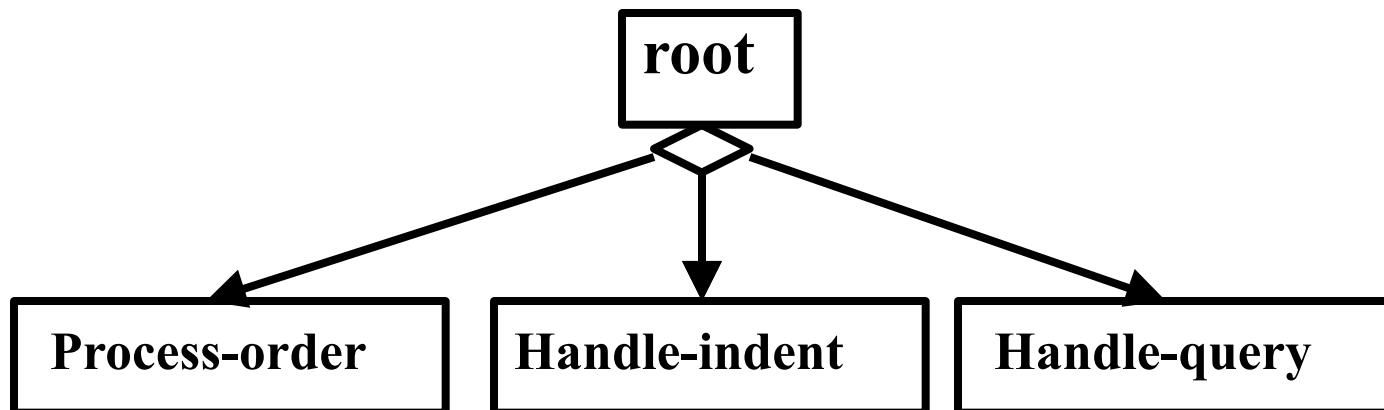
⌘ Library modules represent frequently called modules:

- a rectangle with double side edges.
- Simplifies drawing when a module is called by several modules.

Quick-sort

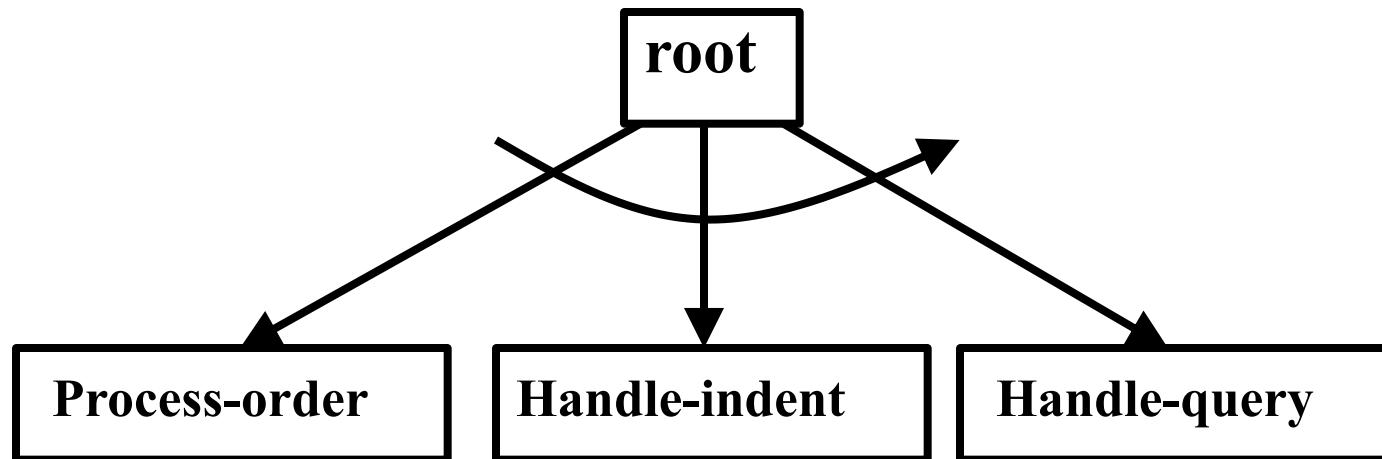
Selection

- ⌘ The diamond symbol represents:
 - one module of several modules connected to the diamond symbol is invoked depending on some condition.



Repetition

⌘ A loop around control flow arrows denotes that the concerned modules are invoked repeatedly.



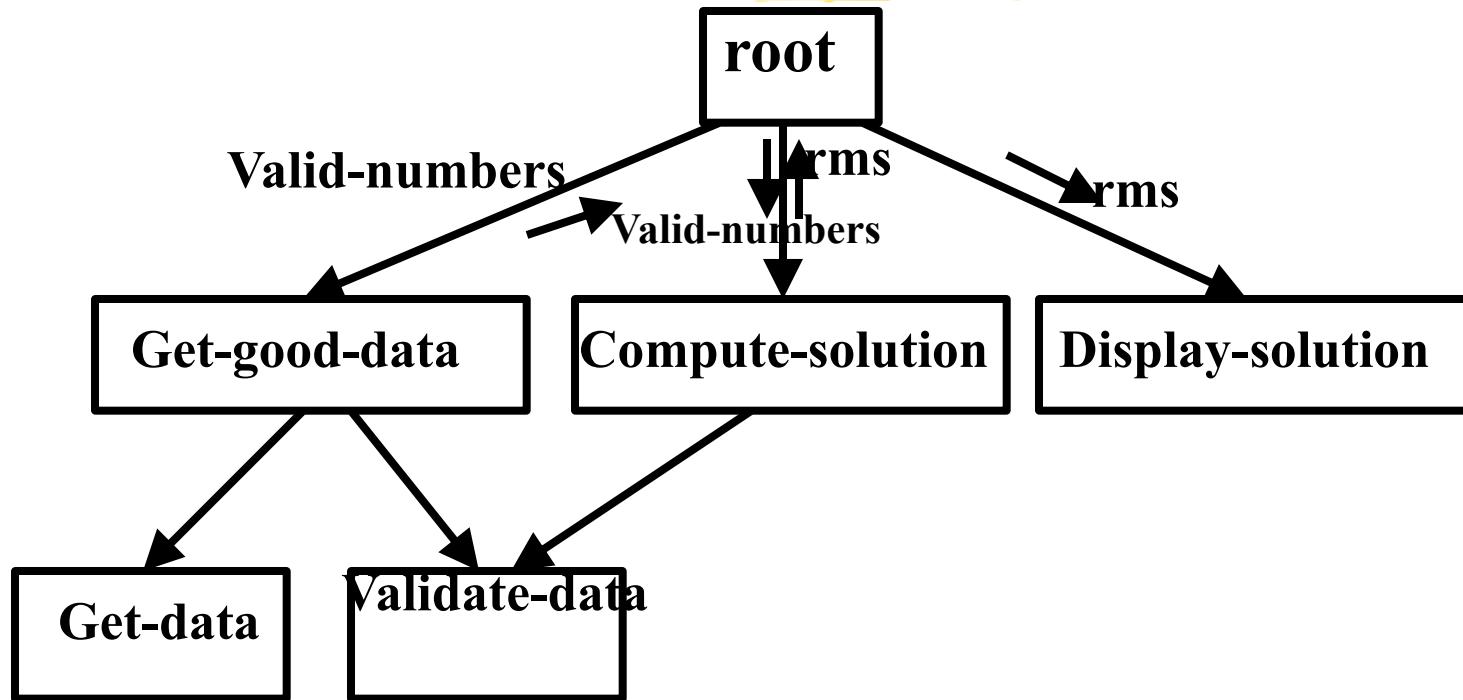
Structure Chart

- ⌘ There is only one module at the top:
 - ↗ the **root module**.
- ⌘ There is at most one control relationship between any two modules:
 - ↗ if module A invokes module B,
 - ↗ module B cannot invoke module A.
- ⌘ The main reason behind this restriction:
 - ↗ consider modules in a **structure chart** to be arranged in layers or levels.

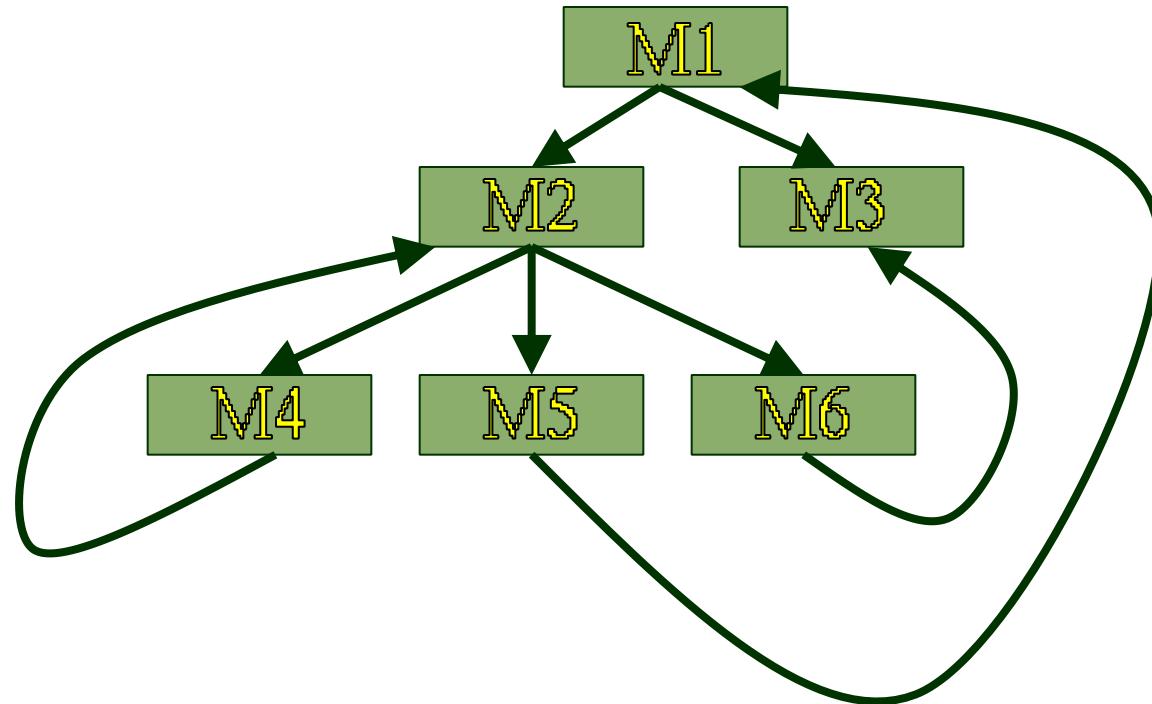
Structure Chart

- ⌘ The principle of abstraction:
 - does not allow lower-level modules to invoke higher-level modules:
 - But, two higher-level modules can invoke the same lower-level module.

Example



Bad Design



Shortcomings of Structure Chart

 **By looking at a structure chart:**

 **we can not say whether a module calls another module just once or many times.**

 **Also, by looking at a structure chart:**

 **we can not tell the order in which the different modules are invoked.**

Flow Chart versus Structure Chart

- ⌘ A structure chart differs from a flow chart in three principal ways:
 - ◻ It is difficult to identify modules of a software from its flow chart representation.
 - ◻ Data interchange among the modules is not represented in a flow chart.
 - ◻ Sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart.

Software Testing



Organization of this Lecture



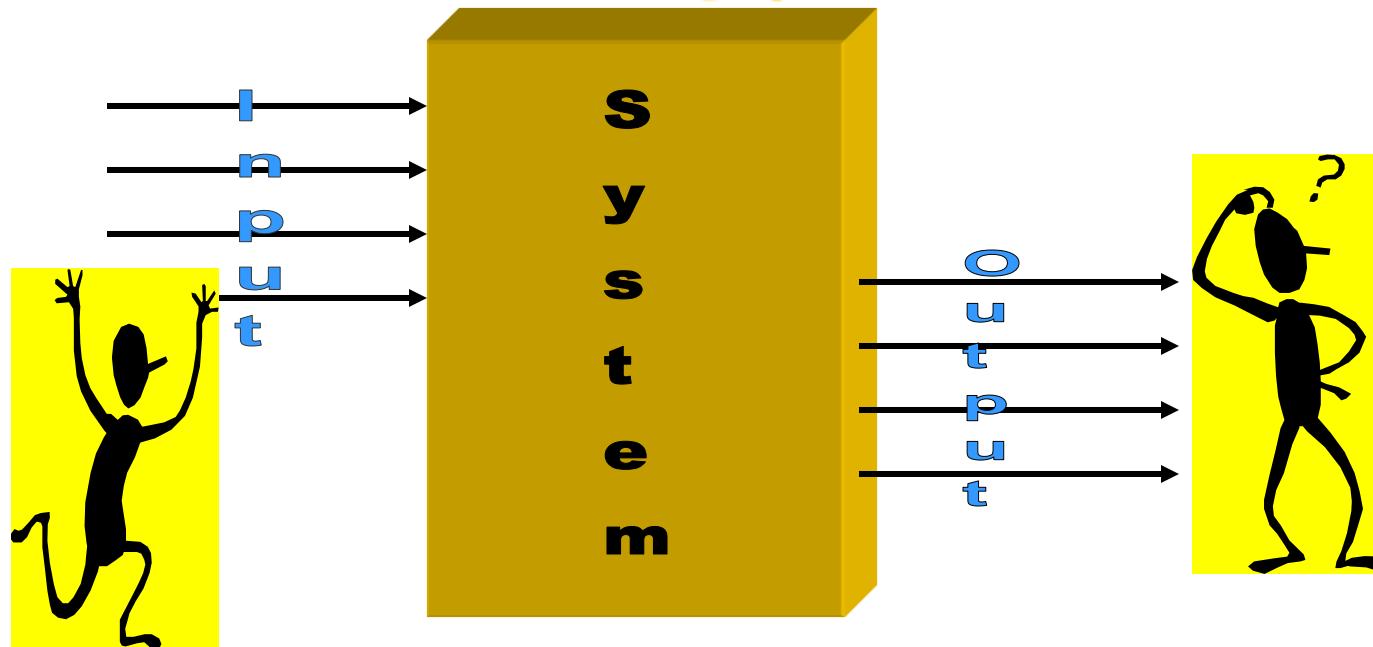
- ⌘ Introduction to Testing.
- ⌘ White-box testing:
 - ⌘ statement coverage
 - ⌘ path coverage
 - ⌘ branch testing
 - ⌘ condition coverage
 - ⌘ Cyclomatic complexity
- ⌘ Summary

How do you test a system?



- # Input test data to the system.
- # Observe the output:
 - ↗ Check if the system behaved as expected.

How do you test a system?



How do you test a system?

- ❖ If the program does not behave as expected:
 - ❑ note the conditions under which it failed.
 - ❑ later debug and correct.

Errors and Failures

⌘ A failure is a event of an error (aka defect or bug).

⚠ mere presence of an error may not lead to a failure.

Test cases and Test suite

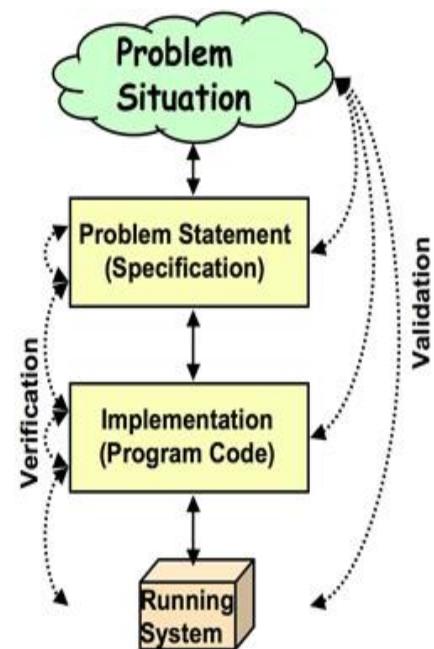


- ⌘ Test a software using a set of carefully designed test cases:
 - ↗ the set of all test cases is called the test suite
- ⌘ A test case is a triplet [I,S,O]:
 - ↗ I is the data to be input to the system,
 - ↗ S is the state of the system at which the data is input,
 - ↗ O is the expected output from the system.

Verification versus Validation

- ⌘ Verification is the process of determining:
 - ↗ whether output of one phase of development conforms to its previous phase.

- ⌘ Validation is the process of determining
 - ↗ whether a fully developed system conforms to its SRS document.



Verification versus Validation



⌘ Aim of Verification:

- ↗ phase containment of errors
- ↗ are we doing right?

⌘ Aim of validation:

- ↗ final product is error free.
- ↗ have we done right?

Design of Test Cases

- # Exhaustive testing of any non-trivial system is impractical:
 - ↗ input data domain is extremely large.
- # Design an optimal test suite:
 - ↗ of reasonable size
 - ↗ to uncover as many errors as possible.

Design of Test Cases

- ⌘ If test cases are selected randomly:
 - ↗ many test cases do not contribute to the significance of the test suite,
 - ↗ do not detect errors not already detected by other test cases in the suite.

- ⌘ The number of test cases in a randomly selected test suite:
 - ↗ not an indication of the effectiveness of the testing.

Design of Test Cases

- ⌘ Testing a system using a large number of randomly selected test cases:
 - ▣ does not mean that many errors in the system will be uncovered.
- ⌘ Consider an example:
 - ▣ finding the maximum of two integers x and y .
- ⌘ If $(x > y)$ max = x ;
 else max = x ;
- ⌘ The code has a simple error:
- ⌘ test suite $\{(x=3,y=2);(x=2,y=3)\}$ can detect the error,
- ⌘ a larger test suite $\{(x=3,y=2);(x=4,y=3); (x=5,y=1)\}$ does not detect the error.

Testing in the large vs. testing in the small

- ⌘ Software products are normally tested first at the individual component (or unit) level. This is referred to as testing in the small.
- ⌘ After testing all the components individually, the components are slowly integrated and tested at each level of integration (integration testing). Finally, the fully integrated system is tested (called system testing) or testing at large.

Unit testing

- ⌘ Unit testing is undertaken after a module has been coded and successfully reviewed. Unit testing (or module testing) is the testing of different units (or modules) of a system in isolation.

Design of Test Cases



- ⌘ Systematic approaches are required to design an optimal test suite:
 - ↗ each test case in the suite should detect different errors.
- ⌘ Two main approaches to design test cases:
 - ↗ Black-box approach
 - ↗ White-box (or glass-box) approach

Black-box Testing

- ⌘ Test cases are designed using only functional specification of the software:
 - ◻ without any knowledge of the internal structure of the software.
- ⌘ For this reason, black-box testing is also known as functional testing.

White-box Testing

 Designing white-box test cases:

-  requires knowledge about the internal structure of software.
-  white-box testing is also called structural testing.

Black-box Testing



- ⌘ Two main approaches to design black box test cases:
 - ─ Equivalence class partitioning
 - ─ Boundary value analysis

Equivalence Class Partitioning

- ⌘ It is a software testing technique that divides the input test data of the application under test into each partition at least once of equivalent data from which test cases can be derived.
- ⌘ An advantage of this approach is it reduces the time required for performing testing of a software due to less number of test cases.
- ⌘ Example: Assume that the application accepts an integer in the range 100 to 999
 - ▣ Valid Equivalence Class partition: 100 to 999 inclusive.
 - ▣ Non-valid Equivalence Class partitions: less than 100, more than 999, decimal numbers and alphabets/non-numeric characters.

Boundary value analysis

- # Boundary value analysis is a type of black box or specification based testing technique in which tests are performed using the boundary values.
- # Example: An exam has a pass boundary at 50 percent, merit at 75 percent and distinction at 85 percent.
 - ↗ The Valid Boundary values for this scenario will be as follows:
 - ↗ 49, 50 - for pass
 - ↗ 74, 75 - for merit
 - ↗ 84, 85 - for distinction
- # Boundary values are validated against both the valid boundaries and invalid boundaries.
- # The Invalid Boundary Cases for the above example can be given as follows:
 - ↗ 0 - for lower limit boundary value
 - ↗ 101 - for upper limit boundary value

White-Box Testing



⌘ There exist several popular white-box testing methodologies:

- ─ Statement coverage
- ─ branch coverage
- ─ path coverage
- ─ condition coverage
- ─ mutation testing
- ─ data flow-based testing

Statement Coverage

- ⌘ In programming language, **statement is the line of code or instruction for the computer to understand** and act accordingly.
- ⌘ A statement becomes an executable statement when it gets compiled and converted into the object code and performs the action when the program is in running mode.
- ⌘ Hence “Statement Coverage”, as the name suggests, is the method of validating that each and every line of code is executed at least once.

Statement Coverage

- ⌘ Statement coverage methodology:
 - ↗ design test cases so that
 - ↙ every statement in a program is executed at least once.
- ⌘ The principal idea:
 - ↗ unless a statement is executed,
 - ↗ we have no way of knowing if an error exists in that statement.

Statement coverage criterion

- ⌘ Based on the observation:
 - ↗ an error in a program can not be discovered:
 - ✗ unless the part of the program containing the error is executed.
- ⌘ Observing that a statement behaves properly for one input value:
 - ↗ no guarantee that it will behave correctly for all input values.

Example

```
#include <iostream.h>
int f1(int x, int y){
    while (x != y){
        if (x>y) then
            x=x-y;
        else y=y-x;
    }
    return x;
}
```

Euclid's GCD Algorithm

By choosing the test set

$\{(x=3,y=3), (x=4,y=3), (x=3,y=4)\}$

all statements are executed at least once.

Branch Coverage

- ❖ Branch Coverage is a testing method which ensures that each branch from each decision point is executed. (e.g., if statements, loops)
- ❖ So in Branch coverage (also called Decision coverage), we validate that each branch is executed at least once.
- ❖ In case of a “IF statement”, there will be two test conditions:
 - ❖ One to validate the **true** branch and
 - ❖ Other to validate the **false** branch

Branch testing

- ⌘ Branch testing is the simplest condition testing strategy:
 - ↗ compound conditions appearing in different branch statements
 - ↗ are given true and false values.
- ⌘ Condition testing
 - ↗ stronger testing than branch testing:
- ⌘ Branch testing
 - ↗ stronger than statement coverage testing.

Branch Coverage

- ⌘ Test cases are designed such that:
 - ↗ different branch conditions
 - ☒ given true and false values in turn.
- ⌘ Branch testing guarantees statement coverage:
 - ↗ a stronger testing compared to the statement coverage-based testing.

Stronger testing

- ⌘ Test cases are a superset of a weaker testing:
 - ◻ discovers at least as many errors as a weaker testing
 - ◻ contains at least as many significant test cases as a weaker test.

Condition Coverage

- ⌘ Test cases are designed such that:
 - ↳ each component of a composite conditional expression
 - ☒ given both true and false values.
- ⌘ Consider the conditional expression
 - ↳ $((c1.\text{and}.c2).\text{or}.c3)$:
- ⌘ Each of $c1$, $c2$, and $c3$ are exercised at least once,
 - ↳ i.e. given true and false values.

Path Coverage



⌘ Design test cases such that:

─ all linearly independent paths in the program are executed at least once.

Path coverage

- ⌘ Path coverage tests all the paths of the program. This is a comprehensive technique which ensures that all the paths of the program are traversed at least once.
- ⌘ Path Coverage is even more powerful than Branch coverage. This technique is useful for testing the complex programs.

Example

```
1 INPUT A & B
2 C = A + B
3 IF C>100
4 PRINT "IT'S DONE"
```

- ⌘ For **Statement Coverage** – we would need only one test case to check all the lines of code.
- ⌘ If consider *TestCase_01* to be ($A=40$ and $B=70$), then all the lines of code will be executed
- ⌘ Is that sufficient?
- ⌘ What if I consider my Test case as $A=33$ and $B=45$?

- ⌘ Because Statement coverage will only cover the true side, for the pseudo code, only one test case would NOT be sufficient to test it.
- ⌘ As a tester, we have to consider the negative cases as well.
- ⌘ Hence for maximum coverage, we need to consider "**Branch Coverage**", which will evaluate the "FALSE" conditions.

So now the pseudo code becomes:

```
1 INPUT A & B  
2 C = A + B  
3 IF C>100  
4 PRINT "IT'S DONE"  
5 ELSE  
6 PRINT "IT'S PENDING"
```

- ⌘ So for Branch coverage, we would require two test cases to complete testing of this pseudo code.
 - ▣ **TestCase_01:** A=33, B=45
 - ▣ **TestCase_02:** A=25, B=80
- ⌘ With this, each and every line of code is executed at least once.
 - ▣ **Branch Coverage ensures more coverage than Statement coverage**
 - ▣ **100% Branch coverage itself means 100% statement coverage,**

- ⌘ Path coverage is used to test the complex code snippets, which basically involves loop statements or combination of loops and decision statements.

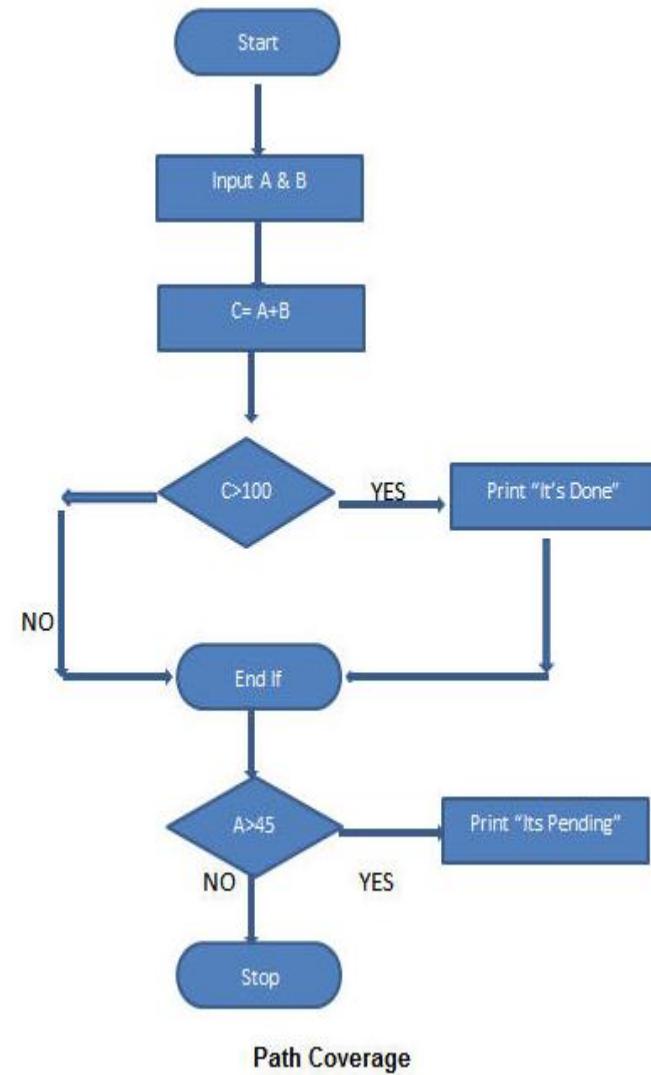
```
1 INPUT A & B  
2 C = A + B  
3 IF C>100  
4 PRINT "IT'S DONE"  
5 END IF  
6 IF A>50  
7 PRINT "IT'S PENDING"  
8 END IF
```

- ⌘ ensure maximum coverage, which require 4 test cases.

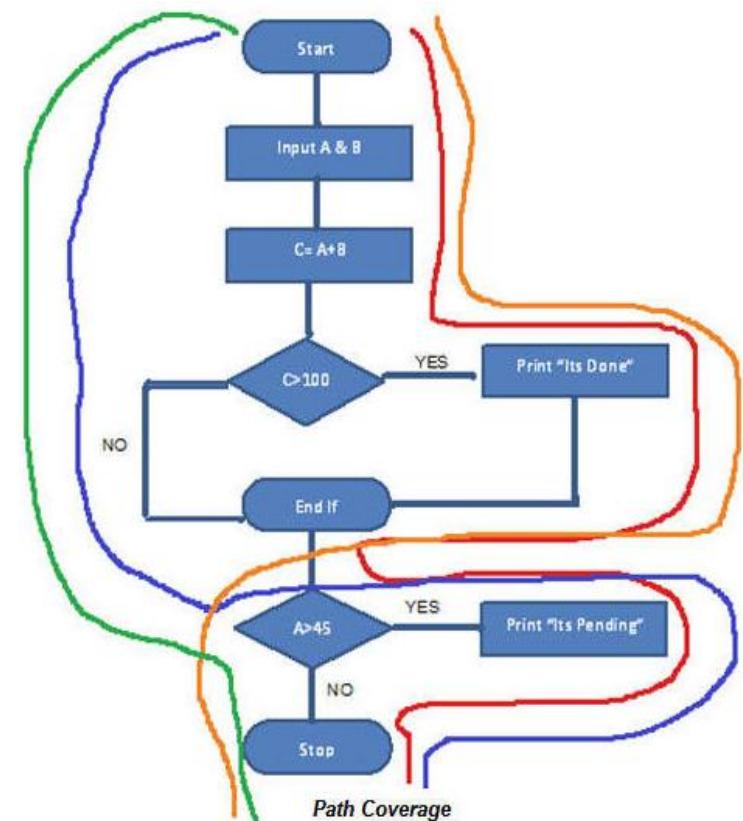
- # there are 2 decision statements, so for each decision statement we would need to branches to test. One for true and other for false condition.

- # So for 2 decision statements, require 2 test cases to test the true side and 2 test cases to test the false side, which makes total of 4 test cases.

- # So, In order to have the full coverage, following test cases are:
- # **TestCase_01:** A=50, B=60
- # **TestCase_02:** A=55, B=40
- # **TestCase_03:** A=40, B=65
- # **TestCase_04:** A=30, B=30



- # Red Line – TestCase_01 = (A=50, B=60)
- # Blue Line = TestCase_02 = (A=55, B=40)
- # Orange Line = TestCase_03 = (A=40, B=65)
- # Green Line = TestCase_04 = (A=30, B=30)



Control flow graph (CFG)

- ⌘ A control flow graph (CFG) describes:
 - ↗ the sequence in which different instructions of a program get executed.
 - ↗ the way control flows through the program.

How to draw Control flow graph?

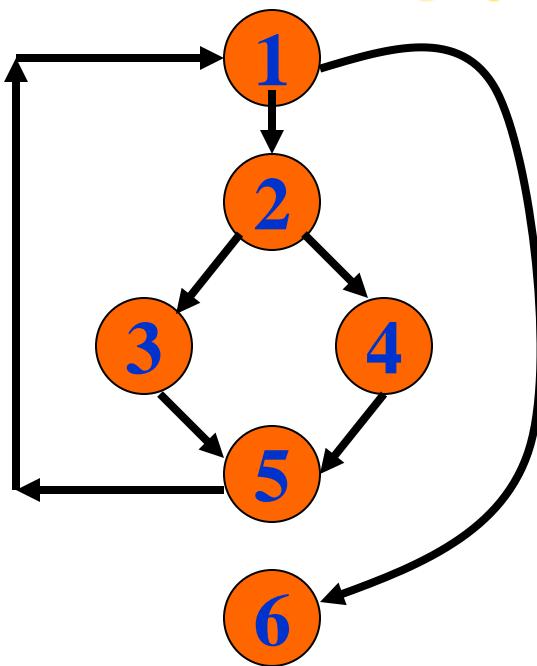
- ☒ Number all the statements of a program.
- ☒ Numbered statements:
 - ☒ represent nodes of the control flow graph
- ⌘ An edge from one node to another node exists:
 - ☒ if execution of the statement representing the first node
 - ☒ can result in transfer of control to the other node.

Example



```
#int f1(int x,int y){  
#1 while (x != y){  
#2   if (x>y) then  
#3     x=x-y;  
#4   else y=y-x;  
#5 }  
#6 return x;      }
```

Example Control Flow Graph

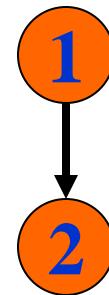


How to draw Control flow graph?

⌘ Sequence:

1 $a = 5;$

2 $b = a * b - 1;$



How to draw Control flow graph?

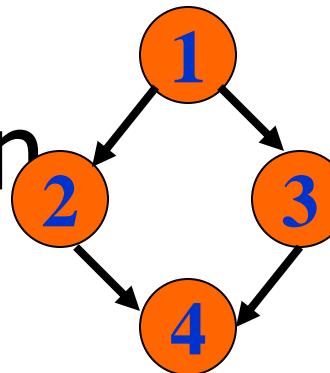
⌘ Selection:

1 if($a > b$) then

2 $c = 3;$

3 else $c = 5;$

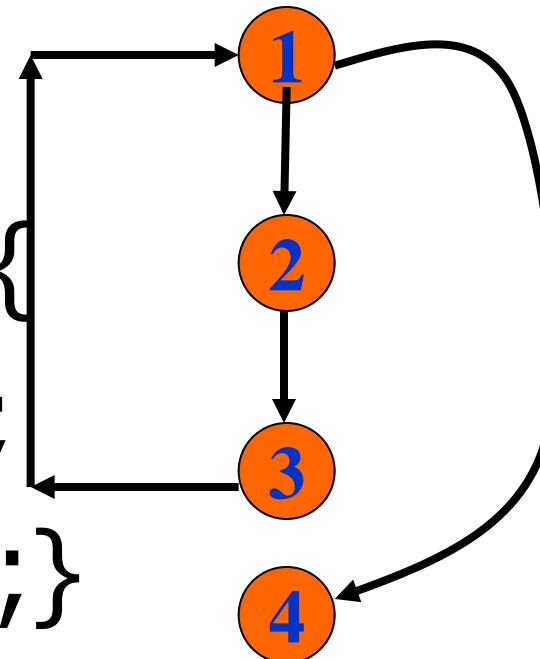
4 $c = c * c;$



How to draw Control flow graph?

⌘ Iteration:

- 1 while(a>b){
- 2 b=b*a;
- 3 b=b-1;}
- 4 c=b+d;



Path

⌘ A path through a program:

- ◻ a node and edge sequence from the starting node to a terminal node of the control flow graph.
- ◻ There may be several terminal nodes for program.

Independent path

⌘ Any path through the program:

 └ introducing at least one new node:

 └ that is not included in any other independent paths.

⌘ It is straight forward:

 └ to identify linearly independent paths of simple programs.

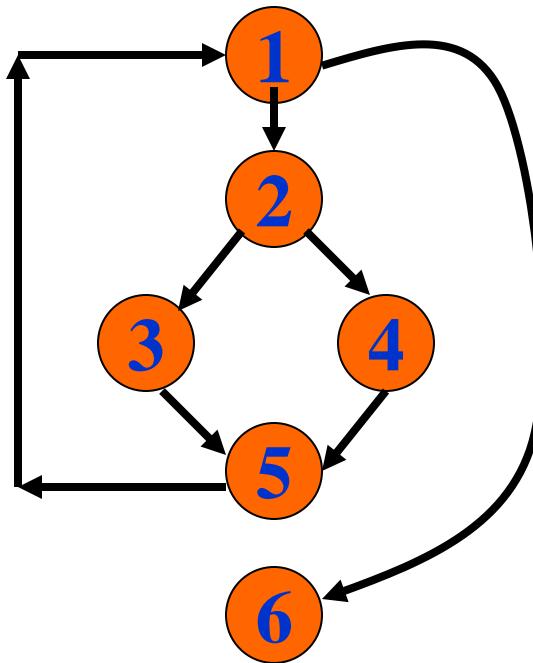
⌘ For complicated programs:

 └ it is not so easy to determine the number of independent paths.

McCabe's cyclomatic metric

- ⌘ An upper bound:
 - ℳ for the number of linearly independent paths of a program
- ⌘ Provides a practical way of determining:
 - ℳ the maximum number of linearly independent paths in a program.
- ⌘ Given a control flow graph G, cyclomatic complexity $V(G)$:
 - ℳ $V(G) = E - N + 2$
 - ℳ N is the number of nodes in G
 - ℳ E is the number of edges in G

Example Control Flow Graph

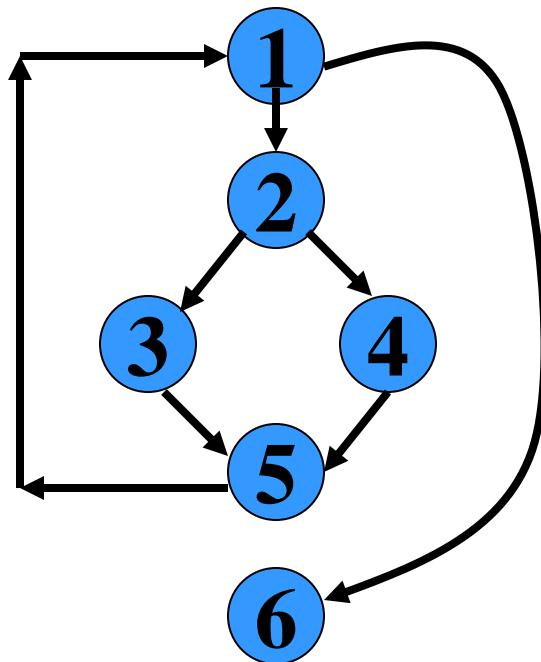


Cyclomatic complexity = $7-6+2 = 3$.

Cyclomatic complexity

- ⌘ Another way of computing cyclomatic complexity:
 - ⌘ inspect control flow graph
 - ⌘ determine number of bounded areas in the graph
 - ⌘ **bounded areas:** Any region enclosed by a nodes and edge sequence.
- ⌘ $V(G) = \text{Total number of bounded areas} + 1$

Example Control Flow Graph



Example



- ⌘ From a visual examination of the CFG:
 - ⌘ the number of bounded areas is 2.
 - ⌘ cyclomatic complexity = $2+1=3$.

Cyclomatic complexity

- ⌘ McCabe's metric provides:
 - ☒ a quantitative measure of testing difficulty and the ultimate reliability
- ⌘ Intuitively,
 - ☒ number of bounded areas increases with the number of decision nodes and loops.
- ⌘ Knowing the number of test cases required:
 - ☒ does not make it any easier to derive the test cases,
 - ☒ only gives an indication of the minimum number of test cases required.

Condition coverage

- ⌘ In this structural testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values.
- ⌘ For example, in the conditional expression $((c1.\text{and}.c2).\text{or}.c3)$, the components $c1$, $c2$ and $c3$ are each made to assume both true and false values.
- ⌘ For a composite conditional expression of n components, for condition coverage, 2^n test cases are required.
- ⌘ The number of test cases increases exponentially with the number of component conditions. It is practical only if n (the number of conditions) is small.

Mutation testing

- # The software is first tested by using an initial test suite built up from the different white box testing strategies.
- # After the initial testing is complete, mutation testing is taken up.
- # The goal of **Mutation Testing** is ensuring the quality of test cases in terms of robustness that it should fail the mutated source code.
- # Idea behind mutation testing is to make few arbitrary changes to a program at a time.
 - ▣ Each time the program is changed, it is called as a **mutated program** and the change effected is called as a **mutant**.
 - ▣ A mutated program is tested against the full test suite of the program.
 - ▣ If there exists at least one test case in the test suite for which a mutant gives an **incorrect result, then the mutant is said to be dead**.
 - ▣ If a mutant remains alive even after all the test cases have been exhausted, the test data is enhanced to kill the mutant.
- # Disadvantage is that it is computationally very expensive, since a large number of possible mutants can be generated.



Software Reliability

- Software reliability,
- SEI CMM,
- Characteristics of software maintenance,
- software reverse engineering,
- software reengineering.

Software Reliability

- Reliability of a software product essentially denotes its trustworthiness or dependability.
- Reliability of a software product can also be defined as the probability of the product working “correctly” over a given period of time.
- Experimentally observed by analyzing the behavior of a large number of programs that 90% of the execution time of a typical program is spent in executing only 10% of the instructions in the program.
- These most used 10% instructions are often called the **core** of the program.
- The rest 90% of the program statements are called **non-core** and are executed only for 10% of the total execution time.

Reasons for software reliability being difficult to measure

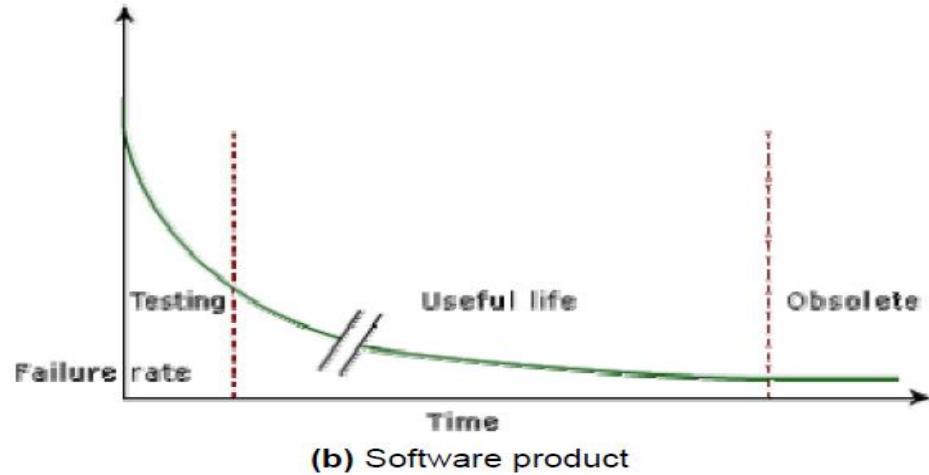
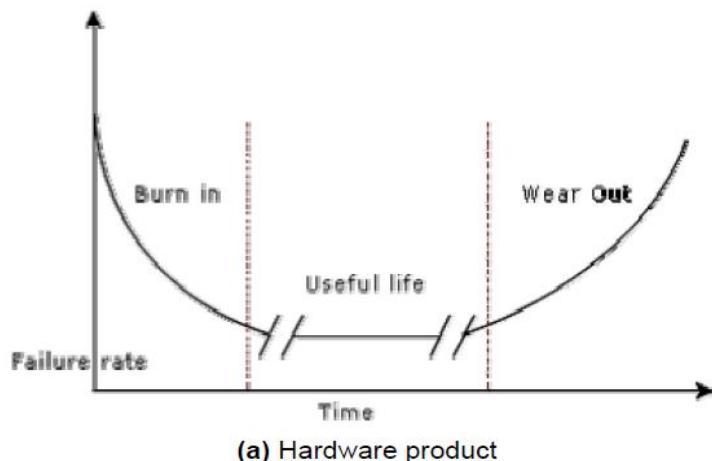
- The reliability improvement due to fixing a single bug depends on where the bug is located in the code.
- The perceived reliability of a software product is highly observer dependent.
- The reliability of a product keeps changing as errors are detected and fixed.

Hardware reliability vs. software reliability differ

- Hardware failures are due to component wear and tear. A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix hardware faults, one has to either replace or repair the failed part.
- Software product would continue to fail until the error is tracked down and either the design or the code is changed.
- For this reason, when a hardware is repaired its reliability is maintained at the level that existed before the failure occurred;
- Whereas when a software failure is repaired, the reliability may either increase or decrease (reliability may decrease if a bug introduces new errors).
- **Hardware reliability study is concerned with stability (for example, inter-failure times remain constant).**
- **Software reliability study aims at reliability growth (i.e. inter-failure times increase).**

Failure rate over the product lifetime

- For hardware products, it can be observed that failure rate is high initially but decreases as the faulty components are identified and removed.
- The system then enters its useful life. After some time (called product life time) the components wear out, and the failure rate increases.
- For software the failure rate is at it's highest during integration and test. As the system is tested, more and more errors are identified and removed resulting in reduced failure rate.
- This error removal continues at a slower pace during the useful life of the product. As the software becomes obsolete no error corrections occurs and the failure rate remains unchanged.



Reliability metrics

Six reliability metrics which can be used to quantify the reliability of software products.

- **Rate of occurrence of failure (ROCOF).** ROCOF measures the frequency of occurrence of unexpected behavior (i.e. failures).
 - ROCOF measure of a software product can be obtained by observing the behavior of a software product in operation over a specified time interval and then recording the total number of failures occurring during the interval.
- **Mean Time To Failure (MTTF).** MTTF is the average time between two successive failures, observed over a large number of failures. To measure MTTF, we can record the failure data for n failures. Let the failure time instants t_1, t_2, \dots, t_n . Then, MTTF can be calculated:
$$\sum_{i=1}^n \frac{t_{i+1}-t_i}{(n-1)}$$
.

- **Mean Time To Repair (MTTR).** Once failure occurs, some time is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and to fix them.
- **Mean Time Between Failure (MTBF).** MTTF and MTTR can be combined to get the MTBF metric: $MTBF = MTTF + MTTR$.
- Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours.
- **Probability of Failure on Demand (POFOD):** This metric does not explicitly involve time. Measures the likelihood of the system failing:
 - when a service request is made.
 - POFOD of 0.001 means: 1 out of 1000 service requests may result in a failure.
- **Availability.** Availability of a system is a measure of how likely shall the system be available for use over a given period of time.
This metric not only considers the number of failures occurring during a time interval, but also takes into account the repair time (down time) of a system when a failure occurs.

Classification of software failures

- **Transient.** Transient failures occur only for certain input values while invoking a function of the system.
- **Permanent.** Permanent failures occur for all input values while invoking a function of the system.
- **Recoverable.** When recoverable failures occur, the system recovers
 - with or without operator intervention.
- **Unrecoverable.** In unrecoverable failures, the system may need to be restarted.
- **Cosmetic.** These classes of failures cause only minor irritations, and do not lead to incorrect results.
 - An example of a cosmetic failure is the case where the mouse button has to be clicked twice instead of once to invoke a given function through the graphical user interface.

SEI CMM

- SEI Capability Maturity Model helped organizations to improve the quality of the software they develop and therefore adoption of SEI CMM model has significant business benefits.
- SEI CMM classifies software development industries into the following five maturity levels.
- **Level 1: Initial.** A software development organization at this level is characterized by ad hoc activities. Very few or no processes are defined and followed.
- **Level 2: Repeatable.** At this level, the basic project management practices such as tracking cost and schedule are established. Size and cost estimation techniques like function point analysis, COCOMO, etc. are used.

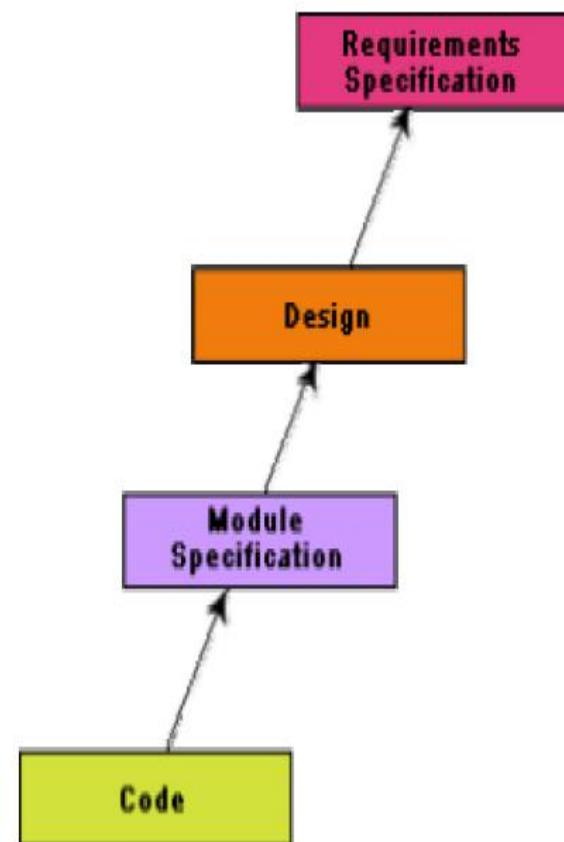
- **Level 3: Defined.** At this level the processes for both management and development activities are defined and documented.
- **Level 4: Managed.** At this level, the focus is on software metrics. Two types of metrics are collected.
 - Product metrics measure the characteristics of the product being developed, such as its size, reliability, time complexity, understandability, etc.
 - Process metrics reflect the effectiveness of the process being used, such as average defect correction time, productivity, average number of defects found per hour inspection, average number of failures detected during testing per LOC, etc.
- **Level 5: Optimizing.** At this stage, process and product metrics are collected. Process and product measurement data are analyzed for continuous process improvement.
 - For example, if from an analysis of the process measurement results, it was found that the code reviews were not very effective and a large number of errors were detected only during the unit testing, then the process may be fine tuned to make the review more effective.

Characteristics of software maintenance

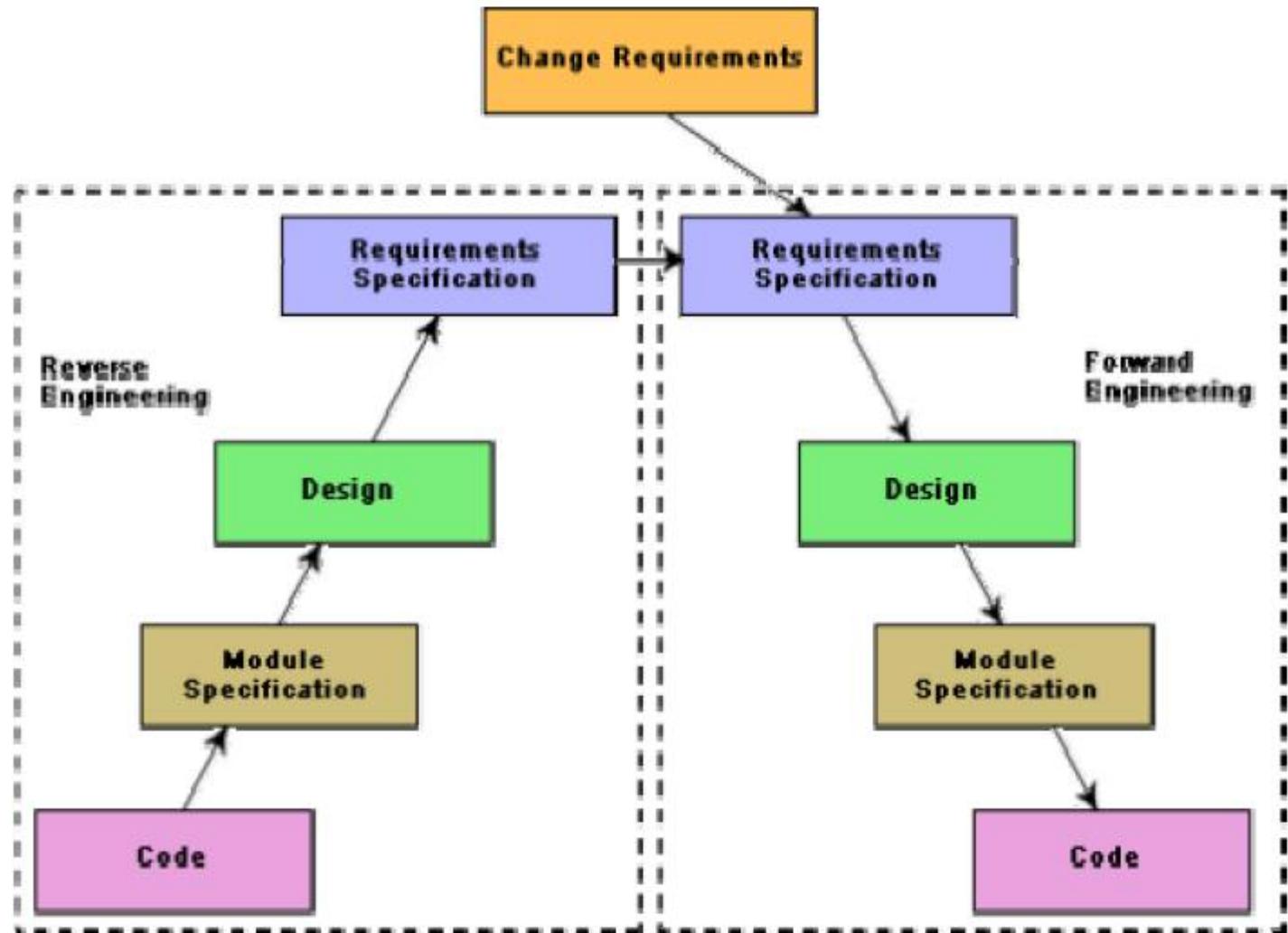
- **Corrective:** Corrective maintenance of a software product is necessary to rectify the bugs observed while the system is in use.
- **Adaptive:** A software product might need maintenance when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware or software.
- **Perfective:** A software product needs maintenance to support the new features that users want it to support, to change different functionalities of

Software reverse engineering

- Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis of its code.
- The purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system.
- Reverse engineering is becoming important, since legacy software products lack proper documentation, and are highly unstructured.
- Even well-designed products become legacy software as their structure degrades through a series of maintenance efforts.



- Software reengineering is a combination of two consecutive processes i.e. software reverse engineering and software forward engineering



What is software reuse?

Software reuse is the process of implementing or updating software systems using existing software assets.

- The systematic development of reusable components
- The systematic reuse of these components as building blocks to create new system

The advantages of reuse

- Increase software productivity
- Shorten software development time
- Improve software system interoperability
- Develop software with fewer people
- Move personnel more easily from project to project
- Reduce software development and maintenance costs
- Produce more standardized software
- Produce better quality software and provide a powerful competitive advantage

What is reusable?

- Application system
- Subsystem
- Component
- Module
- Object
- Function or Procedure

Difference between defect, error, bug, failure and fault:

"A mistake in coding is called error ,error found by tester is called defect, defect accepted by development team then it is called bug ,build does not meet the requirements then it Is failure."

- **Error:** A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. This can be a misunderstanding of the internal state of the software, an oversight in terms of memory management, confusion about the proper way to calculate a value, etc.
- **Failure:** The inability of a system or component to perform its required functions within specified performance requirements. See: bug, crash, exception, and fault.
- **Bug:** A fault in a program which causes the program to perform in an unintended or unanticipated manner. See: anomaly, defect, error, exception, and fault. Bug is terminology of Tester.
- **Fault:** An incorrect step, process, or data definition in a computer program which causes the program to perform in an unintended or unanticipated manner. See: bug, defect, error, exception.
- **Defect:** Commonly refers to several troubles with the software products, with its external behavior or with its internal features.

99% Complete Syndrome

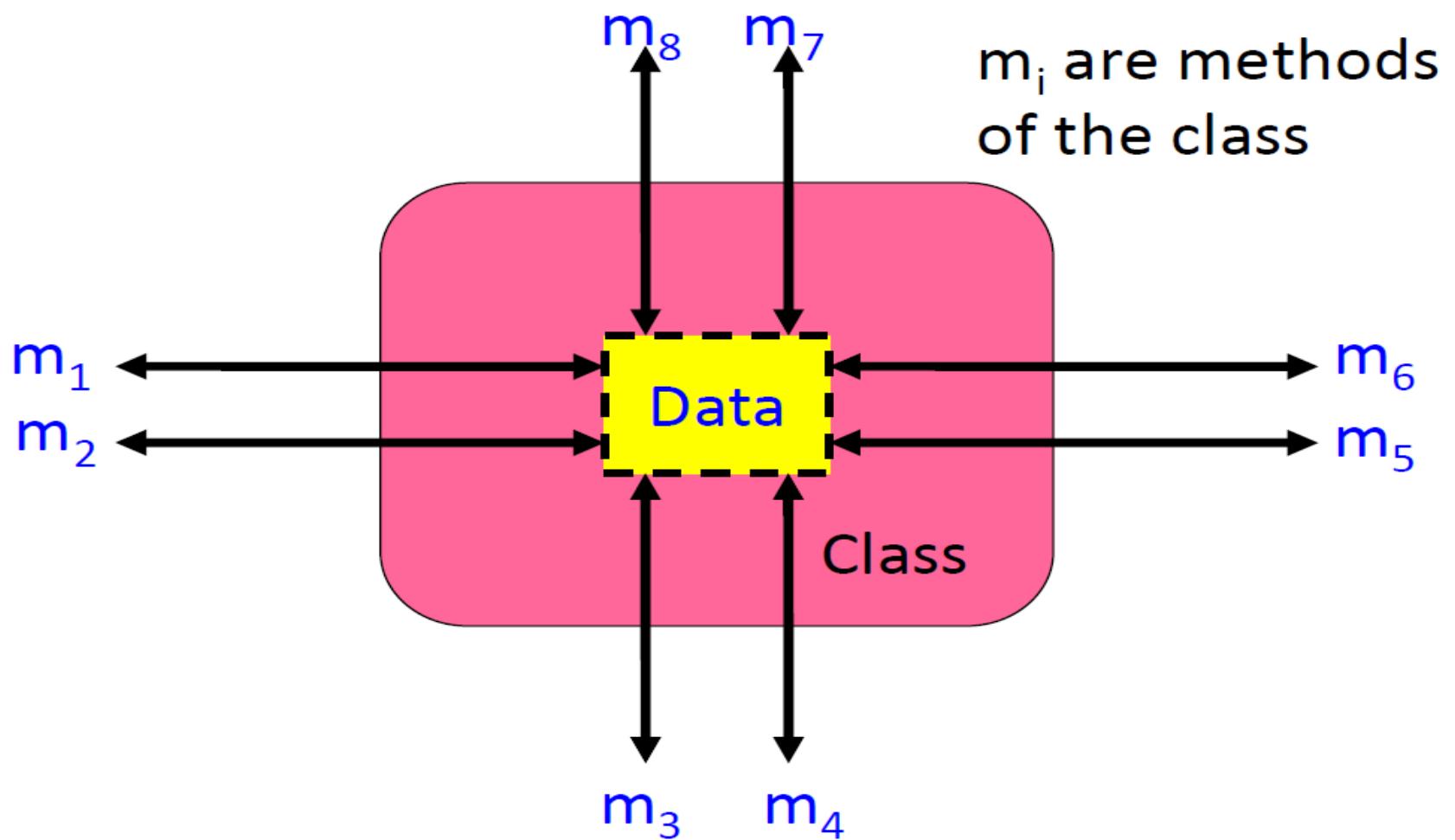
- Last 1% takes longer than the completed 99%. There are many examples to support this statement. For example progress bars we see on computers, the last period last minute, last minute of a match etc. The 90+ percent it runs fast and rest is sluggish.
- In software development if you do not follow SDLC (Software Development Life Cycle) 99% complete syndrome happens. If you ask developer about the progress he/she may say development is in final stage of just 1% is left .
- Most probably, this last 1% will drag on to days, weeks, and even few months. Either something was wrong in the SDLC or they failed to follow up.
- Developers should evaluate the amount of effort needed to complete remaining tasks in hour. Usually instead of this people tend to consider the amount of work gone through against the original estimate.

OBJECT ORIENTED SOFTWARE DESIGN USING UML

Object Oriented

- A system is designed as a set of interacting objects:
 - Often, real-world entities: e.g. an employee, a book etc.
 - Can be conceptual objects also: e.g. Controller, manager, etc.
- Objects consists of data (attributes) and functions (methods) that operate on data.
 - Encapsulation
- Hides organization of internal information
 - Data abstraction

Model of an Object



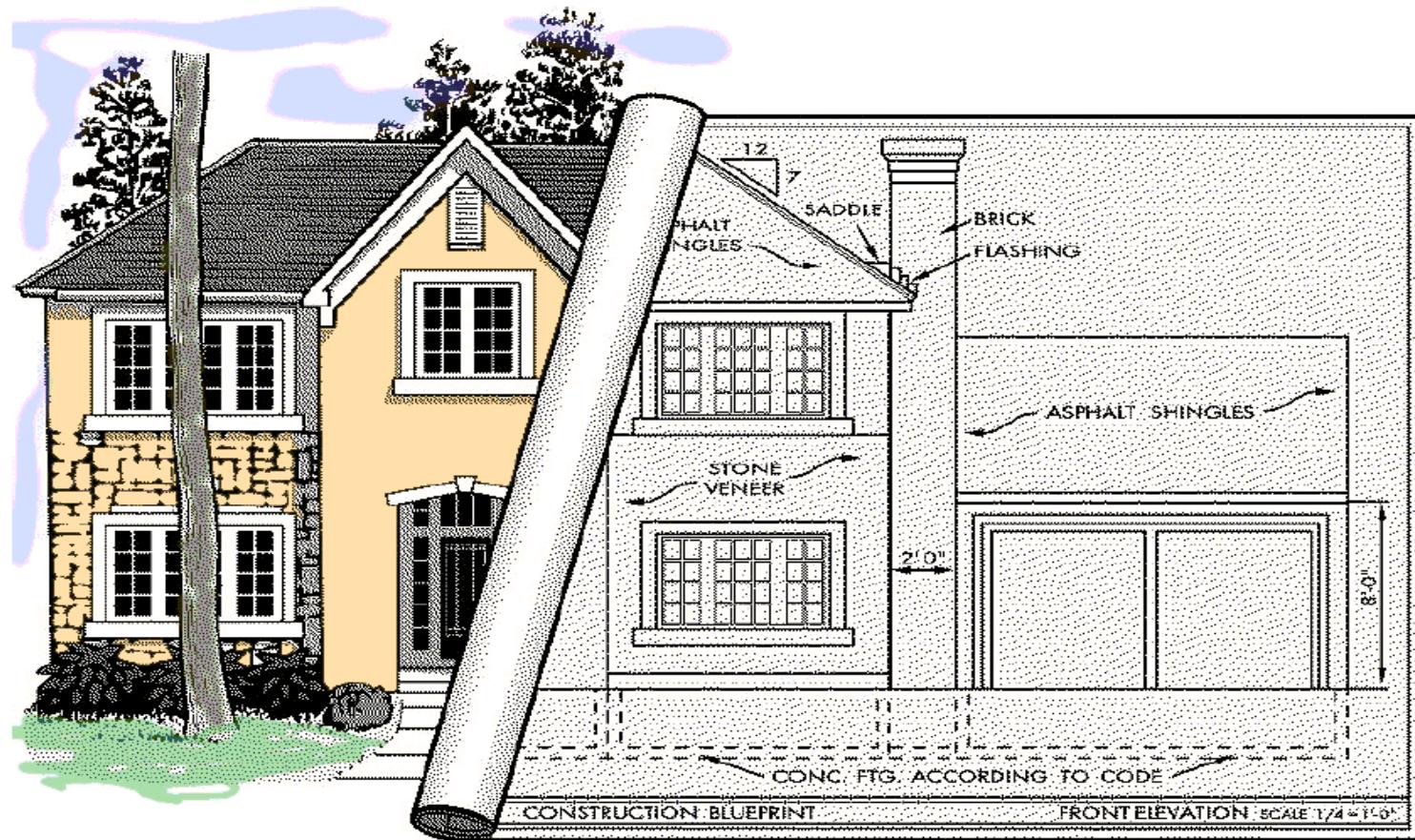
Advantages of Object-Oriented Development

- Code and design reuse
- Increased productivity
- Ease of testing and maintenance
- Better understandability
- Elegant design:
 - Loosely coupled, highly cohesive objects:
 - Essential for solving large problems.
- Initially incurs higher costs
 - After completion of some projects reduction in cost become possible
- Using well-established OO methodology and environment:
 - Projects can be managed with 20% -- 50% of traditional cost of development.

Object modelling using UML

- Unified Modeling Language is a modelling language
 - Not a system design or development methodology
- Used to document object-oriented analysis and design
- Independent of any specific design methodology
- UML developed in early 1990s
 - To standardize the large number of object-oriented modelling notations that existed.
- Current version is UML2.0
- Adopted by Object Management Group (OMG) in 1997
 - OMG an association of industries that promotes consensus notations and techniques

When you want to build a house!

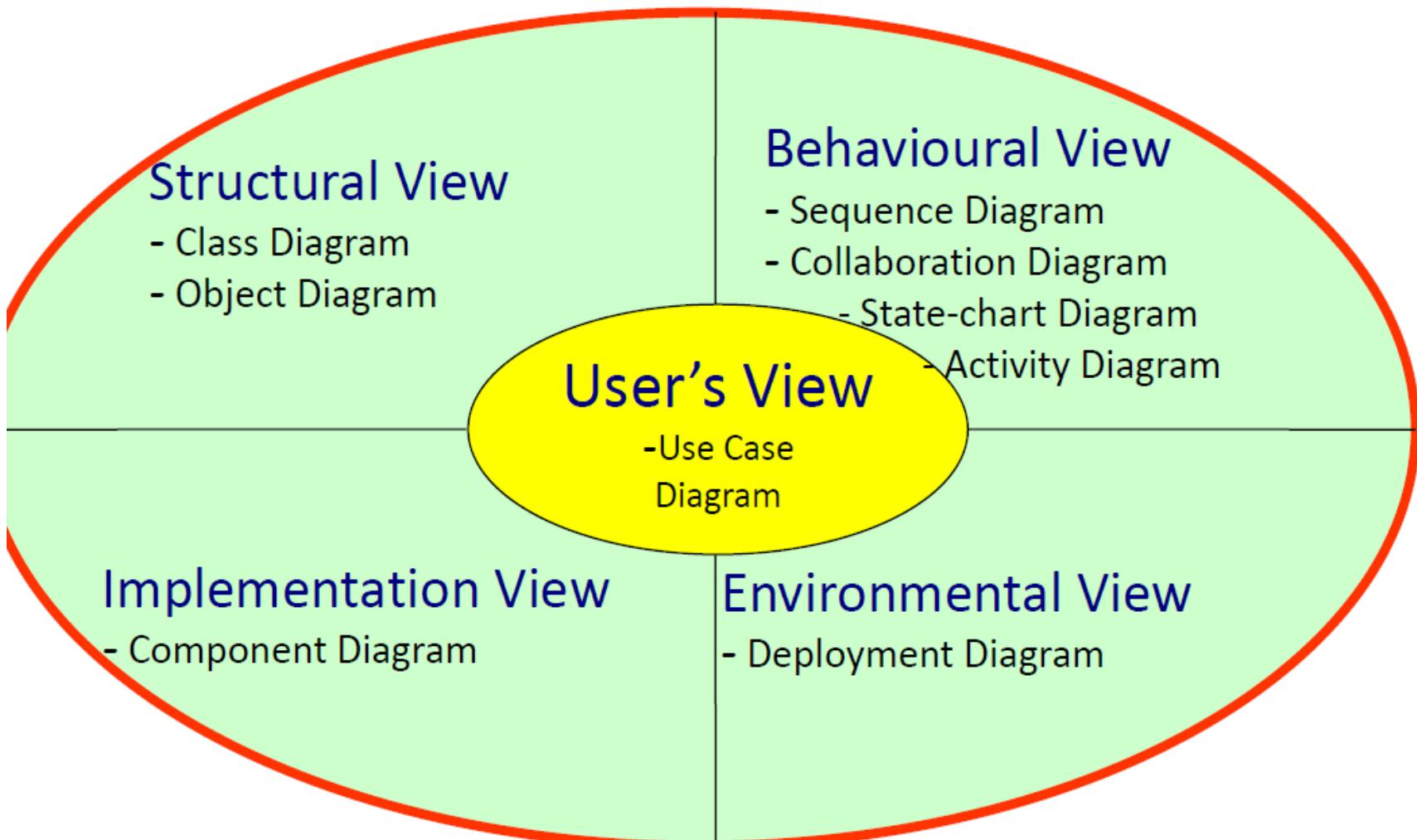


What is UML

- UML a graphical modelling tool, easy to understand and construct
- It provides many diagrams to capture different views of a system
 - Model is required to capture only important aspects
 - Helps in managing complexity

UML 1.x Diagrams

- 9 diagrams supporting 5 views



Use Case Model

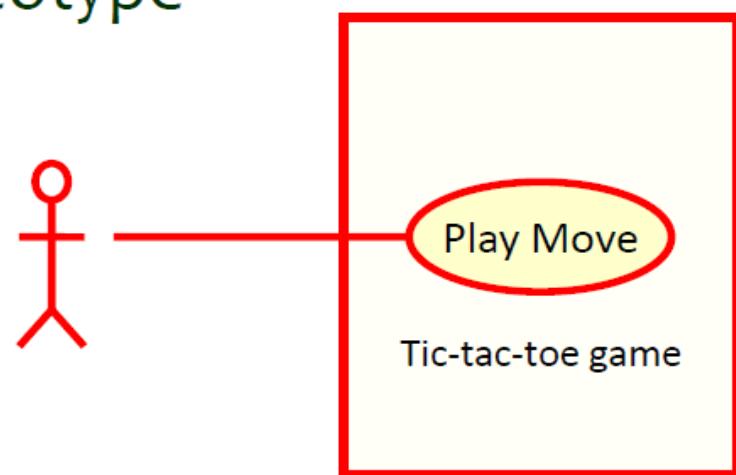
- Consists of a set of “use cases”
- It is the central model:
 - Other models must conform to this model
 - Not really an object-oriented model
 - A functional model of the system
- Use Cases are the main tasks performed by the users of the system.
- Use Cases describe the behavioral aspects of the system.
- Use Cases are used to identify how the system will be used.
- Use Cases are a convenient way to document the functions that the system must support.
- Use Cases are used to identify the components (classes) of the system.

Use Cases

- Normally, use cases are independent of each other
- Implicit dependencies may exist
- **Example:** In **Library Automation System**, renew-book and reserve-book are independent use cases.
 - But in actual implementation of renew-book--- A check is made to see if any book has been reserved using reserve-book.
- Other Possible Use Cases in Library information system
 - issue-book
 - query-book
 - return-book
 - create-member
 - add-book, etc.

Representation of Use Cases

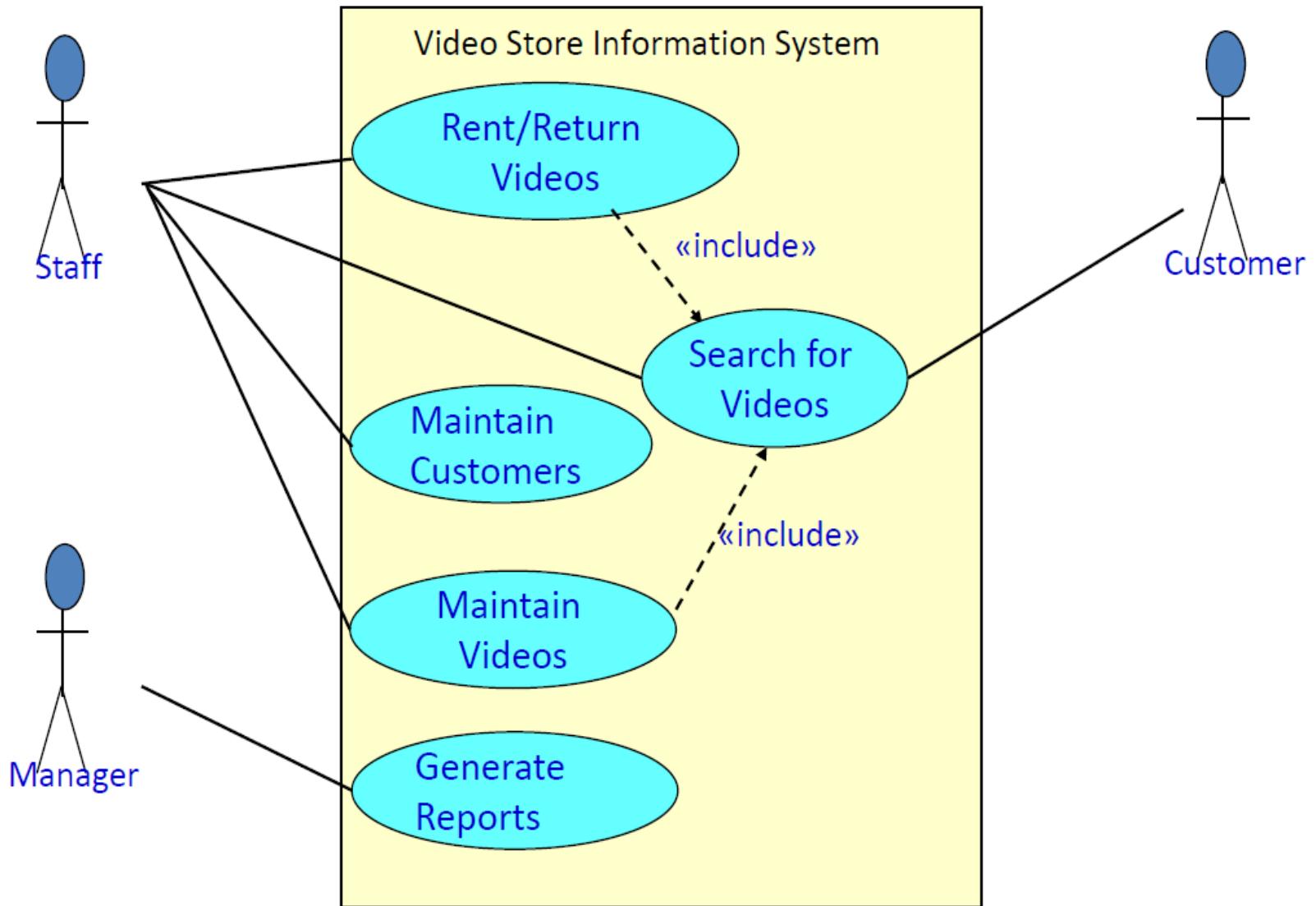
- Represented in a use case diagram
 - A Use Case is represented by an ellipse
 - System boundary is represented by a rectangle
 - Users are represented by stick person icons (actor)
 - Communication relationship between actor and Use Case by a line
- External system by a stereotype



Ex1: Draw a Use Case Model

- Video Store Information System supports the following business functions:
 - Recording information about videos the store owns
 - This database is searchable by staff and all customers
 - Information about a customer's borrowed videos
 - Access by staff and also the customer. It involves video database searching.
 - Staff can record video rentals and returns by customers. It involves video database searching.
 - Staff can maintain customer, video and staff information.
 - Managers of the store can generate various reports.

Ex1: Solution



Development of a Use Case Diagram

- Identify all of the actors who will use the system.
- Interview these actors to identify the functions that they need to perform. (use cases)
- Identify scenarios (sequence of steps to accomplish a use case).
- Identify common steps within the different scenarios. Separate them into different use cases so that they can easily be included in other scenarios.
- Identify relationships between use cases.

Actors

- Actor - an entity external to the system that in some way participates in the use case. Actor represents a role that a user can play.
- An actor typically stimulates the system with input events or receives outputs from the system or does both.
- Actors are treated like classes and can be generalized.
- Primary actor: Use the system to achieve a goal.
(found in left side of the use case diagram)
- Secondary actor: plays a supporting role to facilitate the primary actor to achieve their goal. (right side)

Identification of Use Cases

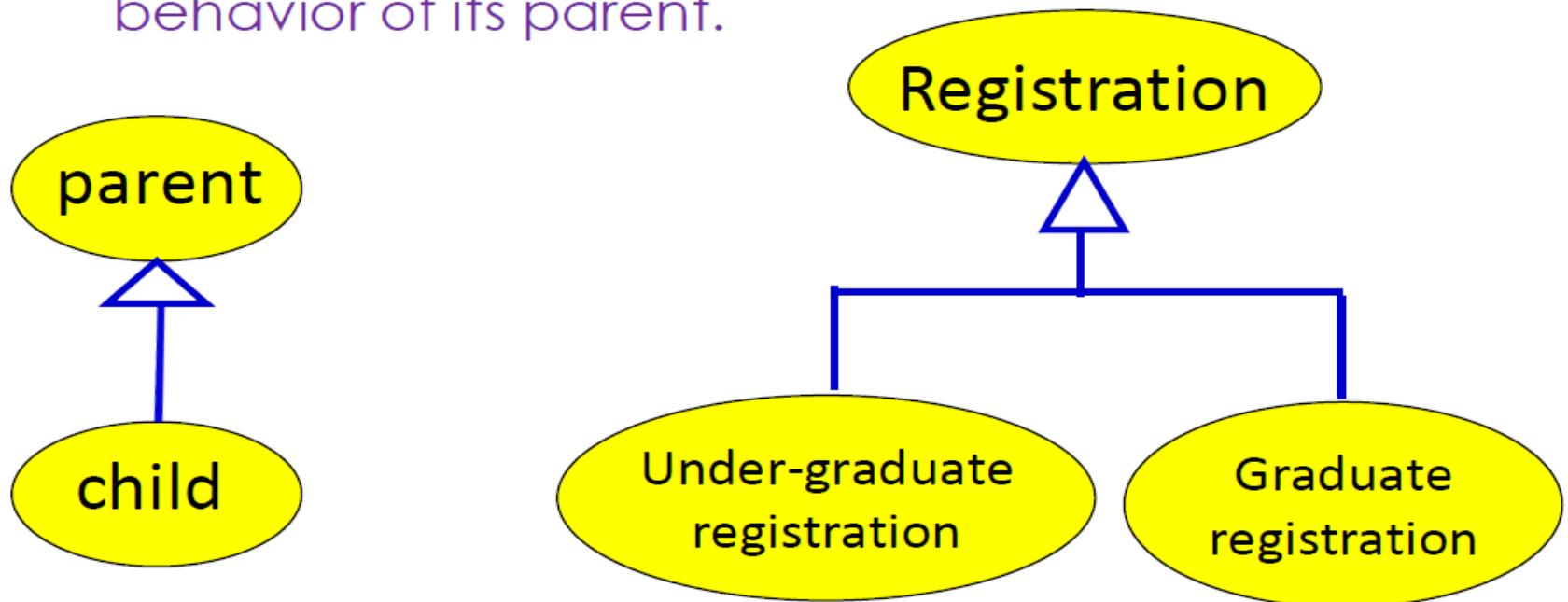
- Actor-based:
 - Identify the actors related to a system or organization.
 - For each actor, identify the processes they initiate or participate in.
- Event-based
 - Identify the external events that the system must respond to.
 - Relate the events to actors and use cases.

Factoring Use Cases

- Two main reasons for factoring:
 - Complex use cases need to be factored into simpler use cases
 - To represent common behaviour across different use cases
- Three ways of factoring:
 - Generalization
 - Include
 - Extend

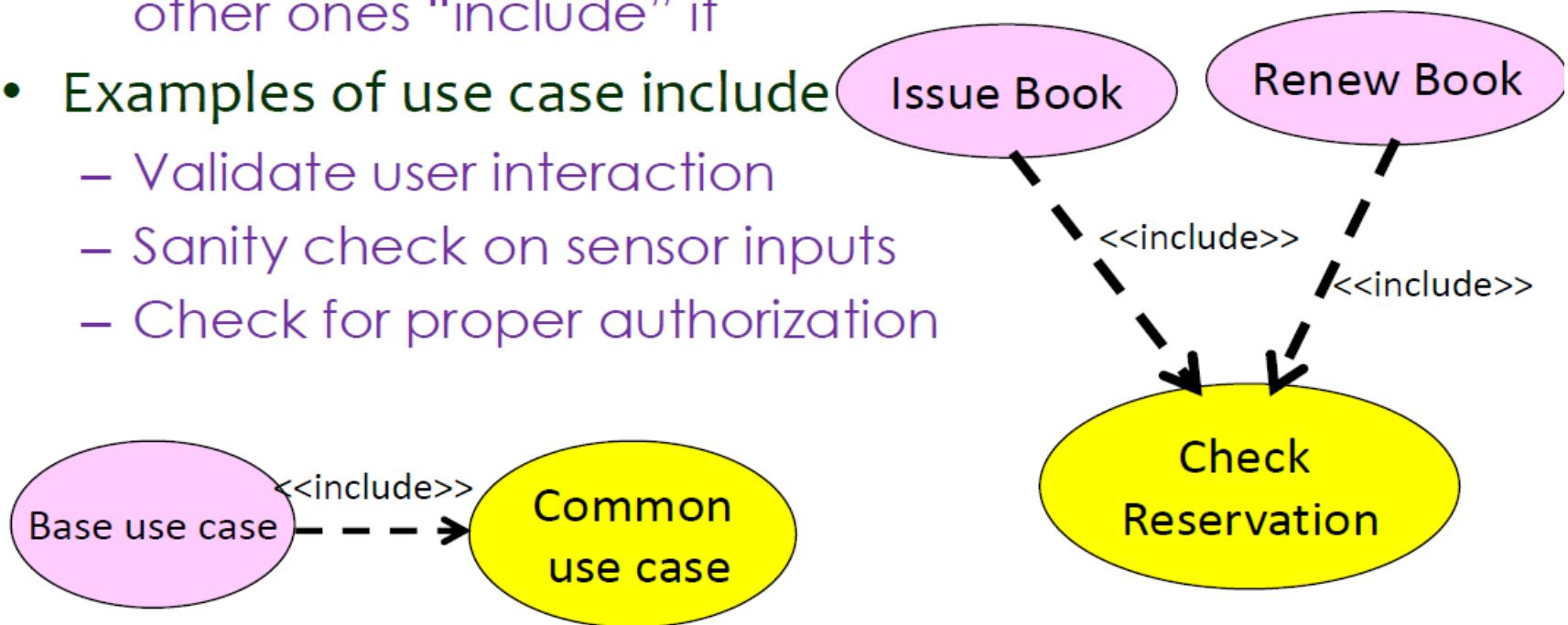
Generalization

- The child use case inherits the behaviour of the parent use case.
 - The child may add to or override some of the behavior of its parent.



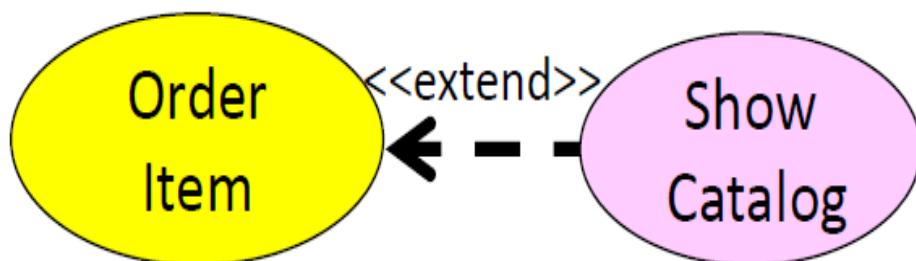
Include

- When you have a piece of behaviour that is similar across many use cases
 - Break this out as a separate use-case and let the other ones “include” it
- Examples of use case include
 - Validate user interaction
 - Sanity check on sensor inputs
 - Check for proper authorization

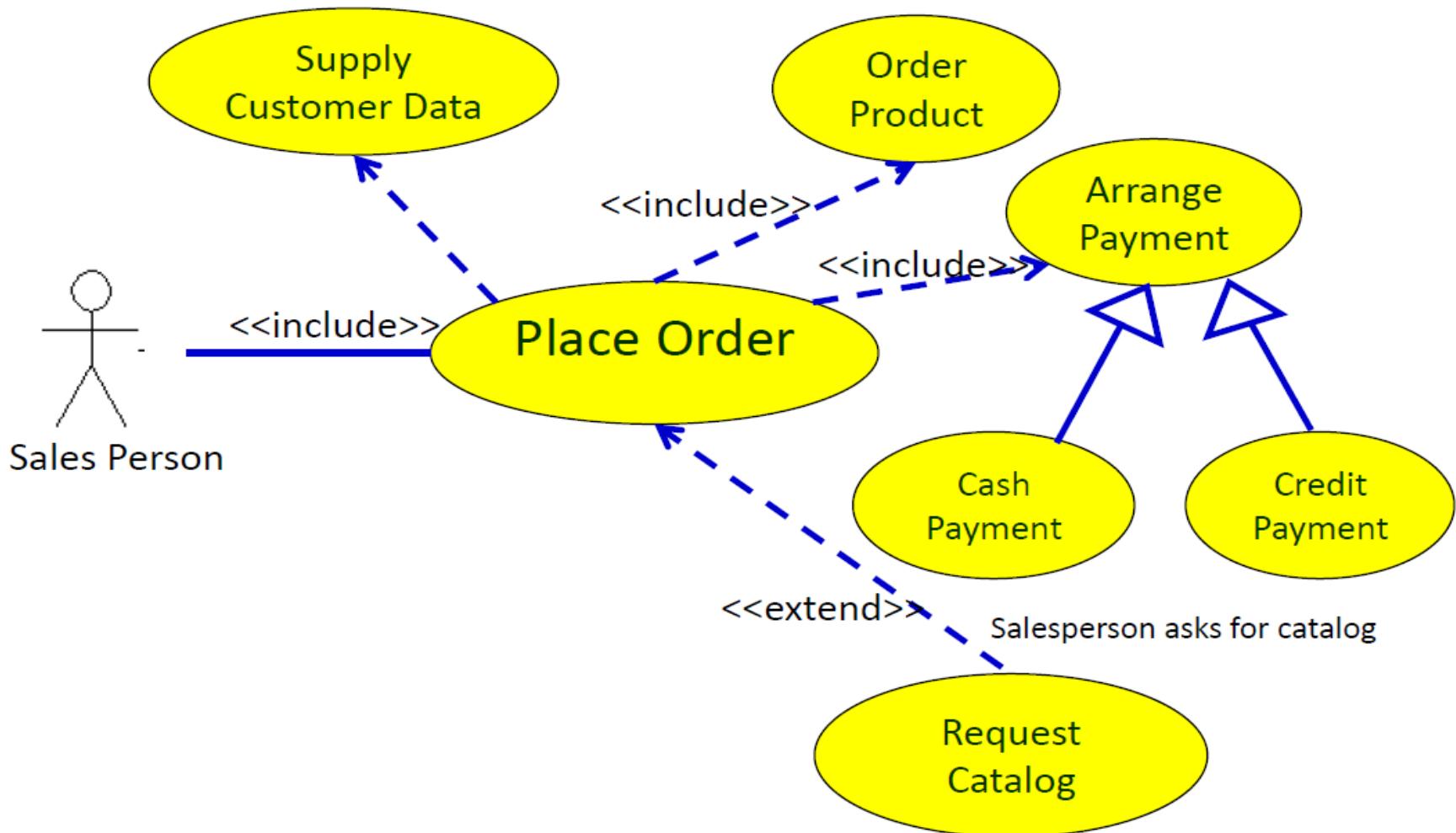


Extends

- Use when a use-case optionally can do a little bit more:
 - Capture the normal behaviour
 - Capture the extra behaviour in a separate use-case
 - Create extends dependency
- Makes it a lot easier to understand



Example



User Interface Design

Types of User Interfaces

- User interfaces can be classified into three categories:
 - Command language-based interface
 - Menu-based interface
 - Direct manipulation interface

Types of User Interfaces

- Each category of interface has its advantages and disadvantages:
 - Modern applications sport a combination of all the three types of interfaces.

Choice of Interface

- Which parts of the interface should be implemented using what type of interface?
 - No simple guidelines available
 - to a large extent depends on the experience and discretion of the designer.
 - a study of characteristics of the different interfaces would give us some idea.

Command Language-Based Interface

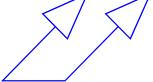
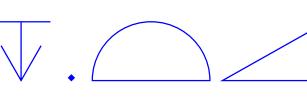
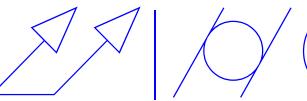
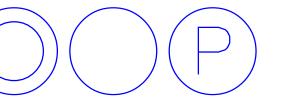
- As the name itself suggests:
 - incorporates some language to form commands.
- Users frame the required commands in the language:
 - type them in whenever required.

Design of command language interface

- Simple command language interface:
 - determine all the commands needed to be supported
 - assign unique names to the different commands.

Design of command language interface

- A more sophisticated command language interface:
 - allow users to compose primitive commands to form more complex commands.

• C o n s i d e r  < 
T.   |  OOP 123

- Like a programming language.

Command Language-Based Interface

- The facility to compose commands:
 - dramatically reduces the number of command names users would have to remember.
 - Commands can be made concise
 - requiring minimal typing by the user.
 - allow faster interaction with the computer
 - simplify input of complex commands.

Advantages of Command Language Interfaces

- Easy to develop:
 - compiler writing techniques are well developed.
- Can be implemented even on cheap alphanumeric terminals.
- Much more efficient:
 - compared to other types of interfaces.

Disadvantages of Command Language Interfaces

- Difficult to learn:
 - Require the user to memorize primitive commands.
- Require the user to type in commands.
- Users make errors while:
 - formulating commands in the command language
 - typing them in.

Disadvantages of Command Language Interfaces

- All interactions are through key-board:
 - cannot take advantage of effective interaction devices such as a mouse.
 - For casual and inexperienced users,
 - command language interfaces are not suitable.

Issues in Designing a Command Language Interface

- Design of a command language interface:
 - involves several issues.
- The designer has to decide
 - what **mnemonics** are to be used for the commands.
 - mnemonics should be meaningful
 - yet be concise to minimize the amount of typing required.

Issues in Designing a Command Language Interface

- The designer has to decide:
 - whether users will be allowed to redefine command names to suit their own preferences.
 - Letting a user define his own mnemonics for various commands is a useful feature,
 - but increases complexity of user interface development.

Issues in Designing a Command Language Interface

- Designer has to decide:
 - whether it should be possible to compose primitive commands to create more complex commands.
 - syntax and semantics of command composition options has to be clearly and unambiguously decided.

Menu-Based Interface

- Advantages of a menu-based interface over a command language interface:
 - users are not required to remember exact command names.
 - typing effort is minimal:
 - menu selections using a pointing device.
 - This factor becomes very important for the occasional users who can not type fast.

Menu-Based Interface

- For experienced users:
 - menu-based interfaces is slower than command language interfaces
 - experienced users can type fast
 - also get speed advantage by composing simple commands into complex commands.

Menu-Based Interface

- Composition of commands in a menu-based interface is not possible.
 - actions involving logical connectives (and, or, all, etc.)
 - awkward to specify in a menu-based system.

Menu-Based Interface

- If the number of choices is large,
 - it is difficult to design a menu-based interface.
 - Even moderate sized software needs hundreds or thousands of menu choices.
- A major problem with the menu-based interface:
 - structuring large number of menu choices into manageable forms.

Structuring Menu Interface

- Any one of the following options is adopted to structure menu items.
 - Walking menu
 - Scrolling menu
 - Hierarchical menu

Scrolling Menu

- Used when the menu options are highly related.
 - For example text height selection in a word processing software.
- Scrolling of menu items
 - lets the user to view and select the menu items that can not be accommodated on one screen.

Walking Menu

- Walking menu is commonly used to structure large menu lists:
 - when a menu item is selected,
 - it causes further menu items to be displayed adjacent to it in a submenu.

Walking Menu

- A walking menu can successfully structure commands only if:
 - there are tens rather than hundreds of choices
 - each adjacently displayed menu does take up some screen space
 - the total screen area is after all limited.

Hierarchical menu

- Menu items are organized in a hierarchy or tree structure.
 - Selecting a menu item causes the current menu display to be replaced by an appropriate submenu.
 - One can consider the menu and its various submenu to form a hierarchical tree-like structure.

Hierarchical menu

- Walking menu are a form of hierarchical menu:
 - practicable when the tree is shallow.
- Hierarchical menu can be used to manage large number of choices,
 - but, users face navigational problems
 - lose track of where they are in the menu tree.

Direct Manipulation Interface

- Present information to the user
 - as **visual models or objects**.
- Actions are performed on the visual representations of the objects, e.g.
 - pull an icon representing a file into an icon representing a trash box, for deleting the file.
- Direct manipulation interfaces are sometimes called as **iconic interfaces**.

Direct Manipulation (Iconic) Interface

- Important advantages of iconic interfaces:
 - icons can be recognized by users very easily,
 - icons are language-independent.
- However, experienced users consider direct manipulation interfaces too slow.

Direct Manipulation (Iconic) Interface

- It is difficult to form complex commands using a direct manipulation interface.
- For example, if one has to drag a file icon into a trash box icon for deleting a file:
 - to delete all files in a directory one has to perform this operation again and again
 - very easily done in a command language-interface by issuing a command **delete**
.

Windowing Systems

- Most modern GUIs are developed using some **windowing system**.
- A windowing system can generate displays through a set of windows.
- Since a window is a basic entity in such a graphical user interface:
 - we need to first discuss what exactly a window is.

Window

- A window is a rectangular area on the screen.
- A window is a virtual screen:
 - it provides an interface to the user for carrying out independent activities,
 - one window can be used for editing a program and another for drawing pictures, etc.
- A window can be divided into two parts:
 - client part,
 - non-client part.

Window

- The client area makes up the whole of the window,
 - except for the borders and scroll bars.
- The client area is available to the programmer.
- Non-client area:
 - under the control of window manager.

Window management system (WMS)

- A graphical interface might consist of a large number of windows,
 - necessary to have some systematic way to manage the windows.
 - Window Management System (WMS)

Window management system (WMS)

- GUIs are developed using a window management system (WMS):
 - A window management system is primarily a resource manager.
 - keeps track of screen area resource
 - allocates it to the different windows which are using the screen.

Window management system (WMS)

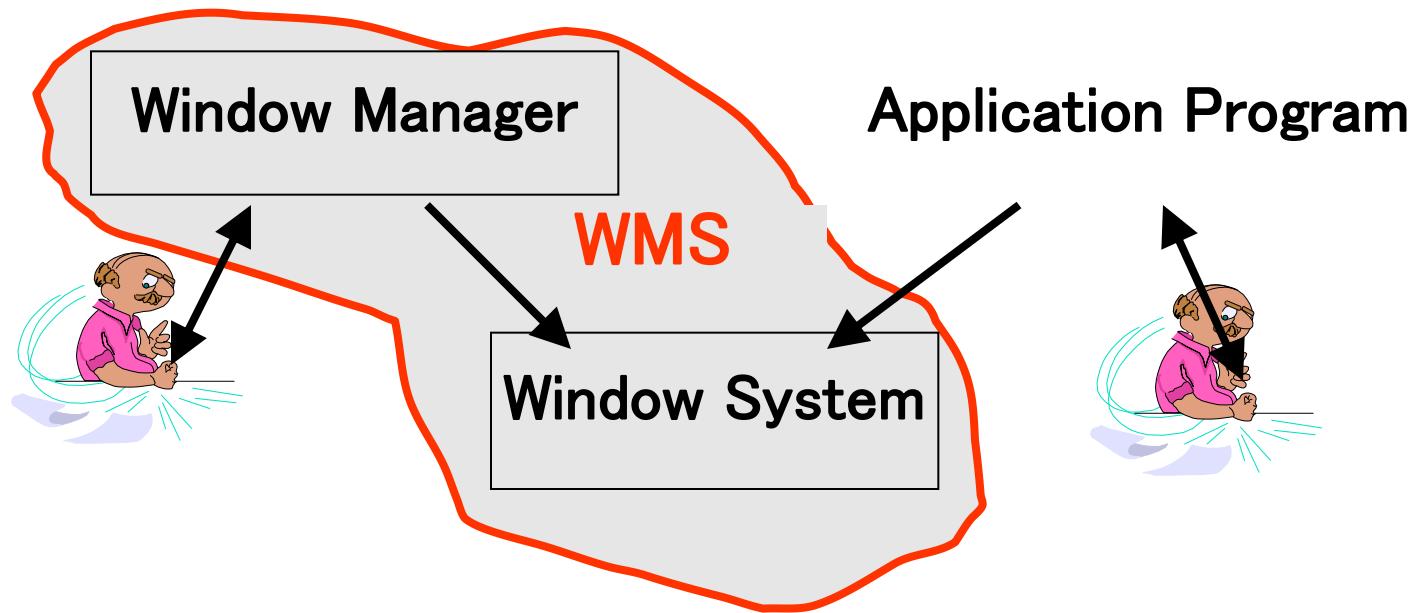
- From a broader perspective, a WMS can be considered as a **user interface management system (UIMS)** ---
 - not only does resource management,
 - also provides the basic behavior to windows
 - provides several utility routines to the application programmer for user interface development.

Window management system (WMS)

- A WMS simplifies the task of a GUI designer to a great extent:
 - provides the basic behaviour to the various windows such as move, resize, iconify, etc.
 - provides routines to manipulate windows such as:
 - creating, destroying, changing attributes of the windows, and drawing text, lines, etc.

Window management system (WMS)

- A WMS consists of two part:
 - a window manager
 - a window system.



Window Manager and Window System

- User interacts with window manager to do various window-related operations such as:
 - window repositioning,
 - window resizing,
 - iconification, etc.
- Window manager also controls the screen's real estate policy.

Window manager

- The window manager is built on the top of the window system:
 - makes use of the basic services provided by the window system.
- The window manager determines how the windows look and behave.
 - several kinds of window managers can be based on the same window system.

Window manager

- Window manager can be considered as a special program:
 - makes use of the services (function calls) of the window system.
- Application programs
 - invoke the window system for user interface-related functions.

Window System

- Provides a large number of routines for the programmer
- It is very cumbersome to use these large set of routines:
 - most WMS provide a higher-level abstraction called [widgets](#).

Window Management System

- A widget is the short form for a window object.
- Widgets are the building blocks in interface design.
- We know that an object is essentially a collection of:
 - related data with several operations defined on these data.

Widgets

- The data of an window object are:
 - the geometric attributes (such as size, location etc.)
 - other attributes such as its background and foreground color, etc.
- The operations defined on these data include, resize, move, draw, etc.

Advantages of Widgets

- One of the most important reasons to use widgets as building blocks:
 - provide consistency.
- Consistent user interfaces
 - improve the user's productivity and
 - lead to higher performance with fewer errors.

Advantages of Widgets

- Widgets make users familiar with standard ways of using an interface -

--

- users can easily extend their knowledge of interface of one application to another
- the learning time for users is reduced to a great extent.

What is regression testing?

Regression testing is a black box testing techniques. It is used to authenticate a code change in the software does not impact the existing functionality of the product.

Regression testing is making sure that the product works fine with new functionality, [bug](#) fixes, or any change in the existing feature.

Regression testing is a type of [software testing](#). Test cases are re-executed to check the previous functionality of the application is working fine, and the new changes have not produced any bugs.

Regression testing can be performed on a new build when there is a significant change in the original functionality. It ensures that the code still works even when the changes are occurring. Regression means Re-test those parts of the application, which are unchanged.

DFD Examples

Yong Choi

BPA

CSUB

Creating Data Flow Diagrams

Steps:

1. Create a list of activities
2. Construct Context Level DFD
(identifies external entities and processes)
3. Construct Level 0 DFD
(identifies manageable sub process)
4. Construct Level 1- n DFD
(identifies actual data flows and data stores)
5. Check against rules of DFD

DFD Naming Guidelines

- External Entity → Noun
- Data Flow → Names of data
- Process → verb phrase
 - a system name
 - a subsystem name
- Data Store → Noun

Creating Data Flow Diagrams

Lemonade Stand Example



Creating Data Flow Diagrams

Example

The operations of a simple lemonade stand will be used to demonstrate the creation of dataflow diagrams.



Steps:

1. Create a list of activities
 - Old way: no Use-Case Diagram
 - New way: use Use-Case Diagram
2. Construct Context Level DFD
(identifies sources and sink)
3. Construct Level 0 DFD
(identifies manageable sub processes)
4. Construct Level 1- n DFD
(identifies actual data flows and data stores)

Creating Data Flow Diagrams

Example

Think through the activities that take place at a lemonade stand.



1. Create a list of activities

- Customer Order
- Serve Product
- Collect Payment
- Produce Product
- Store Product

Creating Data Flow Diagrams

Example

Also think of the additional activities needed to support the basic activities.



1. Create a list of activities

- Customer Order
- Serve Product
- Collect Payment
- Produce Product
- Store Product
- Order Raw Materials
- Pay for Raw Materials
- Pay for Labor

Creating Data Flow Diagrams

Example

Group these activities in some logical fashion, possibly functional areas.



1. Create a list of activities

Customer Order
Serve Product
Collect Payment

Produce Product
Store Product

Order Raw Materials
Pay for Raw Materials

Pay for Labor

Creating Data Flow Diagrams

Example

Create a context level diagram identifying the sources and sinks (users).

Customer Order
Serve Product
Collect Payment

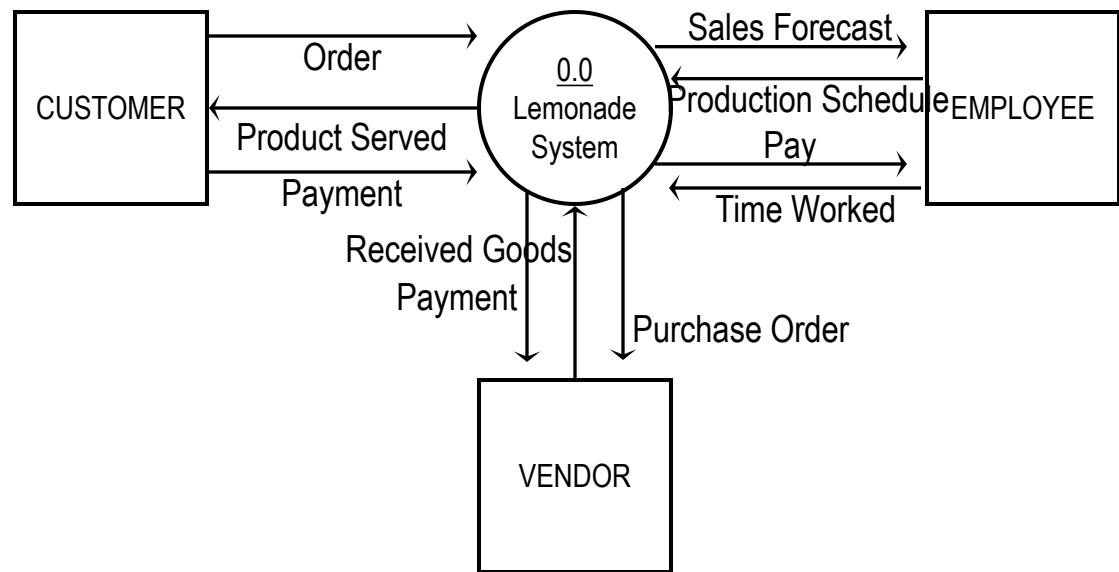
Produce Product
Store Product

Order Raw Materials
Pay for Raw Materials

Pay for Labor

2. Construct Context Level DFD
(identifies sources and sink)

Context Level DFD



Creating Data Flow Diagrams

Example

Create a level 0 diagram identifying the logical subsystems that may exist.

Customer Order
Serve Product
Collect Payment

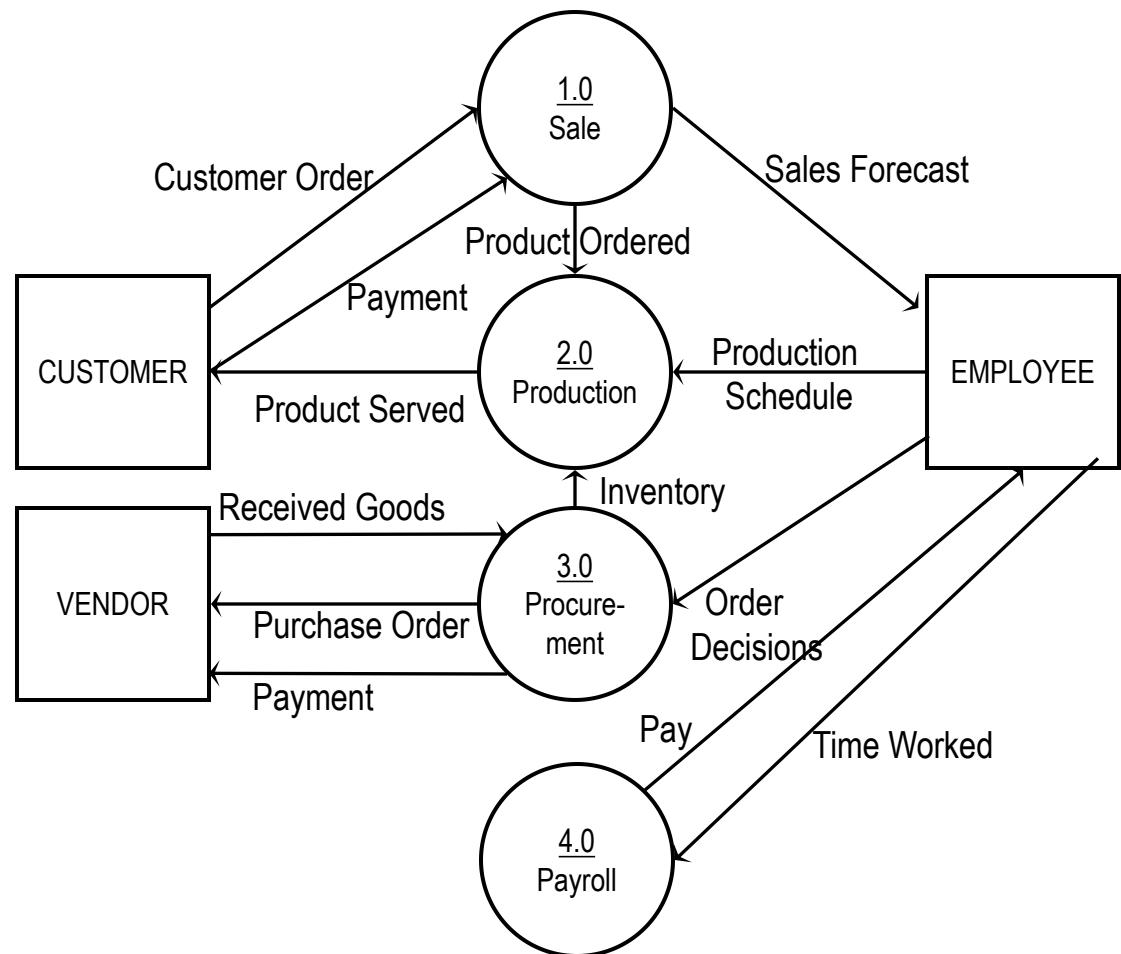
Produce Product
Store Product

Order Raw Materials
Pay for Raw Materials

Pay for Labor

3. Construct Level 0 DFD
(identifies manageable sub processes)

Level 0 DFD



Creating Data Flow Diagrams

Example

Create a level 1
decomposing the processes
in level 0 and identifying
data stores.

Customer Order
Serve Product
Collect Payment

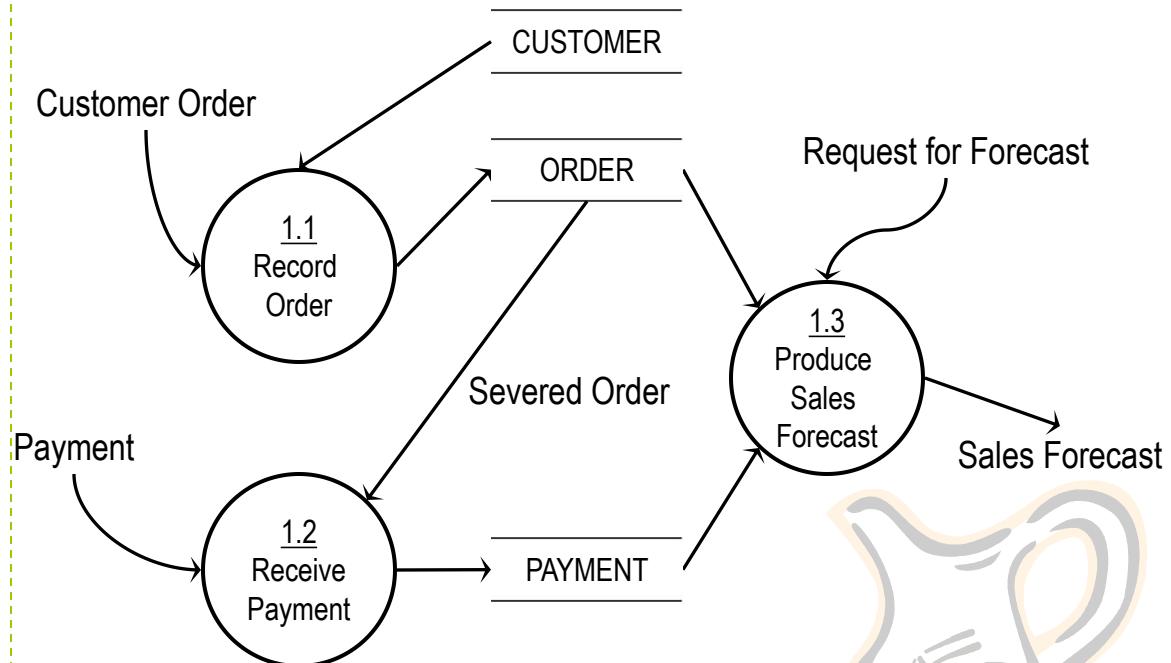
Produce Product
Store Product

Order Raw Materials
Pay for Raw Materials

Pay for Labor

4. Construct Level 1- n DFD
(identifies actual data flows and data stores)

Level 1 DFD



Creating Data Flow Diagrams

Example

Create a level 1
decomposing the processes
in level 0 and identifying
data stores.

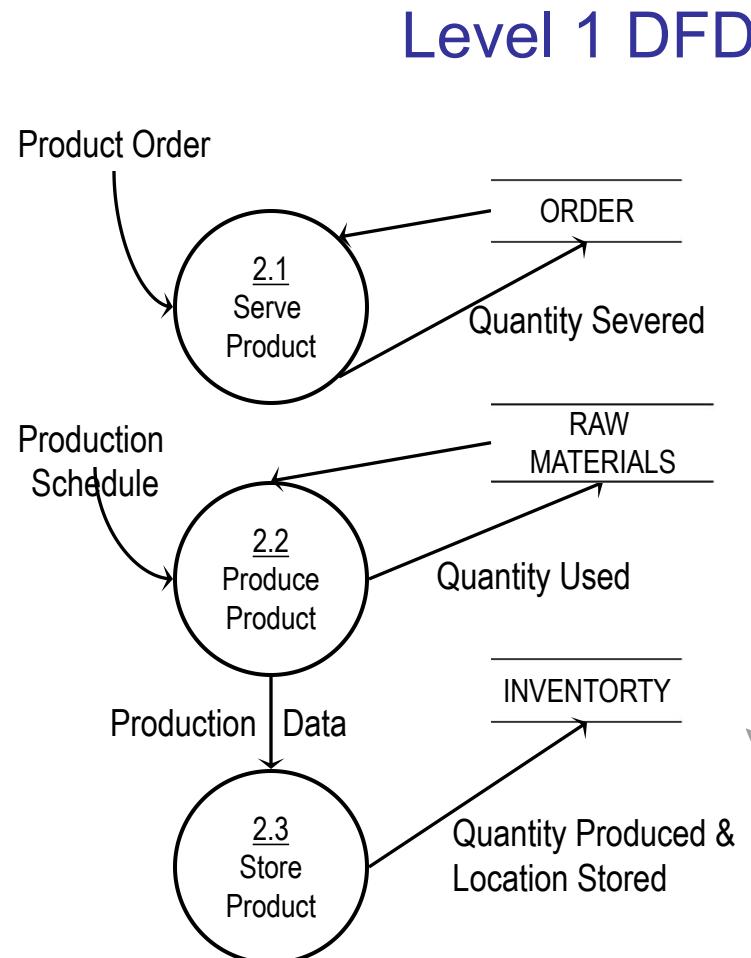
Customer Order
Serve Product
Collect Payment

Produce Product
Store Product

Order Raw Materials
Pay for Raw Materials

Pay for Labor

4. Construct Level 1 (continued)



Creating Data Flow Diagrams

Example

Create a level 1
decomposing the processes
in level 0 and identifying
data stores.

Customer Order
Serve Product
Collect Payment

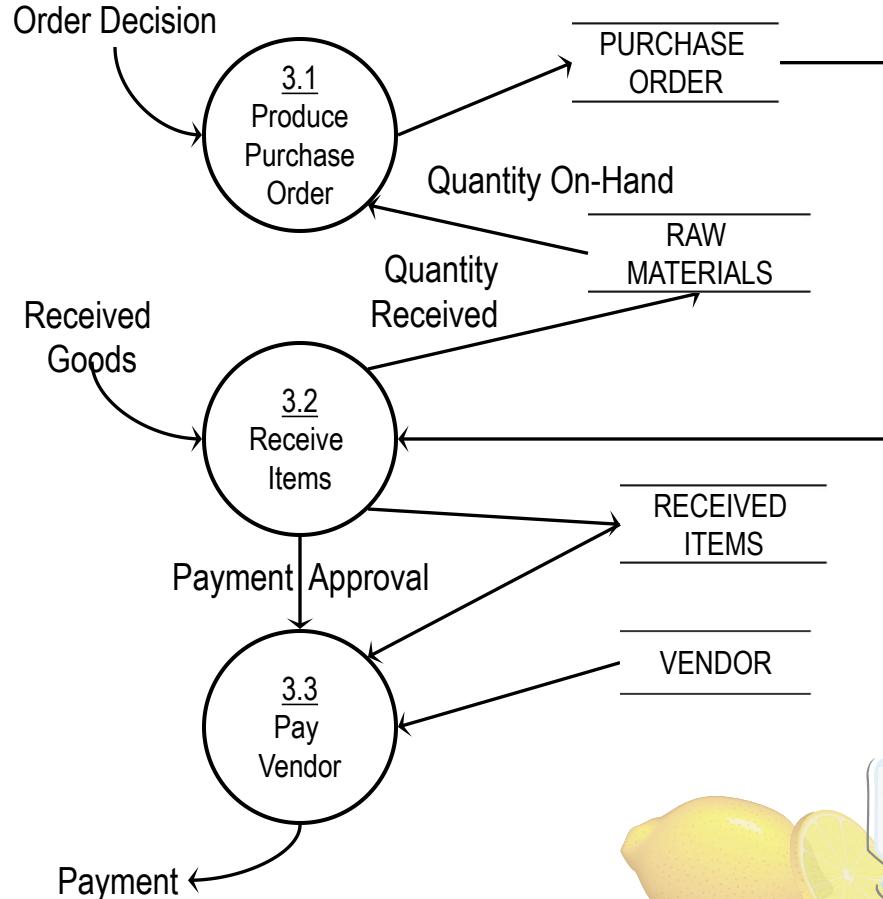
Produce Product
Store Product

Order Raw Materials
Pay for Raw Materials

Pay for Labor

4. Construct Level 1 (continued)

Level 1 DFD



Creating Data Flow Diagrams

Example

Create a level 1
decomposing the processes
in level 0 and identifying
data stores.

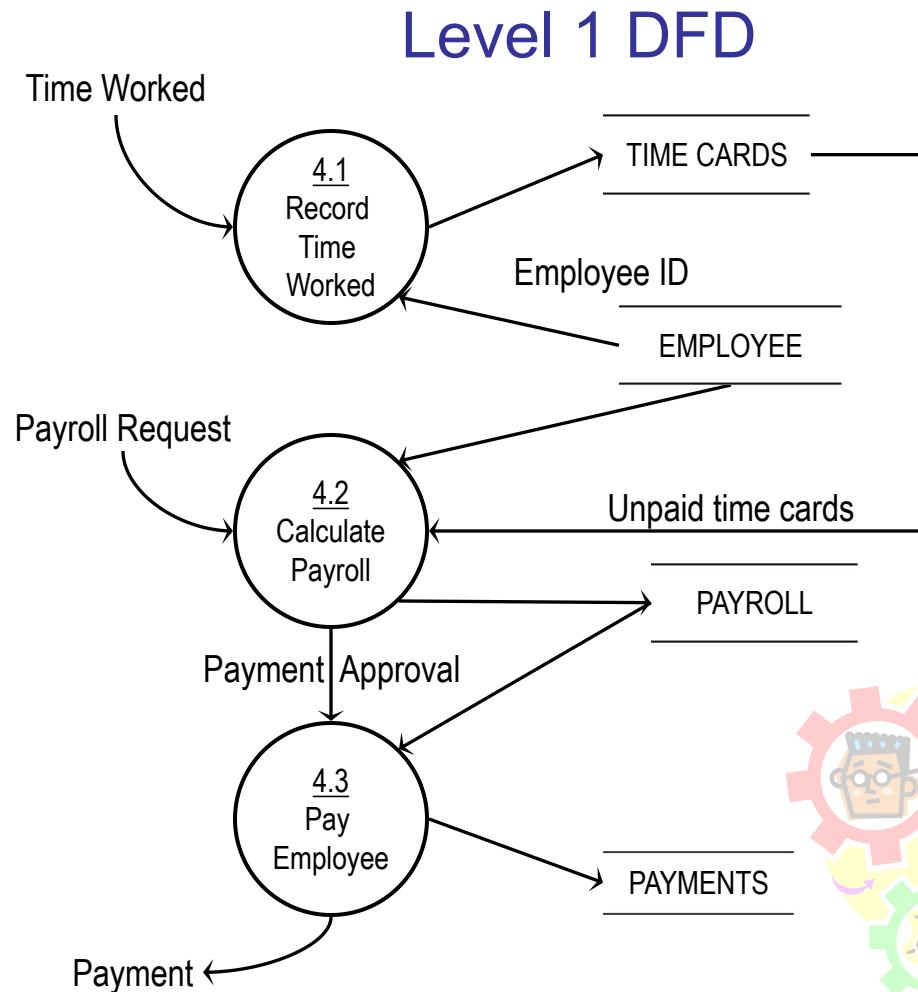
Customer Order
Serve Product
Collect Payment

Produce Product
Store Product

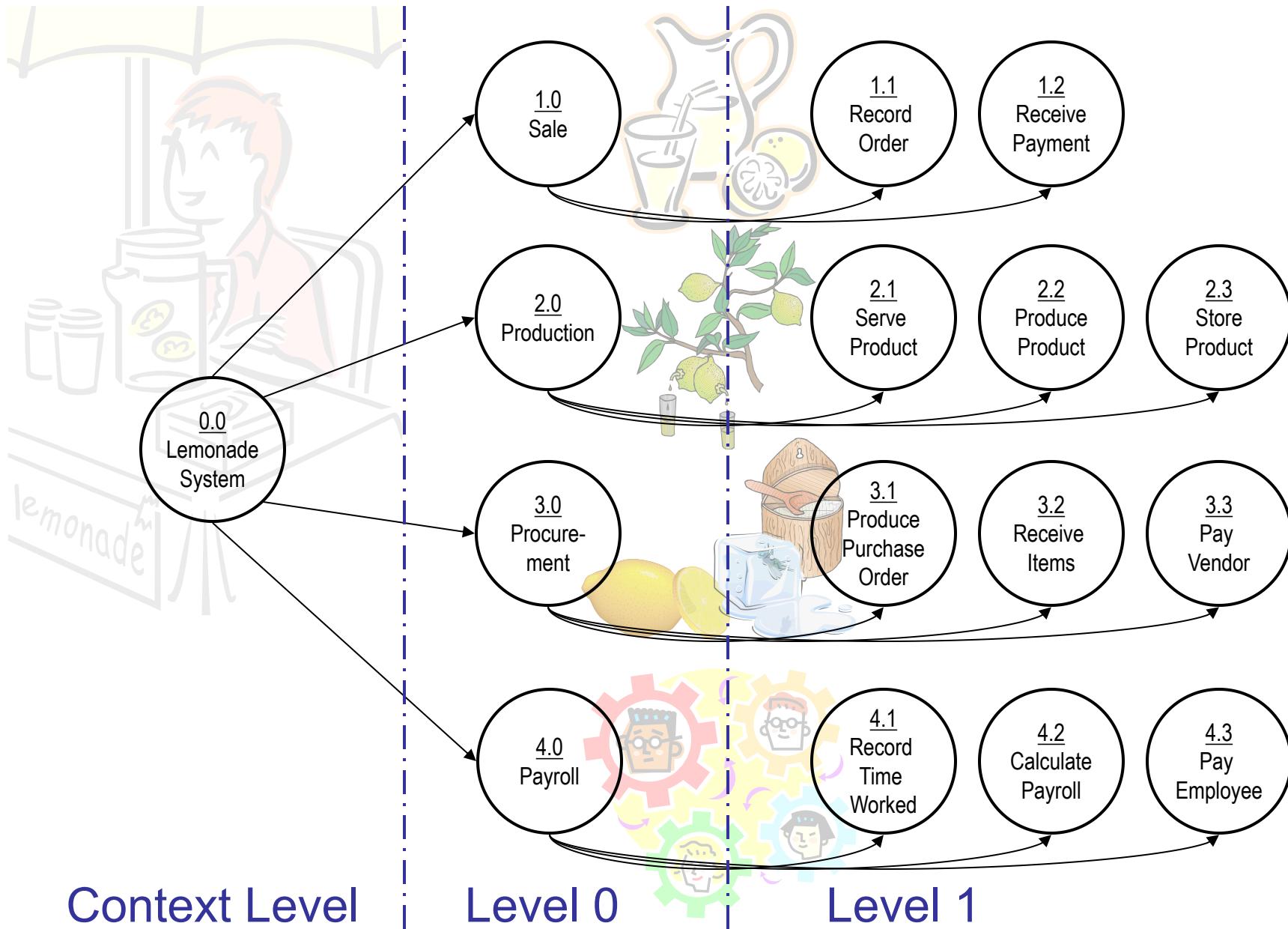
Order Raw Materials
Pay for Raw Materials

Pay for Labor

4. Construct Level 1 (continued)



Process Decomposition



DFD Example: Bus Garage Repairs

- Buses come to a garage for repairs.
- A mechanic and helper perform the repair, record the reason for the repair and record the total cost of all parts used on a Shop Repair Order.
- Information on labor, parts and repair outcome is used for billing by the Accounting Department, parts monitoring by the inventory management computer system and a performance review by the supervisor.

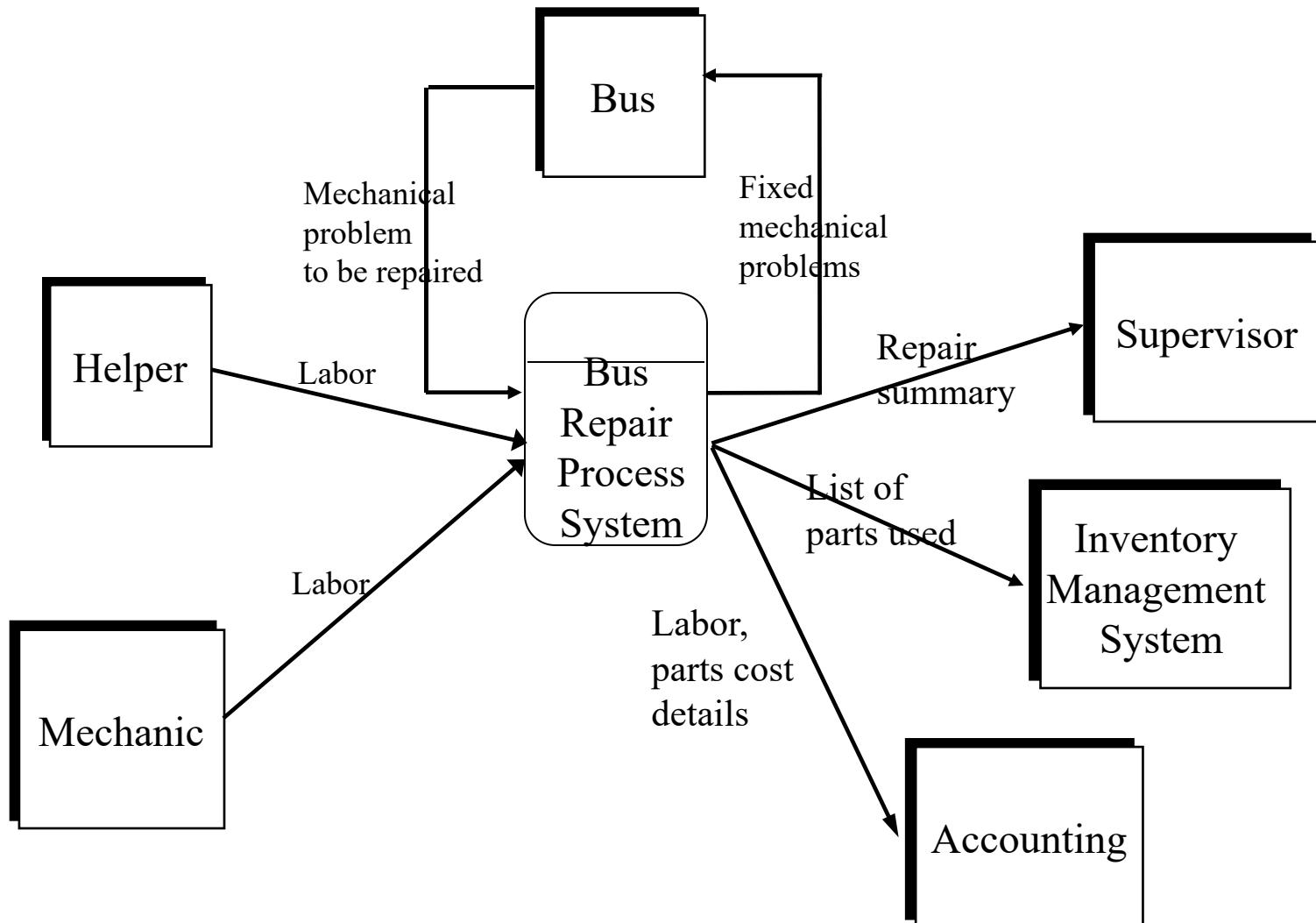
DFD Example: Bus Garage Repairs (cont'd)

- *External Entities*: Bus, Mechanic, Helper, Supervisor, Inventory Management System, Accounting Department, etc.
- *Key process* (“the system”): performing repairs and storing information related to repairs
- *Processes*:
 - Record Bus ID and reason for repair
 - Determine parts needed
 - Perform repair
 - Calculate parts extended and total cost
 - Record labor hours, cost

DFD Example: Bus Garage Repairs (cont'd)

- *Data stores:*
 - Personnel file
 - Repairs file
 - Bus master list
 - Parts list
- *Data flows:*
 - Repair order
 - Bus record
 - Parts record
 - Employee timecard
 - Invoices

Bus Garage Context Diagram



CSUB Burger's Order Processing System

- Draw the CSUB Burger's context diagram
 - System
 - Order processing system
 - External entities
 - Kitchen
 - Restaurant
 - Customer
 - Processes
 - Customer order
 - Receipt
 - Food order
 - Management report

Emerging Trends in Software Engineering

Introduction

- Technology developments occur:
 - To adapt to new environments
 - To respond to new challenges
- Few important developments that have occurred over the last decade or so:
 - Desktops have become more powerful and at the same time more affordable.
 - Internet has become widely accepted.
 - Mobile computing.
 - Outsourcing has become prevalent.

Noticeable Software Engineering Technology Trends

- Following software engineering trends are easily noticeable:
 - Client-server (or Component-based) development
 - Service-Oriented Architecture (SOA)
 - Software as a Service (SaaS)

Client-Server Technology

- Both clients and servers are pieces of software.
- Concepts of Clients and Servers are nothing new --- have existed for ages:
 - Clients are consumers of service.
 - Servers are providers of service.
- Why is then a sudden interest seen in client-server software architecture?

SOA vs. Component Model

cont...

- Several things different.
- Compared to components:
 - SOA's atomic-level objects are often 100 to 1,000 times larger.
- Services may be developed and hosted separately.
 - Possibly pay per use.

SOA vs. Component Model

cont...

- Instead of services embedding calls to each other in their source code:
 - They use defined protocols which describe how services can talk to each other.
- This architecture relies on a business process expert:
 - Links and sequences services, in a process known as **orchestration**, to meet a new business requirement.

SOA vs. Component Model

cont...

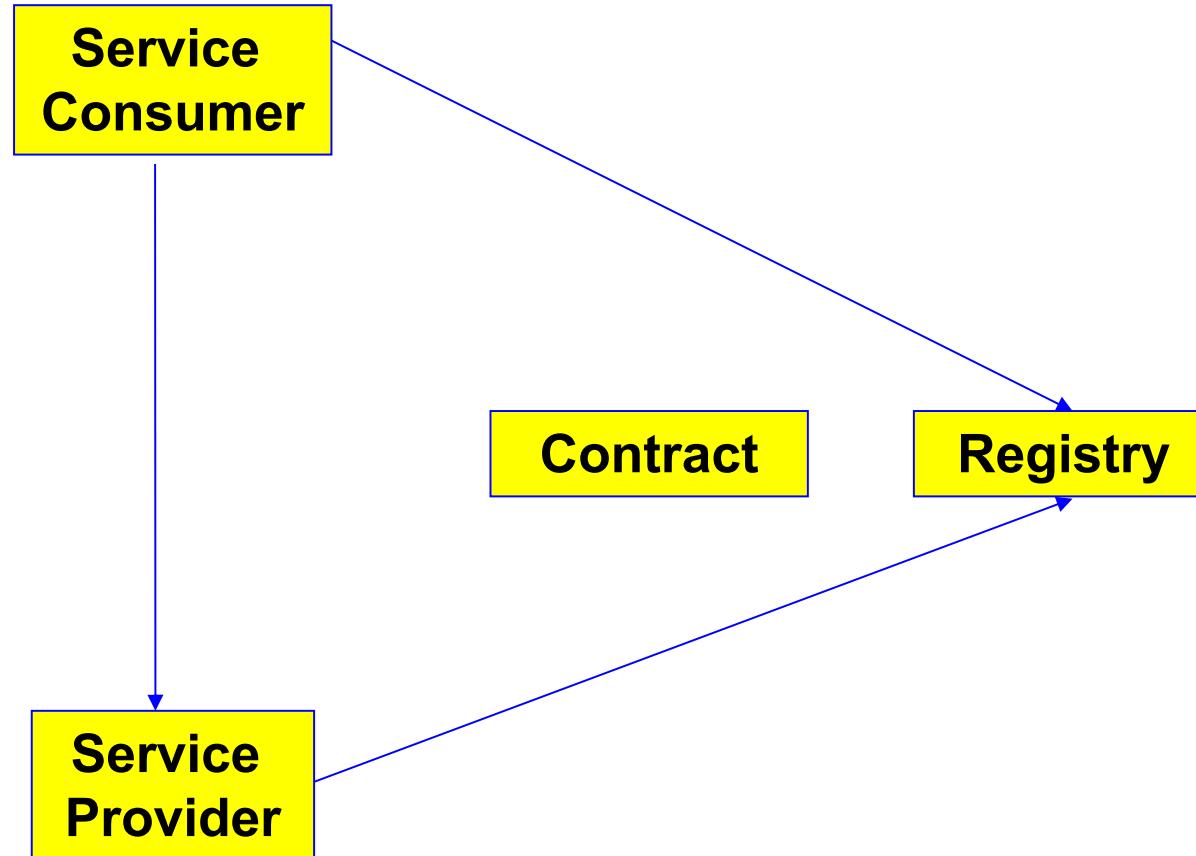
- SOA targets fairly large chunks of functionality to be strung together to form new services:
 - Built almost entirely from existing software services.
- The larger the chunks:
 - The fewer the interfacings required;
 - Faster development;
 - However, very large chunks may not prove easy to reuse.

Principles of Service-Orientation: Services

- Services:

- Abstract the underlying data and logic
- Composable
- Autonomous
- Share a formal contract
- Loosely coupled
- Stateless
- Discoverable

SOA Entities



SOA Challenges

- Building services and infrastructure
- Composing services - formal theory and algebra, languages, semantics
- Testing and verification
- Trust
- Non Functional properties
- Changes
- Business level modeling and translation
- Developing the marketplace

Software as a Service (SaaS)

Challenge Faced

- Owning software is very expensive:
 - An Rs. 50 Lakh software running on an Rs. 1 Lakh computer is common place.
- As with hardware, owning software is the business tradition:
 - Both products and services **bought and used**.
 - Most of IT budget now goes in supporting the software assets.

Background

- To have water supply at home:
 - Do you install system to pump water from river directly to your home?
- To get electricity supply:
 - Do you install a thermal or hydro electric generator?
- It would lead to wastage:
 - You do not need to have all the water pipes open all the time.
 - Unnecessarily expensive proposition.

Background

cont...

- Should you own Rational Suite paying Rs. 50 Lakhs:
 - If you use it only 10 hours every month
 - One hour of usage costs Rs. 100/-
 - Also frees use from maintenance and data storage costs --- Given that maintenance is usually a severe overhead.

SaaS

- Lets customers pay for what they need and when:
 - With price reflecting market place supply and demand.
- SaaS includes:
 - Elements of outsourcing and
 - Application service provisioning

SaaS

- SaaS is a model of software delivery:
 - Software owner provides maintenance, daily technical operation, and support for the software.
 - Services are provided to the clients on amount of usage basis.

SaaS vs. SOA

- SaaS is a software delivery model:
 - SOA is a software construction model.
- SaaS counters the concept of a user as the owner:
 - Owner is a vendor who hosts the software and lets the users execute on-demand charges per usage units.

SaaS versus SOA

- Despite significant differences:
 - Espouse closely related architecture models.
- SaaS and SOA complement each other:
 - SaaS helps to offer components for SOA to use.
 - SOA helps to help quickly realize SaaS.
- Main enabler of SaaS and SOA:
 - Internet and web services technologies.

Key Characteristics

- Network-based access and management of commercially available (not custom) software.
- Activities managed from central location and not at customer site,
 - Enabling customers to access applications remotely via the Web.
- Application delivery follows a one-to-many model (single instance, multi-tenant architecture):
 - In contrast to a one-to-one model.

Summary

- Some of the basic assumptions of software are changing
- This is leading to some different paradigms for software delivery.
- Component-based development:
 - Expected to reduce development time, cost and at the same time improve quality.
- SaaS is changing the way software is delivered.
- SOA conceives a component based assembly model with independent component vendors.²⁰