INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal, Hyderabad – 500 043

Software Engineering

Course Code: ACS008

IV Semester (IARE – R16)

Prepared by:

Ms. B.DHANA LAXMI

Assistant Professor

Mr. A. PRAVEEN

Assistant Professor

UNIT-I

SOFTWARE PROCESS AND PROJECT MANAGEMENT:

Introduction to software engineering, software process, perspective and specialized process models; Software project management: Estimation: LOC and FP based estimation, COCOMO model; Project scheduling: Scheduling, earned value analysis, risk management

1. Roger S. Pressman, “Software Engineering – A Practitioner’s Approach”, McGraw-Hill International Edition, 7th Edition, 2010.
2. Ian Somerville, “Software Engineering”, Pearson Education Asia, 9th Edition, 2011.

What is Software

- The product that software professionals build and then support over the long term.

- Software encompasses:

(1) instructions (computer programs) that when executed provide desired features, function, and performance;

(2) data structures that enable the programs to adequately store and manipulate information and

(3) documentation that describes the operation and use of the programs.

Software products

- Generic products
  - Stand-alone systems that are marketed and sold to any customer
- who wishes to buy them.
  - Examples – PC software such as editing, graphics programs, project management tools; CAD software; software for specific markets such as appointments systems for dentists.
- Customized products
  - Software that is commissioned by a specific customer to meet their own needs.
  - Examples – embedded control systems, air traffic control software, traffic monitoring systems.

3

Why Software is Important?

- The economies of ALL developed nations are dependent on software.
- More and more systems are software controlled (transportation, medical, telecommunications, military, industrial, entertainment, etc…)
- Software engineering is concerned with theories, methods and tools for professional software development.
- Expenditure on software represents asignificant fraction of Gross national product (**GNP**) in all developed countries.

Features of Software?

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

Software Myths

- **Software Myths**
  - **Software myths – beliefs about software and the process used to build it**
  - **Management Myths**
    - **Myth – We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?**
    - **Reality – The book of standards may very well exist but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it adaptable?**
    - **Myth – If we get behind schedule, we can add more programmers and catch up**
    - **Reality – Software development is not a mechanistic process like manufacturing. " Adding more people to a late software project makes it later"**

Software Myths

- **Software Myths**
  - **Software myths – beliefs about software and the process used to build it**
  - **Management Myths**
    - **Myth – If I decide to outsource the software project to a third party, I can relax and let that firm build it**
    - **Reality – If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it is outsourced.**

Software Myths

- **Software Myths**
  - **Customer Myths**
    - **Myth – A general statement of objectives is sufficient to begin writing programs – we can fill in the details later**
    - **Reality – An ambiguous statement of objectives is a recipe for disaster. Unambiguous requirements are developed only through effective and continuous communication between customer and developer**
    - **Myth – Project requirements change, but change can be easily accommodated because software is flexible**
    - **Reality – When requirement changes are requested early, cost impact is relatively small. With time, cost impact grows rapidly, and a change can cause additional resources and major design modifications**

Software Myths

- **Software Myths**
  - **Practitioner Myths**
    - **Myth – Once we write the program and get it to work, our job is done**
    - **Reality – The sooner you begin writing code, the longer it will take you to get done. Between 60 to 80 percent of all effort spent on software will be spent after it is delivered to the customer for the first time**
    - **Myth – Until I get the program running, I have no way of assessing its quality**
    - **Reality – Software reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of software errors**

Software Myths

- **Software Myths**
  - **Practitioner Myths**
    - **Myth – The only deliverable work product for a successful project is the working program**
    - **Reality – A working program is only one part of a software configuration that includes many elements. Documentation provides a foundation for successful engineering and guidance for software support**
    - **Myth – Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down**
    - **Reality – Software engineering is not about creating documents, it si about creating quality. Better quality leads to reduced rework. Reduced rework results in faster delivery times**

Software Applications

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

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).

Importance of Software Engineering

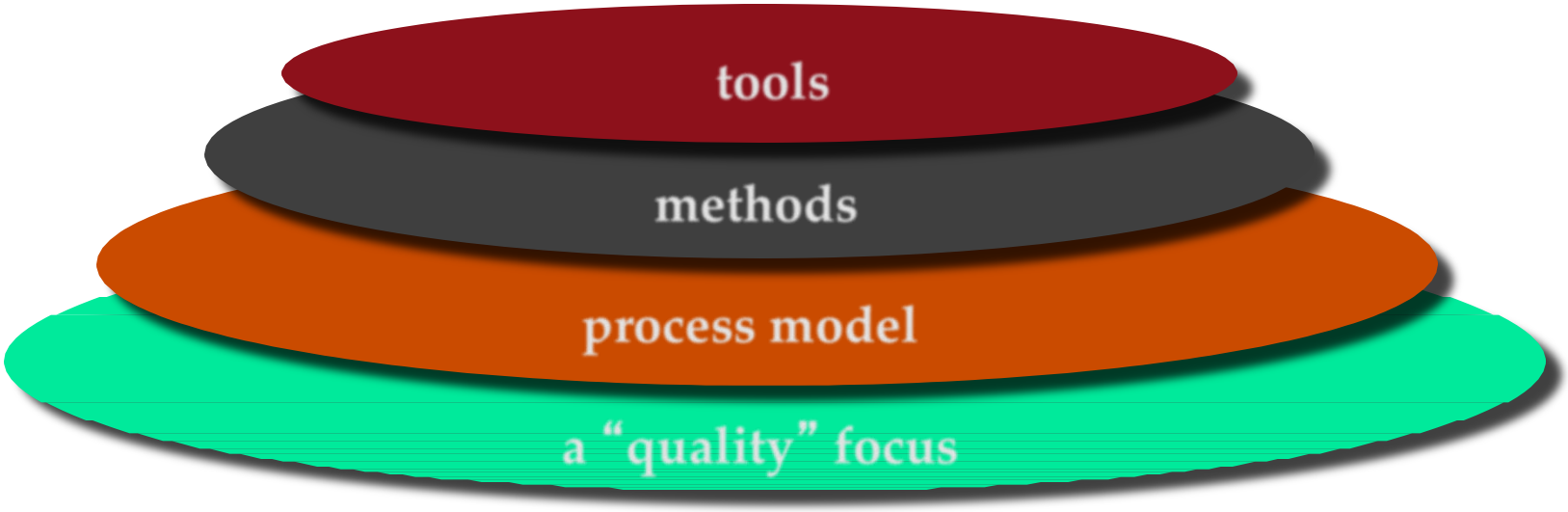
- More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.
- It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems rather than just write the programs as if it was a personal programming project. For most types of system, the majority of costs are the costs of changing the software after it has gone into use.

FAQ about software engineering

Question	Answer
What is software?	Computer programs, data structures and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.
What is software engineering?	Software engineering is an engineering discipline that is concerned with all aspects of software production.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
What is the difference between software and system engineering?	System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this more general process.

Software Engineering

A Layered Technology



- Any engineering approach must rest on organizational commitment to **quality** which fosters a continuous process improvement culture.
- **Process** layer as the foundation defines a framework with activities for effective delivery of software engineering technology. Establish the context where products (model, data, report, and forms) are produced, milestone are established, quality is ensured and change is managed.
- **Method** provides technical how-to's for building software. It encompasses many tasks including communication, requirement analysis, design modeling, program construction, testing and support.
- **Tools** provide automated or semi-automated support for the process and methods.

Software Process

- A process is a collection of activities, actions and tasks that are performed when some work product is to be created. It is **not a rigid prescription** for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work to pick and choose the **appropriate set of work actions** and tasks.
- Purpose of process is to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

Five Activities of a Generic Process Framework

- Communication: communicate with customer to understand objectives and gather requirements.
- Planning: creates a "map" defines the work by describing the tasks, risks and resources, work products and work schedule.
- Modeling: Create a "sketch", what it looks like architecturally, how the constituent parts fit together and other characteristics.
- Construction: code generation and the testing.
- Deployment: Delivered to the customer who evaluates the products and provides feedback based on the evaluation.

- These five framework activities can be used to all software development regardless of the application domain, size of the project, complexity of the efforts etc, though the details will be different in each case.

- For many software projects, these framework activities are applied **iteratively** as a project progresses. Each iteration produces a software increment that provides a subset of overall software features and functionality.

Umbrella Activities

Complement the five process framework activities and help team manage and control progress, quality, change, and risk.

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

Adapting a Process Model

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

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

Prescriptive and Agile Process Models

•The **prescriptive process** models stress detailed definition, identification, and application of process activates and tasks. Intent is to improve system quality, make projects more manageable, make delivery dates and costs more predictable, and guide teams of software engineers as they perform the work required to build a system.

•Unfortunately, there have been times when these objectives were not achieved. If prescriptive models are applied dogmatically and without adaptation, they can increase the level of bureaucracy.

•**Agile process models** emphasize project "agility" and follow a set of principles that lead to a more informal approach to software process. It emphasizes maneuverability and adaptability. It is particularly useful when Web applications are engineered.

Understand the Problem

- *Who has a stake in the solution to the problem?* That is, who are the stakeholders?
- *What are the unknowns?* What data, functions, and features are required to properly solve the problem?
- *Can the problem be compartmentalized?* Is it possible to represent smaller problems that may be easier to understand?
- *Can the problem be represented graphically?* Can an analysis model be created?

Plan the Solution

- *Have you seen similar problems before?* Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- *Has a similar problem been solved?* If so, are elements of the solution reusable?
- *Can subproblems be defined?* If so, are solutions readily apparent for the subproblems?
- *Can you represent a solution in a manner that leads to effective implementation?* Can a design model be created?

Carry Out the Plan

- *Does the solutions conform to the plan?* Is source code traceable to the design model?
- *Is each component part of the solution provably correct?* Has the design and code been reviewed, or better, have correctness proofs been applied to algorithm?

Examine the Result

- *Is it possible to test each component part of the solution?* Has a reasonable testing strategy been implemented?
- *Does the solution produce results that conform to the data, functions, and features that are required?* Has the software been validated against all stakeholder requirements?

What / who / why is Process Models?

What: Go through a series of predictable steps--- a road map that helps you create a timely, high-quality results.

Who: Software engineers and their managers, clients also. People adapt the process to their needs and follow it.

Why: Provides stability, control, and organization to an activity that can if left uncontrolled, become quite chaotic. However, modern software engineering approaches must be agile and demand ONLY those activities, controls and work products that are appropriate.

What Work products: Programs, documents, and data

What are the steps: The process you adopt depends on the software that you are building. One process might be good for aircraft avionic system, while an entirely different process would be used for website creation.

How to ensure right: A number of software process assessment mechanisms that enable us to determine the maturity of the software process. However, the quality, timeliness and long-term viability of the software are the best indicators of the efficacy of the process you use.

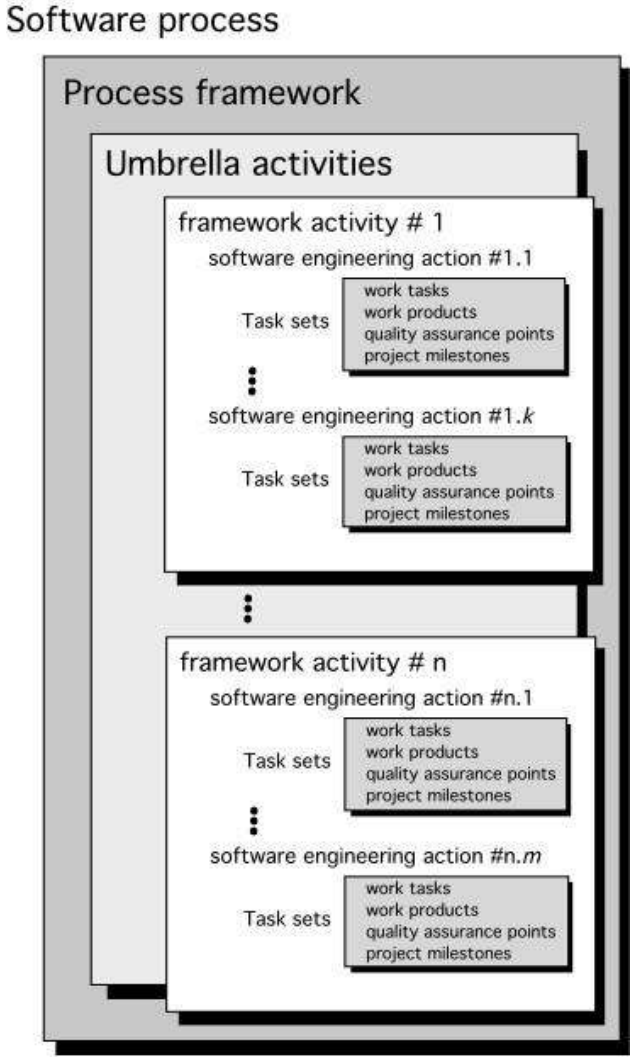
Definition of Software Process

- A **framework** for the activities, actions, and tasks that are required to build high-quality software.

- SP defines the approach that is taken as software is engineered.

- Is not equal to software engineering, which also encompasses **technologies** that populate the process– technical methods and automated tools.

A Generic Process Model



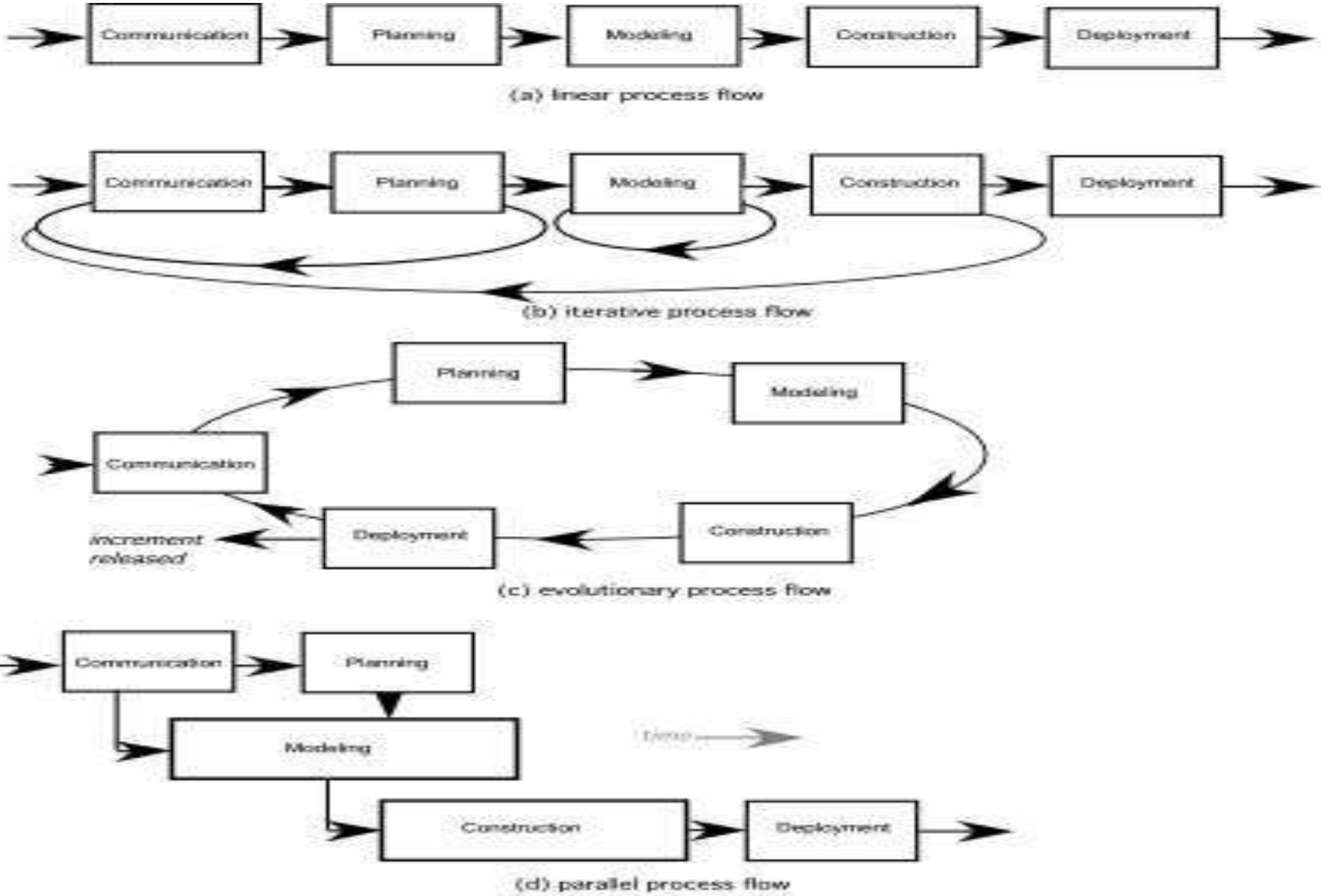
A Generic Process Model

A generic process framework for software engineering defines five framework activities- communication, planning, modeling, construction, and deployment.

In addition, a set of umbrella activities- project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others are applied throughout the process.

Next question is: how the framework activities and the actions and tasks that occur within each activity are organized with respect to sequence and time? See the **process flow** for answer.

Process Flow



Process Flow

1. Linear process flow executes each of the five activities in sequence.

2. An iterative process flow repeats one or more of the activities before proceeding to the next.

3. An evolutionary process flow executes the activities in a circular manner. Each circuit leads to a more complete version of the software.

4. A parallel process flow executes one or more activities in parallel with other activities (modeling for one aspect of the software in parallel with construction of another aspect of the software.

Identifying a Task Set

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

- A task set defines the actual work to be done to accomplish the objectives of a software engineering action.
- A list of the task to be accomplished
- A list of the work products to be produced
- A list of the quality assurance filters to be applied

Identifying a Task Set

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

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

Example of a Task Set for Elicitation

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

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

Example of a Task Set for Elicitation

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

- Both do the same work with different depth and formality. Choose the task sets that achieve the goal and still maintainquality and agility.

Process Patterns

- A *process pattern*
  - describes a process-related problem that is encountered during software engineering work,
  - identifies the environment in which the problem has been encountered, and
  - suggests one or more proven solutions to the problem.
- Stated in more general terms, a process pattern provides you with a *template* [Amb98]—a consistent method for describing problem solutions within the context of the software process.

(defined at different levels of abstraction)

1. Problems and solutions associated with a complete process model (e.g. prototyping).
2. Problems and solutions associated with a framework activity (e.g. planning) or
3. an action with a framework activity (e.g. project estimating).

Process Pattern Types

- *Stage patterns*—defines a problem associated with a framework activity for the process. It includes multiple task patterns as well. For example, EstablishingCommunication would incorporate the task pattern RequirementsGathering and others.
- *Task patterns*—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice
- *Phase patterns*—define the sequence of framework activities that occur with the process, even when the overall flow of activities is iterative in nature. Example includes SprialModel or Prototyping.

An Example of Process Pattern

- **Describes an approach that may be applicable when stakeholders have a general idea of what must be done but are unsure of specific software requirements.**
- **Pattern name. Requirement Unclear**
- **Intent. This pattern describes an approach for building a model that can be assessed iteratively by stakeholders in an effort to identify or solidify software requirements.**
- **Type. Phase pattern**
- **Initial context. Conditions must be met (1) stakeholders have been identified; (2) a mode of communication between stakeholders and the software team has been established; (3) the overriding software problem to be solved has been identified by stakeholders ; (4) an initial understanding of project scope, basic business requirements and project constraints has been developed.**
- **Problem. Requirements are hazy or nonexistent. stakeholders are unsure of what they want.**
- **Solution. A description of the prototyping process would be presented here.**
- **Resulting context. A software prototype that identifies basic requirements. (modes of interaction, computational features, processing functions) is approved by stakeholders. Following this, 1. This prototype may evolve through a series of increments to become the production software or 2. the prototype may be discarded.**
- **Related patterns. CustomerCommunication, IterativeDesign, IterativeDevelopment, Customer Assessment, Requirement Extraction.**

Process Assessment and Improvement

- The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics.
- A number of different approaches to software process assessment and improvement have been proposed over the past few decades:
- **Standard CMMI Assessment Method for Process Improvement (SCAMPI)—**provides a five-step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting, and learning. The SCAMPI method uses the SEI CMMI as the basis for assessment [SEI00].
- **CMM-Based Appraisal for Internal Process Improvement (CBA IPI)—** provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment [Dun01].

Process Assessment and Improvement

- **SPICE (ISO/IEC15504)—**a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process [ISO08].
- **ISO 9001:2000 for Software—**a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies [Ant06].

Prescriptive Models

- Originally proposed to bring order to chaos.
- Prescriptive process models advocate an orderly approach to software engineering. However, will some extent of chaos (less rigid) be beneficial to bring some creativity?

That leads to a few questions ...

- If prescriptive process models strive for structure and order (prescribe a set of process elements and process flow), are they inappropriate for a software world that thrives on change?

- Yet, if we reject traditional process models (and the order they imply) and replace them with something less structured, do we make it impossible to achieve coordination and coherence in software work?

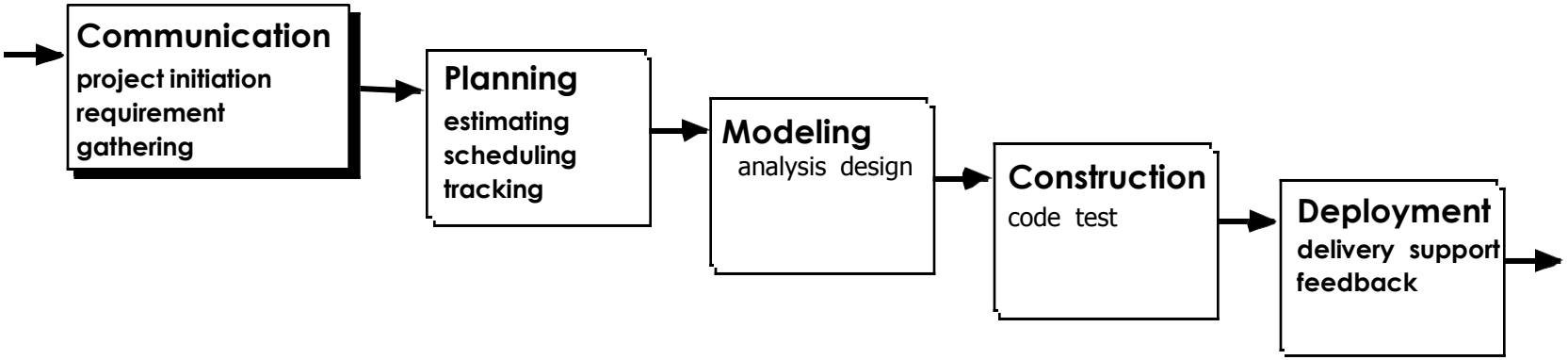
Software Process Models

- **Classic Process Models**
  - **Waterfall Model (Linear Sequential Model)**
- **Incremental Process Models**
  - **Incremental Model**

- **Evolutionary Software Process Models**
  - **Prototyping**
  - **Spiral Model**
  - **Concurrent Development Model**

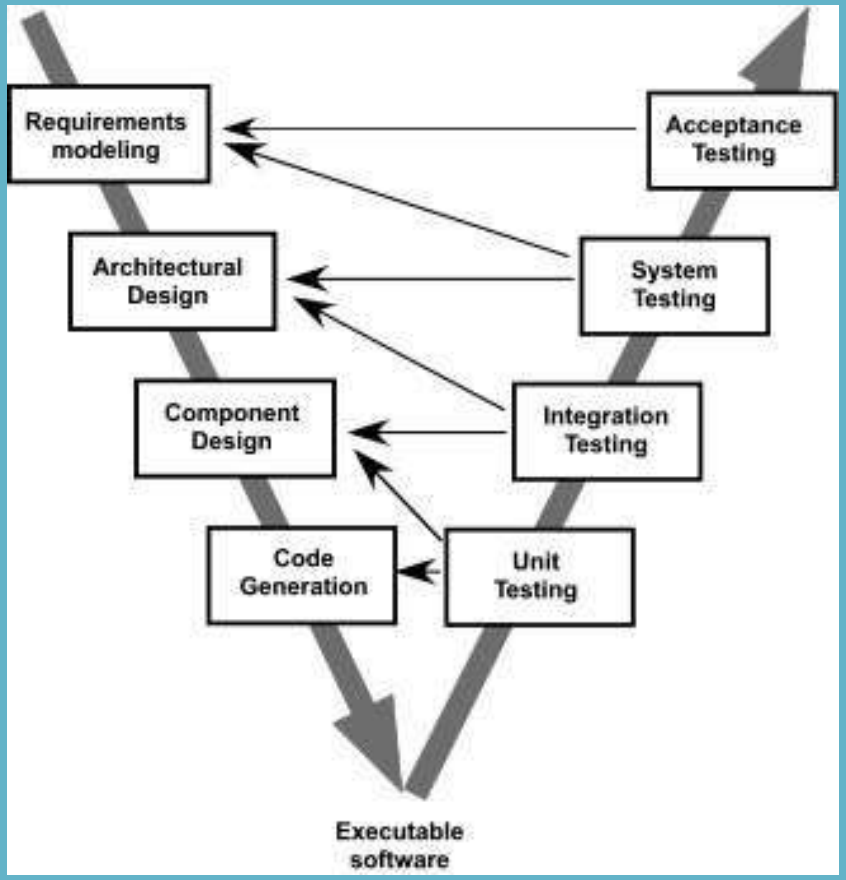
- **Specialized Process Models**
  - **Component-Based Development**
  - **Formal Methods Model**
  - **Aspect-Oriented Software Development**

The Waterfall Model



- The waterfall model, sometimes called the classic life cycle.
- It is the oldest paradigm for Software Engineering. When requirements are well defined and reasonably stable, it leads to a linear fashion
- The waterfall model, sometimes called the classic life cycle, suggests a systematic,sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software.

The V-Model



A variation of waterfall model depicts the relationship of quality assurance actions to the actions associated with communication, modeling and early code construction activates.

Team first moves down the left side of the V to refine the problem requirements. Once code is generated, the team moves up the right side of the V, performing a series of tests that validate each of the models created as the team moved down the left side. The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.

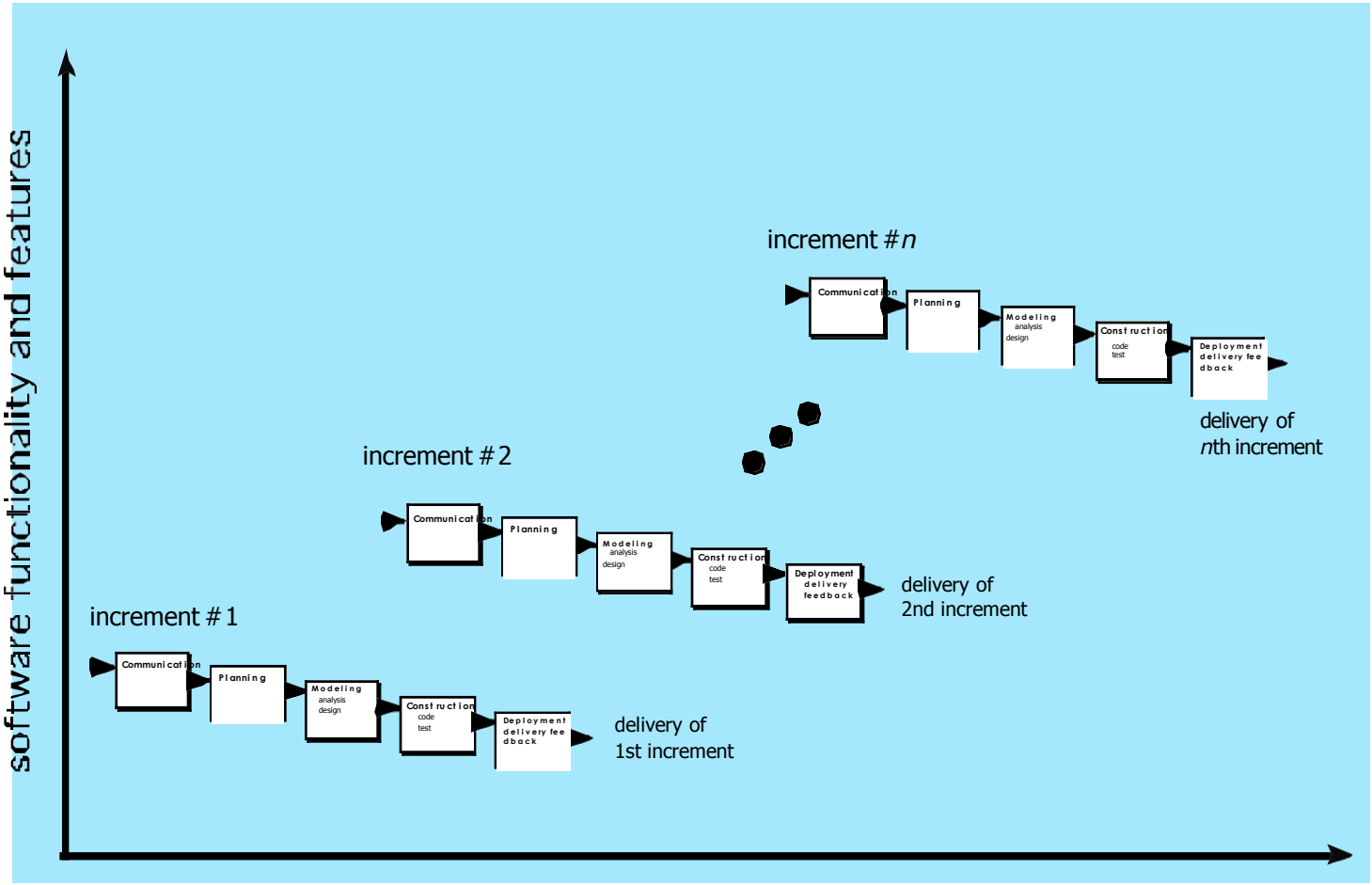
The problems that are sometimes encountered when the waterfall model is applied are:

- Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.

- It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

- The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

The Incremental Model



The Incremental Model

- When initial requirements are reasonably well defined, but the overall scope of the development effort precludes a purely linear process. A compelling need to expand a limited set of new functions to a later system release.
- It combines elements of linear and parallel process flows. Each linear sequence produces deliverable increments of the software.
- The first increment is often a core product with many supplementary features. Users use it and evaluate it with more modifications to better meet the needs.
- The incremental process model focuses on the delivery of an operational product with each increment. Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.
- Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project

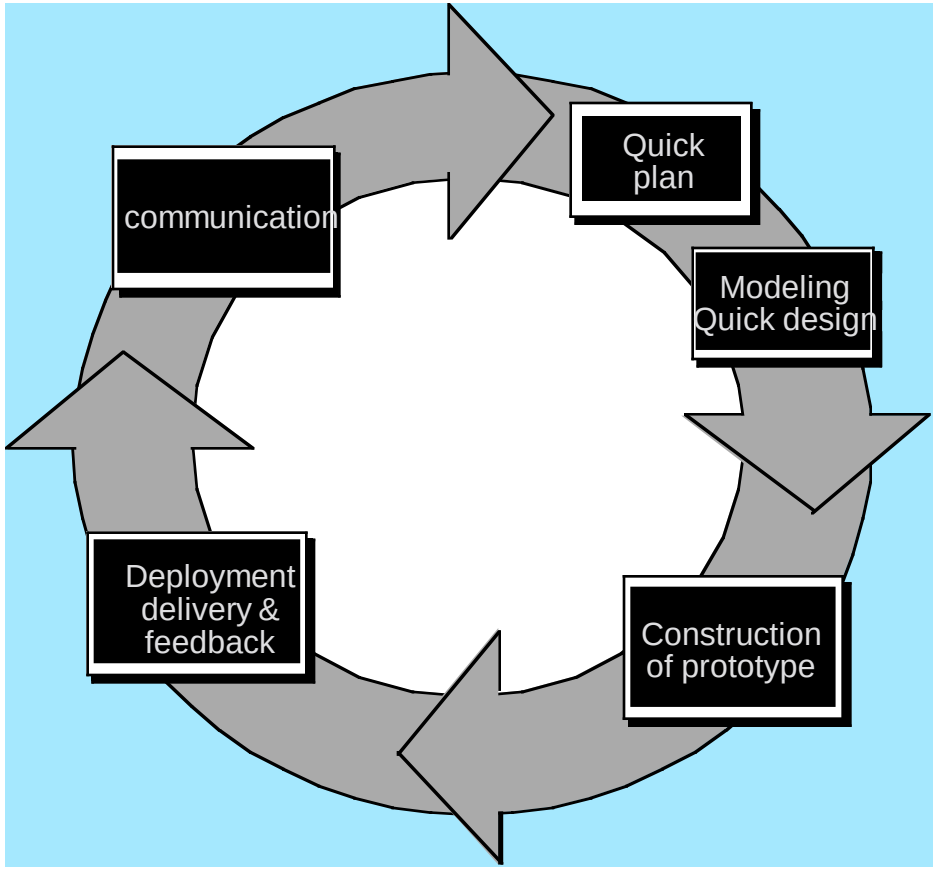
Evolutionary Models

- Software system evolves over time as requirements often change as development proceeds. Thus, a straight line to a complete end product is not possible. However, a limited version must be delivered to meet competitive pressure.
- Usually a set of core product or system requirements is well understood, but the details and extension have yet to be defined.
- You need a process model that has been explicitly designed to accommodate a product that evolved over time.
- It is iterative that enables you to develop increasingly more complete version of the software.
- Two types are introduced, namely Prototyping and Spiral models.

Evolutionary Models: Prototyping

- When to use: Customer defines a set of general objectives but does not identify detailed requirements for functions and features. or Developer may be unsure of the efficiency of an algorithm, the form that human computer interaction should take.
- What step: Begins with communication by meeting with stakeholders to define the objective, identify whatever requirements are known, outline areas where further definition is mandatory. A quick plan for prototyping and modeling (quick design) occur. Quick design focuses on a representation of those aspects the software that will be visible to end users. (interface and output). Design leads to the construction of a prototype which will be deployed and evaluated. Stakeholder's comments will be used to refine requirements.
- Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something immediately. However, engineers may make compromises in order to get a prototype working quickly. The less-than-ideal choice may be adopted forever after you get used to it.

Evolutionary Models: Prototyping



Prototyping can be problematic for the following reasons:

- Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven't considered overall software quality or long-term maintainability.

- As a software engineer, you often make implementation compromises in order to get a prototype working quickly.

- An inappropriate operating system or programming language may be used simply because it is available and known;

- An inefficient algorithm may be implemented simply to demonstrate capability. After a time, you may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system

Evolutionary Models: The Spiral

- It couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model and is a risk-driven process model generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems.
- Two main distinguishing features: one is cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.
- A series of evolutionary releases are delivered. During the early iterations, the release might be a model or prototype. During later iterations, increasingly more complete version of the engineered system are produced.

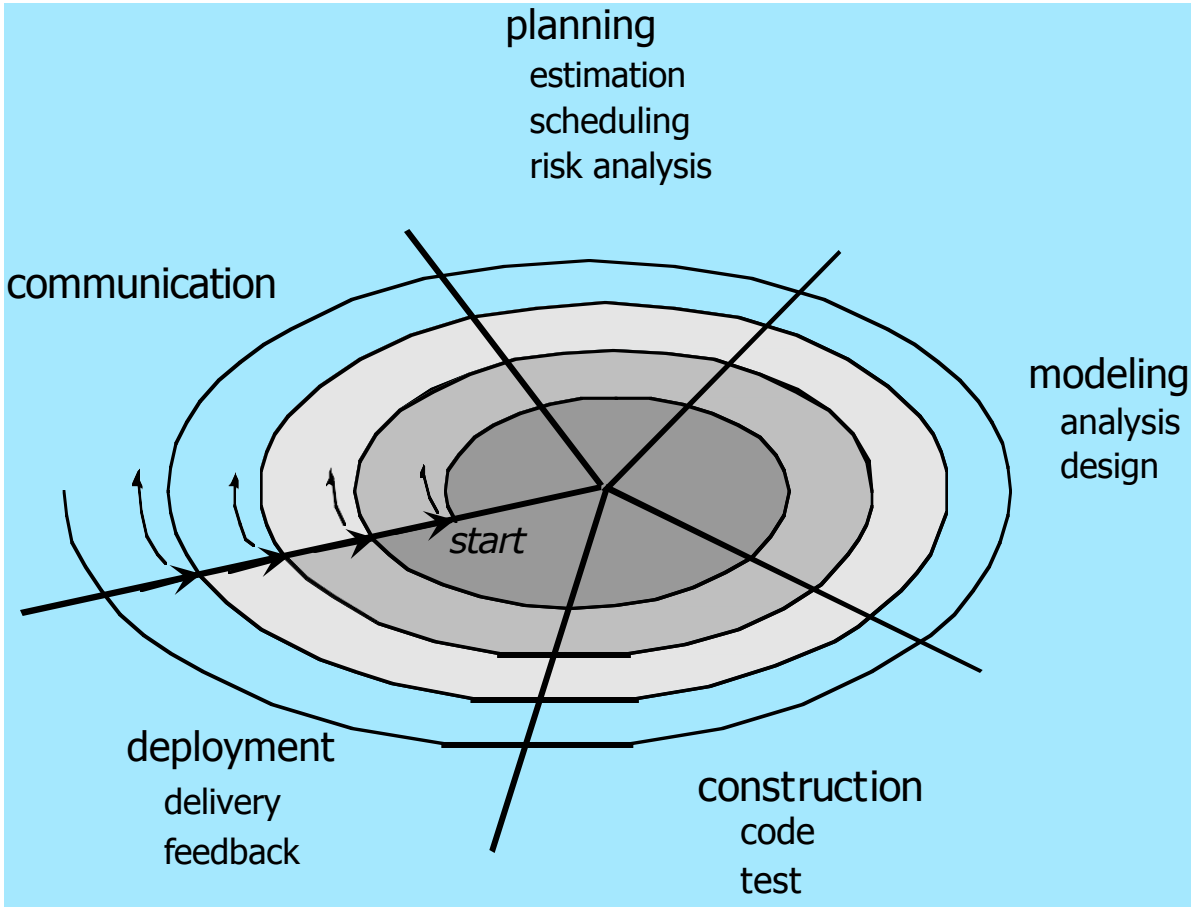
- The first circuit in the clockwise direction might result in the product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.

- Each pass results in adjustments to the project plan. Cost and schedule are adjusted based on feedback. Also, the number of iterations will be adjusted by project manager.

- Good to develop large-scale system as software evolves as the process progresses and risk should be understood and properly reacted to. Prototyping is used to reduce risk.

- However, it may be difficult to convince customers that it is controllable as it demands considerable risk assessment expertise.

Evolutionary Models: The Spiral



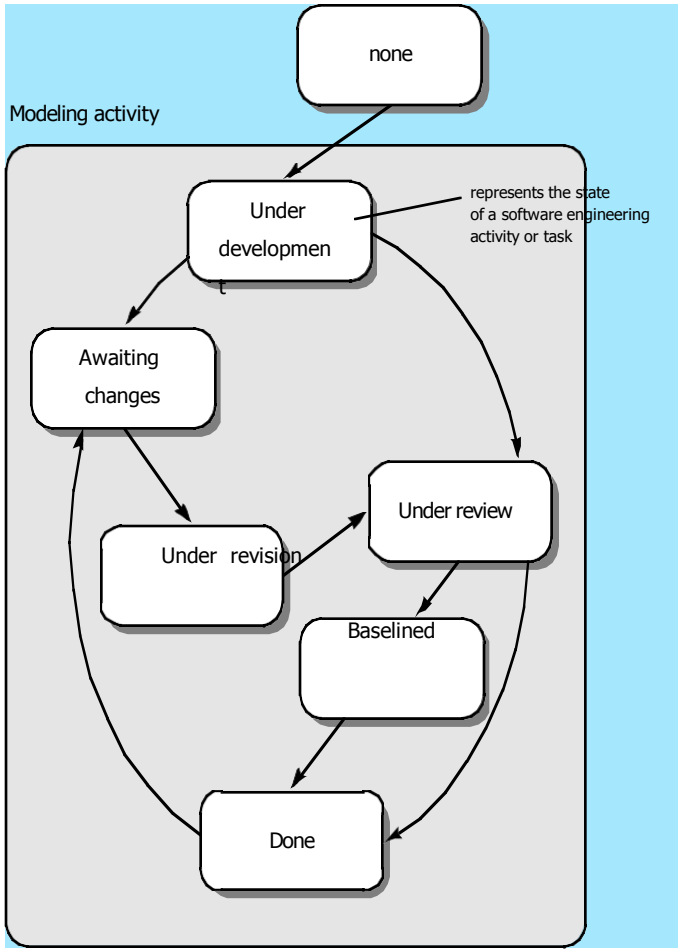
Three Concerns on Evolutionary Processes

- First concern is that prototyping poses a problem to project planning because of the uncertain number of cycles required to construct the product.
- Second, it does not establish the maximum speed of the evolution. If the evolution occur too fast, without a period of relaxation, it is certain that the process will fall into chaos. On the other hand if the speed is too slow then productivity could be affected.
- Third, software processes should be focused on flexibility and extensibility rather than on high quality. We should prioritize the speed of the development over zero defects. Extending the development in order to reach high quality could result in a late delivery of the product when the opportunity niche has disappeared.

Concurrent Model

- Allow a software team to represent iterative and concurrent elements of any of the process models. For example, the modeling activity defined for the spiral model is accomplished by invoking one or more of the following actions: prototyping, analysis and design.
- The Figure shows modeling may be in any one of the states at any given time. For example, communication activity has completed its first iteration and in the awaiting changes state. The modeling activity was in inactive state, now makes a transition into the under development state. If customer indicates changes in requirements, the modeling activity moves from the under development state into the awaiting changes state.
- Concurrent modeling is applicable to all types of software development and provides an accurate picture of the current state of a project. Rather than confining software engineering activities, actions and tasks to a sequence of events, it defines a process network. Each activity, action or task on the network exists simultaneously with other activities, actions or tasks. Events generated at one point trigger transitions among the state.

Concurrent Model



SPECIALIZED PROCESS MODELS

- Component-Based Development
- The Formal Methods Model
- Aspect-Oriented Software Development

58

SPECIALIZED PROCESS MODELS

•Component-Based Development

Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built.

These components can be as either conventional software modules or object-oriented packages or packages of classes

- Steps involved in CBS are
  - Available component-based products are researched and evaluated for the application domain in question.
  - Component Integration issues are considered.

59

SPECIALIZED PROCESS MODELS

- Component-Based Development
  - Steps involved in CBS are
    - A software architecture is designed to accommodate the components
    - Components are integrated into the architecture
    - Comprehensive testing is conducted to ensure proper functionality

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits

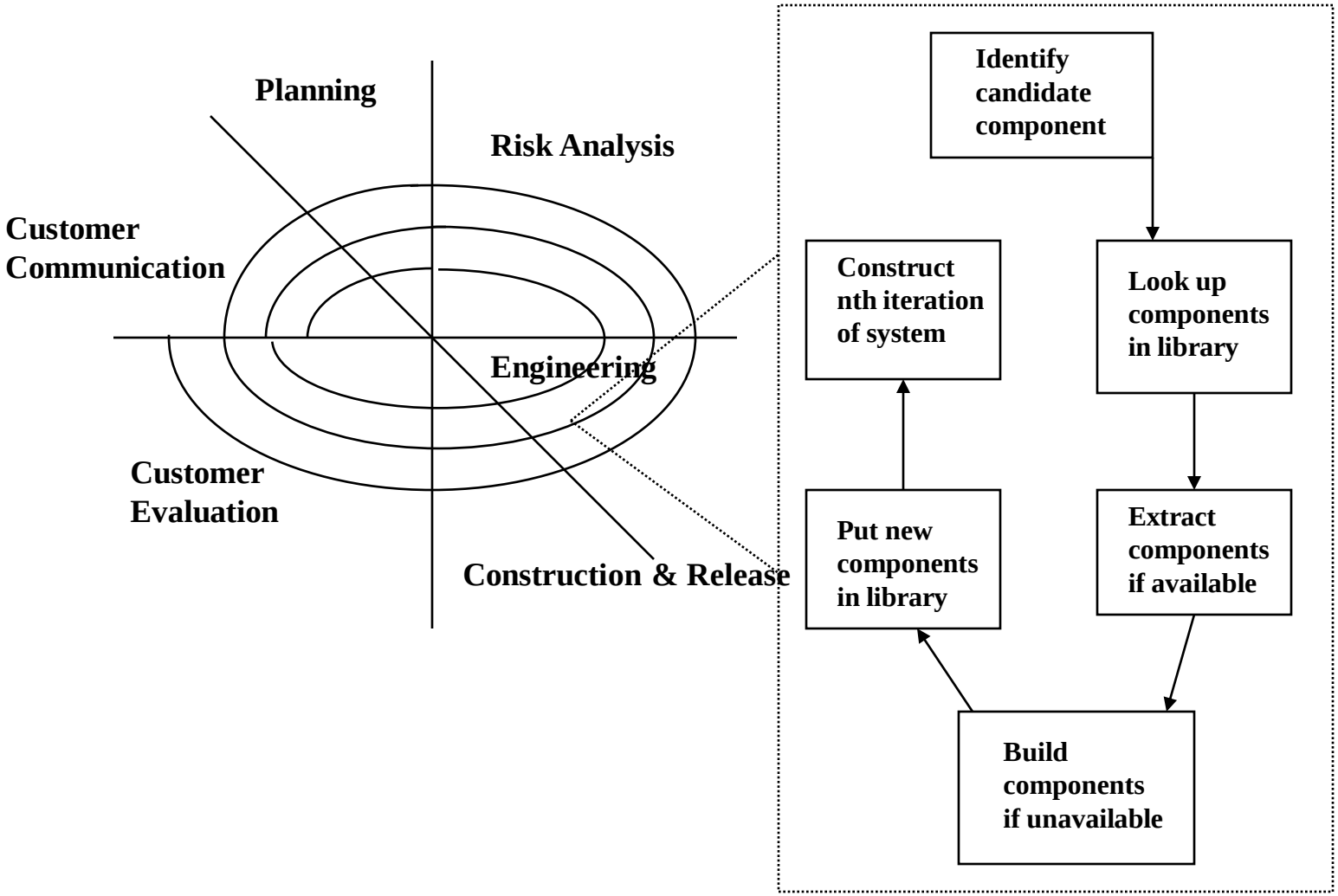
SPECIALIZED PROCESS MODELS

•Component-Based Development

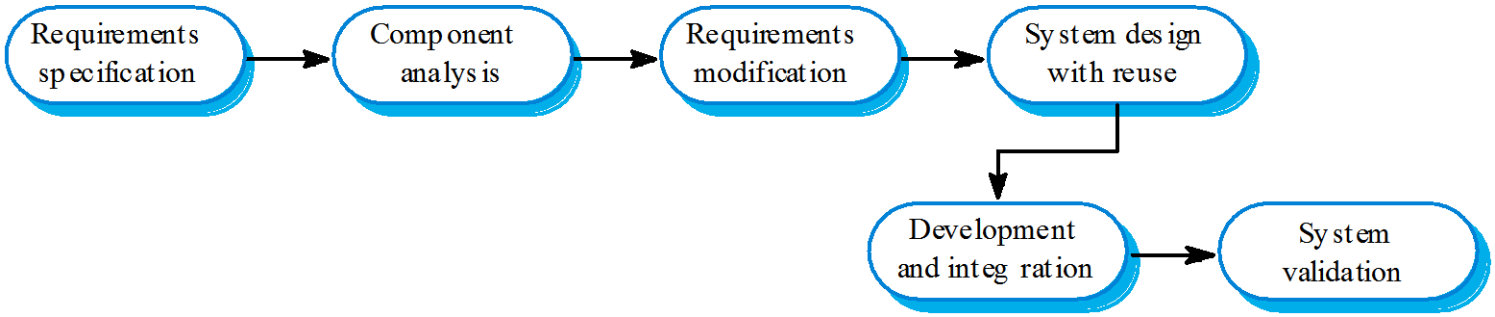
- Component-based development model leads to software reuse and reusability helps software engineers with a number of measurable benefits
- Component-based development leads to a 70 percent reduction in development cycle time, 84 percent reduction in project cost and productivity index of 26.2 compared to an industry norm of 16.9

61

The Component Assembly Model



Reuse-oriented development



SPECIALIZED PROCESS MODELS

- **Formal Methods Model**
  - **Formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software**
  - **They enable software engineers to specify, develop and verify a computer-based system by applying a rigorous mathematical notation**
- **Development of formal models is quite time consuming and expensive**
- **Extensive training is needed in applying formal methods**
- **Difficult to use the model as a communication mechanism for technically unsophisticated customers**

Unified Process

- **The Unified Process is an iterative and incremental development process. Unified Process divides the project into four phases**
  - **Inception**
  - **Elaboration**
  - **Construction**
  - **Transition**
- **The Inception, Elaboration, Construction and Transition phases are divided into a series of timeboxed iterations. (The Inception phase may also be divided into iterations for a large project.)**
- **Each iteration results in an *increment*, which is a release of the system that contains added or improved functionality compared with the previous release.**
- **Although most iterations will include work in most of the process disciplines (*e.g.* Requirements, Design, Implementation, Testing) the relative effort and emphasis will change over the course of the project.**

Unified Process

- **Risk Focused**
  - **The Unified Process requires the project team to focus on addressing the most critical risks early in the project life cycle. The deliverables of each iteration, especially in the Elaboration phase, must be selected in order to ensure that the greatest risks are addressed first. Risk Focused**

Unified Process

- **Inception Phase**
  - **Inception is the smallest phase in the project, and ideally it should be quite short. If the Inception Phase is long then it is usually an indication of excessive up-front specification, which is contrary to the spirit of the Unified Process.**
  - **The following are typical goals for the Inception phase.**
    - **Establish a justification or business case for the project**
    - **Establish the project scope and boundary conditions**
    - **Outline the use cases and key requirements that will drive the design tradeoffs**
    - **Outline one or more candidate architectures**
    - **Identify risks**
    - **Prepare a preliminary project schedule and cost estimate**
  - **The Lifecycle Objective Milestone marks the end of the Inception phase.**

Unified Process

- **Elaboration Phase**
  - **During the Elaboration phase the project team is expected to capture a majority of the system requirements. The primary goals of Elaboration are to address known risk factors and to establish and validate the system architecture.**
  - **Common processes undertaken in this phase include the creation of use case diagrams, conceptual diagrams (class diagrams with only basic notation) and package diagrams (architectural diagrams).**
  - **The architecture is validated primarily through the implementation of an Executable Architectural Baseline. This is a partial implementation of the system which includes the core, most architecturally significant, components. It is built in a series of small, timeboxed iterations.**

Unified Process

- **Elaboration Phase**
  - **By the end of the Elaboration phase the system architecture must have stabilized and the executable architecture baseline must demonstrate that the architecture will support the key system functionality and exhibit the right behavior in terms of performance, scalability and cost.**
  - **The final Elaboration phase deliverable is a plan (including cost and schedule estimates) for the Construction phase. At this point the plan should be accurate and credible, since it should be based on the Elaboration phase experience and since significant risk factors should have been addressed during the Elaboration phase.**
  - **The Lifecycle Architecture Milestone marks the end of the Elaboration phase.**

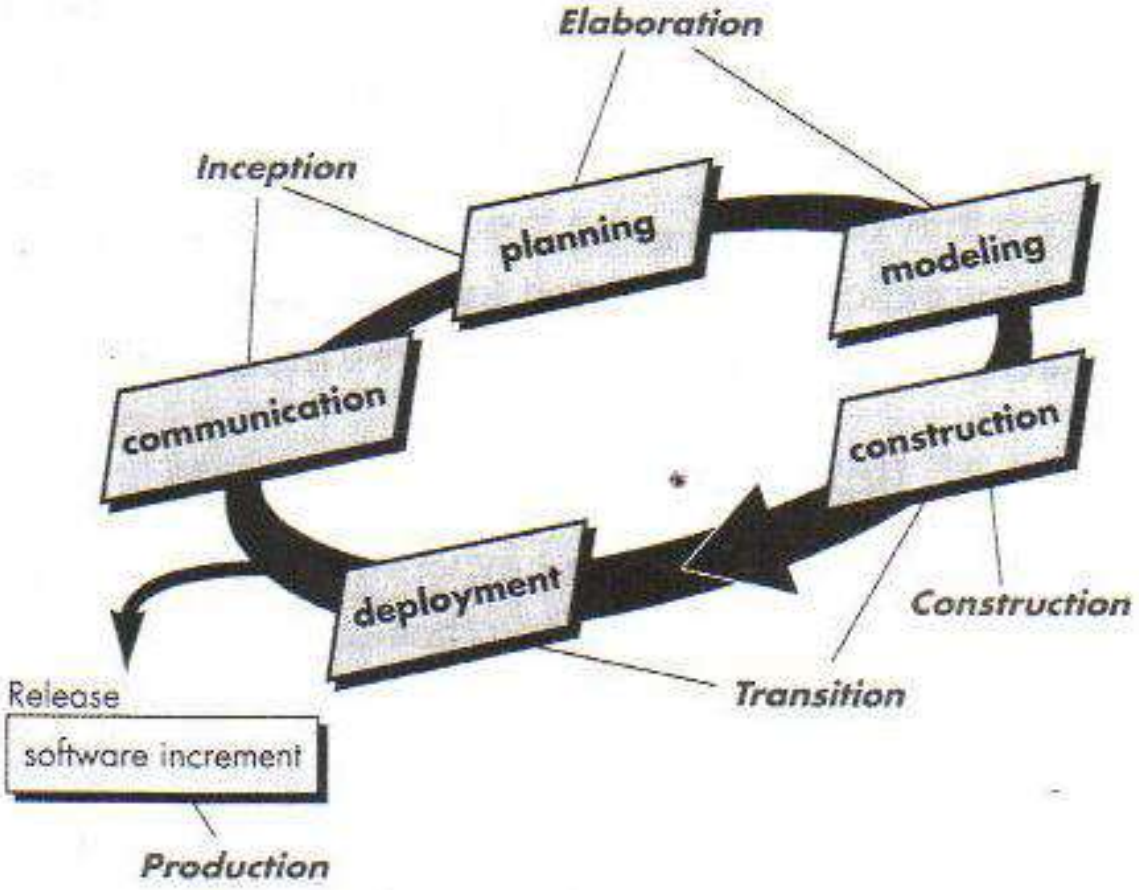
Unified Process

- **Construction Phase**
  - **Construction is the largest phase in the project. In this phase the remainder of the system is built on the foundation laid in Elaboration. System features are implemented in a series of short, timeboxed iterations. Each iteration results in an executable release of the software. It is customary to write full text use cases during the construction phase and each one becomes the start of a new iteration.**
  - **Common UML (Unified Modeling Language) diagrams used during this phase include Activity, Sequence, Collaboration, State (Transition) and Interaction Overview diagrams.**
  - **The Initial Operational Capability Milestone marks the end of the Construction phase.**

Unified Process

- **Transition Phase**
  - **The final project phase is Transition. In this phase the system is deployed to the target users. Feedback received from an initial release (or initial releases) may result in further refinements to be incorporated over the course of several Transition phase iterations. The Transition phase also includes system conversions and user training.**
  - **The Product Release Milestone marks the end of the Transition phase.**

Unified Process



Unified Process

- **Advantages of UP Software Development**
  - **This is a complete methodology in itself with an emphasis on accurate documentation**
  - **It is proactively able to resolve the project risks associated with the client's evolving requirements requiring careful change request management**
  - **Less time is required for integration as the process of integration goes on throughout the software development life cycle.**
  - **The development time required is less due to reuse of components.**

Unified Process

- **Disadvantages of RUP Software Development**
  - **The team members need to be expert in their field to develop a software under this methodology.**
  - **On cutting edge projects which utilise new technology, the reuse of components will not be possible. Hence the time saving one could have made will be impossible to fulfill.**
  - **Integration throughout the process of software development, in theory sounds a good thing. But on particularly big projects with multiple development streams it will only add to the confusion and cause more issues during the stages of testing**

What is estimation?

Estimation is attempt to determine how much money, effort, resources & time it will take to build a specific software based system or project.

Estimation involves answering the following questions:

1. How much effort is required to complete each activity?
2. How much calendar time is needed to complete each activity?
3. What is the total cost of each activity?

Project cost estimation and project scheduling are normally carried out together.

The costs of development are primarily the costs of the effort involved, so the effort computation is used in both the cost and the schedule estimate.

Do some cost estimation before detailed schedules are drawn up.

These initial estimates may be used to establish a budget for the project or to set a price for the software for a customer

There are three parameters involved in computing the total cost of a software development project:

- Hardware and software costs including maintenance
- Travel and training costs
- Effort costs (the costs of paying software engineers).

The following costs are all part of the total effort cost:

1. Costs of providing, heating and lighting office space
2. Costs of support staff such as accountants, administrators, system managers, cleaners and technicians
3. Costs of networking and communications
4. Costs of central facilities such as a library or recreational facilities
5. Costs of Social Security and employee benefits such as pensions and health insurance.

Factors affecting software pricing

Factor	Description
Market opportunity	A development organisation may quote a low price because it wishes to move into a new segment of the software market. Accepting a low profit on one project may give the organisation the opportunity to make a greater profit later. The experience gained may also help it develop new products.
Cost estimate uncertainty	If an organisation is unsure of its cost estimate, it may increase its price by some contingency over and above its normal profit.
Contractual terms	A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged may then be less than if the software source code is handed over to the customer.
Requirements volatility	If the requirements are likely to change, an organisation may lower its price to win a contract. After the contract is awarded, high prices can be charged for changes to the requirements.
Financial health	Developers in financial difficulty may lower their price to gain a contract. It is better to make a smaller than normal profit or break even than to go out of business.

COCOMO Models Motivation

The software cost estimation provides:

- The vital link between the general concepts and techniques of **economic analysis** and the particular world of **software engineering.**
- Software cost estimation techniques also provides an essential part of the foundation for **good software management**.

Cost of a project

- The cost in a project is due to:
  - due the requirements for software, hardware and human resources
  - the cost of software development is due to the human resources needed
  - most cost estimates are measured in ***person-months (PM)***
  - the cost of the project depends on the nature and characteristics of the project, at any point, the accuracy of the estimate will depend on the amount of reliable information we have about the final product.

Software Cost Estimation

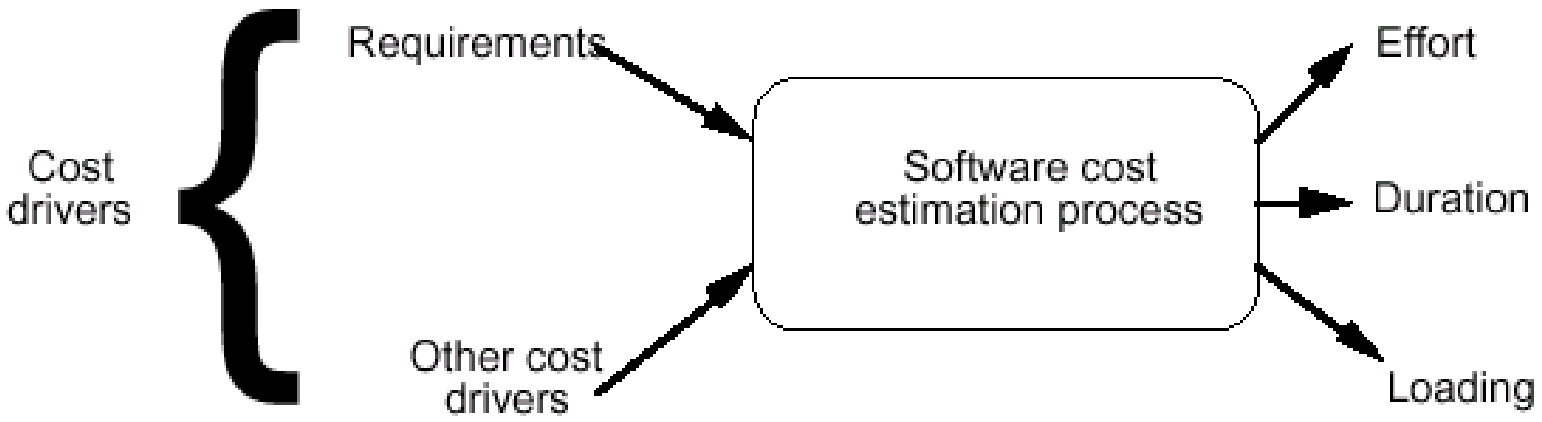


Figure 1. Classical view of software estimation process.

Introduction to COCOMO models

- The **Constructive Cost Model** (COCOMO) is the most widely used software estimation model in the world.
- The COCOMO model predicts the **effort** and **duration** of a project based on inputs relating to the size of the resulting systems and a number of "**cost drives**" that affect productivity.

Effort

- Effort Equation
  - **PM = C * (KDSI)^n (**person-months)
    - where **PM** = number of person-month (=152 working hours),
    - **C** = a constant,
    - **KDSI** = thousands of "delivered source instructions" (DSI) and
    - **n** = a constant.

Introduction to COCOMO models

- Productivity equation
  - **(DSI) / (PM)**
    - where **PM** = number of person-month (=152 working hours),
    - **DSI** = "delivered source instructions“
- Schedule equation
  - **TDEV = C * (PM)^{n} (**months)
    - where TDEV = number of months estimated for software development.
- Average Staffing Equation
  - **(PM) / (TDEV)** **(**FSP)
    - where FSP means Full-time-equivalent Software Personnel.

COCOMO Models

- COCOMO is defined in terms of three different models:
  - the **Basic model**,
  - the **Intermediate model**, and
  - the **Detailed model**.
- The more complex models account for more factors that influence software projects, and make more accurate estimates.

The Development mode

- The most important factors contributing to a project's duration and cost is the Development Mode
    - **Organic Mode:** The project is developed in a familiar, stable environment, and the product is similar to previously developed products. The product is relatively small, and requires little innovation.
    - **Semidetached Mode:** The project's characteristics are intermediate between Organic and Embedded.
    - **Embedded Mode:** The project is characterized by tight, inflexible constraints and interface requirements. An embedded mode project will require a great deal of innovation.

Modes

Feature	Organic	Semidetached	Embedded
Organizational understanding of product and objectives	Thorough	Considerable	General
Experience in working with related software systems	Extensive	Considerable	Moderate
Need for software conformance with pre-established requirements	Basic	Considerable	Full
Need for software conformance with external interface specifications	Basic	Considerable	Full

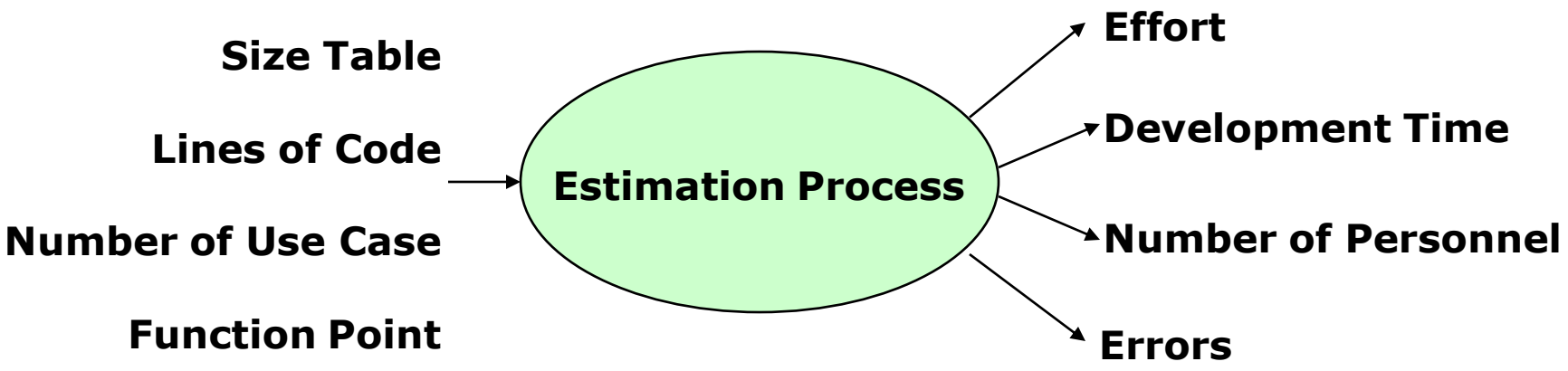
Modes (.)

Feature	Organic	Semidetached	Embedded
Concurrent development of associated new hardware and operational procedures	Some	Moderate	Extensive
Need for innovative data processing architectures, algorithms	Minimal	Some	Considerable
Premium on early completion	Low	Medium	High
Product size range	<50 KDSI	<300KDSI	All

Cost Estimation Process

Cost= Size Of The Project x Productivity

Cost Estimation Process



Function Points

- **STEP 1:** measure size in terms of the amount of functionality in a system. Function points are computed by first calculating an ***unadjusted function point count*** **(UFC).** Counts are made for the following categories
  - *External inputs* – those items provided by the user that describe distinct application-oriented data (such as file names and menu selections)
  - *External outputs* – those items provided to the user that generate distinct application-oriented data (such as reports and messages, rather than the individual components of these)
  - *External inquiries* – interactive inputs requiring a response
  - *External files* – machine-readable interfaces to other systems
  - *Internal files* – logical master files in the system

Function Points(..)

- **STEP 2:** Multiply each number by a weight factor, according to complexity (**simple**, **average** or **complex**) of the parameter, associated with that number. The value is given by a table:

Parameter	simple	average	complex
users inputs	3	4	6
users outputs	4	5	7
users requests	3	4	6
files	7	10	15
external interfaces	5	7	10

- **STEP 3:** Calculate the total **UFP** (Unadjusted Function Points)

- **STEP 4:** Calculate the total **TCF** (Technical Complexity Factor) by giving a value between 0 and 5 according to the importance of the following points:

Function Points(....)

- **Technical Complexity Factors:**
  - **1. Data Communication**
  - **2. Distributed Data Processing**
  - **3. Performance Criteria**
  - **4. Heavily Utilized Hardware**
  - **5. High Transaction Rates**
  - **6. Online Data Entry**
  - **7. Online Updating**
  - **8. End-user Efficiency**
  - **9. Complex Computations**
  - **10. Reusability**
  - **11. Ease of Installation**
  - **12. Ease of Operation**
  - **13. Portability**
  - **14. Maintainability**

Function Points(.....)

- **STEP 5:** Sum the resulting numbers too obtain **DI** (degree of influence)
- **STEP 6: TCF** (Technical Complexity Factor) by given by the formula
  - ***TCF=0.65+0.01*DI***
- **STEP 6:** Function Points are by given by the formula
  - ***FP=UFP*TCF***

Relation between LOC and FP

- *LOC = Language Factor * FP*

- where
  - LOC (Lines of Code)
  - FP (Function Points)

Effort Computation

- The **Basic COCOMO model** computes effort as a function of program size. The Basic COCOMO equation is:
  - ***E = aKLOC^b***
- Effort for three modes of Basic COCOMO.

Mode	**a**	**b**
Organic	2.4	1.05
Semi-detached	3.0	1.12
Embedded	3.6	1.20

Effort Computation

- The **intermediate COCOMO model** computes effort as a function of program size and a set of cost drivers. The Intermediate COCOMO equation is:
  - ***E = aKLOC^b*EAF***
- Effort for three modes of intermediate COCOMO.

Mode	**a**	**b**
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

Effort Computation (..)

Total EAF = Product of the selected factors

Adjusted value of Effort: Adjusted Person Months:

$$\mathbf{APM = (Total\ EAF) * PM}$$

Software Development Time

- **Development Time Equation Parameter Table:**

Parameter	Organic	Semi-detached	Embedded
C	2.5	2.5	2.5
D	0.38	0.35	0.32

Development Time, **TDEV = C * (APM **D)**

Number of Personnel, **NP = APM / TDEV**

Distribution of Effort

- A development process typically consists of the following stages:
- **Requirements Analysis**
- **Design (High Level + Detailed)**
- **Implementation & Coding**
- **Testing (Unit + Integration)**

Error Estimation

- Calculate the estimated number of errors in your design, i.e.total errors found in requirements, specifications, code, user manuals, and bad fixes:
  - Adjust the **Function Point** calculated in step1

AFP = FP ** 1.25

  - Use the following table for calculating error estimates

Error Type	Error / AFP
Requirements	1
Design	1.25
Implementation	1.75
Documentation	0.6
Due to Bug Fixes	0.4

Estimation

- **LOC based estimation**
- **Source lines of code** (**SLOC**), also known as **lines of code** (**LOC**), is a software metric used to measure the size of a computer program by counting the number of lines in the text of the program's source code.
- SLOC is typically used to predict the amount of effort that will be required to develop a program, as well as to estimate programming productivity or maintainability once the software is produced.
- Lines used for commenting the code and header file are ignored.
- **Two major types of LOC:**

1. Physical LOC

- Physical LOC is the count of lines in the text of the program's source code including comment lines.
- Blank lines are also included unless the lines of code in a section consists of more than 25% blank lines.

2. Logical LOC

- Logical LOC attempts to measure the number of executable statements, but their specific definitions are tied to specific computer languages.
- Ex: Logical LOC measure for C-like programming languages is the number of statement-terminating semicolons(;)

Estimation

LOC-based Estimation

The problems of lines of code (LOC)

- Different languages lead to different lengths of code
- It is not clear how to count lines of code
- A report, screen, or GUI generator can generate thousands of lines of code in minutes
- Depending on the application, the complexity of code is different.

PROJECT SCHEDULING

What is PROJECT SCHEDULING?

- Basic Concepts
- Project Scheduling
  - Basic Principles
  - The Relationship Between People and Effort
  - Effort Distribution
- Earned Value Analysis

What is PROJECT SCHEDULING?

- In the late 1960s, a bright-eyed young engineer was chosen to "write" a computer program for an automated manufacturing application. The reason for his selection was simple. He was the only person in his technical group who had attended a computer programming seminar. He knew the ins and outs of assembly language and FORTRAN but nothing about software engineering and even less about project scheduling and tracking. His boss gave him the appropriate manuals and a verbal description of what had to be done. He was informed that the project must be completed in two months. He read the manuals, considered his approach, and began writing code. After two weeks, the boss called him into his office and asked how things were going. "Really great," said the young engineer with youthful enthusiasm. "This was much simpler than I thought. I'm probably close to 75 percent finished."

What is PROJECT SCHEDULING?

- The boss smiled and encouraged the young engineer to keep up the good work. They planned to meet again in a week's time. A week later the boss called the engineer into his office and asked, "Where arewe?" "Everything's going well," said the youngster, "but I've run into a few small snags. I'll get them ironed out and be back on track soon." "How does the deadline look?" the boss asked. "No problem," said the engineer. "I'm close to 90 percent complete." If you've been working in the software world for more than a few years, you can finish the story. It'll come as no surprise that the young engineer1 stayed 90 percent complete for the entire project duration and finished (with the help of others) only one month late. This story has been repeated tens of thousands of times by software developers during the past five decades. The big question is why?

What is PROJECT SCHEDULING?

- ✓ You've selected an app You've selected an appropriate process model Appropriate process model.
- ✓ You've identified the software engineering tasks that have to be performed.
- ✓ You estimated the amount of work and the number of people, you know the deadline, you've even considered the risks.
- ✓ Now it's time to connect the dots. That is, you have to create a network of software engineering tasks that will enable you to get the job done on time.
- ✓ Once the network is created, you have to assign responsibility for each task, make sure it gets done, and adapt the network as risks become reality.

What is PROJECT SCHEDULING?

- **Why it's Important?**
  - ✓ In order to build a complex system, many software engineering tasks occur in parallel.
  - ✓ The result of work performed during one task may have a profound effect on work to be conducted in another task.
  - ✓ These interdependencies are very difficult to understand without a schedule.
  - ✓ It's also virtually impossible to assess progress on a moderate or large software project without a detailed schedule

- **What are the steps?**
  - ✓ The software engineering tasks dictated by the software process model are refined for the functionality to be built.
  - ✓ Effort and duration are allocated to each task and a task network (also called an "activity network") is created in a manner that enables the software team to meet the delivery deadline established.

What is PROJECT SCHEDULING?

Basic Concept of Project Scheduling

- ✓ An unrealistic deadline established by someone outside the software development group and forced on managers and practitioner's within the group.
- ✓ Changing customer requirements that are not reflected in schedule changes.
- ✓ An honest underestimate of the amount of effort and/or the number of resources that will be required to do the job.
- ✓ Predictable and/or unpredictable risks that were not considered when the project commenced.
- ✓ Technical difficulties that could not have been foreseen in advance.

▪**Why should we do when the management demands that we make a dead line I impossible?**

- ✓ Perform a detailed estimate using historical data from past projects.
- ✓ Determine the estimated effort and duration for the project.
- ✓ Using an incremental process model, develop a software engineering strategy that will deliver critical functionality by the imposed deadline, but delay other functionality until later. Document the plan.
- ✓ Meet with the customer and (using the detailed estimate), explain why the imposed deadline is unrealistic.

Project Scheduling

- Project Scheduling
  - Basic Principles
  - The Relationship Between People and Effort
  - Effort Distribution

- Software project scheduling is an action that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks.
- During early stages of project planning, a macroscopic schedule is developed.
- As the project gets under way, each entry on the macroscopic schedule is refined into a detailed schedule.

Project Scheduling

- **Basic Principles of Project Scheduling.**

1. **Compartmentalization:** The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are refined.
2. **Interdependency:** The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence, while others can occur in parallel. Other activities can occur independently.
3. **Time allocation:** Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date. whether work will be conducted on a full-time or part-time basis.
4. **Effort validation:** Every project has a defined number of people on the software team. The project manager must ensure that no more than the allocated number of people have been scheduled at any given time.
5. **Defined responsibilities**. Every task that is scheduled should be assigned to a specific team member.

Project Scheduling

6. **Defined outcomes:** Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a component) or a part of a work product. Work products are often combined in deliverables.
7. **Defined milestones:** Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality and has been approved.

Each of these principles is applied as the project schedule evolves.

The Relationship Between People and Effort

- In a small software development project a single person can analyze requirements, perform design, generate code, and conduct tests. As the size of a project increases, more people must become involved.
- There is a common myth that is still believed by many managers who are responsible for software development projects: “If we fall behind schedule, we can always add more programmers and catch up later in the project.”
- Unfortunately, adding people late in a project often has a disruptive effect on the project, causing schedules to slip even further. The people who are added must learn the system, and the people who teach them are the same people who were doing the work.
- While teaching, no work is done, and the project falls further behind. In addition to the time it takes to learn the system, more people.
- Although communication is absolutely essential to successful software development, every new communication path requires additional effort and therefore additional time.

Effort Distribution

- A recommended distribution of effort across the software process is often referred to as the 40–20–40 rule.
- Forty percent of all effort is allocated to frontend analysis and design. A similar percentage is applied to back-end testing. You can correctly infer that coding (20 percent of effort) is deemphasized.
- Work expended on project planning rarely accounts for more than 2 to 3 percent of effort, unless the plan commits an organization to large expenditures with high risk.
- Customer communication and requirements analysis may comprise 10 to 25 percent of project effort.
- Effort expended on analysis or prototyping should increase in direct proportion with project size and complexity.
- A range of 20 to 25 percent of effort is normally applied to software design. Time expended for design review and subsequent iteration must also be considered.

- Because of the effort applied to software design, code should follow with relatively little difficulty.
- A range of 15 to 20 percent of overall effort can be achieved. Testing and subsequent debugging can account for 30 to 40 percent of software development effort.
- The criticality of the software often dictates the amount of testing that is required. If software is human rated (i.e., software failure can result in loss of life), even higher percentages are typical.

Scheduling

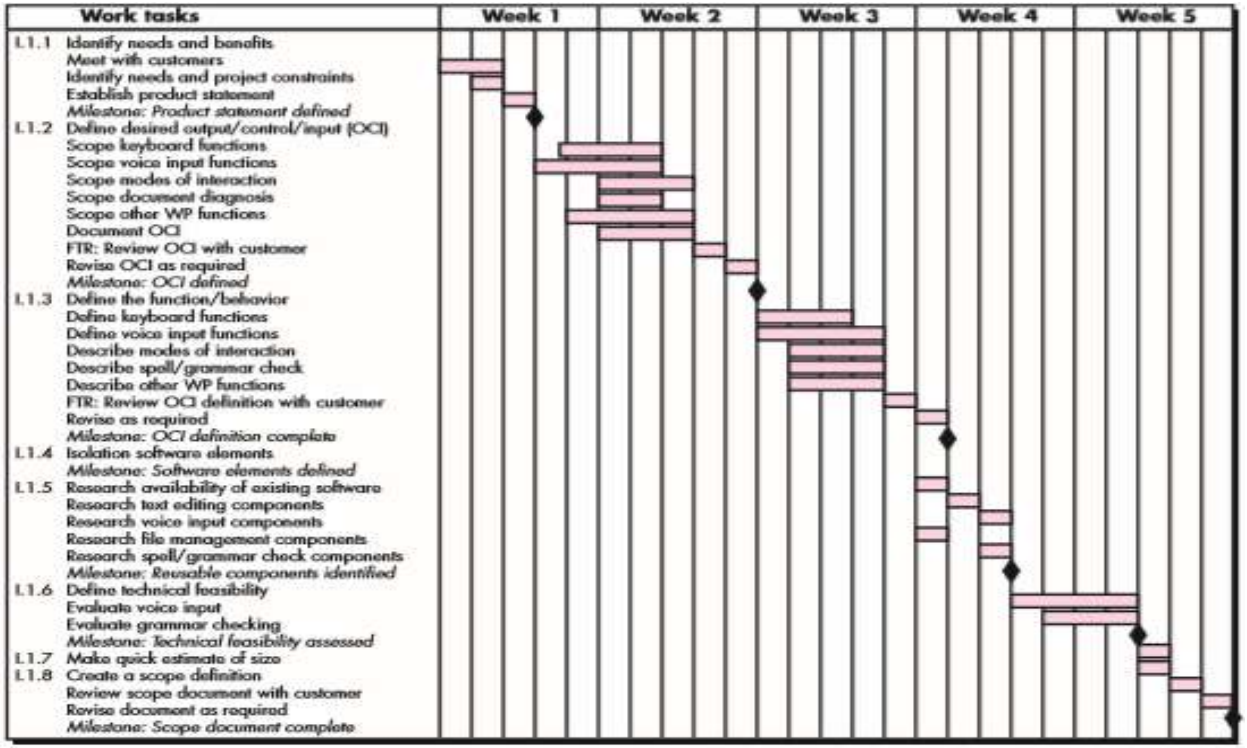
- Scheduling of a software project does not differ greatly from scheduling of any multitask engineering effort. Therefore, generalized project scheduling tools and techniques can be applied with little modification for software projects.
- Program evaluation and review technique (PERT) and the critical path method (CPM) are two project scheduling methods that can be applied to software development.

1. Time-Line Charts:

- When creating a software project schedule, begin with a set of tasks.
- If automated tools are used, the work breakdown is input as a task network or task outline. Effort, duration, and start date are then input for each task. In addition, tasks may be assigned to specific individuals.
- As a consequence of this input, a time-line chart, also called a Gantt chart, is generated.
- A time-line chart can be developed for the entire project. Alternatively, separate charts can be developed for each project function or for each individual working on the project.

Scheduling

- All project tasks (for concept scoping) are listed in the left hand column. The horizontal bars indicate the duration of each task. When multiple bars occur at the same time on the calendar, task concurrency is implied. The diamonds indicate milestones.
- Once the information necessary for the generation of a time-line chart has been input, the majority of software project scheduling tools produce project tables. —a tabular listing of all project tasks, their planned and actual start and end dates, and a variety of related information. Used in conjunction with the time-line chart, project tables enable you to track progress.



Scheduling

2. Tracking the Schedule

- If it has been properly developed, the project schedule becomes a road map that defines the tasks and milestones to be tracked and controlled as the project proceeds.
- Tracking can be accomplished in a number of different ways:
- Conducting periodic project status meetings in which each team member reports progress and problems.
- Evaluating the results of all reviews conducted throughout the software engineering process.
- Determining whether formal project milestones have been accomplished by the scheduled date.
- Comparing the actual start date to the planned start date for each project task listed in the resource table.
- Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon.
- Using earned value analysis to assess progress quantitatively.

In reality, all of these tracking techniques are used by experienced project managers.

Scheduling

3. Tracking Progress for an OO Project

Technical milestone: OO analysis complete

- o All hierarchy classes defined and reviewed
- o Class attributes and operations are defined and reviewed
- o Class relationships defined and reviewed
- o Behavioral model defined and reviewed
- o Reusable classed identified

Technical milestone: OO design complete

- o Subsystems defined and reviewed
- o Classes allocated to subsystems and reviewed
- o Task allocation has been established and reviewed
- o Responsibilities and collaborations have been identified
- o Attributes and operations have been designed and reviewed
- o Communication model has been created and reviewed

Scheduling

- **Technical milestone: OO programming complete**
  - o Each new design model class has been implemented
  - o Classes extracted from the reuse library have been implemented
  - o Prototype or increment has been built
- **Technical milestone: OO testing**
  - o The correctness and completeness of the OOA and OOD models has been reviewed
  - o Class-responsibility-collaboration network has been developed and reviewed
  - o Test cases are designed and class-level tests have been conducted for each class
  - o Test cases are designed, cluster testing is completed, and classes have been integrated
  - o System level tests are complete

Scheduling

4. Scheduling for WebApp Projects

- WebApp project scheduling distributes estimated effort across the planned time line (duration) for building each WebApp increment.
- This is accomplished by allocating the effort to specific tasks.
- The overall WebApp schedule evolves over time.
- During the first iteration, a macroscopic schedule is developed.
- This type of schedule identifies all WebApp increments and projects the dates on which each will be deployed.
- As the development of an increment gets under way, the entry for the increment on the macroscopic schedule is refined into a detailed schedule.
- Here, specific development tasks (required to accomplish an activity) are identified and scheduled.

EARNED VALUE ANALYSIS

- It is reasonable to ask whether there is a quantitative technique for assessing progress as the software team progresses through the work tasks allocated to the project schedule.
- A Technique for performing quantitative analysis of progress does exist. It is called earned value analysis (EVA).
- To determine the earned value, the following steps are performed:
  1. The budgeted cost of work scheduled (BCWS) is determined for each work task represented in the schedule. During estimation, the work (in person-hours or person-days) of each software engineering task is planned. Hence, BCWSi is the effort planned for work task i. To determine progress at a given point along the project schedule, the value of BCWS is the sum of the BCWSi values for all work tasks that should have been completed by that point in time on the project schedule.
  2. The BCWS values for all work tasks are summed to derive the budget at completion (BAC). Hence, BAC (BCWSk) for all tasks k
  3. Next, the value for budgeted cost of work performed (BCWP) is computed. The value for BCWP is the sum of the BCWS values for all work tasks that have actually been completed by a point in time on the project schedule.

EARNED VALUE ANALYSIS

- Given values for BCWS, BAC, and BCWP, important progress indicators can be computed:

  Schedule performance index, SPI = BCWP / BCWS

  Schedule variance, SV = BCWP – BCWS

- SPI is an indication of the efficiency with which the project is utilizing scheduled resources. An SPI value close to 1.0 indicates efficient execution of the project schedule. SV is simply an absolute indication of variance from the planned schedule.
- Percent scheduled for completion = BCWS / BAC

  provides an indication of the percentage of work that should have been completed by time t.
- Percent complete = BCWP / BAC

  provides a quantitative indication of the percent of completeness of the project at a given point in time t. It is also possible to compute the actual cost of work performed (ACWP). The value for ACWP is the sum of the effort actually expended on work tasks that have been completed by a point in time on the project schedule. It is then possible to compute

EARNED VALUE ANALYSIS

Cost performance index, CPI = BCWP /ACWP

Cost variance, CV = BCWP - ACWP

A CPI value close to 1.0 provides a strong indication that the project is within its defined budget. CV is an absolute indication of cost savings (against planned costs) or shortfall at a particular stage of a project.

Risk Management

- Hazard and Risk
- A Hazard is

  Any real or potential condition that can cause injury, illness, or death to personnel; damage to or loss of a system, equipment or property; or damage to the environment. Simpler .... A threat of harm. A hazard can lead to one or several consequences.

- Risk is

The expectation of a loss or damage (consequence)

The combined severity and probability of a loss

The long term rate of loss

A potential problem (leading to a loss) that may - or may not occur in the future.

Risk Management

- Risk Management is A set of practices and support tools to identify, analyze, and treat risks explicitly.
- Treating a risk means understanding it better, avoiding or reducing it (risk mitigation), or preparing for the risk to materialize.
- Risk management tries to reduce the probability of a risk to occur and the impact (loss) caused by risks.

Risk Management

- Reactive versus Proactive Risk Strategies
- Software risks

➢Reactive versus Proactive Risk Strategies

•The majority of software teams rely solely on reactive risk strategies. At best, a reactive strategy monitors the project for likely risks. Resources are set aside to deal with them, should they become actual problems.

•The software team does nothing about risks until something goes wrong. Then, the team flies into action in an attempt to correct the problem rapidly. This is often called a fire-fighting mode.

Risk Management

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

Software Risks

Risk always involves two characteristics:

- Risk always involves two characteristics: uncertainty—the risk may or may not happen; that is, there are no 100 percent probable risks—and loss—if the risk becomes a reality, unwanted consequences or losses will occur.
- When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk.
- Different categories of risks are follows:

1. *Project risks*
   - ❖ Threaten the project plan. That is, if project risks become real, it is likely that the project schedule will slip and that costs will increase.
   - ❖ Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, stakeholder, and requirements problems and their impact on a software project.

Software Risks

2. Technical risks

- Threaten the quality and timeliness of the software to be produced.
- If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems.
- In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors. Technical risks occur because the problem is harder to solve than you thought it would be.

3. Business risks

- Business risks threaten the viability of the software to be built and often jeopardize the project or the product.
- Candidates for the top five business risks are

  (1) building an excellent product or system that no one really wants (market risk)

  (2) building a product that no longer fits into the overall business strategy for the company (strategic risk)

  (3) building a product that the sales force doesn't understand how to sell (sales risk)

  (4) losing the support of senior management due to a change in focus or a change in people (management risk)

  (5) losing budgetary or personnel commitment (budget risks).

Software Risks

Another general categorization of risks has been proposed by Charette.

1. *Known risks* are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment).
2. *Predictable* risks are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced).
3. *Unpredictable risks* are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

UNIT- II

Requirement Analysis and Specification: Software requirements: Functional and nonfunctional, user requirements, system requirements, software requirements document; Requirement engineering process: Feasibility studies, requirements elicitation and analysis, requirements validation, requirements management; Classical analysis: Structured system analysis, petri nets, data dictionary.

SOFTWARE REQUIREMENTS

- IEEE defines Requirement as :

  1. A condition or capability needed by a user to solve a problem or achieve an objective

  2. A condition or capability that must be met or possessed by a system or a system component to satisfy contract, standard, specification or formally imposed document

  3. A documented representation of a condition or capability as in 1 or 2

SOFTWARE REQUIREMENTS

- Requirements may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.
- Requirements may serve a dual function
  - May be the basis for a bid for a contract - therefore must be open to interpretation
  - May be the basis for the contract itself - therefore must be defined in detail

Software Requirements

"If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not pre-defined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organisation's needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the *requirements document* for the system."

Types of Requirements

- Business Requirements
- User requirements
- System requirements
- Functional requirements
- Non-functional requirements
- Domain requirements
- Interface Specifications

Business Requirements

- A high-level business objective of the organization that builds a product or of a customer who procures it
- Generally stated by the business owner or sponsor of the project
  - Example: A system is needed to track the attendance of employees
  - A system is needed to account the inventory of the organization

Business Requirements

Contents of Business Requirements:

- Purpose, Inscope, Out of Scope, Targeted Audiences
- Use Case diagrams
- Data Requirements
- Non Functional Requirements
- Interface Requirements
- Limitations
- Risks
- Assumptions
- Reporting Requirements
- Checklists

User Requirements

- A user requirement refers to a function that the user requires a system to perform.
- Made through statements in natural language and diagrams of the services the system provides and its operational constraints. Written for customers.
- User requirements are set by client and confirmed before system development.
  - For example, in a system for a bank the user may require a function to calculate interest over a set time period.

System Requirements

- A system requirement is a more technical requirement, often relating to hardware or software required for a system to function.
  - System requirements may be something like - "The system must run on a server with IIS
  - System requirements may also include validation requirements such as "File upload is limited to .xls format
- System requirements are more commonly used by developers throughout the development life cycle. The client will usually have less interest in these lower level requirements.
- A structured document setting out detailed descriptions of the system's functions, services and operational constraints.

Functional Requirements

- Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
- A functional requirement defines a function of a software system or its component.
- A function is described as a set of inputs, the behavior, and outputs.
- Functional requirements may be calculations, technical details, data manipulation and processing and other specific functionality that define *what* a system is supposed to accomplish.
- Behavioral requirements describing all the cases where the system uses the functional requirements are captured in *use cases*
- Functional requirements drive the application architecture of a system
- The plan for implementing *functional* requirements is detailed in the system design

Non-Functional Requirements

- A non-functional requirement is a requirement that specifies criteria that can be used to judge the operation of a system, rather than specific behaviors
- The plan for implementing non-functional requirements is detailed in the system architecture.
- Non-functional requirements are often called qualities of a system. Other terms for non-functional requirements are "constraints", "quality attributes", "quality goals", "quality of service requirements" and "non-behavioral requirements
- These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- Process requirements may also be specified mandating a particular CASE system, programming language or development method.
- Non-functional requirements may be more critical than functional requirements. If these are not met, the system may become useless.

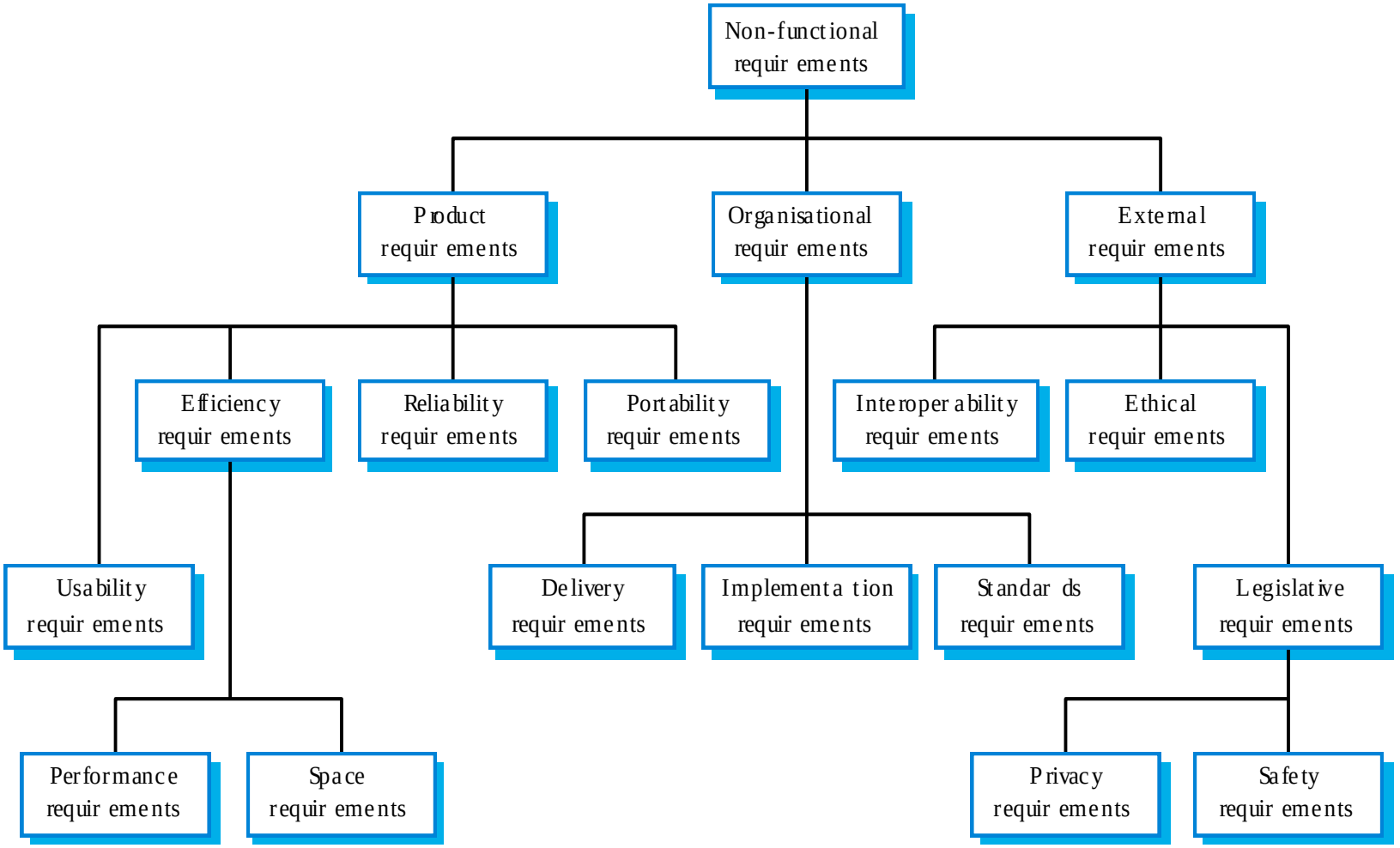
Non-Functional Requirements

- These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- Process requirements may also be specified mandating a particular CASE system, programming language or development method.
- Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless.

Non-functional Requirements classifications

- Product requirements
  - Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.
- Organisational requirements
  - Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.
- External requirements
  - Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

Non-functional requirement types



Non-functional requirements examples

- Product requirement

  The user interface for the system shall be implemented as simple HTML without frames or Java applets.

- Organisational requirement

  The system development process and deliverable documents shall conform to the process and deliverables defined in XYZCo-SP-STAN-95.

- External requirement

  The system shall not disclose any personal information about customers apart from their name and reference number to the operators of the system.

Non-Functional Requirements measures

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

Domain Requirements

- Domain requirements reflect the environment in which the system operates so, when we talk about an application domain it means environments such as train operation, medical records, e-commerce etc.
- Domain requirements may be expressed using specialized domain terminology or reference to domain concepts. Because these requirements are specialized, software engineers often find it difficult to understand how they are related to other system requirements
- Domain requirements are important because they often reflect fundamentals of the application domain. If these requirements are not satisfied, it may be impossible to make the system work satisfactorily.
- For example, the requirements for an insulin pump system that delivers insulin on demand include the following domain requirement:
  - The system safety shall be assured according to standard IEC 60601-1:Medical Electrical Equipment – Part 1:General Requirements for Basic Safety and Essential Performance

Domain requirements problems

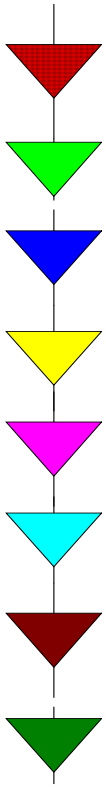
- Understandability
  - Requirements are expressed in the language of the application domain
  - This is often not understood by software engineers developing the system.
- Implicitness
  - Domain specialists understand the area so well that they do not think of making the domain requirements explicit.

Characteristics of Good User Requirements

A user requirement is good

if it is:

1. Verifiable
2. Clear and concise
3. Complete
4. Consistent
5. Traceable
6. Viable
7. Necessary
8. Implementation free



Think of these characteristics as a series of filters. A good requirement will pass through all eight filters.

What Makes a UR Verifiable?

A verifiable requirement …

- is stated in such a way that it **can be tested** by:
  - inspection,
  - analysis, or
  - demonstration.
- makes it **possible to evaluate** whether the system met the requirement, and
- is verifiable by **means that will not contaminate** the product **or compromise** the data integrity.

Is this UR Verifiable?

- **Bad example:**
  - **UR1: The system must be user friendly.**
  - **How should we measure user friendliness?**

- **Good example:**
  - **UR1: The user interface shall be menu driven. It shall provide dialog boxes, help screens, radio buttons, dropdown list boxes, and spin buttons for user inputs.**

What Makes a UR Clear & Concise?

A clear & concise requirement …

- must consist of a **single requirement**,
- should be no more than **30-50 words** in length,
- must be **easily read and understood** by non technical people,
- must be **unambiguous** and not susceptible to multiple interpretations,
- must **not contain** definitions, descriptions of its use, or reasons for its need, and
- must **avoid** subjective or open-ended terms.

Is this UR Clear & Concise?

- **Bad example:**
  - **UR2: All screens must appear on the monitor quickly.**
  - **How long is quickly?**

- **Good example:**
  - **UR2: When the user accesses any screen, it must appear on the monitor within 2 seconds.**

What Makes a UR Complete?

A complete requirement …

- contains **all the information** that is needed to define the system function,
- leaves **no one guessing** (For how long?, 50 % of what?), and
- includes **measurement units** (inches or centimeters?).

Is this UR Complete?

- **Bad example:**
  - **UR3: On loss of power, the battery backup must support normal operations.**
  - **For how long?**

- **Good example:**
  - **UR3: On loss of power, the battery backup must support normal operations for 20 minutes.**

What Makes a UR Consistent?

A consistent requirement …

- **does not conflict** with other requirements in the requirement specification,
- uses the **same terminology** throughout the requirement specification, and
- **does not duplicate** other URs or pieces of other URs or create redundancy in any way.

Is this UR Consistent?

- **Bad example:**
  - **UR4: The electronic batch records shall be Part 11 compliant.**
  - **UR47: An on-going training program for 21 CFR Part 11 needs to be established at the sites.**
  - **Do these refer to the same regulation or different ones?**

- **Good example:**
  - **UR4: The electronic batch records shall be 21 CFR Part 11 compliant.**
  - **UR47: An on-going training program for 21 CFR Part 11 needs to be established at the site.**

What Makes a UR Traceable?

A traceable requirement …

- has a **unique identity** or number,
- **cannot be separated** or broken into smaller requirements,
- can **easily be traced** through to specification, design, and testing.
- **Change Control** on UR level.

Is this UR Traceable?

- **Bad example:**
  - **UR: The system must generate a batch end report and a discrepancy report when a batch is aborted.**
  - **How is this uniquely identified? If the requirement is changed later so that it does not require a discrepancy report, how will you trace it back so you can delete it?**

- **Good example:**
  - **UR6v1: The system must generate a batch end report when a batch is aborted.**
  - **UR7v2: The system must generate a discrepancy report when a batch is completed or aborted.**

What Makes a UR Viable?

A viable requirement …

- can be **met using existing technology**,
- can be **achieved within the budget**,
- can be **met within the schedule**,
- is something the organization has the **necessary skills to utilize**,
- will be **used by the end users**, and
- must be **helpful to build the system**.

Is this UR Viable or Feasible?

- **Bad example:**
  - **The replacement control system shall be installed with no disruption to production.**
  - **This is an unrealistic expectation.**

- **Good example:**
  - **The replacement control system shall be installed causing no more than 2 days of production disruption.**

What Makes a UR Necessary?

A necessary requirement …

- is one that **must be present to meet system objectives**, and
- is **absolutely critical** for the operation of the system,
- leads to a **deficiency in the system if it is removed**.

Is this UR Necessary?

- **Bad example:**
  - **All desktop PCs for the project must be configured with 512MB of memory, DVD ROM/CD-RW multifunction drive and a 21-inch flat screen monitor.**
  - **This may not be needed for all PCs for the project.**

- **Good example:**
  - **The desktop PCs for the developers on the project must be configured with 512MB of memory, DVD ROM/CD-RW multifunction drive and a 21-inch flat screen monitor.**

What Makes a UR Free of Implementation Details?

A requirement that is free of implementation details …

- defines **what functions are provided** by the system,
- **does NOT specify how** a function can or should be implemented, and
- allows the **system developer to decide what technology is best** suited to achieve the function.

Is this UR Free of Implementation Details?

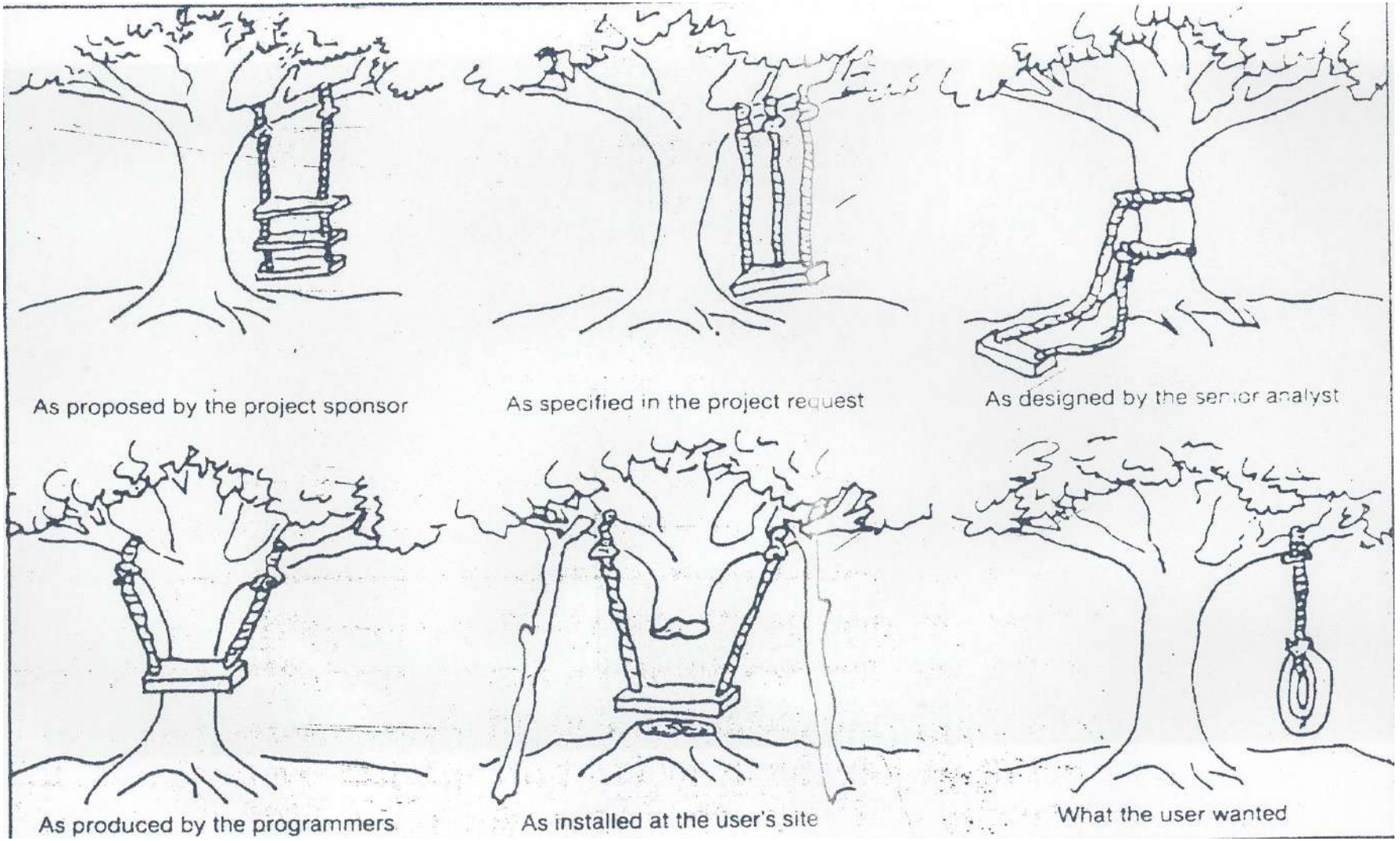
- **Bad example:**
  - **After 3 unsuccessful attempts to log on, a Java Script routine must run and lock the user out of the system.**
  - **Specifying a JavaScript routine concerns how the requirement will be implemented.**

- **Good example:**
  - **After 3 unsuccessful attempts to log on, the user must be locked out of the system.**

Requirements imprecision

- Problems arise when requirements are not precisely stated.
- Ambiguous requirements may be interpreted in different ways by developers and users.
- Consider the term ‘appropriate viewers’
  - User intention - special purpose viewer for each different document type;
  - Developer interpretation - Provide a text viewer that shows the contents of the document.

Requirements Mismatch



Interface specification

- Most systems must operate with other systems and the operating interfaces must be specified as part of the requirements.
- Three types of interface may have to be defined
  - Procedural interfaces;
  - Data structures that are exchanged;
  - Data representations.
- Formal notations are an effective technique for interface specification.

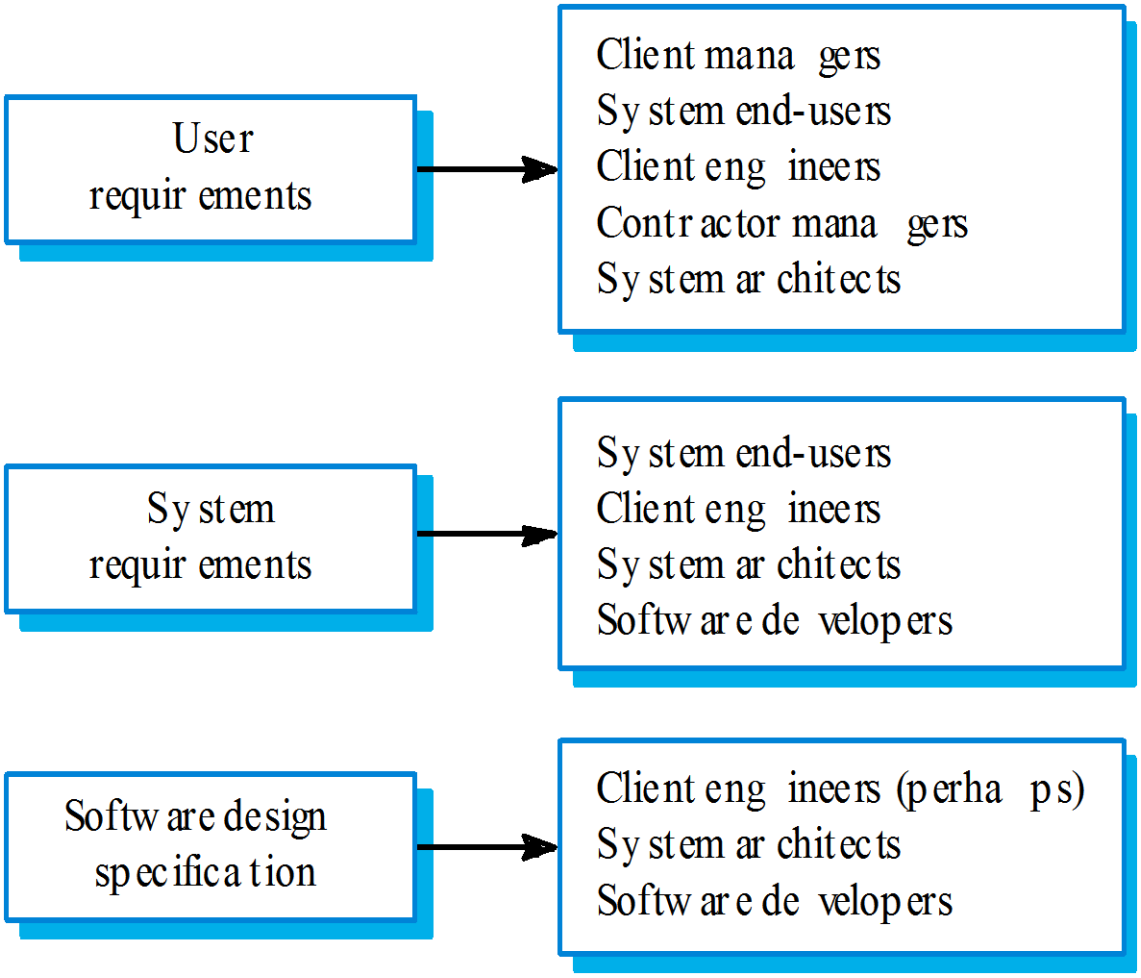
Example interface description

```
interface PrintServer {

// defines an abstract printer server
// requires:          interface Printer, interface PrintDoc
// provides: initialize, print, displayPrintQueue, cancelPrintJob, switchPrinter

        void initialize ( Printer p ) ;
        void print ( Printer p, PrintDoc d ) ;
        void displayPrintQueue ( Printer p ) ;
        void cancelPrintJob (Printer p, PrintDoc d) ;
        void switchPrinter (Printer p1, Printer p2, PrintDoc d) ;
} //PrintServer
```

Requirements readers



Requirements and design

- In principle, requirements should state what the system should do and the design should describe how it does this.
- In practice, requirements and design are inseparable
  - A system architecture may be designed to structure the requirements;
  - The system may inter-operate with other systems that generate design requirements;
  - The use of a specific design may be a domain requirement.

Definitions and specifications

User requirement definition

1. The software must provide a means of representing and accessing external files created by other tools.

System requirements specification

1.1 The user should be provided with facilities to define the type of external files.
1.2 Each external file type may have an associated tool which may be applied to the file.
1.3 Each external file type may be represented as a specific icon on the user's display.
1.4 Facilities should be provided for the icon representing an external file type to be defined by the user.
1.5 When a user selects an icon representing an external file, the effect of that selection is to apply the tool associated with the type of the external file to the file represented by the selected icon.

Specifying User requirements

- Should describe functional and non-functional requirements in such a way that they are understandable by system users who don’t have detailed technical knowledge.
- User requirements are defined using natural language, tables and diagrams as these can be understood by all users.

Problems with natural language

- Lack of clarity
  - Precision is difficult without making the document difficult to read.
- Requirements confusion
  - Functional and non-functional requirements tend to be mixed-up.
- Requirements amalgamation
  - Several different requirements may be expressed together.

Problems with NL specification

- Ambiguity
  - The readers and writers of the requirement must interpret the same words in the same way. NL is naturally ambiguous so this is very difficult.
- Over-flexibility
  - The same thing may be said in a number of different ways in the specification.
- Lack of modularisation
  - NL structures are inadequate to structure system requirements.

Alternatives to NL specification

Notation	Description
Structured natural language	This approach depends on defining standard forms or templates to express the requirements specification.
Design description languages	This approach uses a language like a programming language but with more abstract features to specify the requirements by defining an operational model of the system. This approach is not now widely used although it can be useful for interface specifications.
Graphical notations	A graphical language, supplemented by text annotations is used to define the functional requirements for the system. An early example of such a graphical language was SADT. Now, use-case descriptions and sequence diagrams are commonly used .
Mathematical specifications	These are notations based on mathematical concepts such as finite-state machines or sets. These unambiguous specifications reduce the arguments between customer and contractor about system functionality. However, most customers don't understand formal specifications and are reluctant to accept it as a system contract.

Structured language specifications

- The freedom of the requirements writer is limited by a predefined template for requirements.
- All requirements are written in a standard way.
- The terminology used in the description may be limited.
- The advantage is that the most of the expressiveness of natural language is maintained but a degree of uniformity is imposed on the specification.

Form-based specifications

- Definition of the function or entity.
- Description of inputs and where they come from.
- Description of outputs and where they go to.
- Indication of other entities required.
- Action to be performed
- Pre and post conditions (if appropriate).
- The side effects (if any) of the function.

Form-based node specification

Insulin Pump/Control Software/SRS/3.3.2

Function Compute insulin dose: Safe sugar level

Description Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.

Inputs Current sugar reading (r2), the previous two readings (r0 and r1)

Source Current sugar reading from sensor. Other readings from memory.

Outputs CompDose Ğthe dose in insulin to be delivered

Destination Main control loop

Action: CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.

Requires Two previous readings so that the rate of change of sugar level can be computed.

Pre-condition The insulin reservoir contains at least the maximum allowed single dose of insulin..

Post-condition r0 is replaced by r1 then r1 is replaced by r2

Side-effects None

Tabular specification

- Used to supplement natural language.
- Particularly useful to define a number of possible alternative courses of action.

Condition	Action
Sugar level falling (r2 < r1)	CompDose = 0
Sugar level stable (r2 = r1)	CompDose = 0
Sugar level increasing and rate of increase decreasing ((r2-r1)<(r1-r0))	CompDose = 0
Sugar level increasing and rate of increase stable or increasing. ((r2-r1) □ (r1-r0))	CompDose = round ((r2-r1)/4) If rounded result = 0 then CompDose = MinimumDose

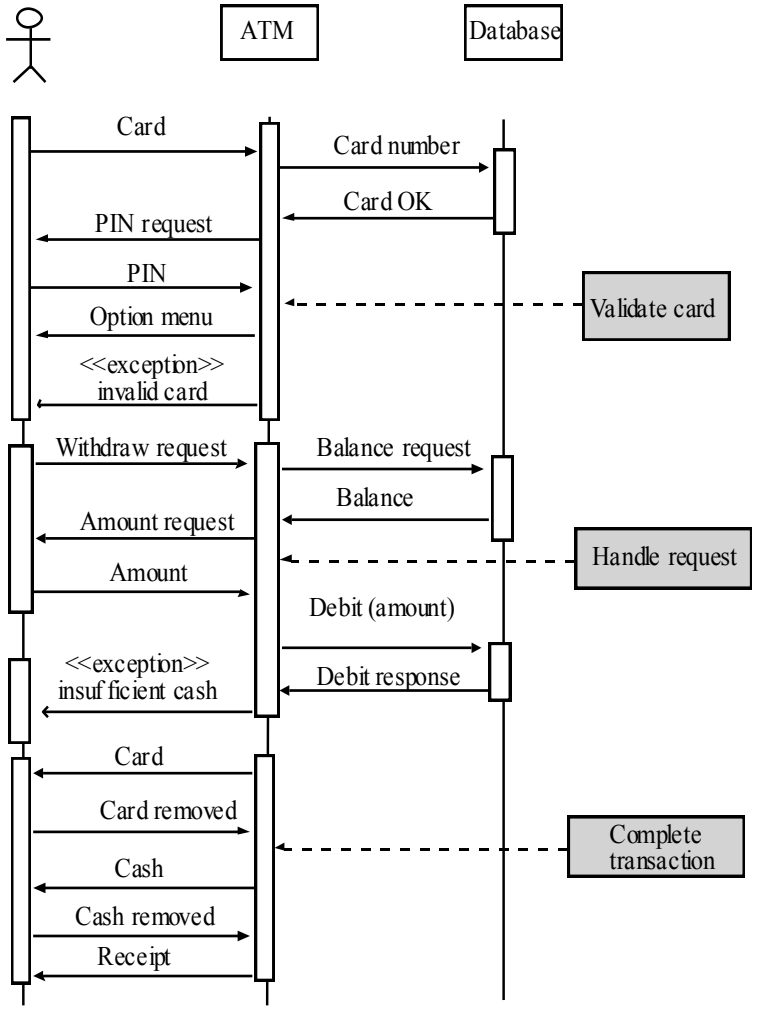
Graphical models

- Graphical models are most useful when you need to show how state changes or where there is a need to describe a sequence of actions.

Sequence diagrams

- These show the sequence of events that take place during some user interaction with a system.
- You read them from top to bottom to see the order of the actions that take place.
- Cash withdrawal from an ATM
  - Validate card;
  - Handle request;
  - Complete transaction.

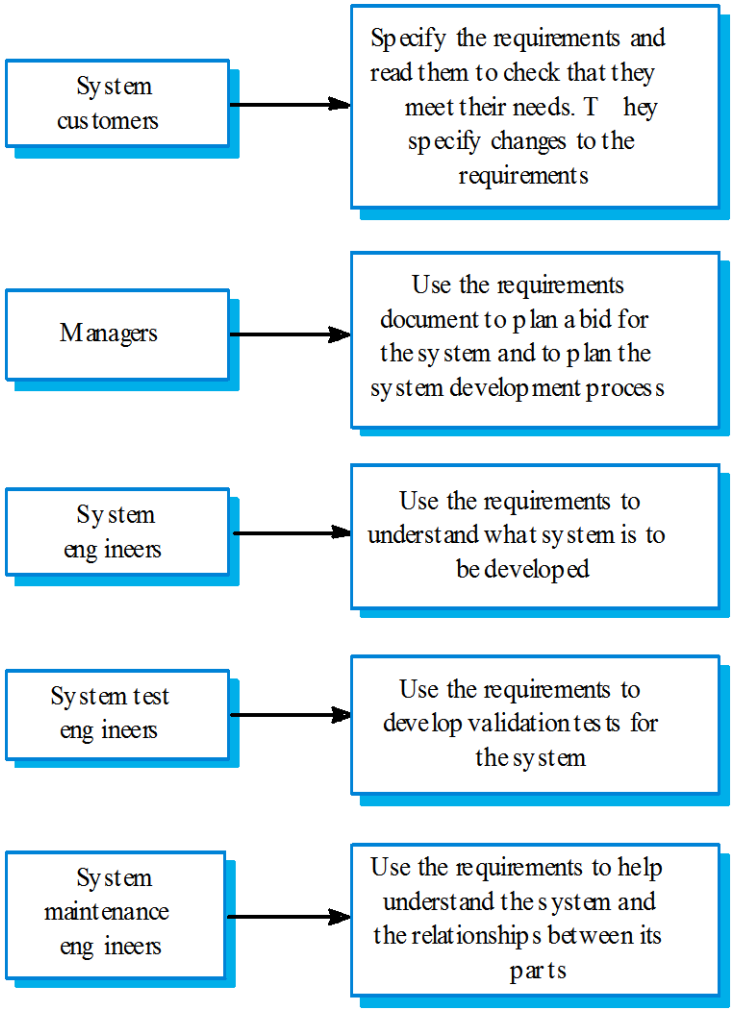
Sequence diagram of ATM withdrawal



The Software Requirements Specifications (SRS) Document

- The requirements document is the official statement of what is required of the system developers.
- Should include both a definition of user requirements and a specification of the system requirements.
- It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it.

Users of a requirements document



Purpose of SRS

- Communication between the Customer, Analyst, System Developers, Maintainers
- Firm foundation for the design phase
- Support system testing activities
- Support project management and control
- Controlling the evolution of the system

IEEE Requirements Standard

- Defines a generic structure for a requirements document that must be instantiated for each specific system.
  - Introduction.
  - General description.
  - Specific requirements.
  - Appendices.
  - Index.

IEEE Requirements Standard

1. Introduction
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definitions, Acronyms and Abbreviations
   - 1.4 References
   - 1.5 Overview

IEEE requirements standard

2. General description
    - 2.1 Product perspective
    - 2.2 Product function summary
    - 2.3 User characteristics
    - 2.4 General constraints
    - 2.5 Assumptions and dependencies

IEEE Requirements Standard

3. Specific Requirements
   - Functional requirements
   - External interface requirements
   - Performance requirements
   - Design constraints
   - Attributes
     eg.security, availability, maintainability,
     transferability/conversion
   - Other requirements
- Appendices
- Index

Suggested SRS Document Structure

- Preface
  - Should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version
- Introduction
  - This should describe the for the system. It should briefly describe its functions and explain how it will work with other. It should describe how it will with other systems. It should describe how the system fits into the overall business or strategic objectives of the organization commissioning the software
- Glossary
  - This should define the technical terms used in the document. Should not make assumptions about the experience or expertise of the reader

Suggested SRS Document Structure

- User Requirements Definition
  - The services provided for the user and the non-functional system requirements should be described in this section. This description may use natural language, diagrams or other notations that are understandable by customers. Product and process standards which must be followed should be specified
- System Architecture
  - This chapter should present a high-level overview of the anticipated system architecture showing the distribution of functions across modules. Architectural components that re reused should be highlighted
- System Requirements Specification
  - This should describe the functional and non-functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements e.g. interfaces to other systems may be defined

Suggested SRS Document Structure

- System Models
  - This should set out one or more system models showing the relationships between the system components and the system and its environment. These might be object models and data-flow models
- System Evolution
  - This should describe the fundamental assumptions on which the system is based and anticipated changes due to hardware evolution, changing user needs etc
- Appendices
  - These should provide detailed, specific information which is related to the application which is being developed. E.g. Appendices that may include hardware and database descriptions.
- Index
  - Several indexes to the document may be included

Requirements Engineering Processes

- A customer says “ I know you think you understand what I said, but what you don’t understand is what I said is not what I mean”
- Requirement engineering helps software engineers to better understand the problem to solve.
- It is carried out by software engineers (analysts) and other project stakeholders
- It is important to understand what the customer wants before one begins to design and build a computer based system
- Work products include user scenarios, functions and feature lists, analysis models

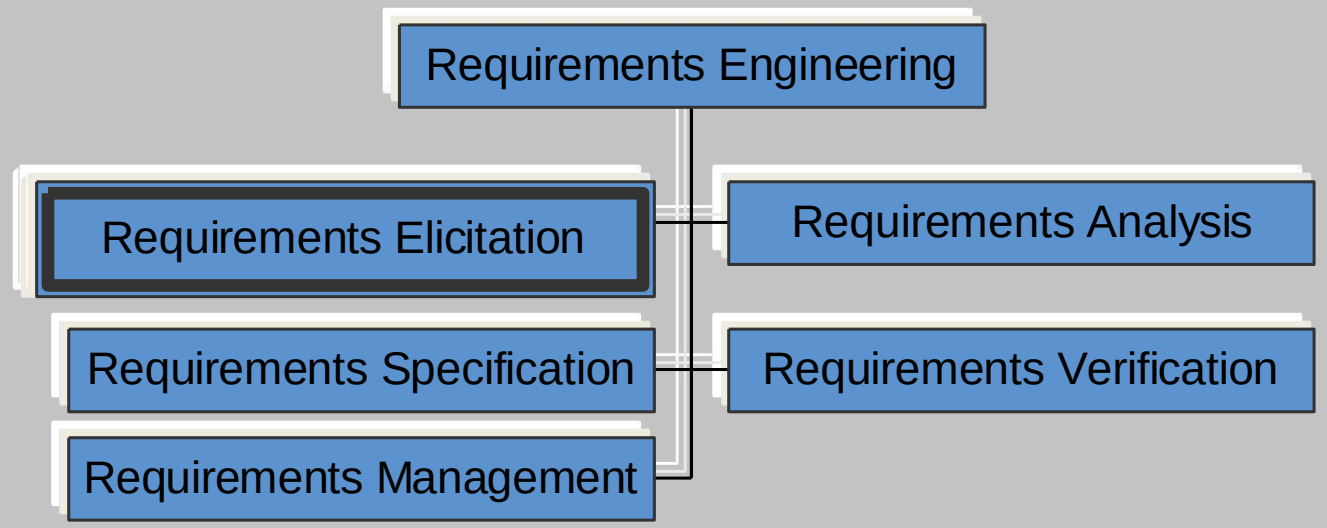
Requirements Engineering Processes

- Requirements engineering (RE) is a systems and software engineering process which covers all of the activities involved in discovering, documenting and maintaining a set of requirements for a computer-based system
- The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements.
- Activities within the RE process may include:
  - Requirements elicitation - discovering requirements from system stakeholders
  - Requirements Analysis and negotiation - checking requirements and resolving stakeholder conflicts

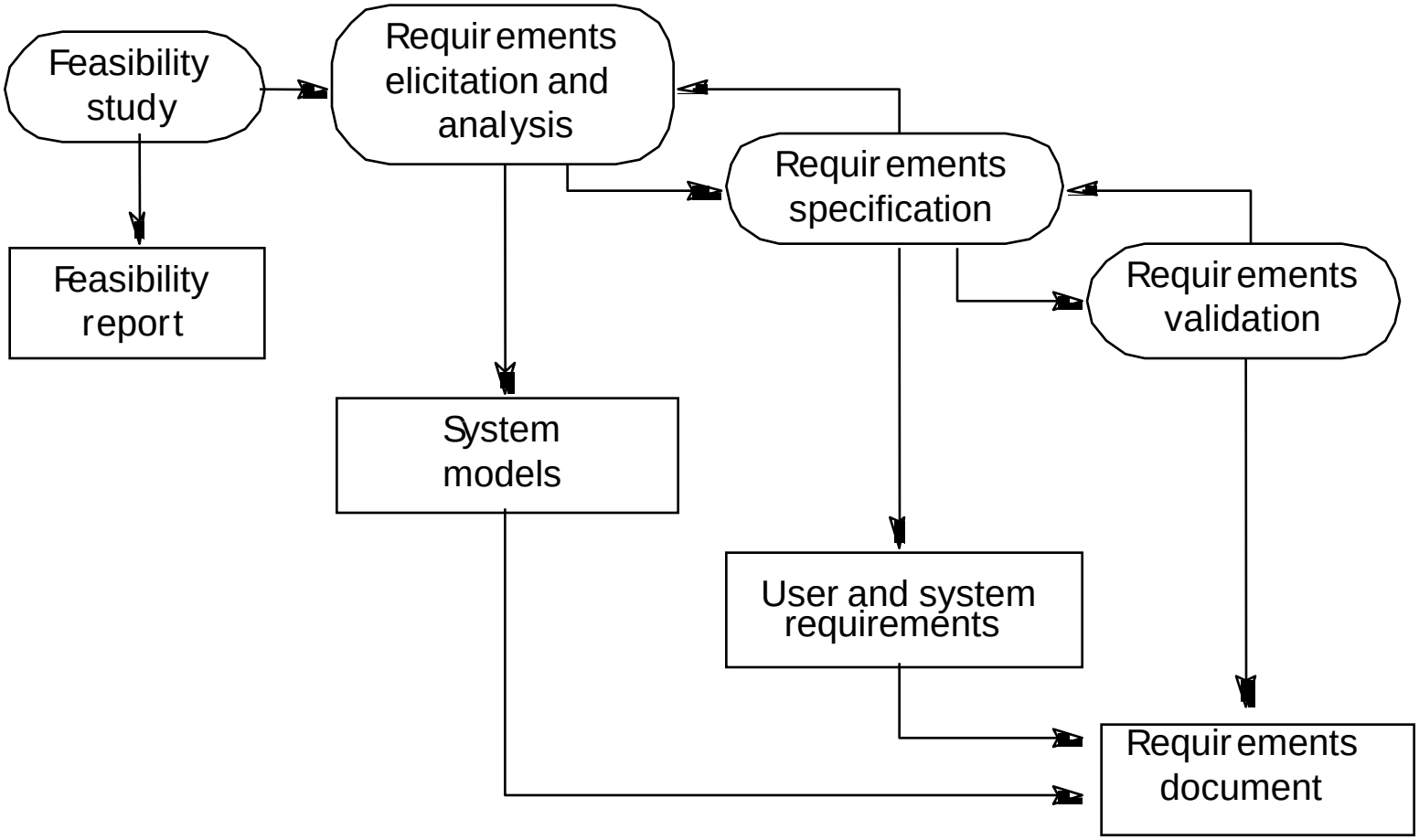
Requirements Engineering Processes

- Activities within the RE process may include:
  - Requirements specification (Software Requirements Specification)- documenting the requirements in a requirements document
  - System modeling - deriving models of the system, often using a notation such as the Unified Modeling Language
  - Requirements validation - checking that the documented requirements and models are consistent and meet stakeholder needs
  - Requirements management - managing changes to the requirements as the system is developed and put into use

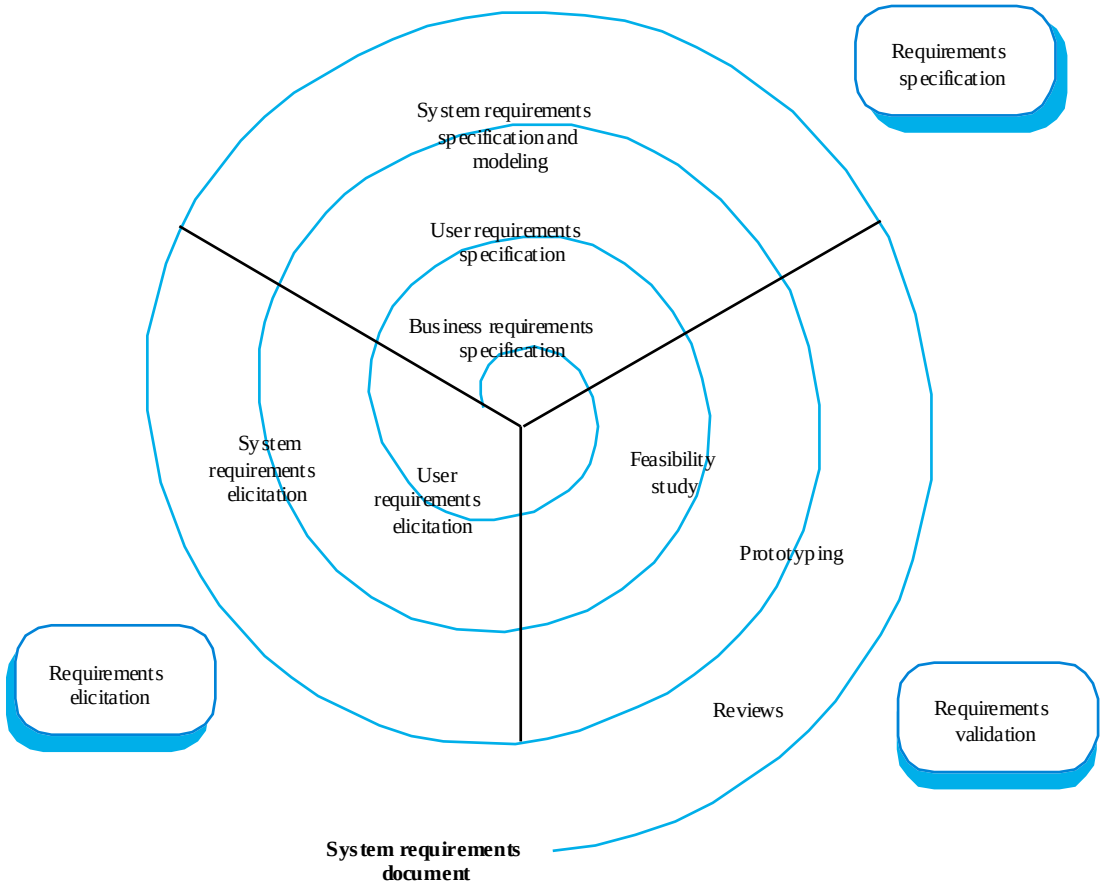
Requirements Engineering Processes



The Requirements Engineering Process



Requirements Engineering Processes



Feasibility studies

- The purpose of feasibility study is not to solve the problem, but to determine whether the problem is worth solving.
- A feasibility study decides whether or not the proposed system is worthwhile.
- The feasibility study concentrates on the following area.
  - Operational Feasibility
  - Technical Feasibility
  - Economic Feasibility

Feasibility studies

- A short focused study that checks
  - If the system contributes to organisational objectives;
  - If the system can be engineered using current technology and within budget;
  - If the system can be integrated with other systems that are used.

Feasibility study implementation

- Based on information assessment (what is required), information collection and report writing.
- Questions for people in the organisation
  - What if the system wasn't implemented?
  - What are current process problems?
  - How will the proposed system help?
  - What will be the integration problems?
  - Is new technology needed? What skills?
  - What facilities must be supported by the proposed system?

Requirements Analysis

- Requirements analysis in systems engineering and software engineering, encompasses those tasks that go into determining the needs or conditions to meet for a new or altered product, taking account of the possibly conflicting requirements of the various stakeholders, such as beneficiaries or users.

- Requirements analysis is critical to the success of a systems or software project. The requirements should be documented, actionable, measurable, testable, traceable, related to identified business needs or opportunities, and defined to a level of detail sufficient for system design.

Requirement Analysis

- Requirements analysis includes three types of activities
  - Eliciting requirements: The task of identifying the various types of requirements from various sources including project documentation, (e.g. the project charter or definition), business process documentation, and stakeholder interviews. This is sometimes also called requirements gathering.
  - Analyzing requirements: determining whether the stated requirements are clear, complete, consistent and unambiguous, and resolving any apparent conflicts.
  - Recording requirements: Requirements may be documented in various forms, usually including a summary list and may include natural-language documents, use cases, user stories, or process specifications.

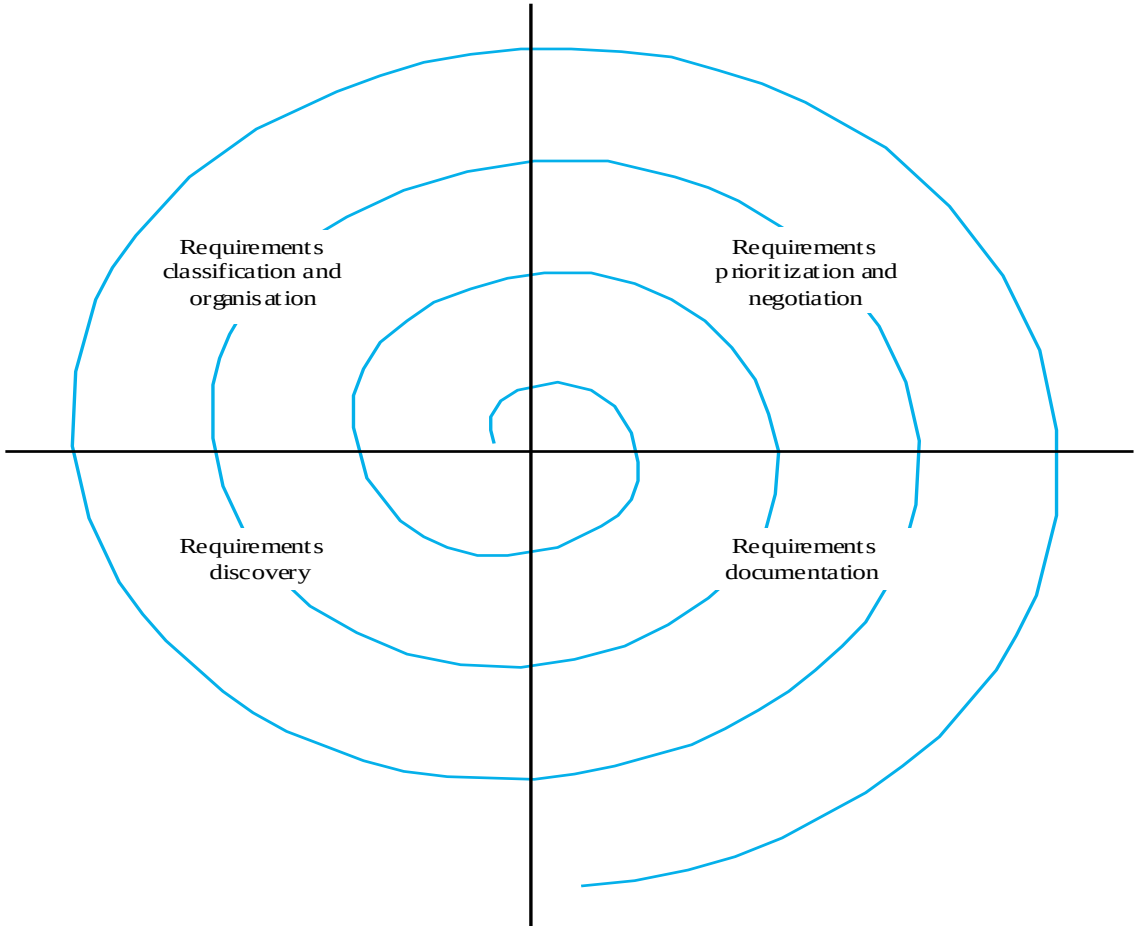
Problems of Requirements Analysis

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

Requirement Elicitation

- Requirements elicitation is the practice of collecting the requirements of a system from users, customers and other stakeholders. Sometimes called Requirements Discovery
- Requirements elicitation is important because one can never be sure to get all requirements from the user and customer by just asking them what the system should do
- Requirements elicitation practices include interviews, questionnaires, user observation, workshops, brain storming, use cases, role playing and prototyping.
- Before requirements can be analyzed, modeled, or specified they must be gathered through an elicitation process.
- Requirements elicitation is a part of the requirements engineering process, usually followed by analysis and specification of the requirements.

The Requirements Analysis Spiral



Requirements Analysis Process activities

- **Requirements discovery**
  - **Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.**
- **Requirements classification and organisation**
  - **Groups related requirements and organises them into coherent clusters.**
- **Prioritisation and negotiation**
  - **Prioritising requirements and resolving requirements conflicts.**
- **Requirements documentation**
  - **Requirements are documented and input into the next round of the spiral.**

Requirements discovery

- The process of gathering information about the proposed and existing systems and distilling(complete separation) the user and system requirements from this information.
- Sources of information include documentation, system stakeholders and the specifications of similar systems.

Requirements discovery

- Stakeholder Identification
  - Stakeholders (SH) are people or organizations (legal entities such as companies, standards bodies) that have a valid interest in the system. They may be affected by it either directly or indirectly
  - Stakeholders are not limited to the organization employing the analyst. Other stakeholders will include:
    - Anyone who operates the system (normal and maintenance operators)
    - Anyone who benefits from the system (functional, political, financial and social beneficiaries)

Requirements discovery

- Stakeholder Identification
  - Other stakeholders will include:
    - Anyone involved in purchasing or procuring the system. In a mass-market product organization, product management, marketing and sometimes sales act as surrogate consumers (mass-market customers) to guide development of the product
    - Organizations which regulate aspects of the system (financial, safety, and other regulators)
    - People or organizations opposed to the system (negative stakeholders
    - Organizations responsible for systems which interface with the system under design

Requirements Discovery

- E.g. ATM stakeholders
  - Bank customers
  - Representatives of other banks
  - Bank managers
  - Counter staff
  - Database administrators
  - Security managers
  - Marketing department
  - Hardware and software maintenance engineers
  - Banking regulators

Requirements Discovery -Viewpoints

- Viewpoints are a way of structuring the requirements to represent the perspectives of different stakeholders. Stakeholders may be classified under different viewpoints.
- This multi-perspective analysis is important as there is no single correct way to analyse system requirements.

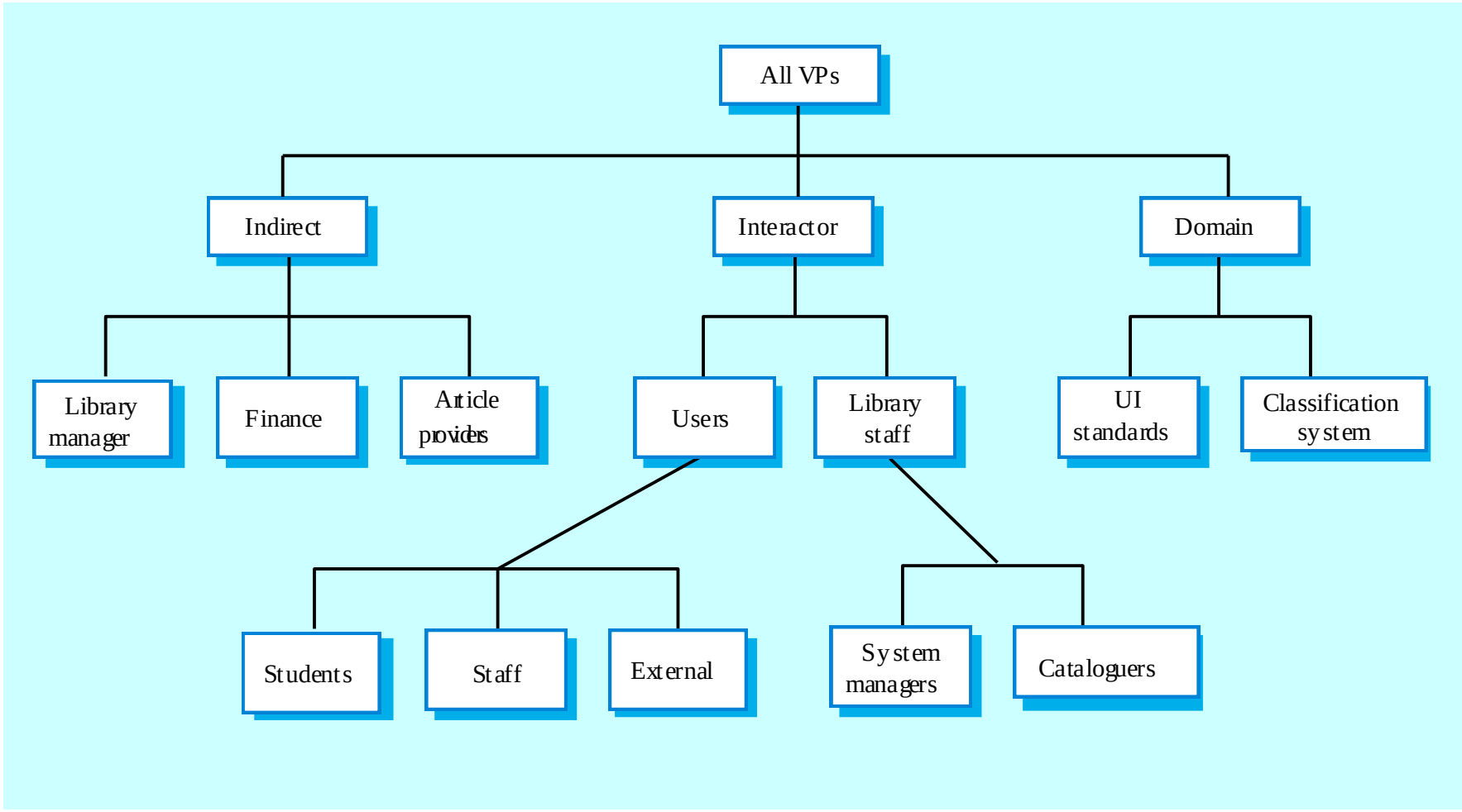
Requirements Discovery -Viewpoints

- Types of viewpoint
  - Interactor viewpoints
    - People or other systems that interact directly with the system. In an ATM, the customers and the account database are interactor VPs.
  - Indirect viewpoints
    - Stakeholders who do not use the system themselves but who influence the requirements. In an ATM, management and security staff are indirect viewpoints.
  - Domain viewpoints
    - Domain characteristics and constraints that influence the requirements. In an ATM, an example would be standards for inter-bank communications.

Viewpoint Identification

- Identify viewpoints using
  - Providers and receivers of system services;
  - Systems that interact directly with the system being specified;
  - Regulations and standards;
  - Sources of business and non-functional requirements.
  - Engineers who have to develop and maintain the system;
  - Marketing and other business viewpoints.

LIBSYS viewpoint hierarchy



Interviewing

- The interview is the primary technique for information gathering during the systems analysis phases of a development project. It is a skill which must be mastered by every analyst.
- The interviewing skills of the analyst determine what information is gathered, and the quality and depth of that information. Interviewing, observation, and research are the primary tools of the analyst.
- In formal or informal interviewing, the RE team puts questions to stakeholders about the system that they use and the system to be developed.
- Interviews are good for getting an overall understanding of what stakeholders do and how they might interact with the system.

Interviewing

- Goals of the Interview
  - At each level, each phase, and with each interviewee, an interview may be conducted to:
    - Gather information on the company
    - Gather information on the function
    - Gather information on processes or activities
    - Uncover problems
    - Conduct a needs determination
    - Verification of previously gathered facts
    - Gather opinions or viewpoints
    - Provide information
    - Obtain leads for further interviews

Interviews In Practice

- There are two types of interview
  - Closed interviews where a pre-defined set of questions are answered.
  - Open interviews where there is no pre-defined agenda and a range of issues are explored with stakeholders.
- Normally a mix of closed and open-ended interviewing is undertaken.
- Interviews are not good for understanding domain requirements
  - Requirements engineers cannot understand specific domain terminology;
  - Some domain knowledge is so familiar that people find it hard to articulate or think that it isn’t worth articulating.

Interviews

- Effective Interviewers
  - Interviewers should be open-minded, willing to listen to stakeholders and should not have pre-conceived ideas about the requirements.
  - They should prompt the interviewee with a question or a proposal and should not simply expect them to respond to a question such as ‘what do you want’.
- Information form interviews supplement other information about the system from documents, user observations, and so on
- Sometimes, apart from information from documents, interviews may be the only source of information about the system requirements
- It should be used alongside other requirements elicitation techniques

Scenarios

- Scenarios are real-life examples of how a system can be used.
- Scenarios can be particularly useful for adding detail to an outline requirements description.
- Each scenario covers one or more possible interactions
- Several forms of scenarios can be developed, each of which provides different types of information at different levels of detail about the system
- Scenarios may be written as text, supplemented by diagrams, screen shots and so on

Scenarios

- Scenarios are real-life examples of how a system can be used.
- A scenario may include
  - A description of the starting situation;
  - A description of the normal flow of events;
  - A description of what can go wrong;
  - Information about other concurrent activities that might be going on at the same time
  - A description of the system state when the scenario finishes.

LIBSYS scenario (1)

Initial assumption: The user has logged on to the LIBSYS system and has located the journal containing the copy of the article.

Normal: The user selects the article to be copied. He or she is then prompted by the system to either provide subscriber information for the journal or to indicate how they will pay for the article. Alternative payment methods are by credit card or by quoting an organisational account number.

The user is then asked to fill in a copyright form that maintains details of the transaction and they then submit this to the LIBSYS system.

The copyright form is checked and, if OK, the PDF version of the article is downloaded to the LIBSYS working area on the userÕs computer and the user is informed that it is available. The user is asked to select a printer and a copy of the article is printed. If the article has been flagged as Ôprint-onlyÕ it is deleted from the userÕs system once the user has confirmed that printing is complete.

LIBSYS scenario (2)

What can go wrong: The user may fail to fill in the copyright form correctly. In this case, the form should be re-presented to the user for correction. If the resubmitted form is still incorrect then the userÕs request for the article is rejected.

The payment may be rejected by the system. The userÕs request for the article is rejected.

The article download may fail. Retry until successful or the user terminates the session.

It may not be possible to print the article. If the article is not flagged as Ôprint-onlyÕ then it is held in the LIBSYS workspace. Otherwise, the article is deleted and the userÕs account credited with the cost of the article.

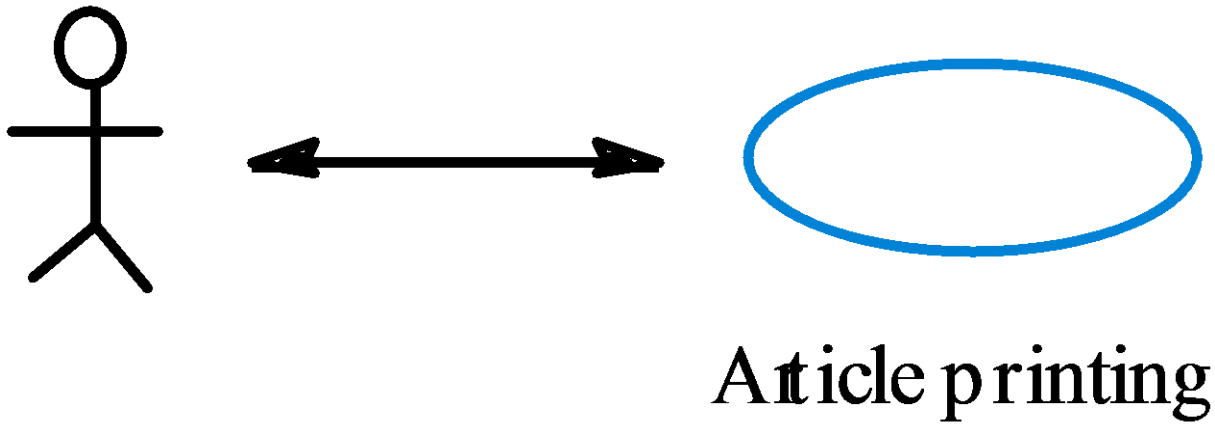
Other activities: Simultaneous downloads of other articles.

System state on completion: User is logged on. The downloaded article has been deleted from LIBSYS workspace if it has been flagged as print-only.

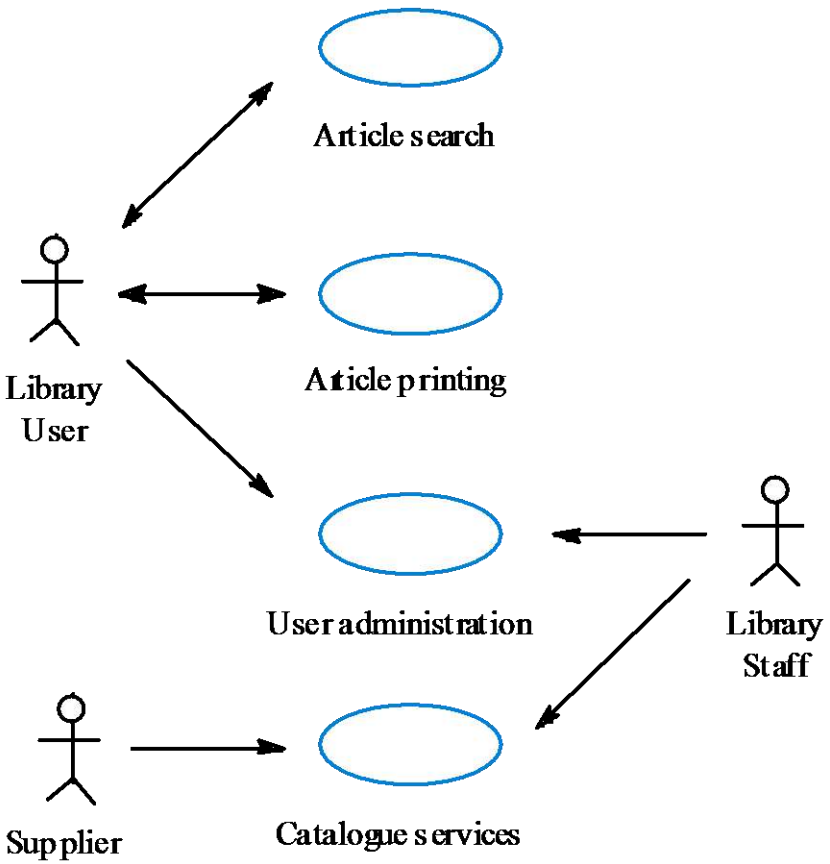
Use cases

- Use-cases are a scenario based technique in the UML which identify the actors in an interaction and which describe the interaction itself.
- A set of use cases should describe all possible interactions with the system.
- Sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system.
- Use-case approach helps with requirements prioitization

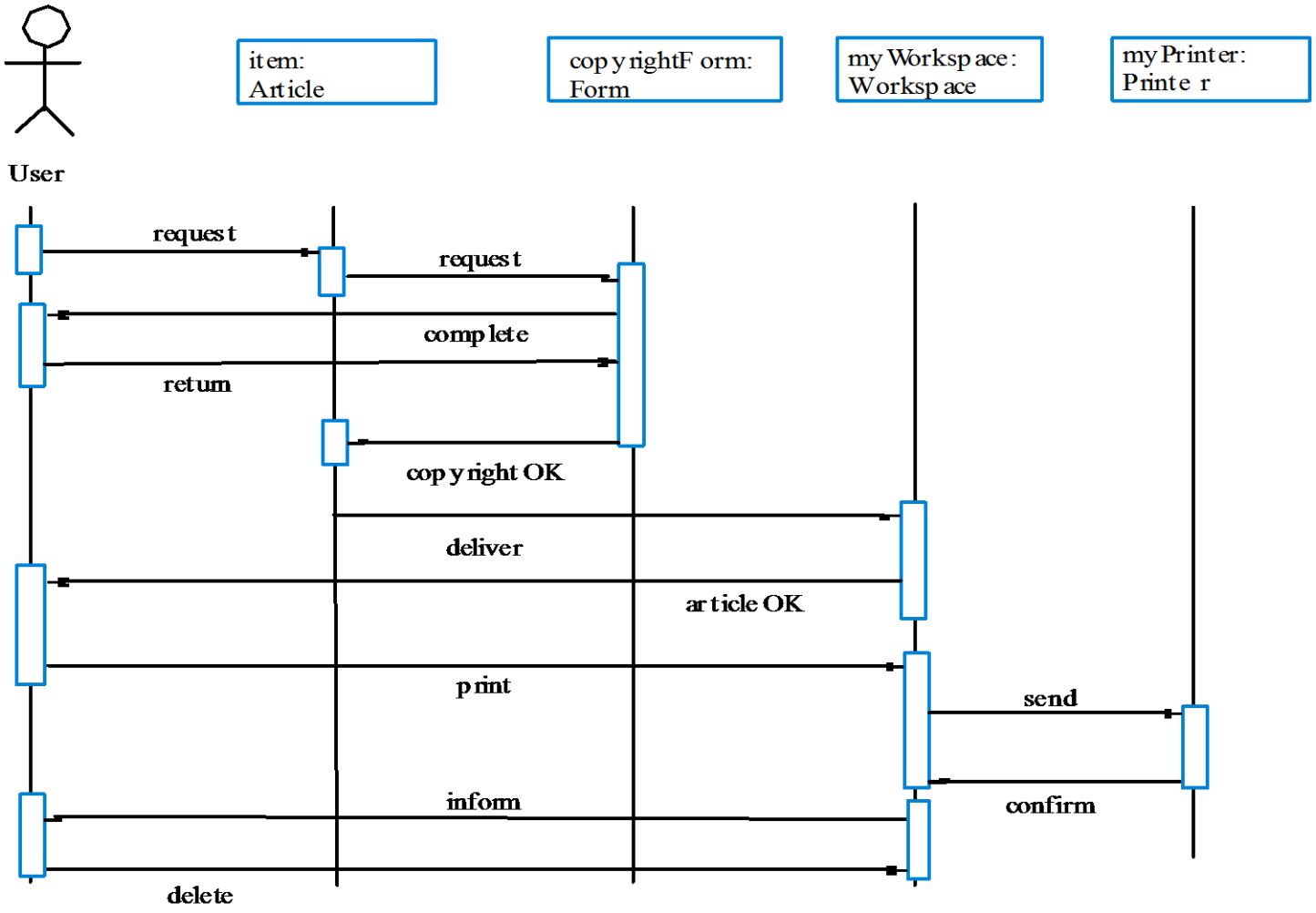
Article printing use-case



LIBSYS use cases



Print article sequence



USE CASES

- A Use case can have high priority for
  - It describes one of the business process that the system enables
  - Many users will use it frequently
  - A favoured user class requested it
  - It provides capability that’s required for regularoty compliance
  - Other system functions depend on its presence

Social and organisational factors

- Software systems are used in a social and organisational context. This can influence or even dominate the system requirements.
- Social and organisational factors are not a single viewpoint but have influences on all viewpoints.
- Good analysts must be sensitive to these factors but currently no systematic way to tackle their analysis.

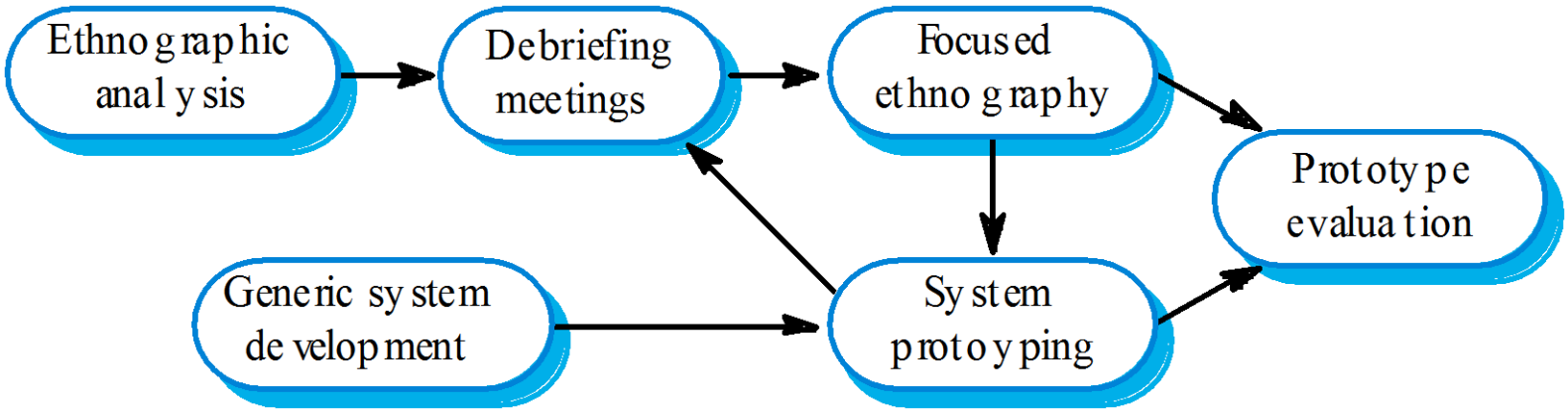
Ethnography

- A social scientists spends a considerable time observing and analysing how people actually work.
- People do not have to explain or articulate their work.
- Social and organisational factors of importance may be observed.
- Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models.

Focused ethnography

- Developed in a project studying the air traffic control process
- Combines ethnography with prototyping
- Prototype development results in unanswered questions which focus the ethnographic analysis.
- The problem with ethnography is that it studies existing practices which may have some historical basis which is no longer relevant.

Ethnography and prototyping



Scope of Ethnography

- Requirements that are derived from the way that people actually work rather than the way which process definitions suggest that they ought to work.
- Requirements that are derived from cooperation and awareness of other people's activities.

Requirements Classification

In order to better understand and manage the large number of requirements, it is important to organize them in logical clusters

- It is possible to classify the requirements by the following categories (or any other clustering that appears to be convenient)
  - Features
  - Use cases
  - Mode of operation
  - User class
  - Responsible subsystem
- This makes it easier to understand the intended capabilities of the product
- And more effective to manage and prioritize large groups rather than single requirements

Requirements Classification – Features

- A Feature is
  - a set of logically related (functional) requirements that provides a capability to the user and enables the satisfaction of a business objective
- The description of a feature should include[1]
  - Name of feature (e.g. Spell check)
  - Description and Priority
  - Stimulus/response sequences
  - List of associated functional requirements

Requirements Classification – Feature Example

- 3.1 Order Meals
  - 3.1.1 Description and Priority
    - A cafeteria Patron whose identity has been verified may order meals either to be delivered to a specified company location or to be picked up in the cafeteria. A Patron may cancel or change a meal order if it has not yet been prepared.
    - Priority = High.
  - 3.1.2 Stimulus/Response Sequences
    - Stimulus: Patron requests to place an order for one or more meals.
    - Response: System queries Patron for details of meal(s), payment, and delivery instructions.
    - Stimulus: Patron requests to change a meal order.
    - Response: If status is “Accepted,” system allows user to edit a previous meal order.

Requirements Classification – Feature Example

- Stimulus: Patron requests to cancel a meal order.
- Response: If status is “Accepted, ”system cancels a meal order.

– 3.1.3 Functional Requirements

- 3.1.3.1. The system shall let a Patron who is logged into the Cafeteria Ordering System place an order for one or more meals.
- 3.1.3.2. The system shall confirm that the Patron is registered for payroll deduction to place an order.
- .....

Requirements Validation

- Concerned with demonstrating that the requirements define the system that the customer really wants.
- Requirements error costs are high so validation is very important
  - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

Requirements Checking

- Validity. Does the system provide the functions which best support the customer's needs?
- Consistency. Are there any requirements conflicts?
- Completeness. Are all functions required by the customer included?
- Realism. Can the requirements be implemented given available budget and technology
- Verifiability. Can the requirements be checked?

Requirements Validation Techniques

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

Requirements Reviews

- Regular reviews should be held while the requirements definition is being formulated.
- Both client and contractor staff should be involved in reviews.
- Reviews may be formal (with completed documents) or informal. Good communications between developers, customers and users can resolve problems at an early stage.

Review Checks

- Verifiability. Is the requirement realistically testable?
- Comprehensibility. Is the requirement properly understood?
- Traceability. Is the origin of the requirement clearly stated?
- Adaptability. Can the requirement be changed without a large impact on other requirements?

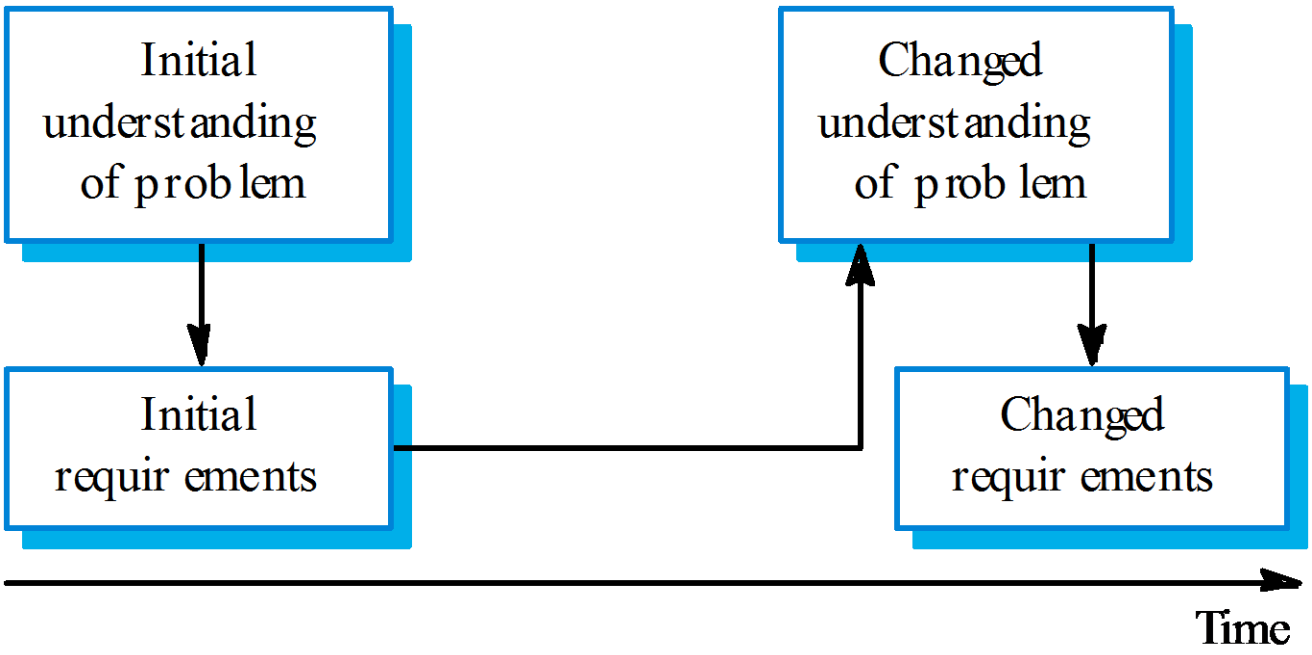
Requirements Management

- Requirements management is the process of managing changing requirements during the requirements engineering process and system development.
- Requirements are inevitably incomplete and inconsistent
  - New requirements emerge during the process as business needs change and a better understanding of the system is developed;
  - Different viewpoints have different requirements and these are often contradictory.

Requirements Change

- The priority of requirements from different viewpoints changes during the development process.
- System customers may specify requirements from a business perspective that conflict with end-user requirements.
- The business and technical environment of the system changes during its development.

Requirements Evolution



Enduring and Volatile Requirements

- Enduring requirements
  - These are relatively stable requirements that derive from the core activity of the organization
  - Relate directly to the domain of the system
  - These requirements may be derived from domain models that show the entities and relations which characterise an application domain
  - For example, in a hospital there will always be requirements concerned with patients, doctors, nurses, treatments, etc

Enduring and Volatile Requirements

- Volatile requirements
  - These are requirements that are likely to change during the system development process or after the system has been become operational.
  - Examples of volatile requirements are requirements resulting from government health-care policies or healthcare charging mechanisms.

Enduring and Volatile Requirements

- **Volatile requirements can be classified as**

Requirement type	Description
Mutable requirements	Requirements that change because of changes to the environment in which the organisation is operating. For example, in hospital systems, the funding of patient care may change and thus require different treatment information to be collected
Consequential requirements	Requirements that result from the introduction of the computer system. Introducing the computer system may change the organisations processes and open up new ways of working which generate new system requirements.
Compatibility requirements	Requirements that depend on the particular systems or business processes within an organisation. As these change, the compatibility requirements on the commissioned or delivered system may also have to evolve.
Emergent requirements	Requirements that emerge as the customer's understanding of the system develops during the system development. The design process may reveal new emergent requirements.

Traceability

- Traceability is concerned with the relationships between requirements, their sources and the system design
- Source traceability
  - Links from requirements to stakeholders who proposed these requirements;
- Requirements traceability
  - Links between dependent requirements;
- Design traceability
  - Links from the requirements to the design;

A traceability Matrix

Req. id	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1		D	R					
1.2			D			D		D
1.3	R			R				
2.1			R		D			D
2.2								D
2.3		R		D				
3.1								R
3.2							R	

CASE tool support

- Requirements storage
  - Requirements should be managed in a secure, managed data store.
- Change management
  - The process of change management is a workflow process whose stages can be defined and information flow between these stages partially automated.
- Traceability management
  - Automated retrieval of the links between requirements.

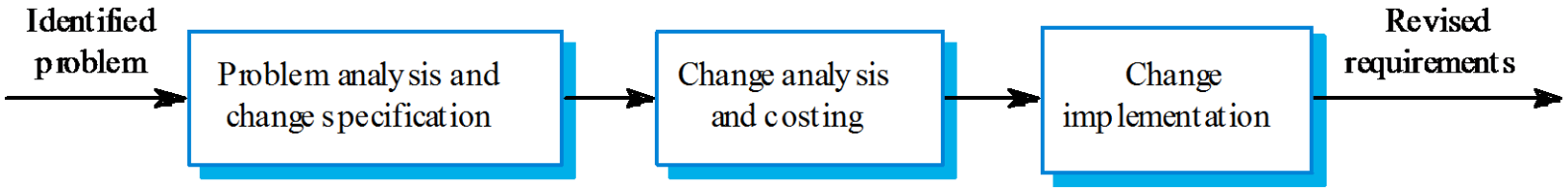
Requirements Management Planning

- During the requirements engineering process, one has to plan:
  - Requirements identification
    - How requirements are individually identified;
  - A change management process
    - The process followed when analysing a requirements change;
  - Traceability policies
    - The amount of information about requirements relationships that is maintained;
  - CASE tool support
    - The tool support required to help manage requirements change;

Requirements Change Management

- Should apply to all proposed changes to the requirements.
- Principal stages
  - Problem analysis. Discuss requirements problem and propose change;
  - Change analysis and costing. Assess effects of change on other requirements;
  - Change implementation. Modify requirements document and other documents to reflect change.

Change Management



Classical Analysis

- **Structured system analysis**
- Throughout the phases of analysis and design, the analyst should proceed step by step, obtaining feedback from users and analyzing the design for omissions and errors.
- Moving too quickly to the next phase may require the analyst to rework portions of the design that were produced earlier.
- They Structured a project into small, well-defines activities and specify the sequence and interaction of these activities.
- They use diagrammatic and other modeling techniques to give a more precise (structured) definition that is understandable by both users and developers.
- Structured analysis provides a clear requirements statements that everyone can understand and is a firm foundation for subsequent design and implementation.
- Part of the problem with systems analysts just asking 'the right questions' that it is often difficult for a technical person to describe the system concepts back to the user can understand.
- Structured methods generally include the use of easily understood, non technical diagrammatic techniques.
- It is important that these diagram do not contain computer jargon and technical detail that the user wont understand – and does not need understand.
- Use graphics whenever possible to help communicate better with the user.
- Differentiate between logical and physical system.
- Build a logical system model to familiarize the user with system characteristics and interrelationships before implementation.

Structured system analysis

- A Data Flow Diagram (DFD) is a graphical representation of the "flow" of data through an information system.
- It differs from the system flowchart as it shows the flow of data through processes instead of computer hardware.
- It is common practice to draw a System Context Diagram first which shows the interaction between the system and outside entities.
- The DFD is designed to show how a system is divided into smaller portions and to highlight the flow of data between those parts.
- This context-level Data flow diagram is then "exploded" to show more detail of the system being modeled. 10

High-Level Petri Nets

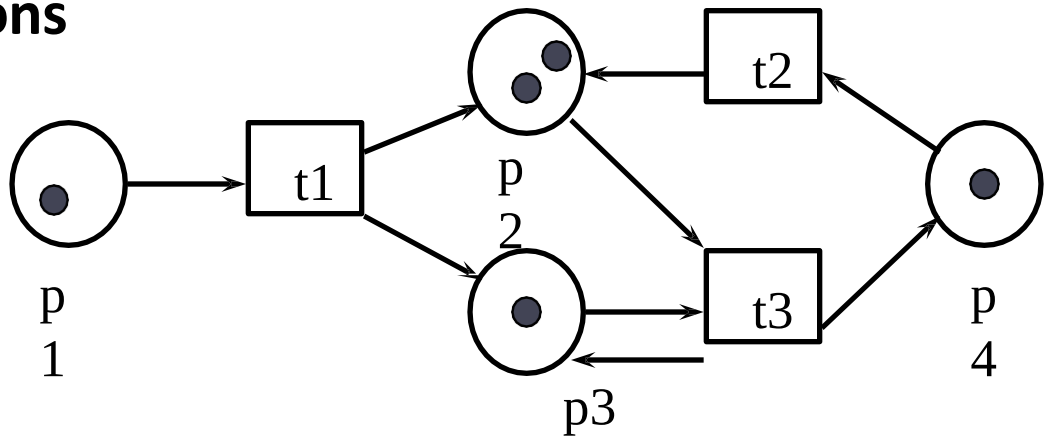
- The classical Petri net was invented by Carl Adam Petri in 1962.
- A lot of research has been conducted (>10,000 publications).
- Until 1985 it was mainly used by theoreticians.
- Since the 80's their practical use has increased because of the introduction of high-level Petri nets and the availability of many tools.
- **High-level Petri nets** are Petri nets extended with
  - colour (for the modelling of attributes)
  - time (for performance analysis)
  - hierarchy (for the structuring of models, DFD's)

Why do we need Petri Nets?

- Petri Nets can be used to rigorously define a system (reducing ambiguity, making the operations of a system clear, allowing us to prove properties of a system etc.)
- They are often used for **distributed systems** (with several subsystems acting independently) and for systems with **resource sharing**.
- Since there may be more than one transition in the Petri Net active at the same time (and we do not know which will 'fire' first), they are **non-deterministic**.

The Classical Petri Net Model

A **Petri net** is a network composed of **places** ( ) and **transitions**

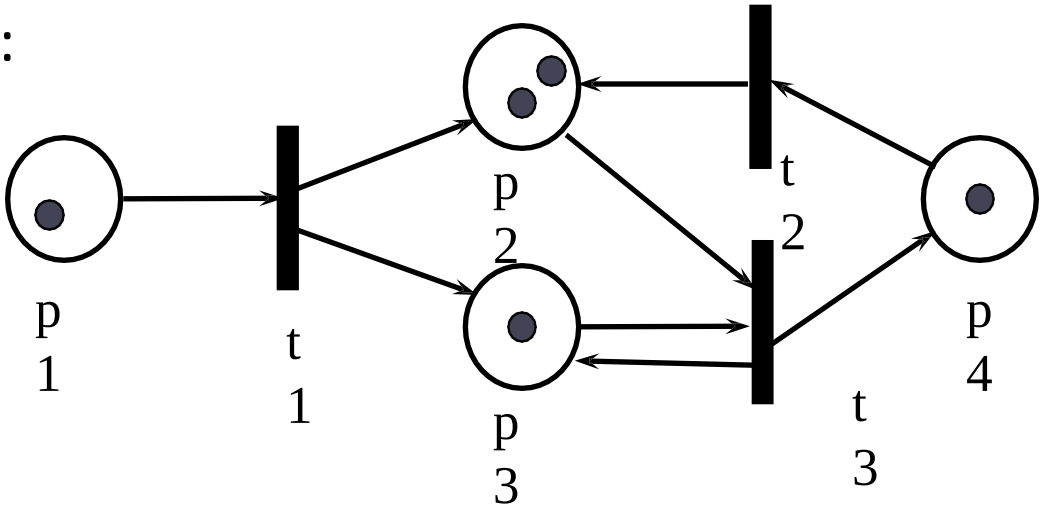


Connections are directed and between a place and a transition, or a transition and a place (e.g. Between "p1 and t1" or "t1 and p2" above).

Tokens ( ) are the dynamic objects.

The Classical Petri Net Model

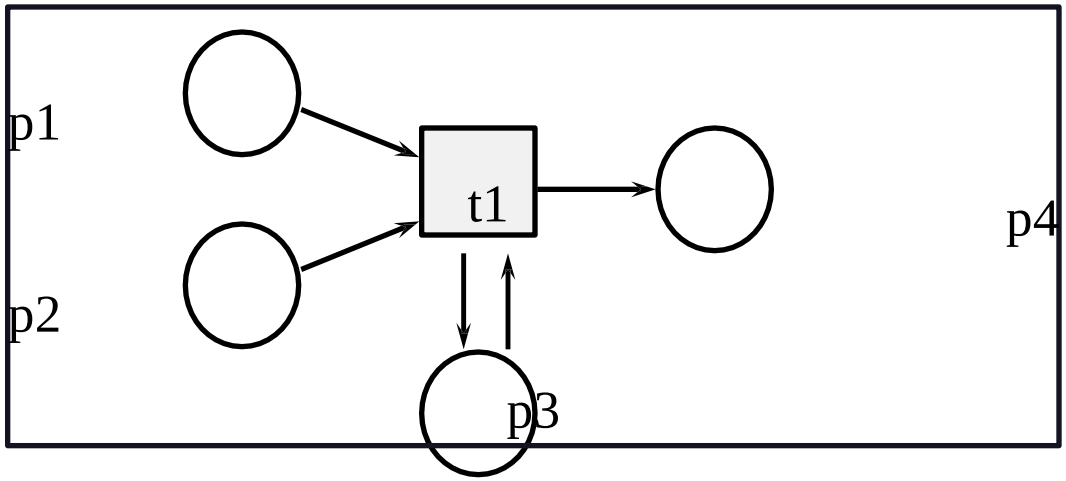
Another (**equivalent**) notation is to use a solid bar for the transitions:



We may use either notation since they are equivalent, sometimes one makes the diagram easier to read than the other..

The **state** of a Petri net is determined by the distribution of tokens over the places (we could represent the above **state** as (1,2,1,1) for (p1,p2,p3,p4))

Transitions with Multiple Inputs and Outputs

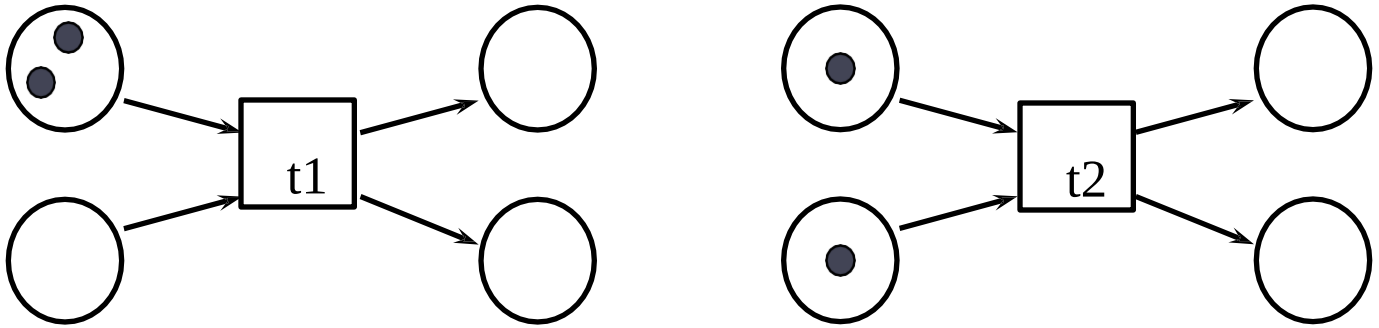


Transition t1 has three **input places** (p1, p2 and p3) and two **output places** (p3 and p4).

Place p3 is both an input and an output place of t1.

Enabling Condition

- Transitions are the **active** components and places and tokens are
  passive components.
- A transition is **enabled** if each of the input places contains tokens.

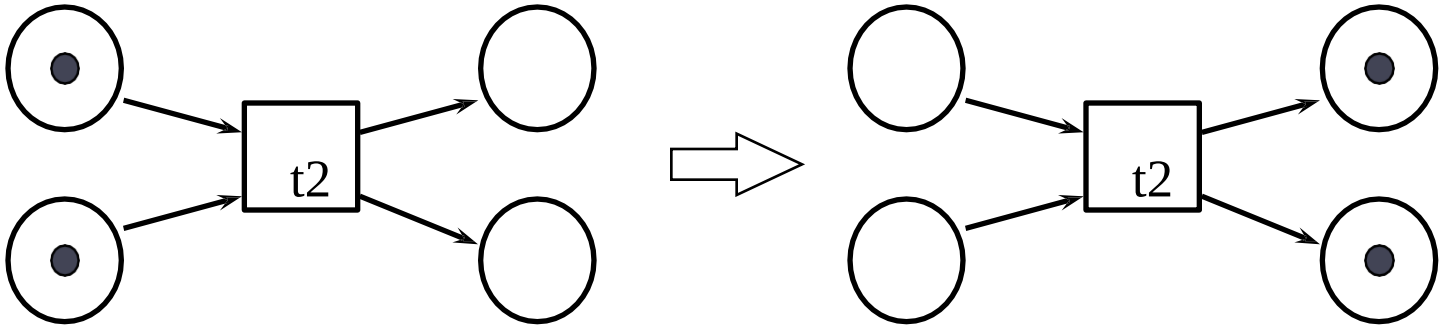


Transition t1 is not enabled, transition t2 is enabled.

Firing

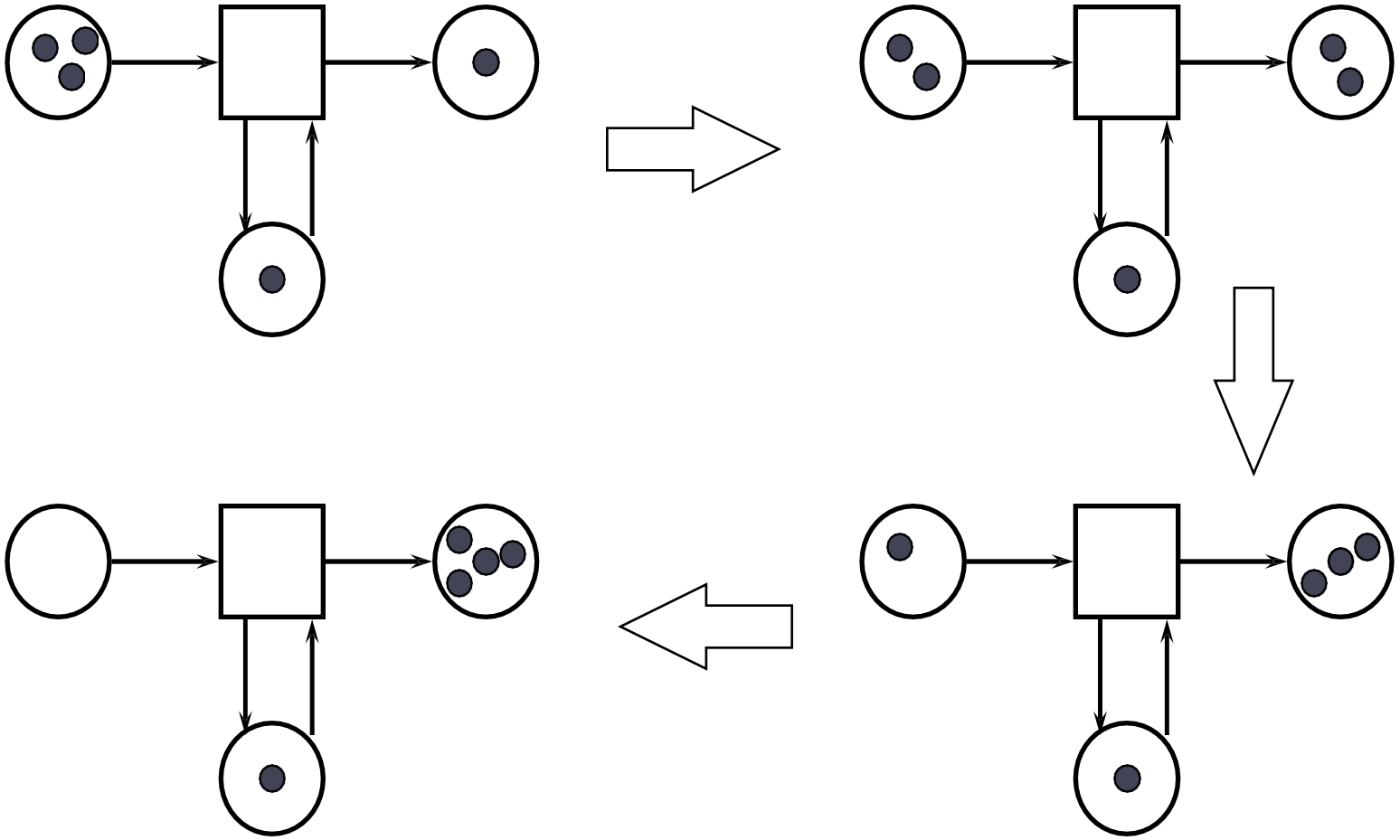
An enabled transition may **fire**.

Firing corresponds to **consuming** tokens from the input
places and **producing** tokens for the output places.

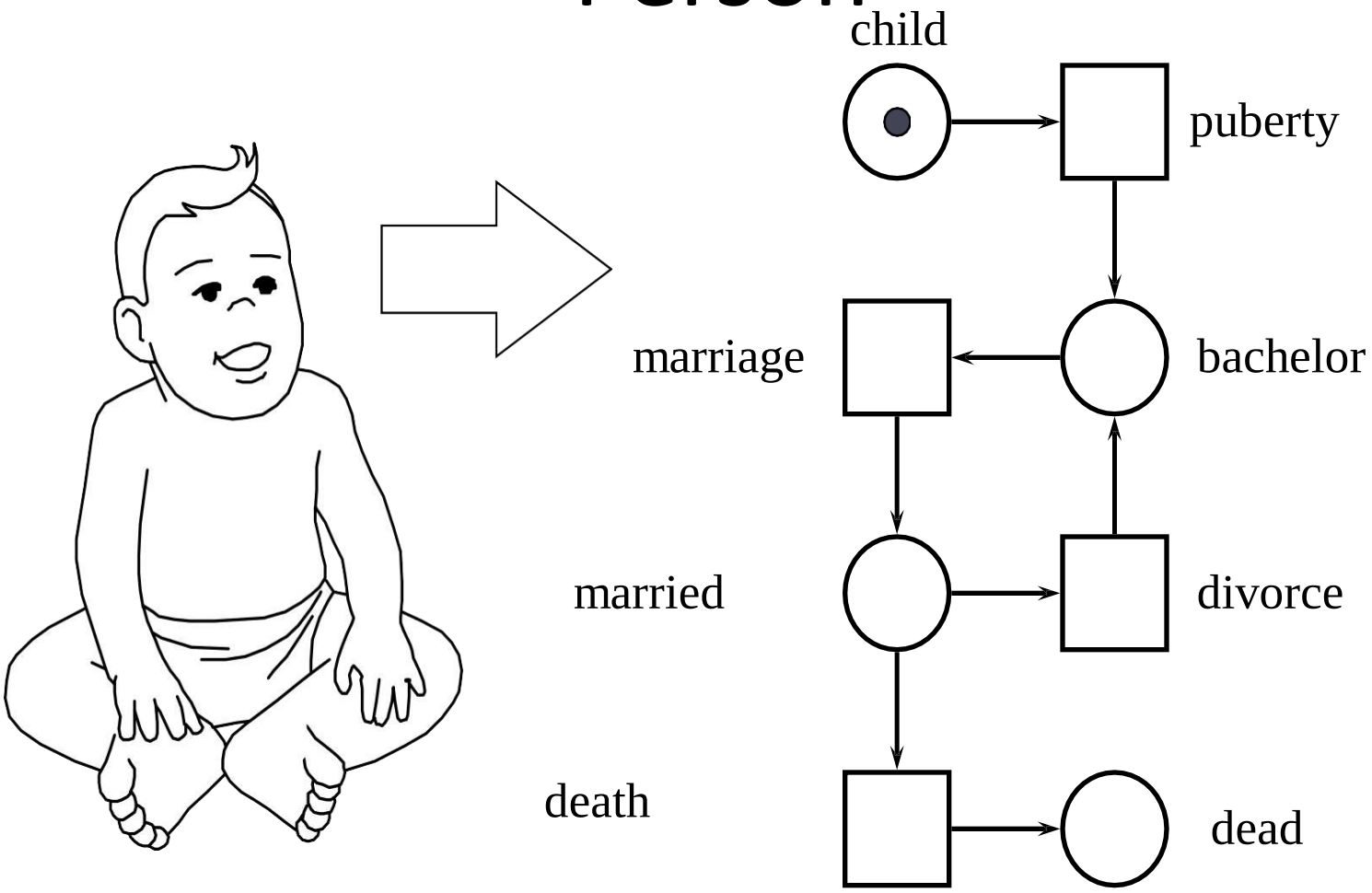


Firing is **atomic** (only one transition fires at a time, even if more than one is enabled)

An Example Petri Net

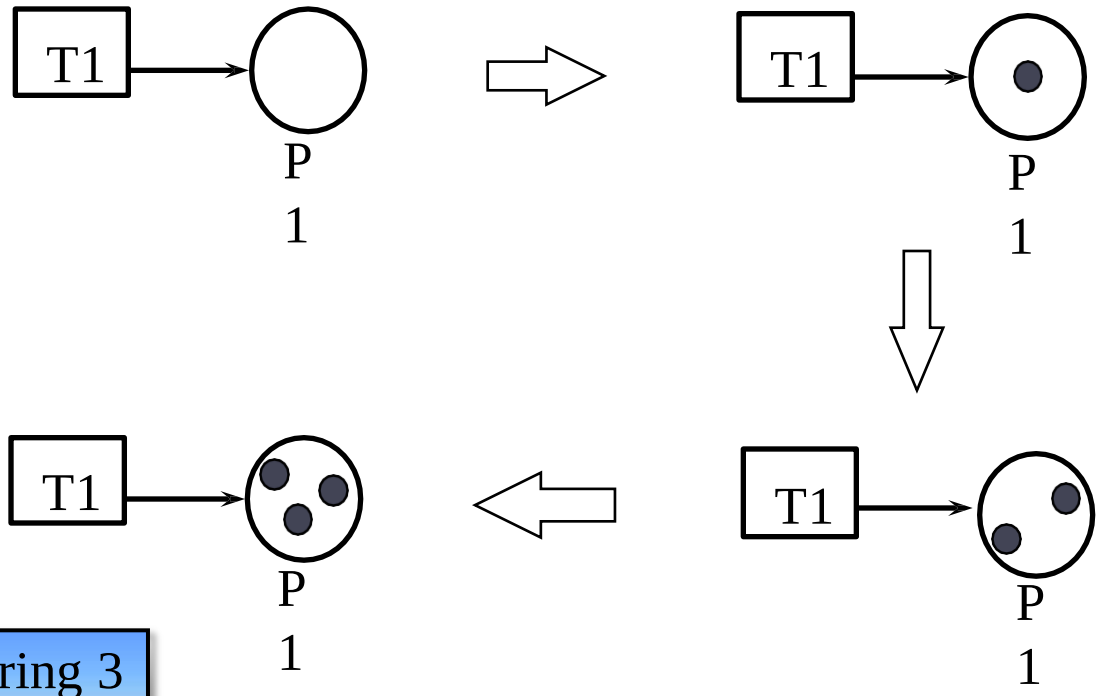


Example: Life-Cycle of a Person



Creating/Consuming Tokens

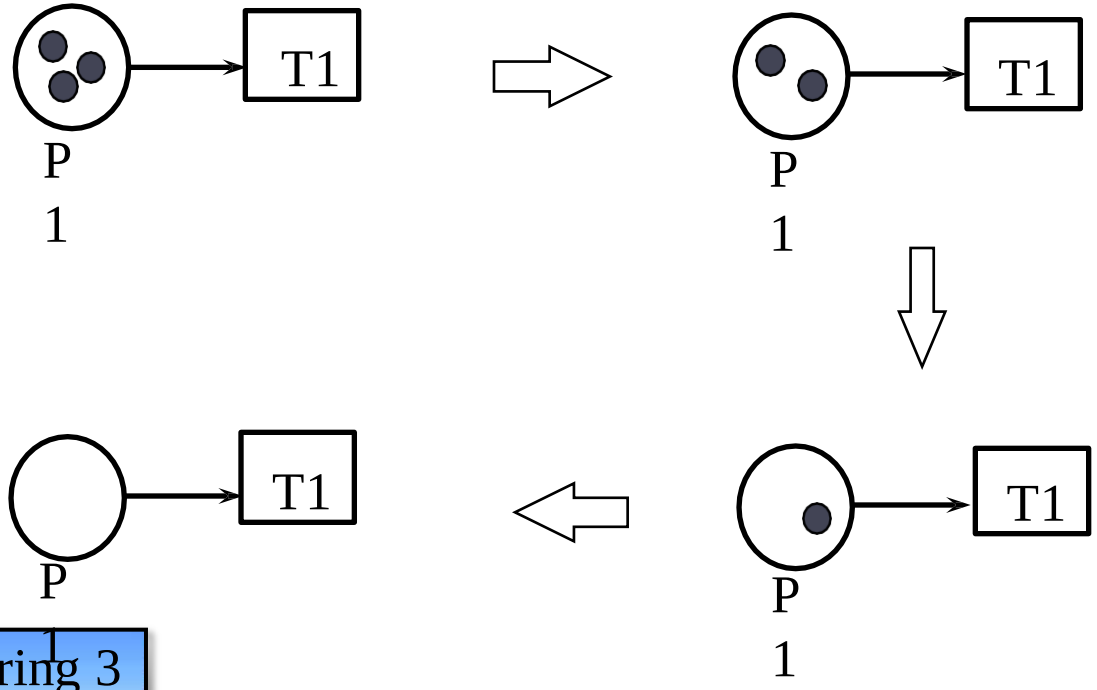
A transition without any input can fire at any time and produces tokens in the connected places:



After firing 3 times..

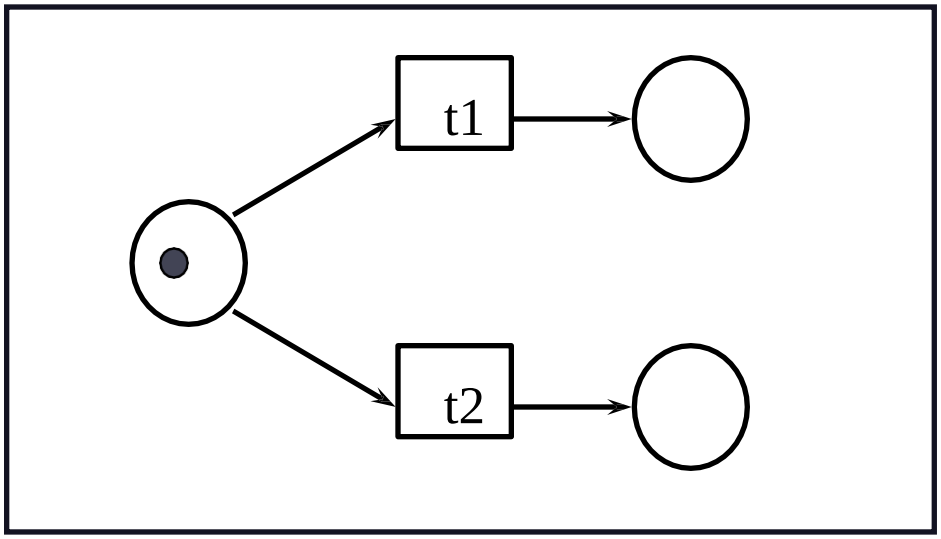
Creating/Consuming Tokens

A transition without any output must be enabled to fire
and deletes (or consumes) the incoming token(s):



After firing 3 times..

Non-Determinism in Petri Nets



Two transitions fight for the same token: **conflict**.

Even if there are two tokens, there is still a conflict.

The next transition to fire (t1 or t2) is arbitrary (non-deterministic).

Data Dictionary

1. A tool for recording and processing information (metadata) about the data that an organisation uses.
2. A central catalogue for metadata.
3. Can be integrated within the DBMS or be separate.
4. May be referenced during system design, programming, and by actively-executing programs.
5. Can be used as a repository for common code (e.g. library routines).

Benefits of a DDS

Benefits of a DDS are mainly due to the fact that it is a central store of information about the database.

- Benefits include -
- improved documentation and control
- consistency in data use
- easier data analysis
- reduced data redundancy
- simpler programming
- the enforcement of standards
- better means of estimating the effect of change.

DDS Facilities

A DDS should provide two sets of facilities:

- To record and analyse data requirements independently of how they are going to be met - conceptual data models (entities, attributes, relationships).
- To record and design decisions in terms of database or file structures implemented and the programs which access them - internal schema.

One of the main functions of a DDS is to show the relationship between the conceptual and implementation views. The mapping should be consistent - inconsistencies are an error and can be detected here.

DD Management

- With so much detail held on the DDS, it is essential that an indexing and cross-referencing facility is provided by the DDS.
- The DDS can produce reports for use by the data administration staff (to investigate the efficiency of use and storage of data), systems analysts, programmers, and users.
- DDS can provide a pre-printed form to aid data input into the database and DD.
- A query language is provided for ad-hoc queries. If the DD is tied to the DBMS, then the query language will be that of the DBMS itself.

Management Objectives

From an management point of view, the DDS should

- provide facilities for documenting information collected during all stages of a computer project.
- provide details of applications usage and their data usage once a system has been implemented, so that analysis and redesign may be facilitated as the environment changes.
- make access to the DD information easier than a paper-based approach by providing cross-referencing and indexing facilities.
- make extension of the DD information easier.
- encourage systems analysts to follow structured methodologies.

Advanced Facilities

Extra facilities which may be supported by DDS are:

- Automatic input from source code of data definitions (at compile time).
- The recognition that several versions of the same programs or data structures may exist at the same time.
  - live and test states of the programs or data.
  - programs and data structures which may be used at different sites.
  - data set up under different software or validation routine.
- The provision of an interface with a DBMS.
- Security features such as password protection, to restrict DDS access.
- Generation of update application programs and programs to produce reports and validation routines.

Management Advantages

A number of possible benefits may come from using a DDS:

- improve control and knowledge about the data resource.
- allows accurate assessment of cost and time scale to effect any changes.
- reduces the clerical load of database administration, and gives more control
- over the design and use of the database.
- accurate data definitions can be provided securely directly to programs.
- aid the recording, processing, storage and destruction of data and associated documents.

Management Disadvantages

A DDS is a useful management tool, but at a price.

- The DDS 'project' may itself take two or three years.
- It needs careful planning, defining the exact requirements designing its contents, testing, implementation and evaluation.
- The cost of a DDS includes not only the initial price of its installation and any hardware requirements, but also the cost of collecting the information entering it into the DDS, keeping it up-to-date and enforcing standards.
- The use of a DDS requires management commitment, which is not easy to achieve, particularly where the benefits are intangible and long term.

UNIT- III
SOFTWARE DESIGN

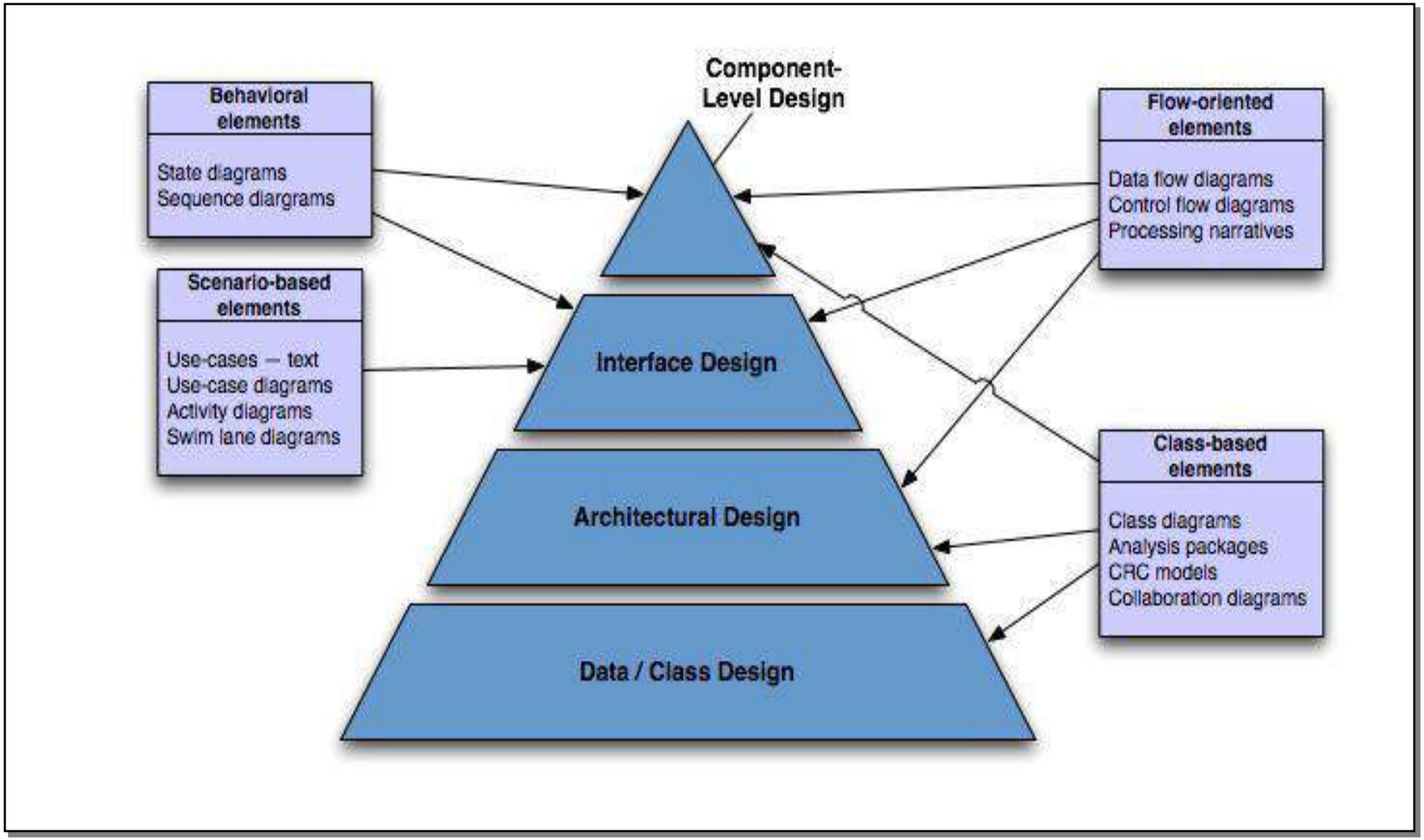
Design process, Design concepts, design model, design heuristic, architectural design, architectural styles, Assessing alternative architectural designs, and architectural mapping using data flow.

User interface design: Interface analysis, interface design; Component level design: Designing class based components, traditional components.

Design Engineering

- **A good software should exhibit the following properties**
  - **Firmness – A program should not have any bugs that inhibit its function.**
  - **Commodity – A program should be suitable for the purposes for which it was intended**
  - **Delight – The experience of using the program should be a pleasurable one.**
- **The goal of design engineering is to produce a model or representation that exhibits firmness, commodity and delight. Design engineering for computer software changes continuously as new methods, better analysis, and broader understanding evolve.**

Analysis → Design



Design Engineering

- **Design creates a representation or model of the software, but unlike the analysis model, the design model provides detail about software data structures, architecture, interfaces, and components that are necessary to implement the system**

- **Design allows a software engineer to model the system or product that is to be built**

- **A design model that encompasses architectural, interface, component-level, and deployment representations is the primary work product that is produced during software design**

Design Engineering

- **The design model is assessed by the software team in an effort to determine whether it contains errors, inconsistencies, or omission, whether better alternatives exist, and whether the model can be implemented within the constraints, schedule, and cost that have been established**
- **Design produces a data /class design, an architectural design, an interface design, and a component design**
- **Data /analysis design transforms analysis –class models into design class realizations and the requisite data structures required to implement the software**

Design Engineering

- **The architectural design defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that effect the way in which architectural can be implemented**
- **The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it.**
- **The component level design transforms structural elements of the software architecture into a procedural description of software components**

The Design Process:

- **Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blue print depicts a holistic view of software. That is, the design is represented at a high level of abstraction— a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements.**

1. Software Quality Guidelines and Attributes

- **Design Guidelines**
- **A good design should**
  - **exhibit good architectural structure**
  - **be modular**
  - **contain distinct representations of data, architecture, interfaces, and components (modules)**
  - **lead to data structures that are appropriate for the objects to be implemented and be drawn from recognizable design patterns**
  - **lead to components that exhibit independent functional characteristics**
  - **lead to interfaces that reduce the complexity of connections between modules and with the external environment**
  - **be derived using a reputable method that is driven by information obtained during software requirements analysis**

- **Quality Guidelines**
  - **A design should exhibit an architecture that**

    (1) has been created using recognizable architectural styles or patterns

    (2) is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and (3) can be implemented in an evolutionary fashion,2 thereby facilitating implementation and testing.
  - **A design should be modular; that is, the software should be logically partitioned into elements or subsystems.**
  - **A design should contain distinct representations of data, architecture, interfaces, and components.**
  - **A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.**

- **Quality Guidelines**

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

- **Quality attributes**
  - **Functionality is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are derived and the security of the overall system**
  - **Usability is assessed by considering human factors, overall aesthetics, consistency and documentation**
  - **Reliability is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure, the ability to recover form failure, and the predictability of the program.**
  - **Performance is measured by processing speed, response time, resource consumption, throughput, and efficiency**
  - **Supportability combines the ability to extend the program, adaptability, serviceability, testability, compatibility, configurability, the ease with which a system can be installed, and the ease with which problems can be localized**

The Design Process:

2. The Evolution of Software Design:

- **The evolution of software design is a continuing process that has now spanned almost six decades.**
- **All these methods have a number of common characteristics:**
  1. **a mechanism for the translation of the requirements model into a design representation,**
  2. **a notation for representing functional components and their interfaces,**
  3. **heuristics for refinement and partitioning, and**
  4. **guidelines for quality assessment.**

Regardless of the design method that is used, you should apply a set of basic concepts to data, architectural, interface, and component-level design. These concepts are considered in the sections that follow

Design Concepts

- A set of fundamental software design concepts has evolved over the history of software engineering
- Fundamental software design concepts
  - Abstraction
  - Architecture
  - Patterns
  - Separation of Concerns
  - Modularity
  - Information hiding
  - Functional independence
  - Refinement
  - Aspects
  - Refactoring
  - Object-Oriented Design Concepts
  - Design Classes

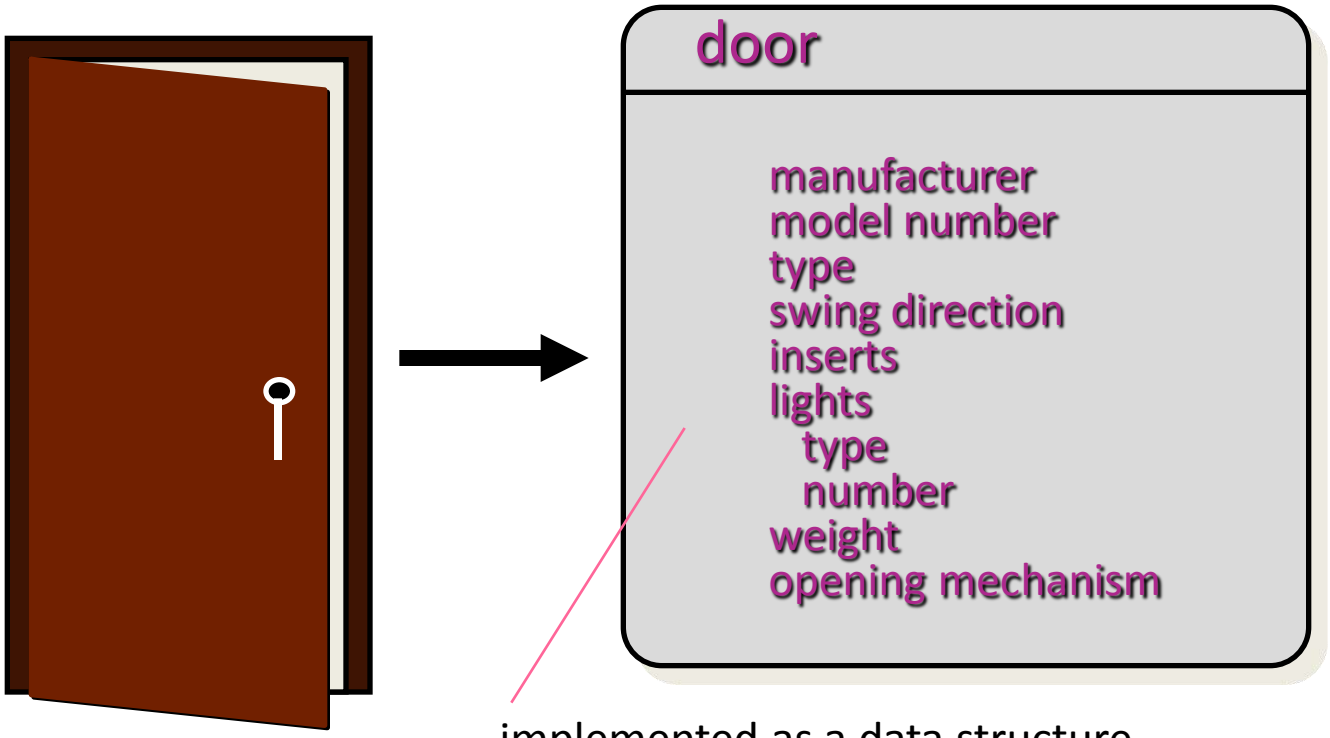
Design Concepts

- **Abstraction**
  - **Abstraction is the process by which data and programs are defined with a representation similar in form to its meaning (semantics), while hiding away the implementation details.**
  - **Abstraction tries to reduce and factor out details so that the programmer can focus on a few concepts at a time**
  - **At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At the lower levels of abstraction, a more detailed description of the solution is provided.**

Design Concepts

- **Abstraction**
  - **Abstraction can be**
    - **Data abstraction is a named collection of data that describes a data object. Data abstraction for ‘door’ would encompass a set of attributes that describe the door (e.g. door type, swing direction, opening mechanism, weight, dimensions).**
    - **The procedural abstraction ‘open’ would make use of information contained in the attributes of the data abstraction ‘door’**

Data Abstraction



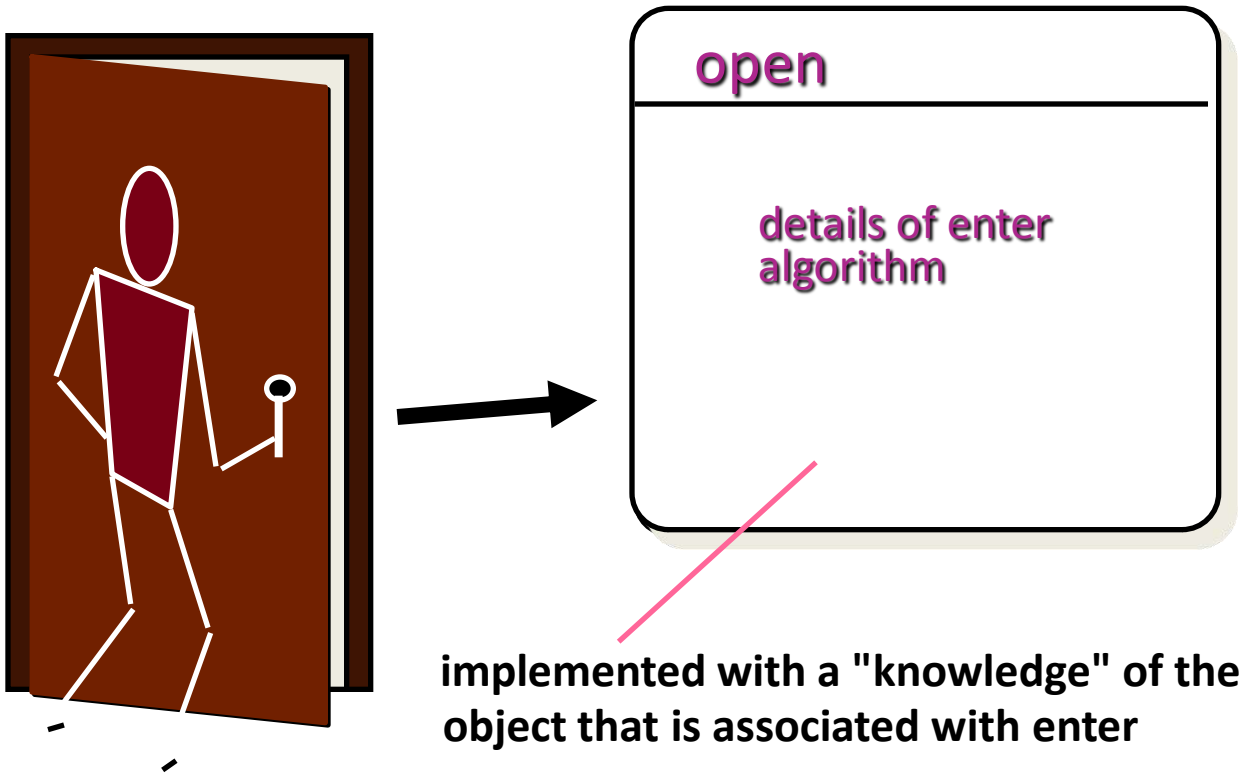
implemented as a data structure

Design Concepts

- **Abstraction**
  - **Procedural abstraction refers to a sequence of instructions that have as specific and limited function. The name of the procedural abstraction implies these functions, but specific details are suppressed.**

    e.g. ‘open’ for a door. ‘open’ implies a long sequence of procedural steps (e.g. walk to the door, reach out and grasp knob, turn knob, turn knob and pull door, step away from moving door, etc)

Procedural Abstraction



Design Concepts

- **Architecture**
  - **Architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components. Architectural design can be represented using**
    - **Structural models**
    - **Framework models**
    - **Dynamic models**
    - **Procedural models**
    - **Function models**

Design Concepts

- **Architecture**
  - **Structural models represent architecture as an organized collection of program components**
  - **Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of application**
  - **Dynamic models address the behavioural aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events**
  - **Procedural models focus on the design of the business or technical process that the system must accommodate**
  - **Function models can be used to represent the functional hierarchy of a system**

Design Concepts

- **Patterns**
  - **Design pattern describes a design structure that solves a particular design problem within a specific context.**
  - **Each design pattern is to provide a description that enables a designer to determine**
    - **Whether the pattern is applicable to the current work**
    - **Whether the pattern can be reused**
    - **Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern**

Design Concepts

Separation of Concerns

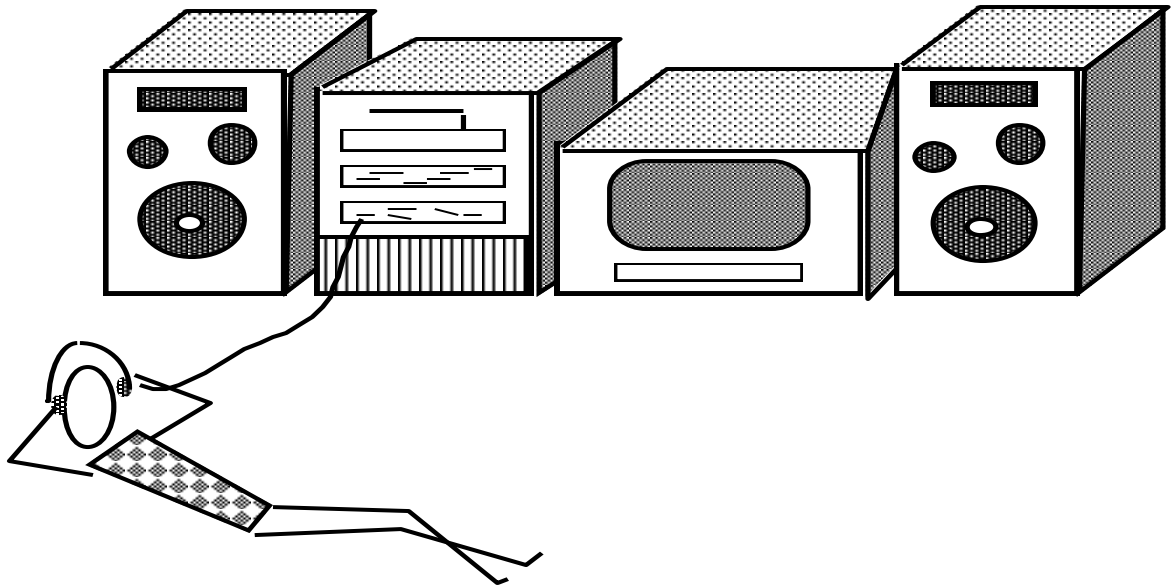
- Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.
- A concern is a feature or behavior that is specified as part of the requirements model for the software.
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.
- For two problems, p1 and p2, if the perceived complexity of p1 is greater than the perceived complexity of p2, it follows that the effort required to solve p1 is greater than the effort required to solve p2.

Design Concepts

- **Modularity**
  - **Software is divided into separately named and addressable components, called modules that are integrated to satisfy problem requirements**
  - **Design has to be modularized so that development can be more easily planned, software increments can be defined and delivered, changes can be more easily accommodated, testing and debugging can be conducted more efficiently and long-term maintenance can be conducted without serious side effects**

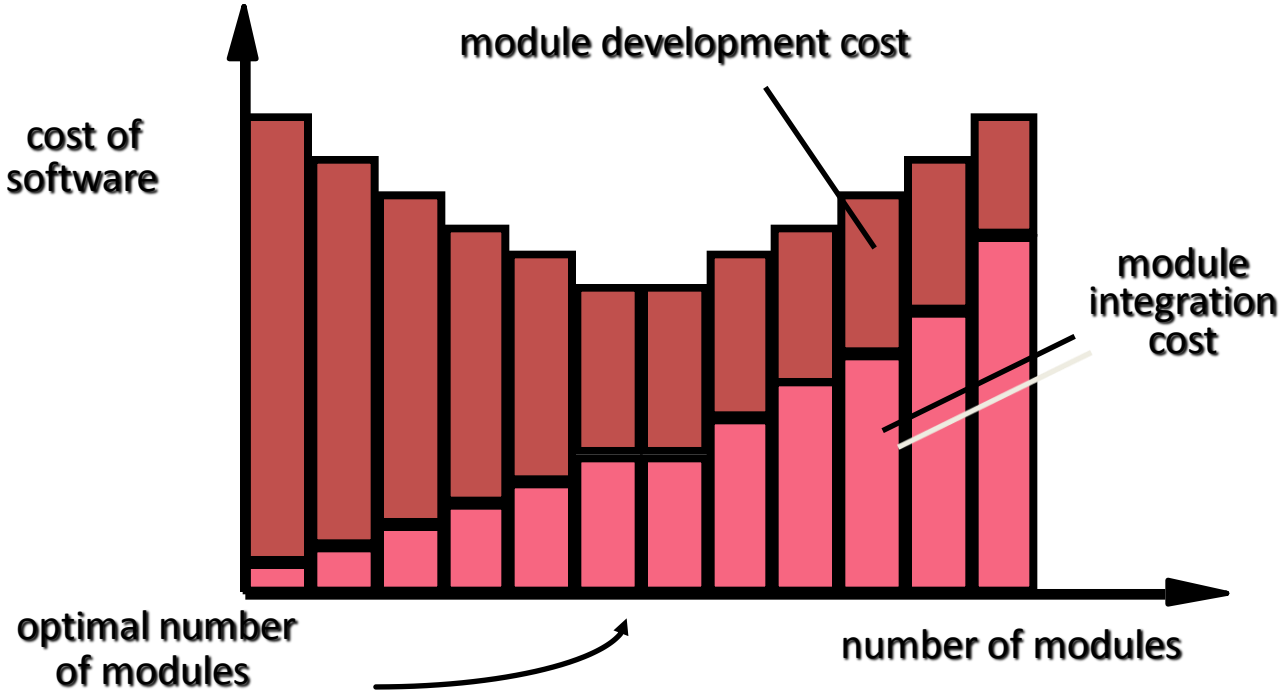
Modular Design

easier to build, easier to change, easier to fix ...



Modularity: Trade-offs

What is the "right" number of modules
for a specific software design?



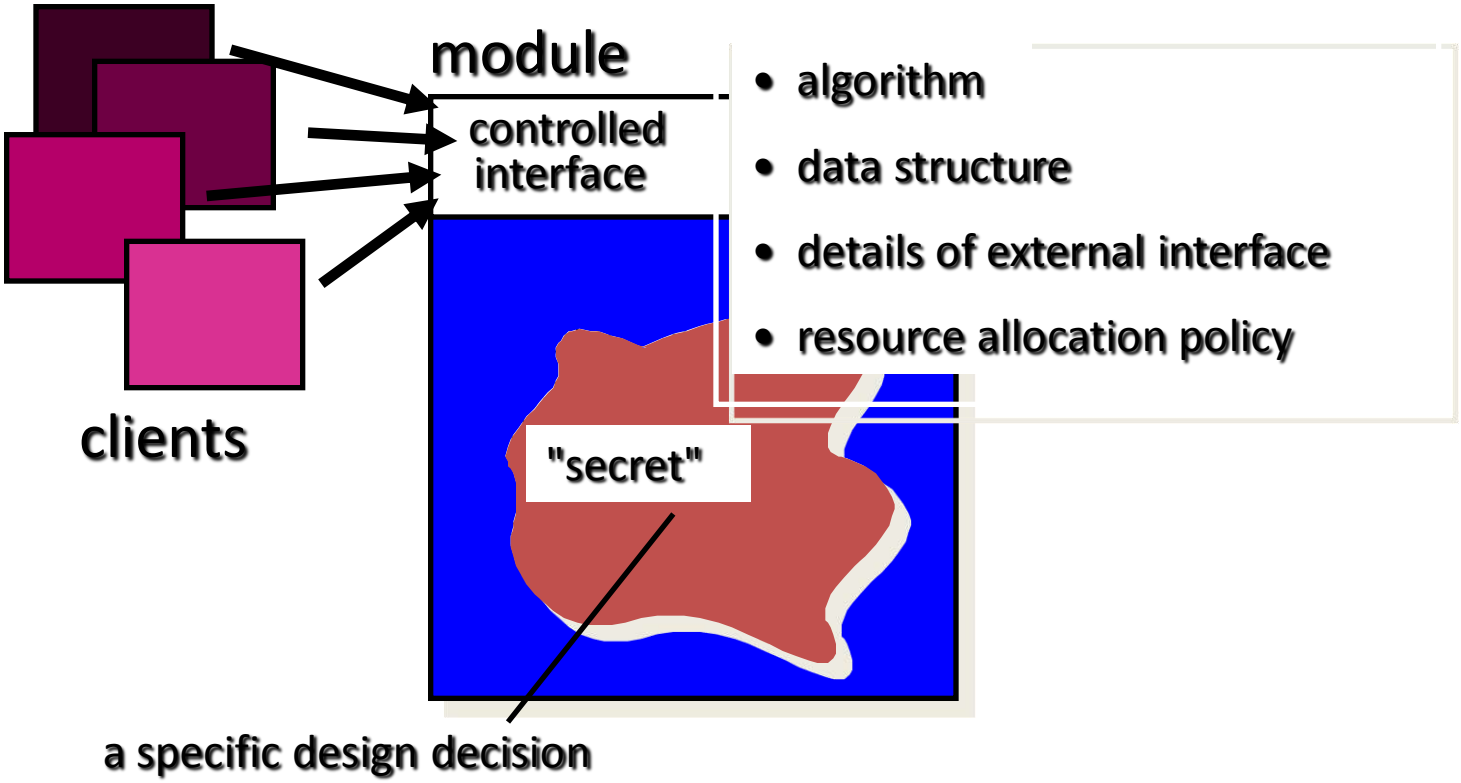
Design Concepts

- **Information Hiding**
  - **Modules should be specified and designed so that information (algorithms and data) contained in a module is inaccessible to other modules that have no need for such information**
  - **Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function**
  - **Hiding defines and enforces access constraints to both data procedural detail within a module and any local data structure used by the module**
  - **The use of information hiding as a design criterion for modular systems provides the benefits when modifications are required during testing and later during software maintenance**

Design Concepts

- Information Hiding
  - Reduces the likelihood of "side effects"
  - Limits the global impact of local design decisions
  - Emphasizes communication through controlled interfaces
  - Discourages the use of global data
  - Leads to encapsulation—an attribute of high quality design
  - Results in higher quality software

Information Hiding



Design Concepts

- **Functional Independence**
  - **Functional independence is achieved by developing modules with 'single minded' function and avoids excessive interaction with other modules.**
  - **Independent modules are easier to maintain because secondary effects caused by design or code modification are limited, error propagation is reduced**

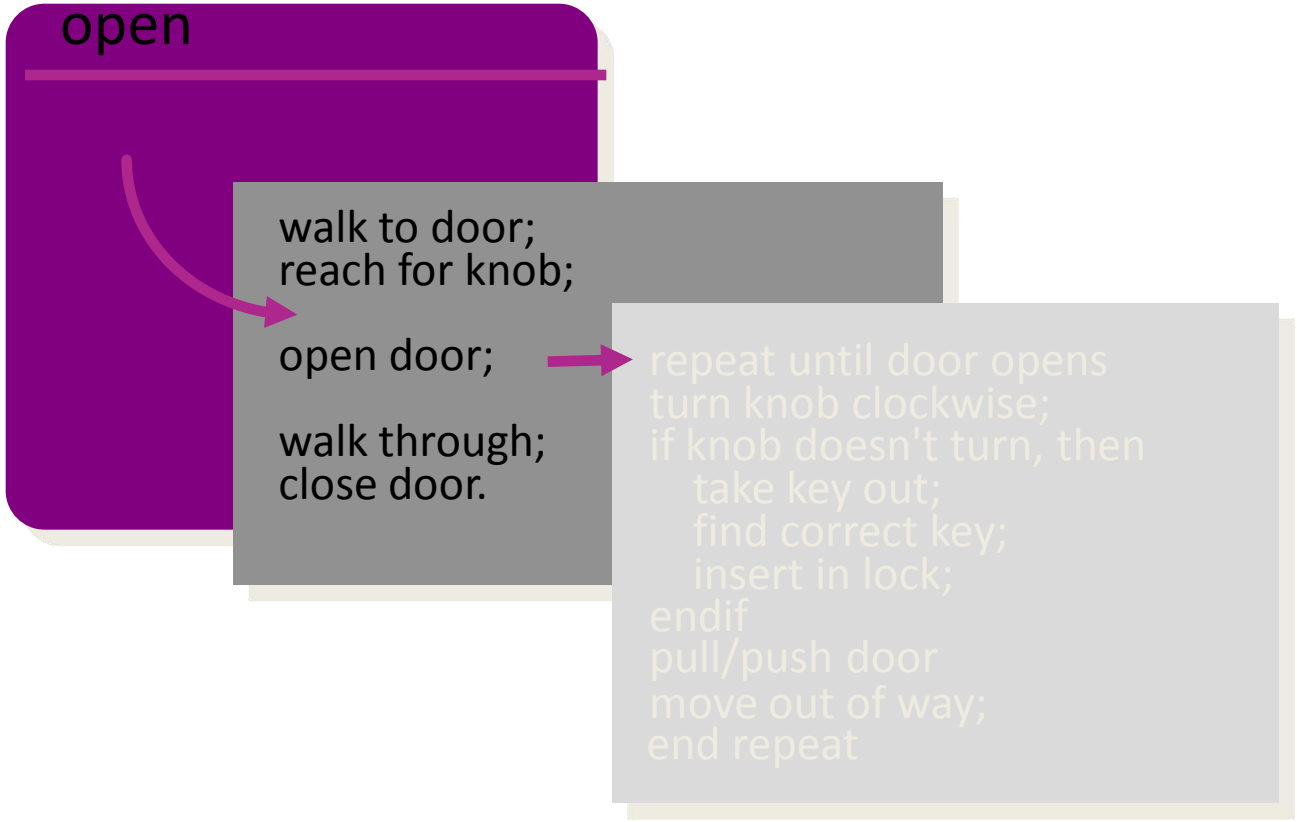
Design Concepts

- **Functional Independence**
  - **Independence is assessed using two qualitative criteria**
    - **Cohesion which is an indication of the relative functional strength of a module. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program**
    - **Coupling is an indication of the relative interdependence among modules. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface**

Design Concepts

- **Refinement**
  - **Refinement is a process of elaboration. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement occurs.**
  - **Refinement helps the designer to reveal low-level details as design progresses.**

Stepwise Refinement



305

Design Concepts

- **Refactoring**
  - **Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure**
  - **When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design**

Design Concepts

- **Aspects**
  - **As requirements analysis occurs, a set of "concerns" is uncovered. These concerns "include requirements, use cases, features, data structures, quality-of-service issues, variants, intellectual property boundaries, collaborations, patterns and contracts"**
  - **Ideally, a requirements model can be organized in a way that allows you to isolate each concern (requirement) so that it can be considered independently.**
  - **In practice, however, some of these concerns span the entire system and cannot be easily compartmentalized. As design begins, requirements are refined into a modular design representation.**
  - **Consider two requirements, A and B. Requirement A crosscuts requirement B "if a software decomposition [refinement] has been chosen in which B cannot be satisfied without taking A into account".**

Design Concepts

- **Refactoring**
  - **An important design activity suggested for many agile methods, refactoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior**
  - **When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design .**

Design Concepts

Object-Oriented Design Concepts

- **The object-oriented (OO) paradigm is widely used in modern software engineering.**
- **OO design concepts such as classes and objects, inheritance, messages, and polymorphism, among others.**

Design Concepts

- **Design classes**
  - **As the design model evolves, a set of design classes are to be defined that**
    - **Refine the analysis classes by providing design detail that will enable the classes to be implemented**
    - **Create a new set of design classes that implement a software infrastructure to support the business solution**

Design Concepts

- **Design classes**
  - **Types of design classes**
    - **User Interface classes define all abstractions that are necessary for human-computer interaction (HCI). Design classes for the interface may be visual representations of the elements of the metaphor**
    - **Business domain classes are refinements of the analysis classes . The classes identify the attributes and services that are required to implement some element of the business domain**
    - **Process classes implement lower-level business abstractions required to fully manage the business domain classes**
    - **Persistent classes represent data stores that will persist beyond the execution of the software**
    - **System classes implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world**

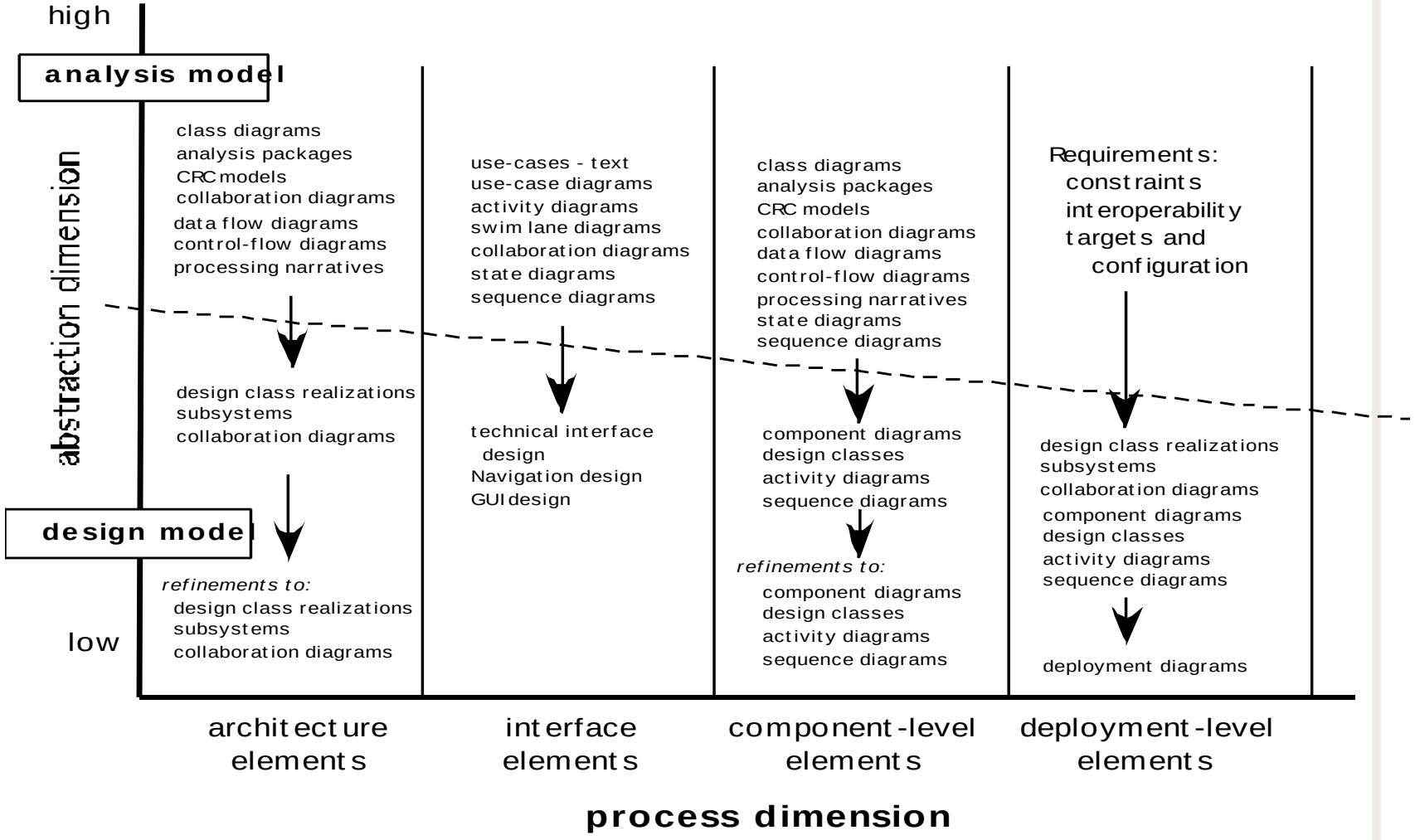
Design Concepts

- **Design classes**
  - **Characteristics of well-formed design classes**
    - **Complete and sufficient – a design class should be the complete encapsulation of all attributes and methods that can reasonably be expected to exist for the class.**
    - **Primitiveness – methods associated with a design class should be focussed on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing.**
    - **High cohesion – a cohesive design class has a small, focussed set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities.**
    - **Low coupling – Within the design model, it is necessary for design classes to collaborate with one another collaboration should be kept to an acceptable minimum.**

Design Model

- Design model can be viewed as
  - Process dimension indicating the evolution of the design model as design tasks are executed as part of the software process.
  - Abstraction dimension represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively
- Design Model Elements are as follows
  - Data design elements
  - Architectural design elements
  - Interface design elements
  - Component-level design elements
  - Deployment-level design elements

The Design Model



Design Model

- **Data design elements**
  - **Data design creates a model of data and/or information that is represented at a high level of abstraction.**
  - **Data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system**
    - **Architectural level → databases and files**
    - **Component level → data structures**

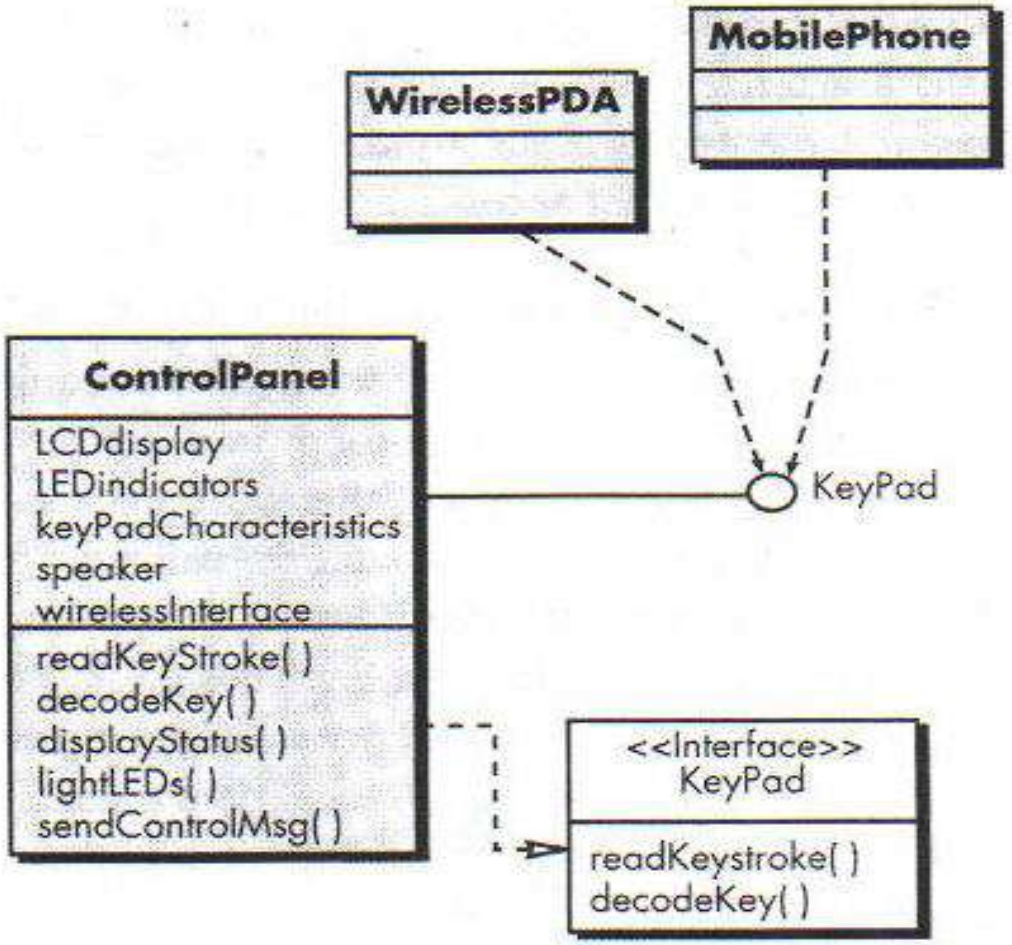
Design Model

- **Architectural design elements**
- **The architectural design for software is the equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows that allow movement into and out of the rooms. The floor plan gives us an overall view of the house. Architectural design elements give us an overall view of the software.**
  - **The architectural model is derived from**
    - **Information about the application domain for the software to be built**
    - **Specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand**
    - **The availability of architectural patterns and styles**

Design Model

- **Interface design elements**
  - **The interface design elements for software tell how information flows into and out of the system and how it is communicated among the components defined as part of the architecture**
  - **Important elements of interface design**
    - **The user interface (UI): Usability design incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components). In general, the UI is a unique subsystem within the overall application architecture.**
    - **External interfaces to other systems, devices, networks or other producers or consumers of informationThe design of external interfaces requires definitive information about the entity to which information is sent or received.**
    - **Internal interfaces between various design componentsThe design of internal interfaces is closely aligned with component-level design**

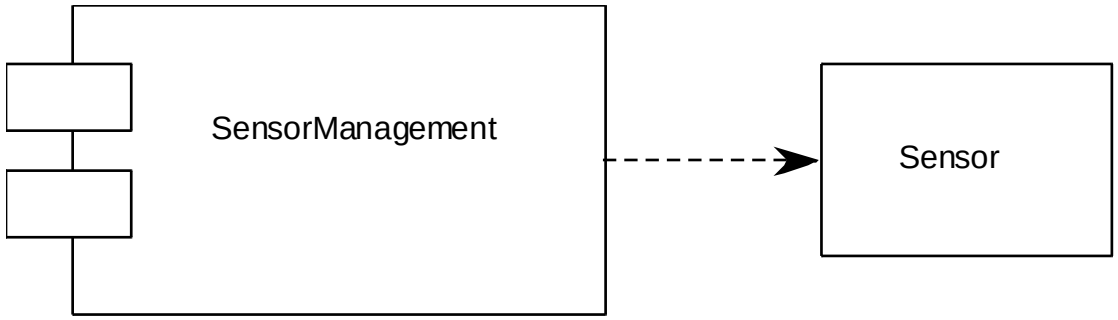
Design Model - Interface Elements



Design Model

- **Component-level design elements**
- **The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches, faucets, sinks, showers, tubs, drains, cabinets, and closets.**
  - **Component-level design for software fully describes the internal detail of each software component.**
  - **Component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations**
  - **The design details of a component can be modelled at many different levels of abstraction.**
  - **An UML activity diagram can be used to represent processing logic. Detailed procedural flow for a component can be represented using either pseudocode or diagrammatic form (e.g., flowchart or box diagram.**

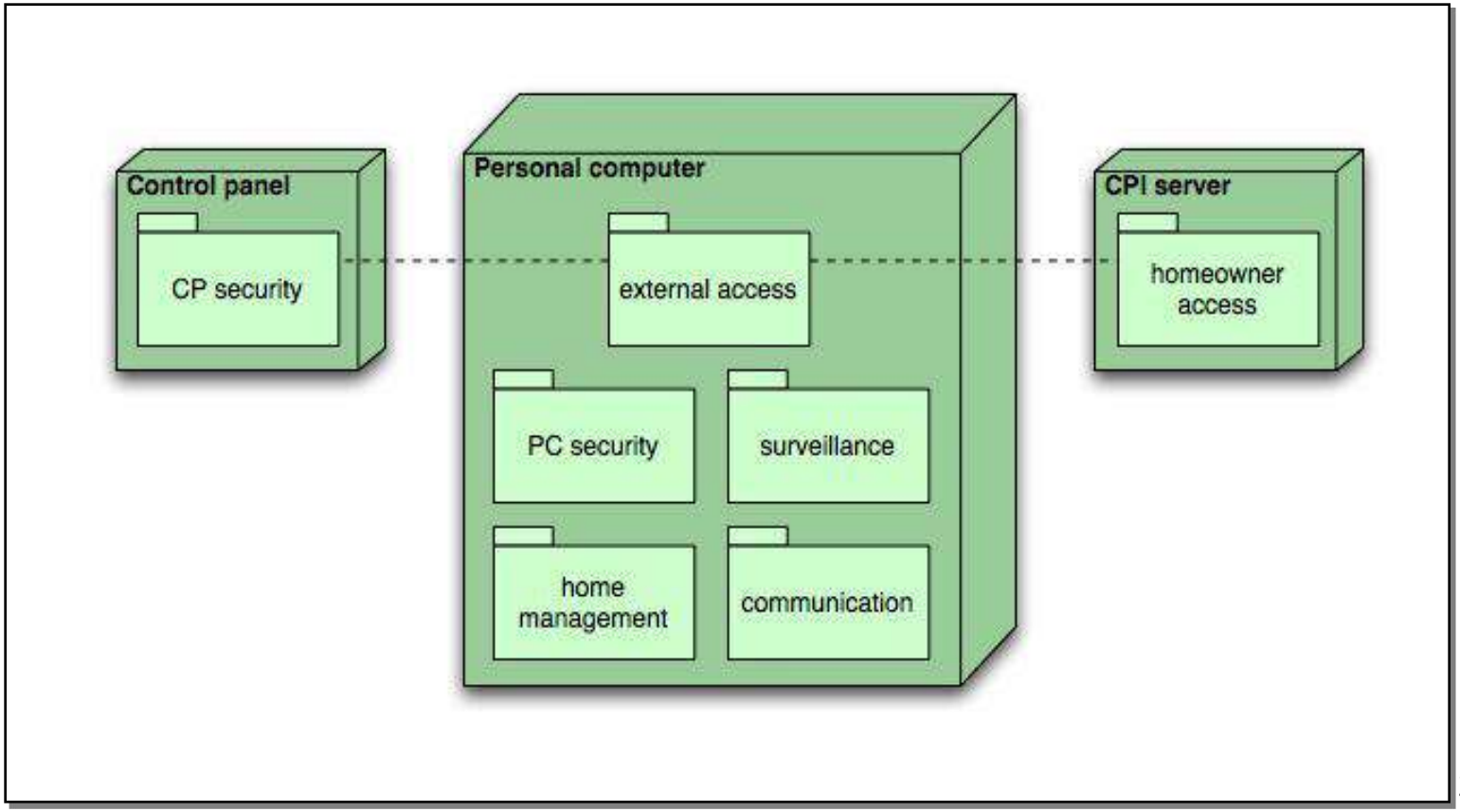
Component Elements



Design Model

- **Deployment-level design elements**
  - **Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.**
  - **Deployment diagrams shows the computing environment but does not explicitly indicate configuration details**

Deployment Diagram



322

Creating an architectural design

Software Architecture

What is software Architecture

- When you consider the architecture of a building, many different attributes come to mind. At the most simplistic level, you think about the overall shape of the physical structure. But in reality, architecture is much more. It is the manner in which the various components of the building are integrated to form a cohesive whole.
- The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them
- Software architecture enables to
  - Analyze the effectiveness of the design in meeting its stated requirements
  - Consider architectural alternatives at a stage when making design changes is still relatively easy
  - Reduce the risks associated with the construction of the software

Software Architecture

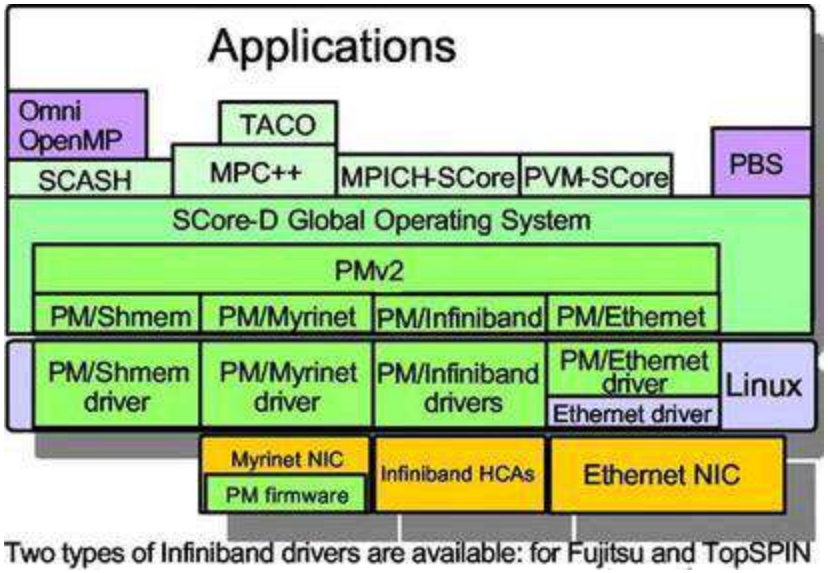
- **Architectural design represents the structure of data and program components that are required to build a computer-based system.**
- **It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system**
- **Architecture considers two levels of the design – data design and architectural design. Data design enables us to represent the data component of the architecture.**
- **Architectural design focuses on the representation of the structure of software components, their properties, and interactions**

Software Architecture

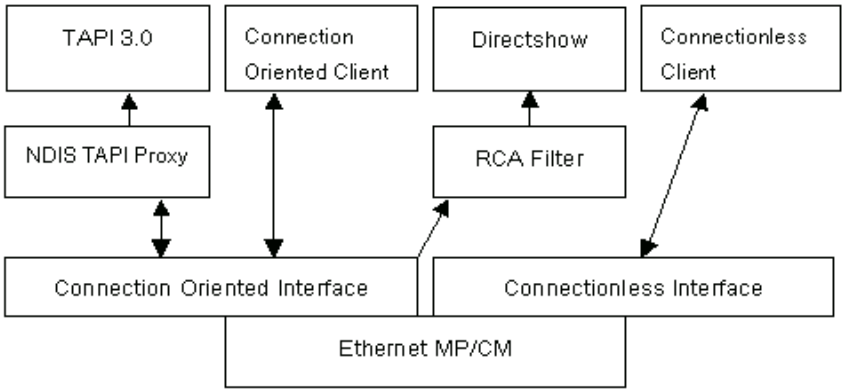
Why Is Architecture Important?

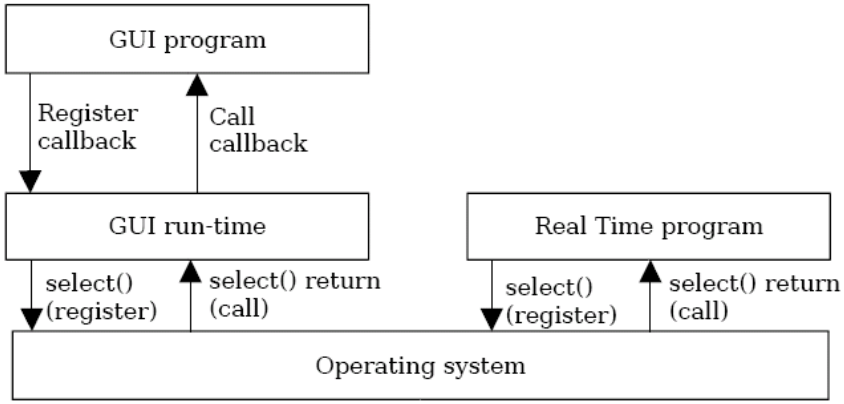
- **Representations of software architecture are an enabler for communication between all parties, interested in the development of a computer-based system.**
- **The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on ultimate success of the system as an operational entity.**
- **Architecture constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together**

Example Software Architecture Diagrams

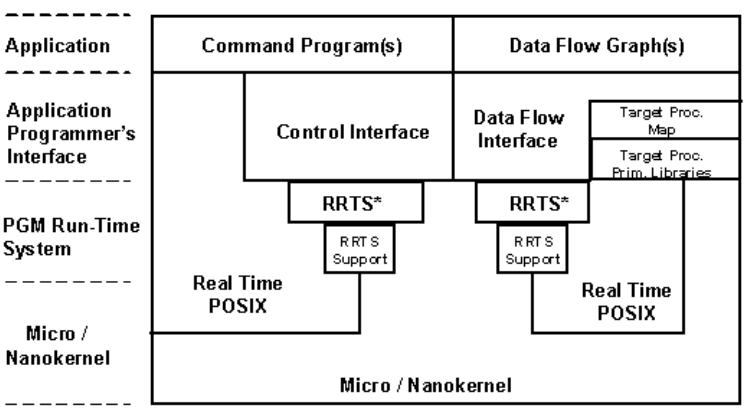


Two types of Infiniband drivers are available: for Fujitsu and TopSPIN





Software Architecture Diagram



*RRTS: RASSP Run-Time Support

Software Architecture

Architectural Descriptions

- Each of us has a mental image of what the word architecture means. In reality, however, it means different things to different people.
- The implication is that different stakeholders will see an architecture from different viewpoints that are driven by different sets of concerns.
- An architectural description is actually a set of work products that reflect different views of the system.
- An architectural description of a software-based system must exhibit characteristics that are analogous to those noted for the office building.
- Developers want clear, decisive guidance on how to proceed with design.
- Customers want a clear understanding on the environmental changes that must occur and assurances that the architecture will meet their business needs.
- Other architects want a clear, salient understanding of the architecture's key aspects." Each of these "wants" is reflected in a different view represented using a different viewpoint.

Software Architecture

Architectural Decisions

- Each view developed as part of an architectural description addresses a specific stakeholder concern.
- To develop each view (and the architectural description as a whole) the system architect considers a variety of alternatives and ultimately decides on the specific architectural features that best meet the concern.
- Therefore, architectural decisions themselves can be considered to be one view of the architecture.
- The reasons that decisions were made provide insight into the structure of a system and its conformance to stakeholder concerns.

Software Architectural Styles

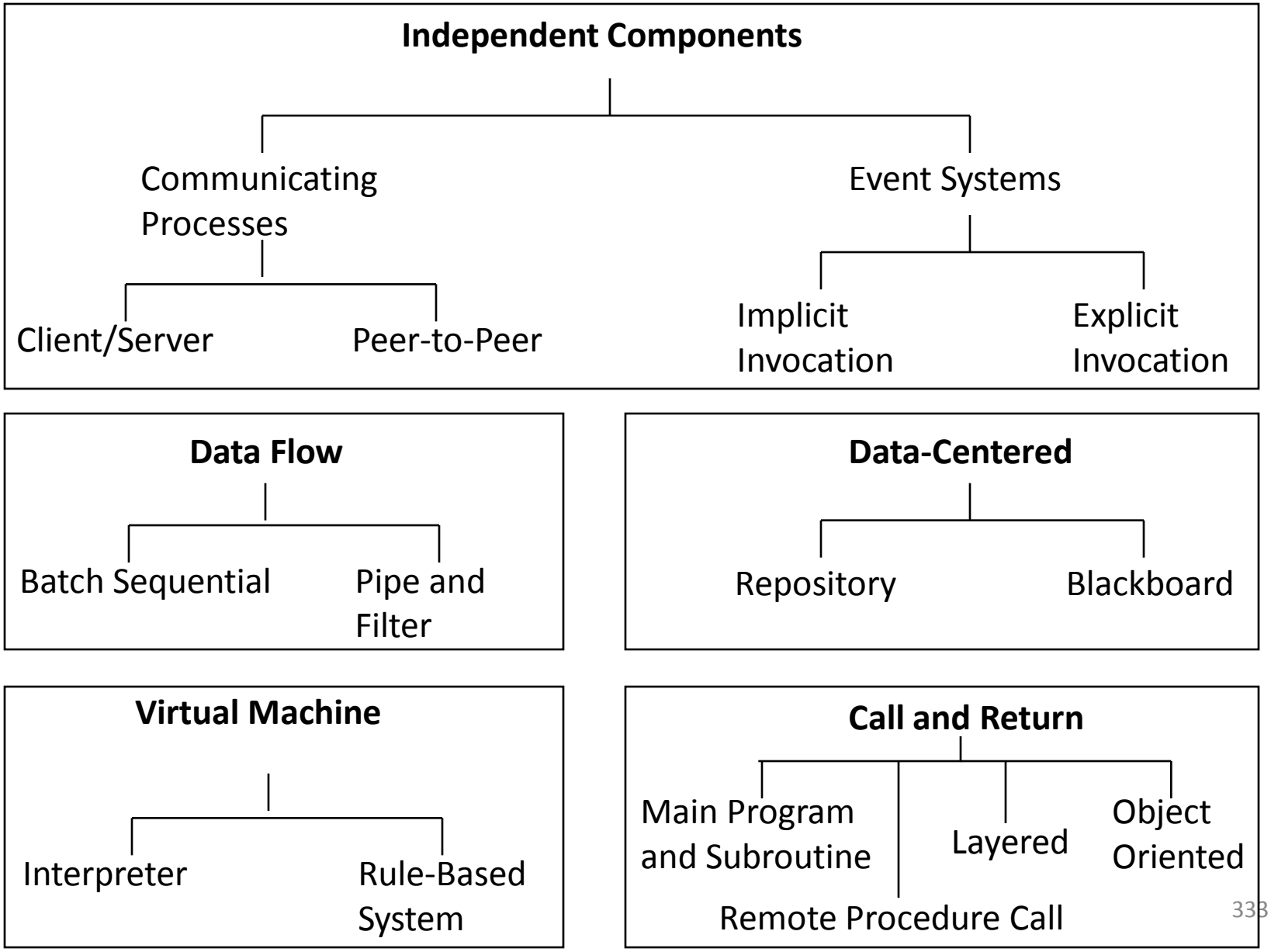
1. A Brief Taxonomy of Architectural Styles
2. Architectural Patterns
3. Organization and Refinement

Common Architectural Styles of Homes

Software Architectural Style

- **The software that is built for computer-based systems exhibit one of many architectural styles**
- **Each style describes a system category that encompasses**
  - **A set of component types that perform a function required by the system**
  - **A set of connectors (subroutine call, remote procedure call, data stream, socket) that enable communication, coordination, and cooperation among components**
  - **constraints that define how components can be integrated to form the system;**
  - **semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts**

A brief Taxonomy of Architectural Styles

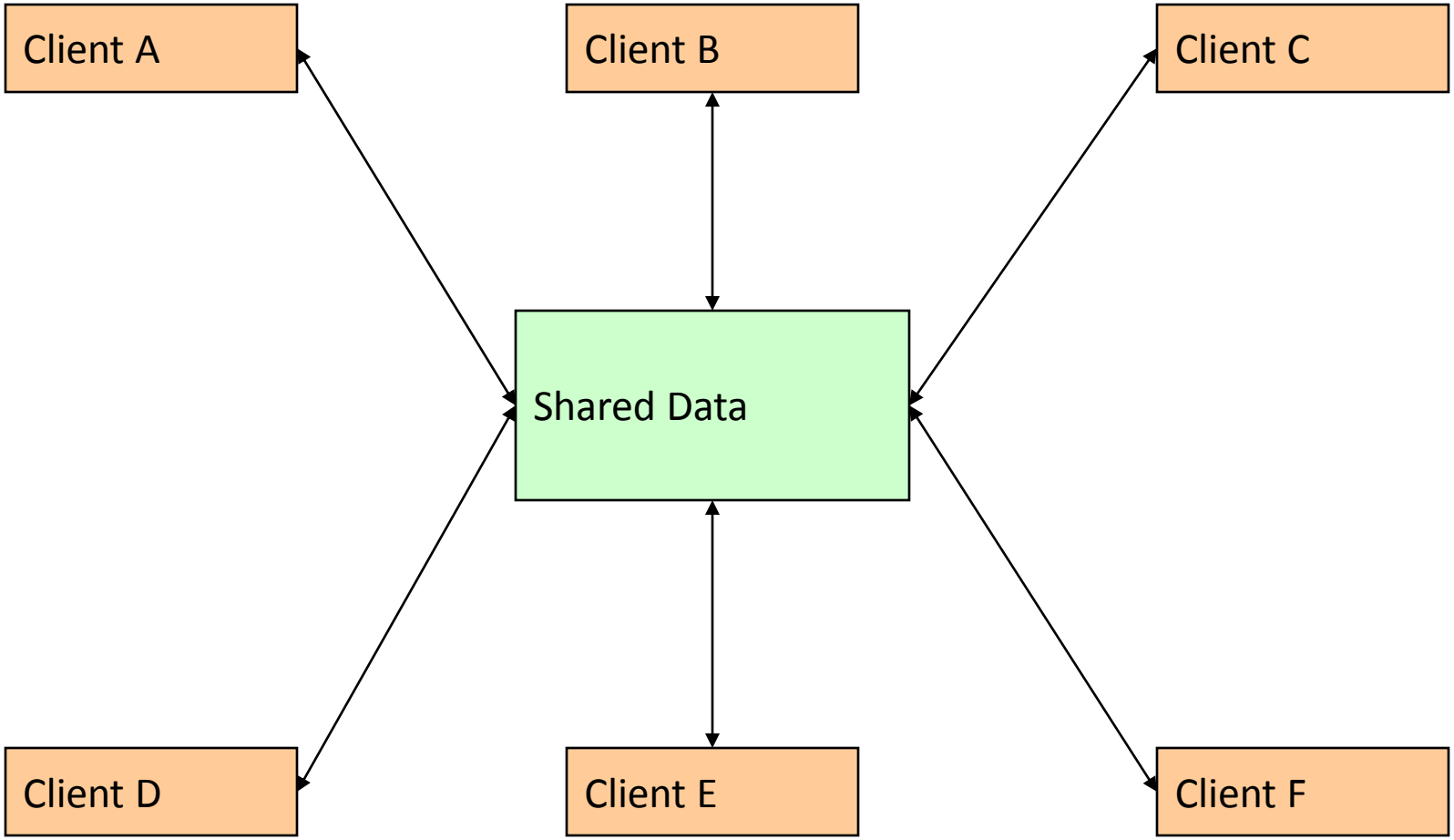


Software Architectural Style

A Brief Taxonomy of Architectural Styles

- Data-centered architectures. A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store.
- illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a “blackboard

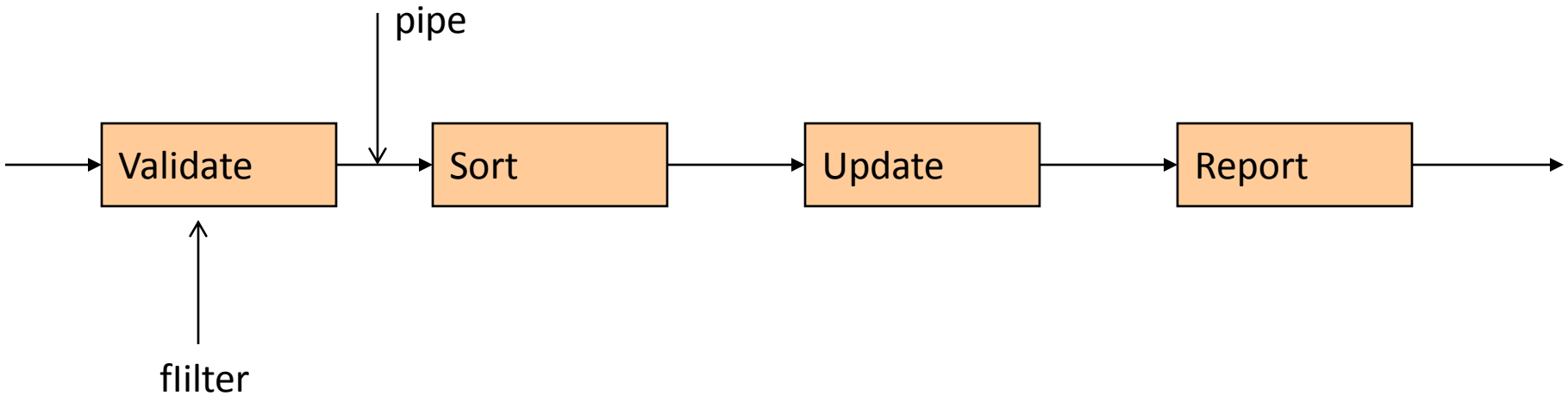
Data-Centered Style



Software Architectural Style

- **Data-flow architectures.**
- **This architecture is applied when input data are to be**
- transformed through a series of computational or manipulative components into output data.
- A pipe-and-filter pattern shows has a set of components, called *filters, connected by pipes that transmit data from one component to the next.*
- *Each* filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form.
- However, the filter does not require knowledge of the workings of its neighboring filters.

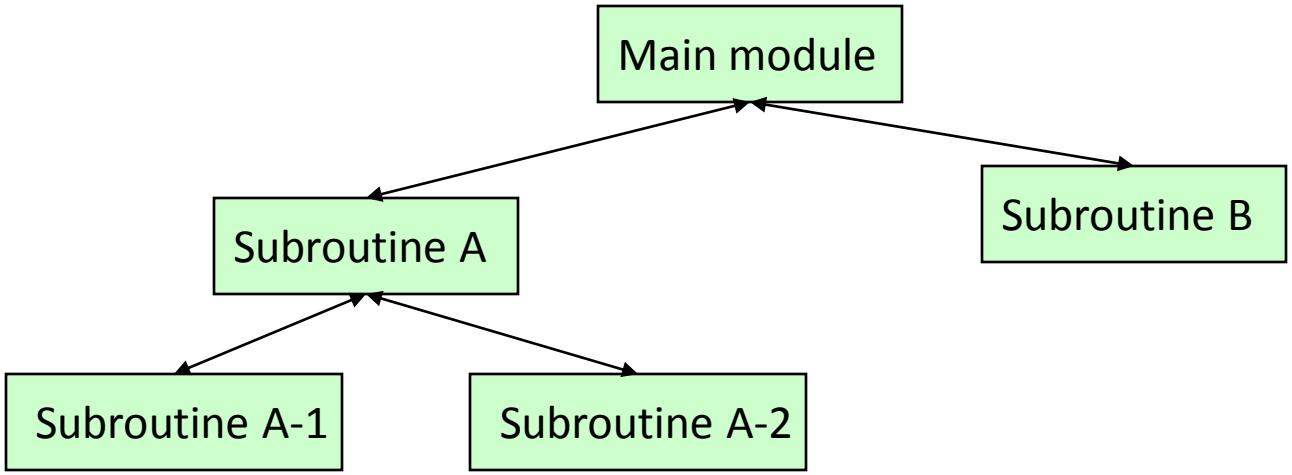
Data Flow Style

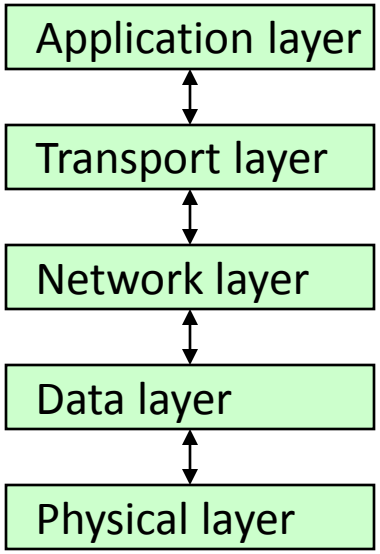


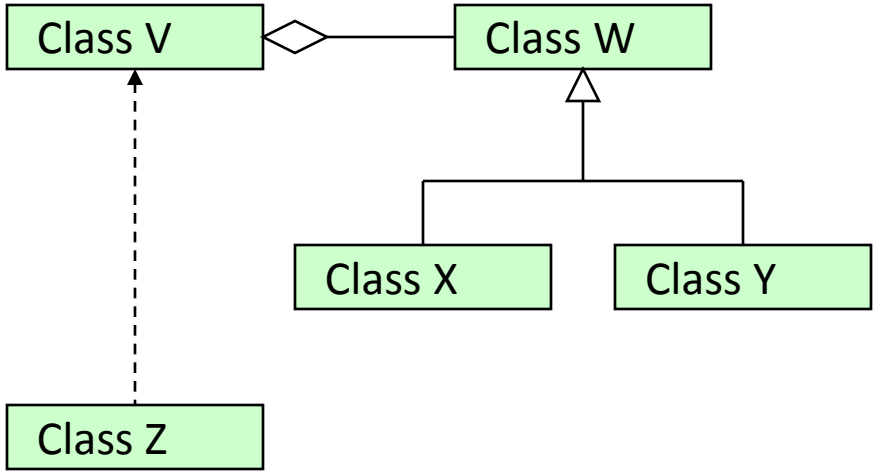
Software Architectural Style

- **Call and return architectures.**
- This architectural style enables you to achieve a program structure that is relatively easy to modify and scale.
- A number of substyles exist within this category:
- *Main program/subprogram architectures: This classic program structure* decomposes function into a control hierarchy where a “main” program invokes a number of program components that in turn may invoke still other components. Figure illustrates an architecture of this type.
- *Remote procedure call architectures: The components of a main* program/subprogram architecture are distributed across multiple computers on a network.

Call-and-Return Style







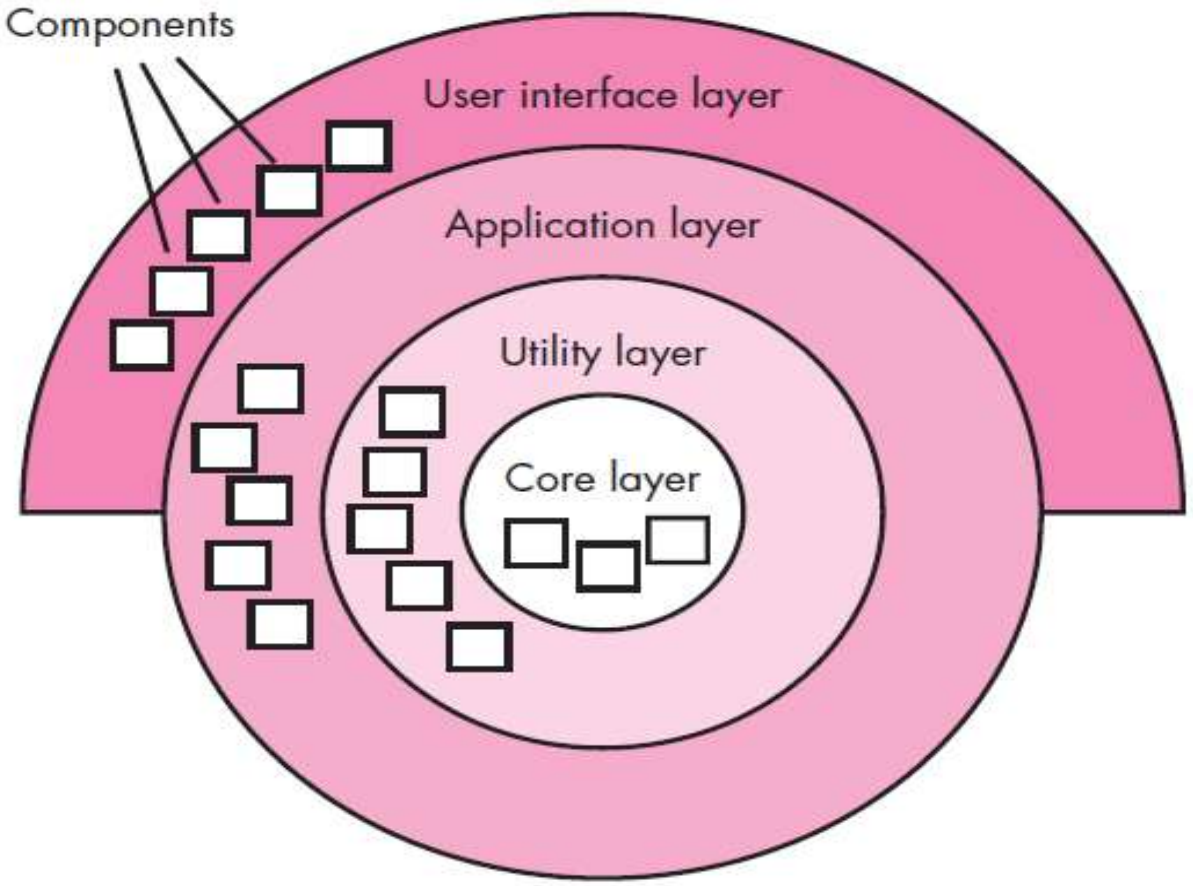
Software Architectural Style

- **Object-oriented architectures.**
- The components of a system encapsulate data and the operations that must be applied to manipulate the data.
- Communication and coordination between components are accomplished via message passing.

Software Architectural Style

- **Layered architectures.**
- The basic structure of a layered architecture is illustrated in Figure.
- A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
- At the outer layer, components service user interface operations.
- At the inner layer, components perform operating system interfacing.
- Intermediate layers provide utility services and application software functions.

Layered architectures



Architectural Patterns

- As the requirements model is developed, you'll notice that the software must address a number of broad problems that span the entire application.

- For example, the requirements model for virtually every e-commerce application is faced with the following problem: *How do we offer a broad array of goods to a broad array of customers and allow those customers to purchase our goods online?*

- Architectural patterns address an application-specific problem within a specific context and under a set of limitations and constraints. The pattern proposes an architectural solution that can serve as the basis for architectural design.

Software Architectural Style

- **Organization and Refinement**
- Because the design process often leaves you with a number of architectural alternatives, it is important to establish a set of design criteria that can be used to assess an architectural design that is derived.
- **Control.**
- How is control managed within the architecture? Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy? How do components transfer control within the system? How is control shared among components? What is the control topology (i.e., the geometric form that the control takes)? Is control synchronized or do components operate asynchronously?
- **Data.**
- How are data communicated between components? Is the flow of data continuous, or are data objects passed to the system sporadically? What is the mode of data transfer (i.e., are data passed from one component to another or are data available globally to be shared among system components)? Do data components (e.g., a blackboard or repository) exist, and if so, what is their role? How do functional components interact with data components? Are data components passive or active (i.e., does the data component actively interact with other components in the system)? How do data and control interact within the system?

Architectural Design Process

Architectural Design Steps

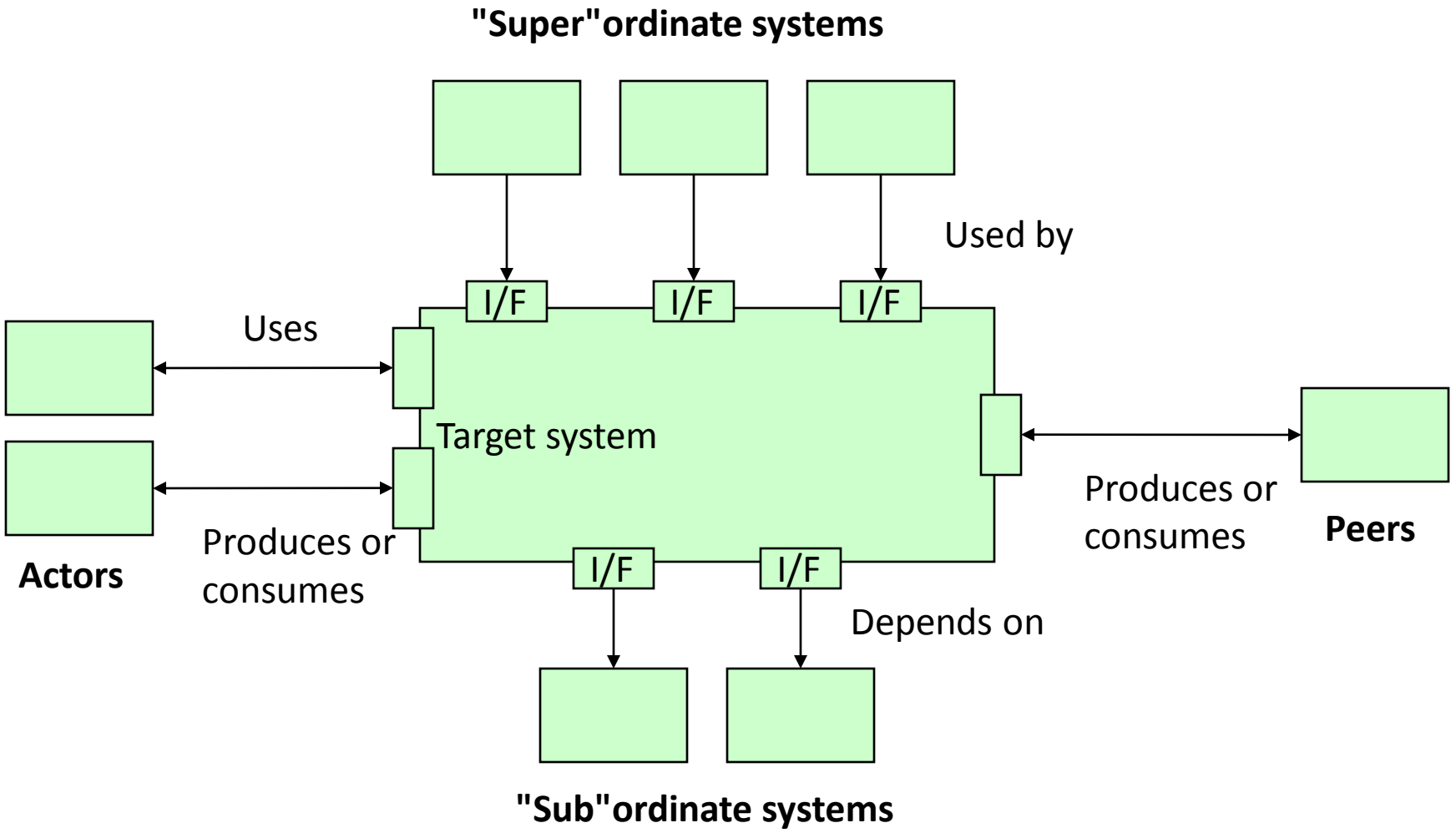
- As architectural design begins, the software to be developed must be put into context—that is, the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction.

1. **Represent the system in context**
2. **Define archetypes**
3. **Refine the architecture into components**
4. **Describe instantiations of the system**

Represent the System in Context

- Use an architectural context diagram (ACD) that shows
  - The identification and flow of all information into and out of a system
  - The specification of all interfaces
  - Any relevant support processing from/by other systems
- An ACD models the manner in which software interacts with entities external to its boundaries

Represent the System in Context



Represent the System in Context

- An ACD identifies systems that interoperate with the target system
  - Super-ordinate systems
    - Use target system as part of some higher level processing scheme
  - Sub-ordinate systems
    - those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality
  - Peer-level systems
    - Interact on a peer-to-peer basis with target system to produced or consumed by peers and target system
  - Actors
    - People or devices that interact with target system to produce or consume data

Define Archetypes

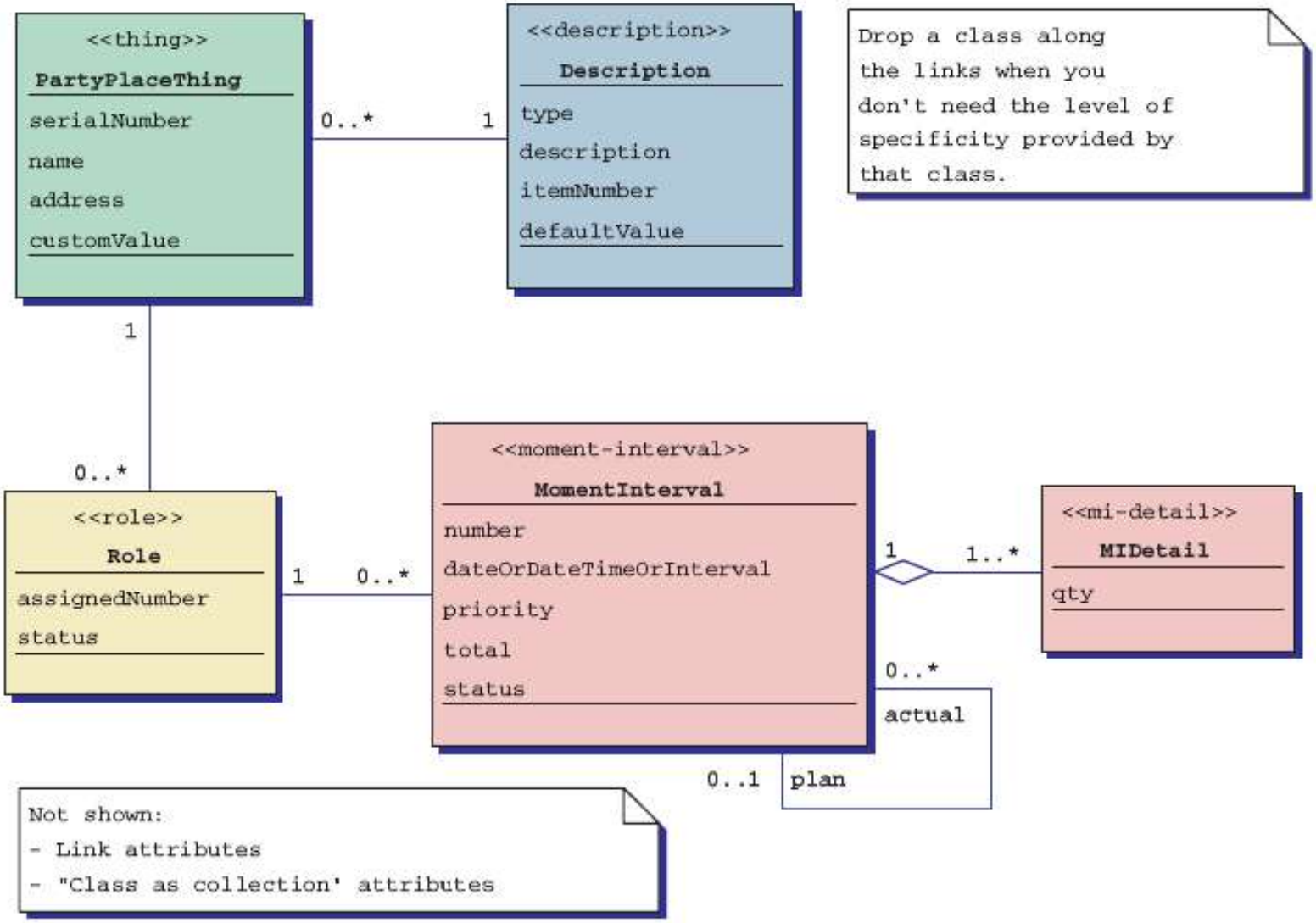
- Archetypes indicate the important abstractions within the problem domain (i.e., they model information)
- An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system
- Only a relatively small set of archetypes is required in order to design even relatively complex systems
- The target system architecture is composed of these archetypes
  - They represent stable elements of the architecture
  - They may be instantiated in different ways based on the behavior of the system
  - They can be derived from the analysis class model
- The archetypes and their relationships can be illustrated in a UML class diagram

Define Archetypes

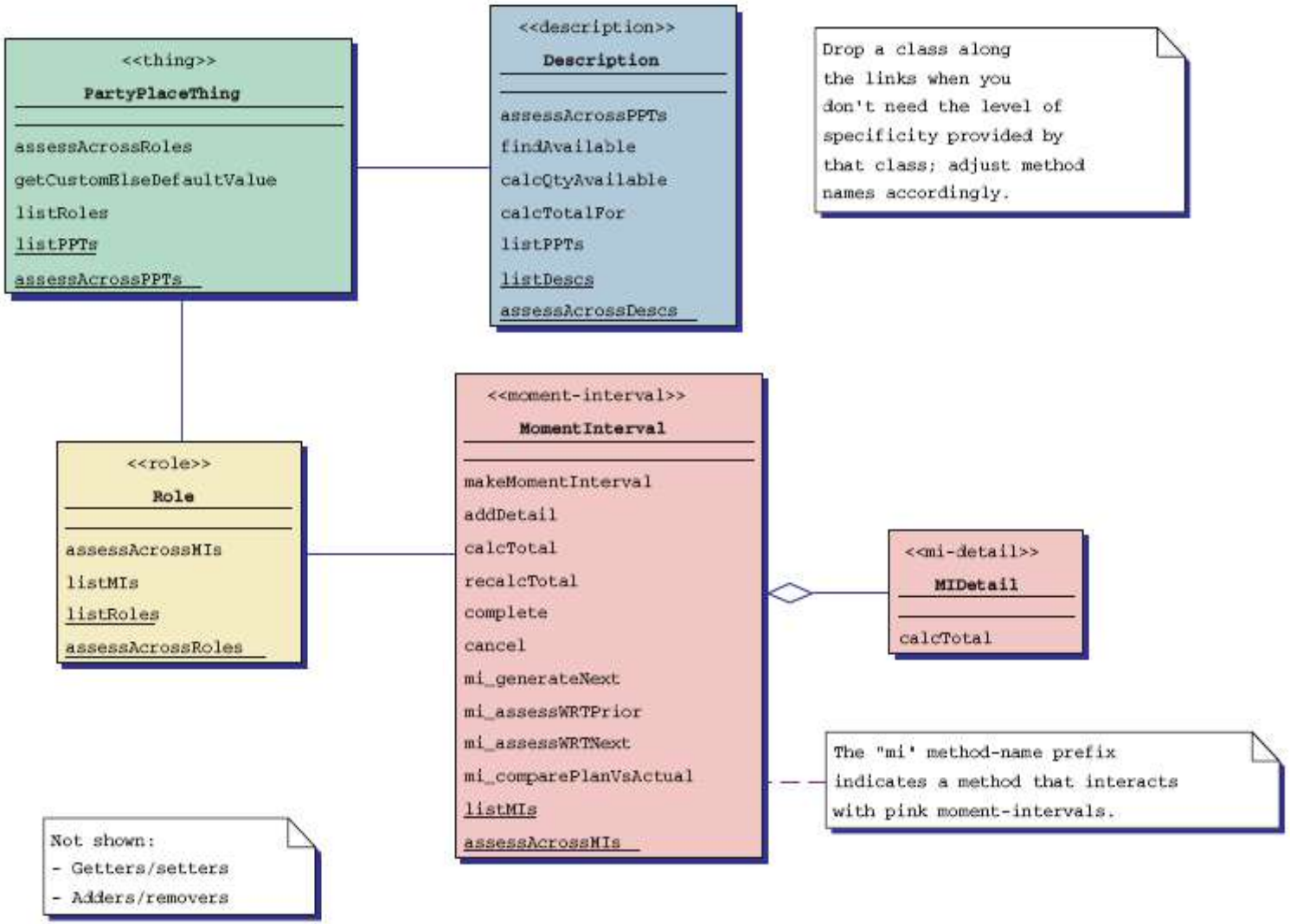
- **Archetypes in Software Architecture**

- **Node -** Represents a cohesive collection of input and output elements of the home security function
- **Detector/Sensor -** An abstraction that encompasses all sensing equipment that feeds information into the target system.
- **Indicator -** An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.
- **Controller -** An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

Archetypes – their attributes



Archetypes – their methods



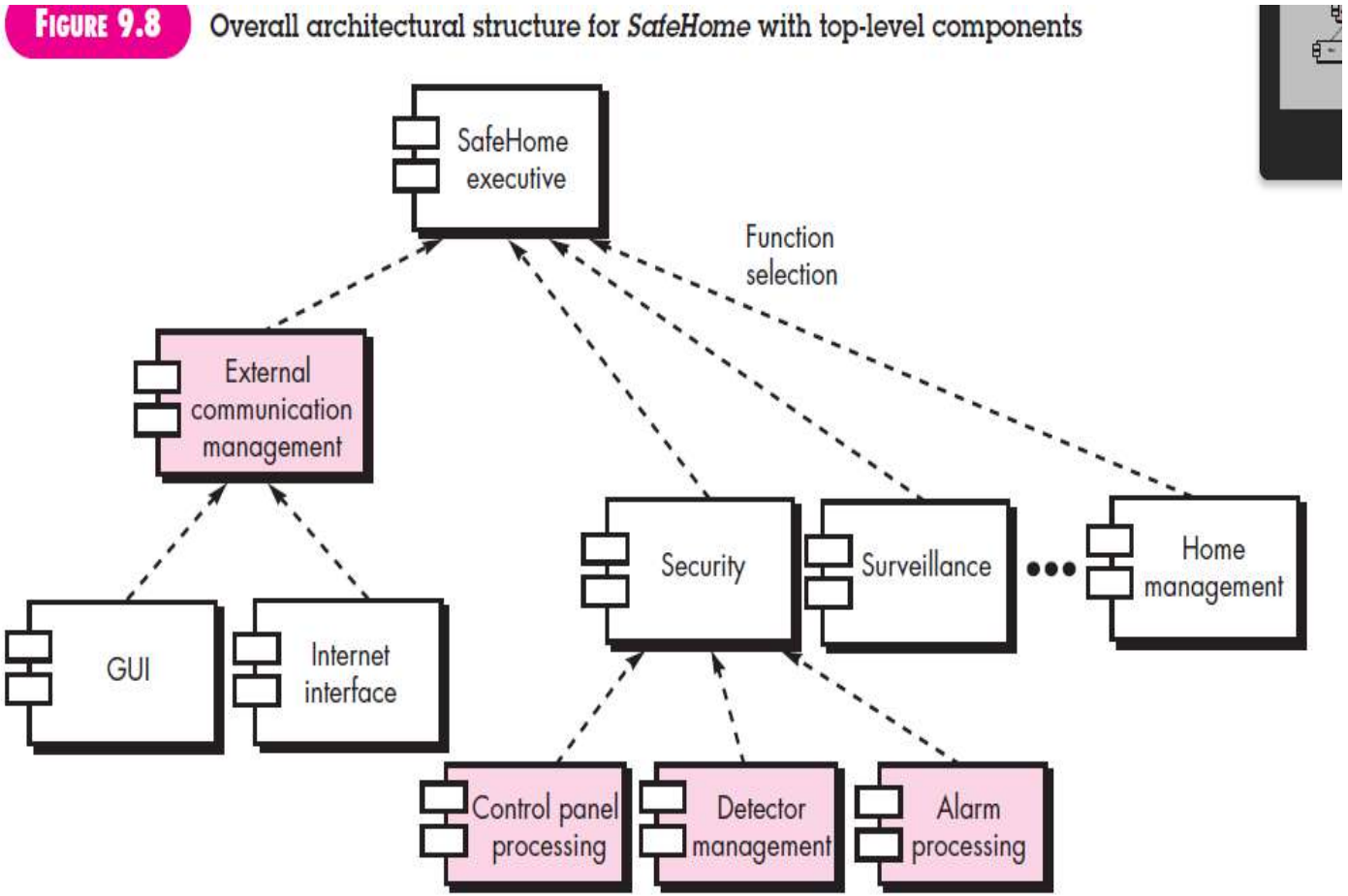
Refine the Architecture into Components

- Based on the archetypes, the architectural designer refines the software architecture into components to illustrate the overall structure and architectural style of the system

- These components are derived from various sources
  - The application domain provides application components, which are the domain classes in the analysis model that represent entities in the real world
  - The infrastructure domain provides design components (i.e., design classes) that enable application components but have no business connection
    - Examples: memory management, communication, database, and task management
- These components are derived from various sources
  - The interfaces in the ACD imply one or more specialized components that process the data that flow across the interface

Refine the Architecture into Components

- A UML class diagram can represent the classes of the refined architecture and their relationships



Describe Instantiations of the System

- The architectural design that has been modeled to this point is still relatively high level.
- The context of the system has been represented, archetypes that indicate the important abstractions within the problem domain have been defined, the overall structure of the system is apparent, and the major software components have been identified.
- However, further refinement (recall that all design is iterative) is still necessary.

Assessing alternative architectural design

- Design results in a number of architectural alternatives that are each assessed to determine which is the most appropriate for the problem to be solved.

1. An Architecture Trade-Off Analysis Method

The Software Engineering Institute (SEI) has developed an architecture trade-off analysis method that establishes an iterative evaluation process for software architectures. The design analysis activities that follow are performed iteratively:

- Collect scenarios**.** A set of use cases is developed to represent the system from the user's point of view.
- Elicit requirements, constraints, and environment description. This information is determined as part of requirements engineering and is used to be certain that all stakeholder concerns have been addressed.
- Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements.
- Evaluate quality attributes by considering each attribute in isolation.
- Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.
- Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted

Assessing alternative architectural design

2. Architectural Complexity

A useful technique for assessing the overall complexity of a proposed architecture is to consider dependencies between components within the architecture. These dependencies are driven by information/control flow within the system.

3. Architectural Description Languages

- Architectural description language (ADL) provides a semantics and syntax for describing a software architecture.
- ADL should provide the designer with the ability to decompose architectural components, compose individual components into larger architectural blocks, and represent interfaces (connection mechanisms) between components.
- Once descriptive, language based techniques for architectural design have been established, it is more likely that effective assessment methods for architectures will be established as the design evolves.

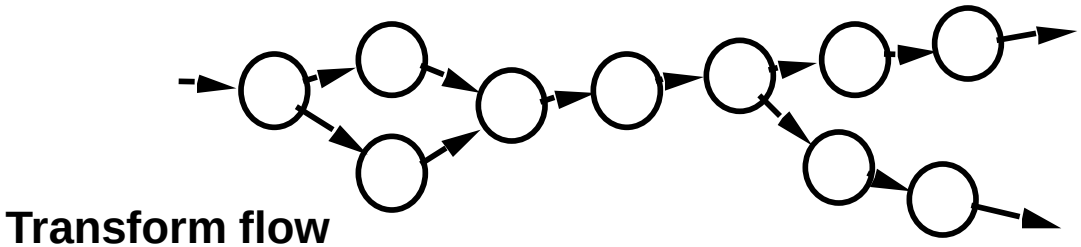
Architectural Mapping using Data Flow

- Transform Mapping
- Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style.
  - Information must enter and exit software in an “external world”. Such externalized data must be converted into an internal form for processing. Information enters along paths that transform external data into an internal form. These paths are identified are *Incoming flow*.
  - Incoming data are transformed through a transform center and move along the paths that now lead “out” of the software. Data moving along these paths are called *Outgoing flow*

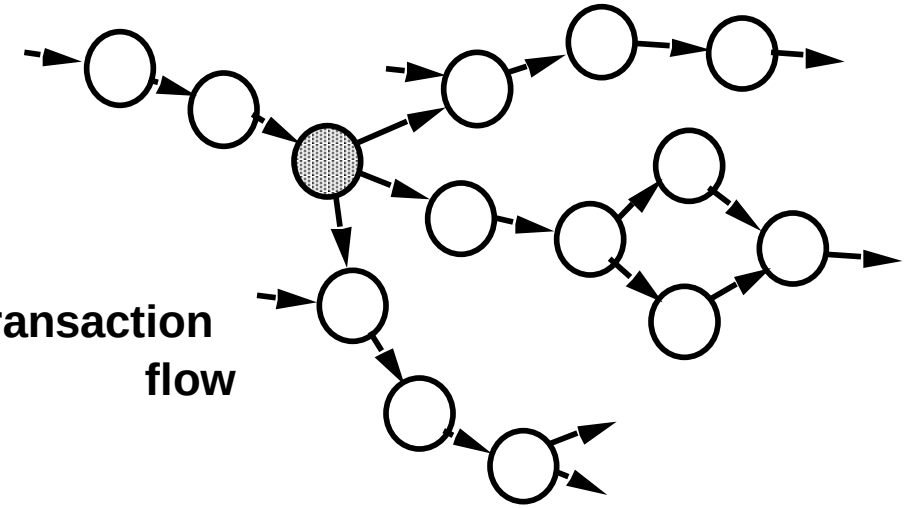
Mapping Data Flow to Architecture

- **Transaction Flow**
  - Information flow is often characterized by a single data item, called *transaction*, that triggers other data flow along one of many paths
  - Transaction flow is characterized by data moving along an incoming path that converts external world information into a transaction
  - The transaction is evaluated and, based on its value, flow along one of many action paths is initiated. The hub of information from which many action paths emanate is called a *transaction center*

Flow Characteristics



Transform flow

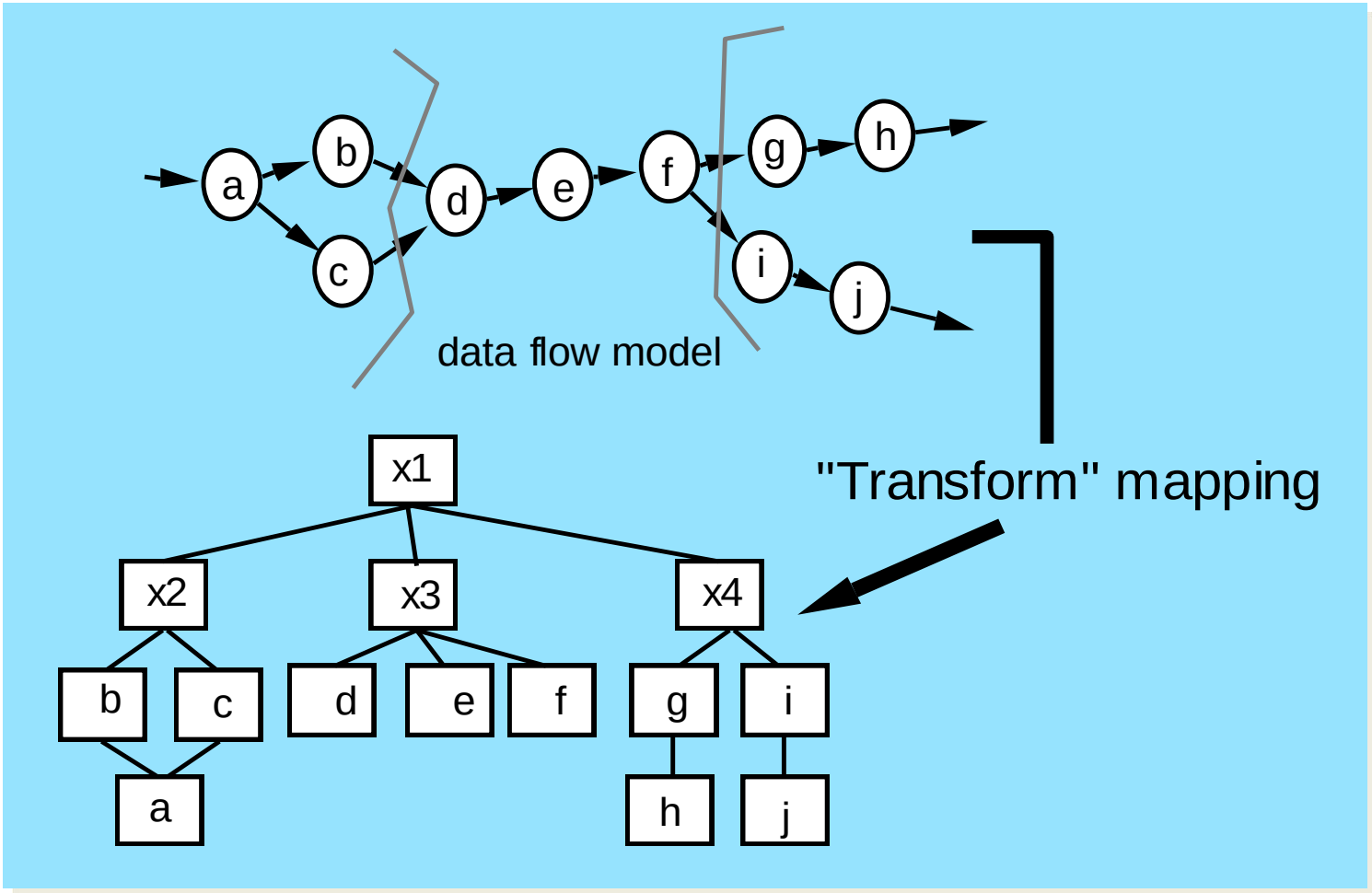


Transaction flow

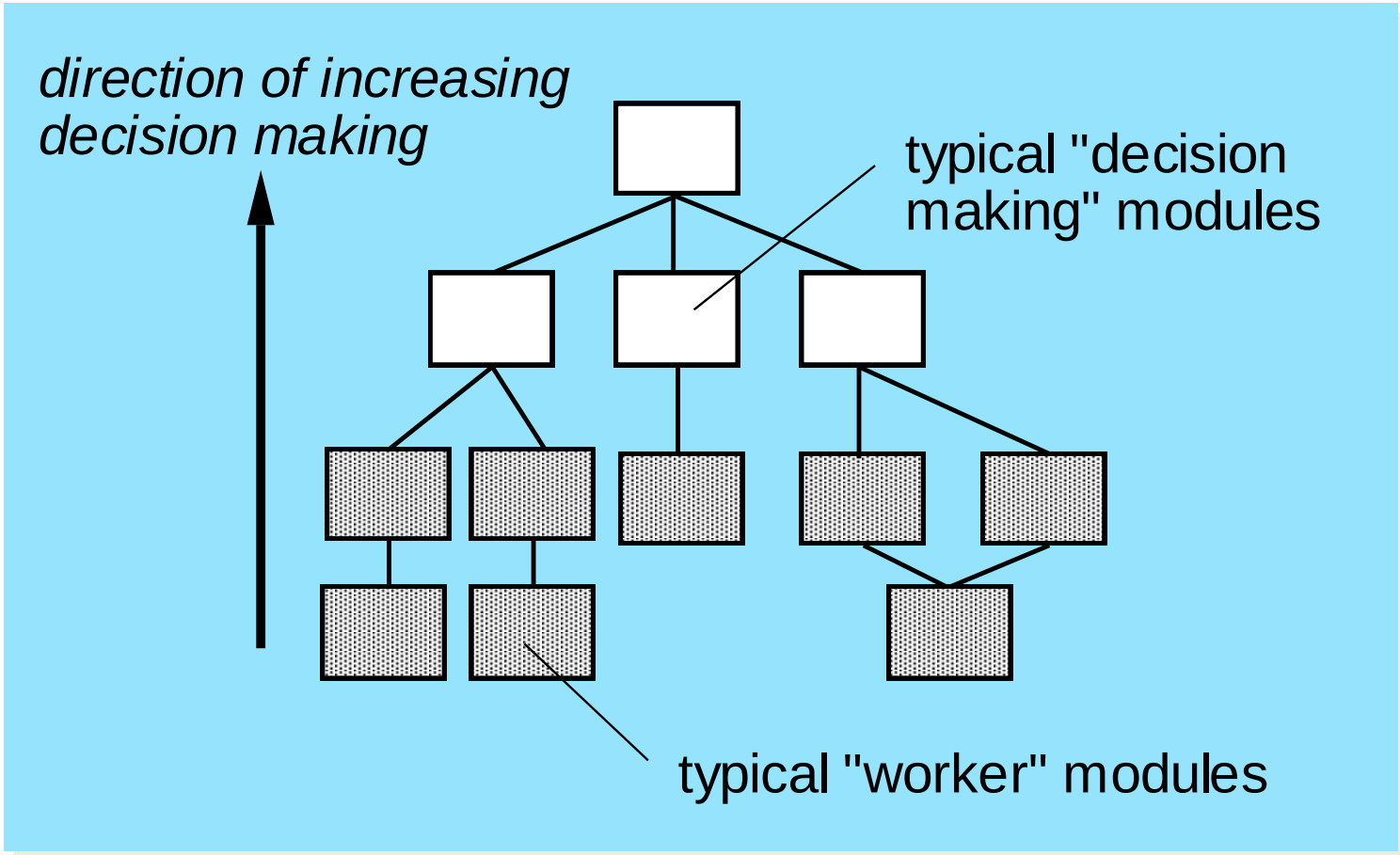
Mapping Data Flow to Architecture

- **Transform Mapping**
  1. Review the fundamental system model.
  2. Review and refine data flow diagrams for the software
  3. Determine whether the DFD has transform or transaction flow characteristics.
  4. Isolate the transform center by specifying incoming and outgoing flow boundaries.
  5. Perform “first-level factoring”
  6. Perform “second-level factoring”
  7. Refine the first-iteration architecture using design heuristics for improved software quality.

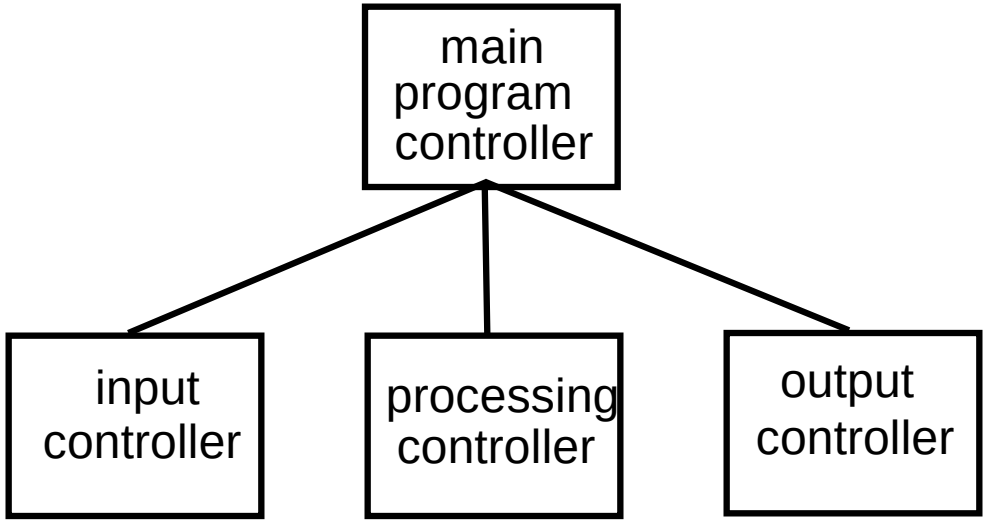
Transform Mapping



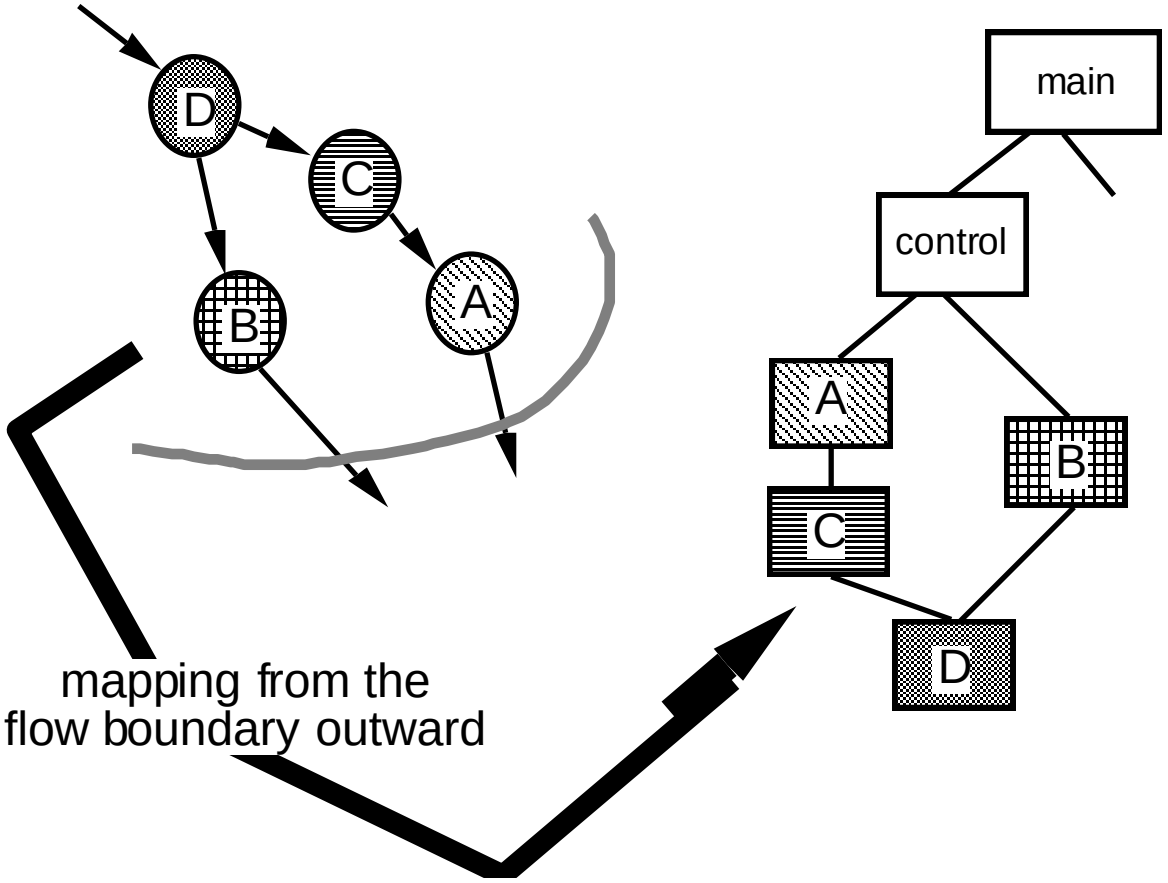
Factoring


direction of increasing
decision making
typical "decision
making" modules
typical "worker" modules

First Level Factoring



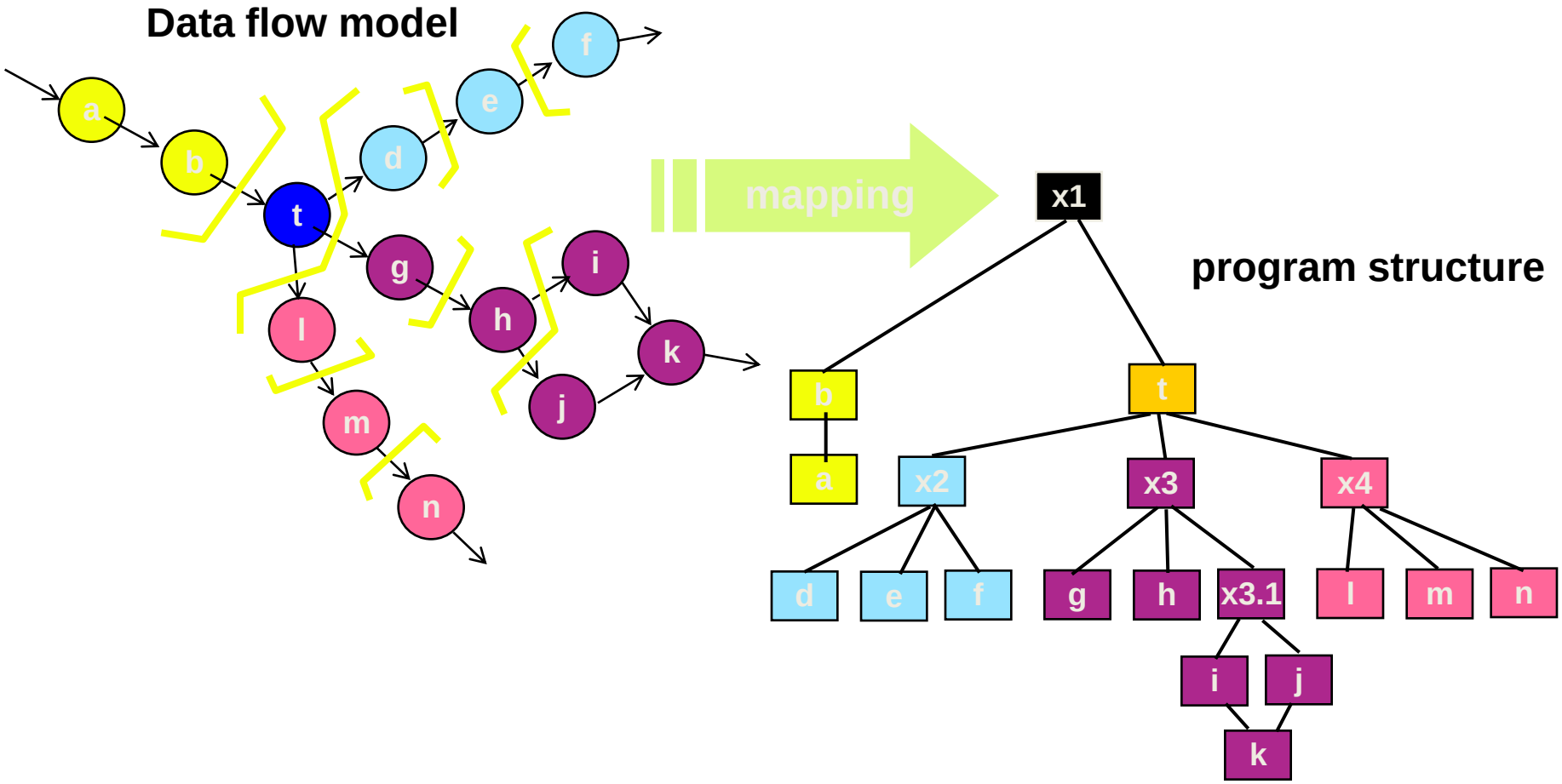
Second Level Factoring



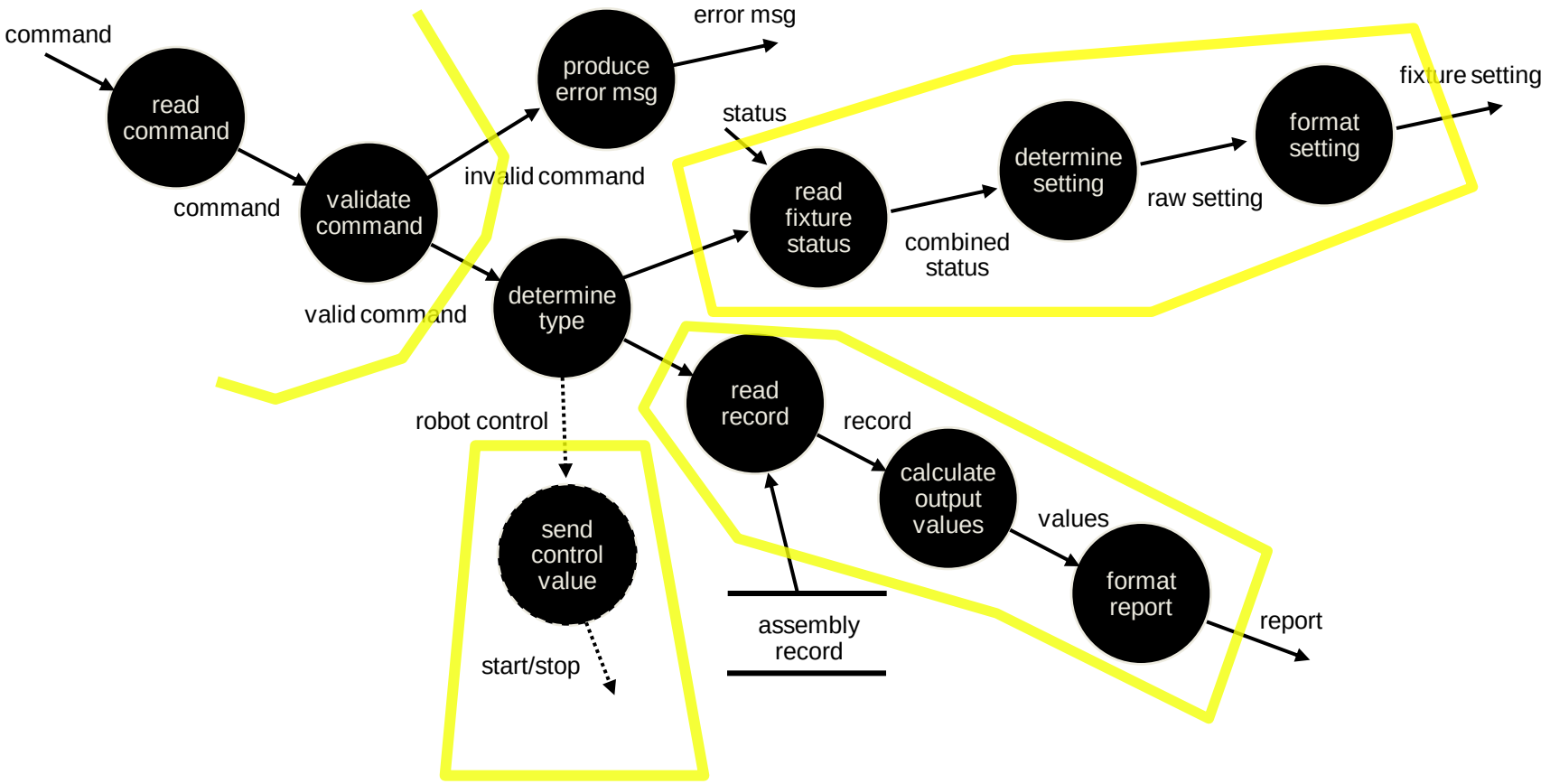
Mapping Data Flow to Architecture

- **Transaction Mapping**
  1. **Review the fundamental system model.**
  2. **Review and refine data flow diagrams for the software**
  3. **Determine whether the DFD has transform or transaction flow characteristics.**
  4. **Isolate the transaction center and the flow characteristics along each of the action paths.**
  5. **Map the DFD in a program structure amenable to transaction processing.**
  6. **Factor and refine the transaction structure and the structure of each action path.**
  7. **Refine the first-iteration architecture using design heuristics for improved software quality.**

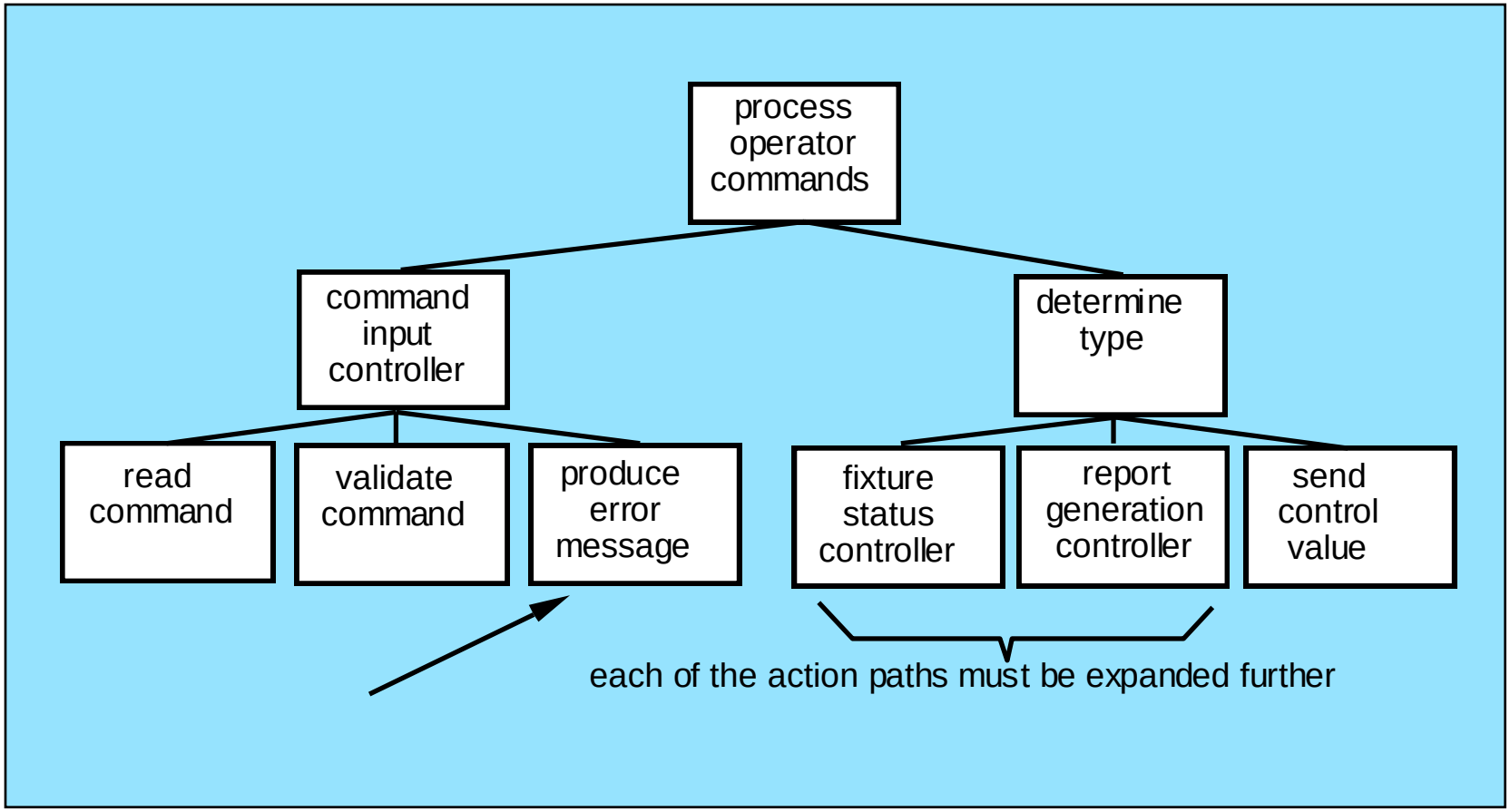
Transaction Mapping



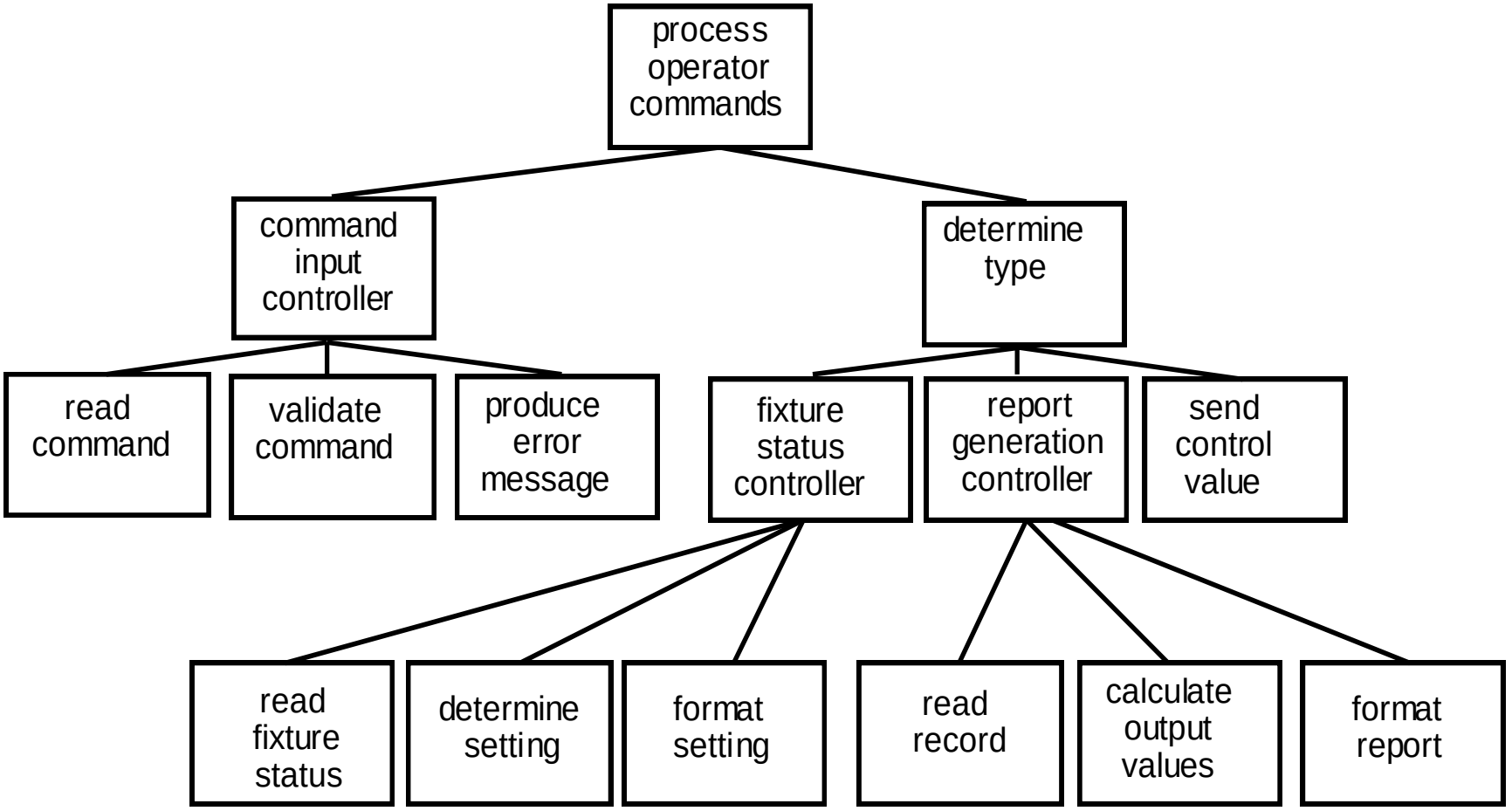
Isolate Flow Paths



Map the Flow Model



Refining


process operator commands
command input controller
determine type
read command
validate command
produce error message
fixture status controller
report generation controller
send control value
read fixture status
determine setting
format setting
read record
calculate output values
format report

Refining the Architectural Design

- Any discussion of design refinement should be prefaced with the following comment:
- “Remember that an ‘optimal design’ that doesn’t work has questionable merit.”
- You should be concerned with developing a representation of software that will meet all functional and performance requirements and merit acceptance based on design measures and heuristics.
- Refinement of software architecture during early stages of design is to be encouraged.

Syllabus

- **User interface design :** Golden Rules, interface analysis, interface design steps
- **Modeling component-level design :** Designing class-based components, conducting component level design, object constraint language, designing conventional components

Performing User interface design

User interface design

- **Background**
  - **Interface design focuses on the following**
    - **The design of interfaces between software components**
    - **The design of interfaces between the software and other nonhuman producers and consumers of information**
    - **The design of the interface between a human and the computer**
  - **Graphical user interfaces (GUIs) have helped to eliminate many of the most horrific interface problems**
  - **However, some are still difficult to learn, hard to use, confusing, counterintuitive, unforgiving, and frustrating**
  - **User interface analysis and design has to do with the study of people and how they relate to technology**

The User Interface

- **User interfaces creates an effective communication medium between a human and a computer**
- **Using a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis of a user interface prototype**
- **User interfaces should be designed to match the skills, experience and expectations of its anticipated users.**
- **System users often judge a system by its interface rather than its functionality.**
- **A poorly designed interface can cause a user to make catastrophic errors.**
- **Poor user interface design is the reason why so many software systems are never used.**

The User Interface

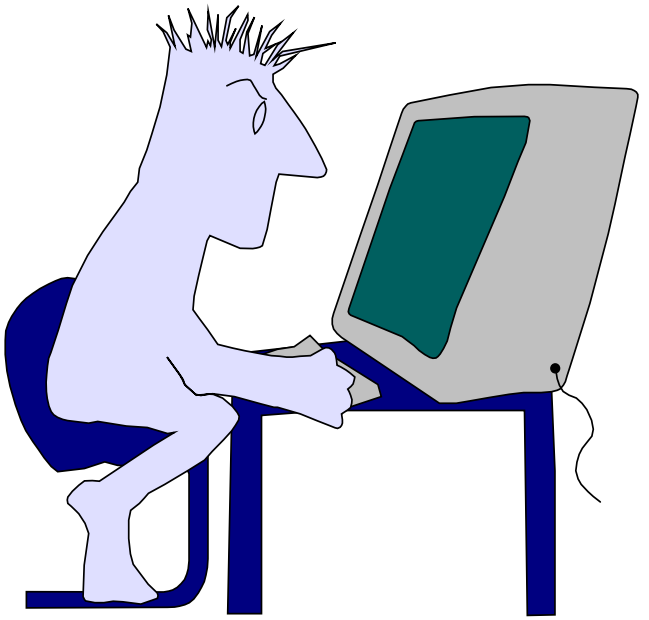
- **Easy to learn?**
- **Easy to use?**
- **Easy to Understand?**

User Interface Design

- **User interface design has as much to do with the study of people as it does with technology issues**
- **Some of the questions that are to be asked and answered are**
  - **Who is the user?**
  - **How does the user learn to interact with a new computer-based system?**
  - **How does the user interpret information produced by the system?**
  - **What will the user expect of the system?**
- **The designer might introduce constraints and limitations to simplify implementation of the interface.**
- **The result is may be an interface that is easy to build, but frustrating to use**

User Interface Design

- **Typical Design Errors**
  - Lack of consistency
  - Too much memorization
  - No guidance / help
  - No context sensitivity
  - Poor response
  - Arcane/unfriendly



Human factors in interface design

- **Limited short-term memory**
  - **People can instantaneously remember about 7 items of information. If you present more than this, they are more liable to make mistakes.**
- **People make mistakes**
  - **When people make mistakes and systems go wrong, inappropriate alarms and messages can increase stress and hence the likelihood of more mistakes.**
- **People are different**
  - **People have a wide range of physical capabilities. Designers should not just design for their own capabilities.**
- **People have different interaction preferences**
  - **Some like pictures, some like text.**

User Interface Design

- **Golden Rules**
  - **Place the user in control**
  - **Reduce the user's memory load**
  - **Make the interface consistent**

User Interface Design

- **Place the User in Control**
  - **Define interaction modes in a way that does not force a user into unnecessary or undesired actions**
    - **The user shall be able to enter and exit a mode with little or no effort (e.g., spell check → edit text → spell check)**
  - **Provide for flexible interaction**
    - **The user shall be able to perform the same action via keyboard commands, mouse movement, or voice recognition**
  - **Allow user interaction to be interruptible and "undo"able**
    - **The user shall be able to easily interrupt a sequence of actions to do something else (without losing the work that has been done so far)**
    - **The user shall be able to "undo" any action**

User Interface Design

- **Place the User in Control**
  - **Streamline interaction as skill levels advance and allow the interaction to be customized**
    - **The user shall be able to use a macro mechanism to perform a sequence of repeated interactions and to customize the interface**
  - **Hide technical internals from the casual user**
    - **The user shall not be required to directly use operating system, file management, networking. etc., commands to perform any actions. Instead, these operations shall be hidden from the user and performed "behind the scenes" in the form of a real-world abstraction**
  - **Design for direct interaction with objects that appear on the screen**
    - **The user shall be able to manipulate objects on the screen in a manner similar to what would occur if the object were a physical thing (e.g., stretch a rectangle, press a button, move a slider)**

User Interface Design

- **Reduce the User's Memory Load**
  - **Reduce demand on short-term memory**
    - **The interface shall reduce the user's requirement to remember past actions and results by providing visual cues of such actions**
  - **Establish meaningful defaults**
    - **The system shall provide the user with default values that make sense to the average user but allow the user to change these defaults**
    - **The user shall be able to easily reset any value to its original default value**
  - **Define shortcuts that are intuitive**
    - **The user shall be provided mnemonics (i.e., control or alt combinations) that tie easily to the action in a way that is easy to remember such as the first letter**

User Interface Design

- **Reduce the User's Memory Load**
  - The visual layout of the interface should be based on a real world metaphor
    - The screen layout of the user interface shall contain well-understood visual cues that the user can relate to real-world actions
  - Disclose information in a progressive fashion
    - When interacting with a task, an object or some behavior, the interface shall be organized hierarchically by moving the user progressively in a step-wise fashion from an abstract concept to a concrete action (e.g., text format options → format dialog box)

The more a user has to remember, the more error-prone interaction with the system will be

User Interface Design

- **Make the Interface Consistent**
  - **The interface should present and acquire information in a consistent fashion**
    - **All visual information shall be organized according to a design standard that is maintained throughout all screen displays**
    - **Input mechanisms shall be constrained to a limited set that is used consistently throughout the application**
    - **Mechanisms for navigating from task to task shall be consistently defined and implemented**
  - **Allow the user to put the current task into a meaningful context**
    - **The interface shall provide indicators (e.g., window titles, consistent color coding) that enable the user to know the context of the work at hand**
    - **The user shall be able to determine where he has come from and what alternatives exist for a transition to a new task**

User Interface Design

- **Make the Interface Consistent**
  - **Maintain consistency across a family of applications**
    - **A set of applications performing complimentary functionality shall all implement the same design rules so that consistency is maintained for all interaction**
  - **If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so**
    - **Once a particular interactive sequence has become a de facto standard (e.g., alt-S to save a file), the application shall continue this expectation in every part of its functionality**

User Interface Design Models

- **Four different models come into play when a user interface is analyzed and designed**
  - **User profile model – Established by a human engineer or software engineer**
  - **Design model – Created by a software engineer**
  - **Implementation model – Created by the software implementers**
  - **User's mental model – Developed by the user when interacting with the application**
- **The role of the interface designer is to reconcile these differences and derive a consistent representation of the interface**

User Profile Model

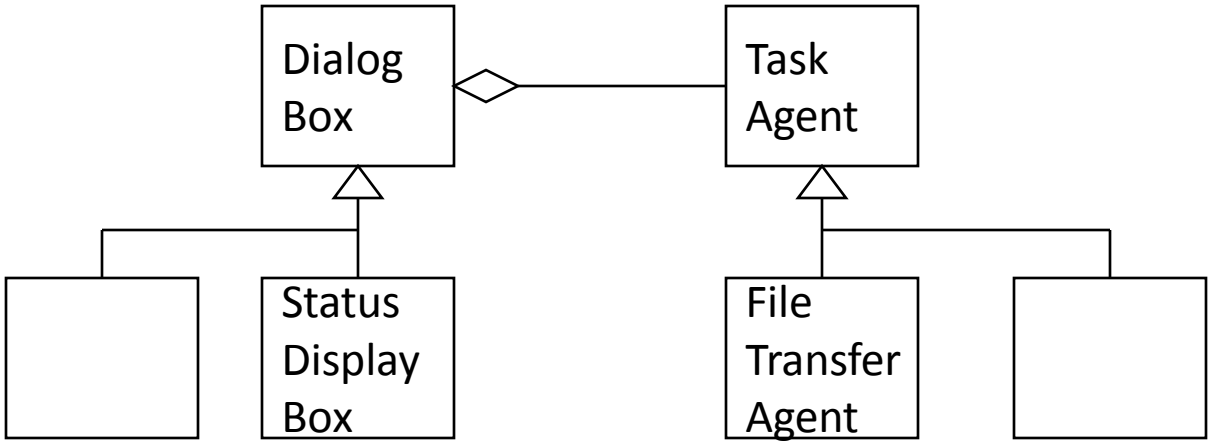
- Establishes the profile of the end-users of the system
  - Based on age, gender, physical abilities, education, cultural or ethnic background, motivation, goals, and personality
- Considers syntactic knowledge of the user
  - The mechanics of interaction that are required to use the interface effectively
- Considers semantic knowledge of the user
  - The underlying sense of the application; an understanding of the functions that are performed, the meaning of input and output, and the objectives of the system

User Profile Model

- Categorizes users as
  - Novices
    - No syntactic knowledge of the system, little semantic knowledge of the application, only general computer usage
  - Knowledgeable, intermittent users
    - Reasonable semantic knowledge of the system, low recall of syntactic information to use the interface
  - Knowledgeable, frequent users
    - Good semantic and syntactic knowledge (i.e., power user), look for shortcuts and abbreviated modes of operation

Design Model

- **Derived from the analysis model of the requirements**
- **Incorporates data, architectural, interface, and procedural representations of the software**
- **Constrained by information in the requirements specification that helps define the user of the system**
- **Normally is incidental to other parts of the design model**
  - **But in many cases it is as important as the other parts**



Implementation Model

- **Consists of the look and feel of the interface combined with all supporting information (books, videos, help files) that describe system syntax and semantics**
- **Strives to agree with the user's mental model; users then feel comfortable with the software and use it effectively**
- **Serves as a translation of the design model by providing a realization of the information contained in the user profile model and the user’s mental model**

User's Mental Model

- **Often called the user's system perception**
- **Consists of the image of the system that users carry in their heads**
- **Accuracy of the description depends upon the user’s profile and overall familiarity with the software in the application domain**

User Interface Development

- User interface development follows a spiral process
  - Interface analysis (user, task, and environment analysis)
    - Focuses on the profile of the users who will interact with the system
    - Concentrates on users, tasks, content and work environment
    - Studies different models of system function (as perceived from the outside)
    - Delineates the human- and computer-oriented tasks that are required to achieve system function
  - Interface design
    - Defines a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system

User Interface Development

- User interface development follows a spiral process
  - Interface construction
    - Begins with a prototype that enables usage scenarios to be evaluated
    - Continues with development tools to complete the construction
  - Interface validation, focuses on
    - The ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements
    - The degree to which the interface is easy to use and easy to learn
    - The users' acceptance of the interface as a useful tool in their work

User Interface Analysis

- **Elements of the User Interface**
  - **To perform user interface analysis, the practitioner needs to study and understand four elements**
    - **The users who will interact with the system through the interface**
    - **The tasks that end users must perform to do their work**
    - **The content that is presented as part of the interface**
    - **The work environment in which these tasks will be conducted**

User Interface Analysis

- **User Analysis**
  - **The analyst strives to get the end user's mental model and the design model to converge by understanding**
    - **The users themselves**
    - **How these people use the system**
  - **Information can be obtained from**
    - **User interviews with the end users**
    - **Sales input from the sales people who interact with customers and users on a regular basis**
    - **Marketing input based on a market analysis to understand how different population segments might use the software**
    - **Support input from the support staff who are aware of what works and what doesn't, what users like and dislike, what features generate questions, and what features are easy to use**
  - **A set of questions should be answered during user analysis**

User Interface Analysis

- **User Analysis Questions**
  1) **Are the users trained professionals, technicians, clerical or manufacturing workers?**
  2) **What level of formal education does the average user have?**
  3) **Are the users capable of learning on their own from written materials or have they expressed a desire for classroom training?**
  4) **Are the users expert typists or are they keyboard phobic?**
  5) **What is the age range of the user community?**
  6) **Will the users be represented predominately by one gender?**
  7) **How are users compensated for the work they perform or are they volunteers?**

User Interface Analysis

- **User Analysis Questions**

8. **Do users work normal office hours, or do they work whenever the job is required?**
9. **Is the software to be an integral part of the work users do, or will it be used only occasionally?**
10. **What is the primary spoken language among users?**
11. **What are the consequences if a user makes a mistake using the system?**
12. **Are users experts in the subject matter that is addressed by the system?**
13. **Do users want to know about the technology that sits behind the interface?**

User Interface Analysis

- **Task Analysis and Modeling**
  - **Task analysis strives to know and understand**
    - **The work the user performs in specific circumstances**
    - **The tasks and subtasks that will be performed as the user does the work**
    - **The specific problem domain objects that the user manipulates as work is performed**
    - **The sequence of work tasks (i.e., the workflow)**
    - **The hierarchy of tasks**
  - **Use cases**
    - **Show how an end user performs some specific work-related task**
    - **Enable the software engineer to extract tasks, objects, and overall workflow of the interaction**
    - **Helps the software engineer to identify additional helpful features**

User Interface Analysis

- **Content Analysis**
  - **The display content may range from character-based reports, to graphical displays, to multimedia information**
  - **Display content may be**
    - **Generated by components in other parts of the application**
    - **Acquired from data stored in a database that is accessible from the application**
    - **Transmitted from systems external to the application in question**
  - **The format and aesthetics of the content (as it is displayed by the interface) needs to be considered**
  - **A set of questions should be answered during content analysis**

User Interface Analysis

- **Content Analysis Guidelines**
  1) **Are various types of data assigned to consistent locations on the screen (e.g., photos always in upper right corner)?**
  2) **Are users able to customize the screen location for content?**
  3) **Is proper on-screen identification assigned to all content?**
  4) **Can large reports be partitioned for ease of understanding?**
  5) **Are mechanisms available for moving directly to summary information for large collections of data?**
  6) **Is graphical output scaled to fit within the bounds of the display device that is used?**
  7) **How is color used to enhance understanding?**
  8) **How are error messages and warnings presented in order to make them quick and easy to see and understand?**

User Interface Analysis

- **Work Environment Analysis**
  - **Software products need to be designed to fit into the work environment, otherwise they may be difficult or frustrating to use**
  - **Factors to consider include**
    - **Type of lighting**
    - **Display size and height**
    - **Keyboard size, height and ease of use**
    - **Mouse type and ease of use**
    - **Surrounding noise**
    - **Space limitations for computer and/or user**
    - **Weather or other atmospheric conditions**
    - **Temperature or pressure restrictions**
    - **Time restrictions (when, how fast, and for how long)**

Interface Design Steps

1. **Applying Interface Design Steps**
2. **User Interface Design Patterns**
3. **Design Issues**

User Interface Design

- **User interface design is an iterative process, where each iteration elaborate and refines the information developed in the preceding steps**
- **General steps for user interface design**
  1) **Using information developed during user interface analysis, define user interface objects and actions (operations)**
  2) **Define events (user actions) that will cause the state of the user interface to change; model this behavior**
  3) **Depict each interface state as it will actually look to the end user**
  4) **Indicate how the user interprets the state of the system from information provided through the interface**
- **During all of these steps, the designer must**
  - **Always follow the three golden rules of user interfaces**
  - **Model how the interface will be implemented**
  - **Consider the computing environment (e.g., display technology, operating system, development tools) that will be used**

User Interface Design

- **Applying Interface Design Steps**
- **Interface objects and actions are obtained from a grammatical parse of the use cases and the software problem statement**
  - **Interface objects are categorized into types: source, target, and application**
    - **A source object is dragged and dropped into a target object such as to create a hardcopy of a report**
    - **An application object represents application-specific data that are not directly manipulated as part of screen interaction such as a list**
  - **After identifying objects and their actions, an interface designer performs screen layout which involves**
    - **Graphical design and placement of icons**
    - **Definition of descriptive screen text**
    - **Specification and titling for windows**
    - **Definition of major and minor menu items**
    - **Specification of a real-world metaphor to follow**

User Interface Design

- User Interface Design Patterns
- Graphical user interfaces have become so common that a wide variety of user interface design patterns has emerged.
- A design pattern is an abstraction that prescribes a design solution to a specific, well-bounded design problem.
- As an example of a commonly encountered interface design problem, consider a situation in which a user must enter one or more calendar dates, sometimes months in advance. There are many possible solutions to this simple problem, and a number of different patterns that might be proposed. Laakso [Laa00] suggests a pattern called **CalendarStrip** that produces a continuous, scrollable calendar in which the current date is highlighted and future dates may be selected by picking them from the calendar. The calendar metaphor is well known to every user and provides an effective mechanism for placing a future date in context.

User Interface Design

- **Design Issues**
- **Four common design issues usually surface in any user interface**
  - **System response time**
    - **System response time is the primary complaint for many interactive applications.**
    - **System response time is measured from the point at which the user performs some control action (e.g., hits the return key or clicks a mouse) until the software responds with desired output or action.**
    - **System response time has two important characteristics: length and variability. If system response is too long, user frustration and stress are inevitable. Variability refers to the deviation from average response time, and in many ways, it is the most important response time characteristic.**

User Interface Design

- **Help facilities**
  - **Almost every user of an interactive, computer-based system requires help now and then.**
  - **Modern software provides online help facilities that enable a user to get a question answered or resolve a problem without leaving the interface.**
  - **A number of design issues must be addressed when a help facility is considered:**
  - **When is it available, how is it accessed, how is it represented to the user, how is it structured, what happens when help is exited**
- **Error handling**
  - **Error messages and warnings are "bad news" delivered to users of interactive systems when something has gone awry. At their worst, error messages and warnings impart useless or misleading information and serve only to increase user frustration.**

User Interface Design

- **In general, every error message or warning produced by an interactive system should have the following characteristics:**
  - **The message should describe the problem in plain language that a typical user can understand**
  - **The message should provide constructive advice for recovering from the error**
  - **The message should indicate any negative consequences of the error (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have)**
  - **The message should be accompanied by an audible or visual cue such as a beep, momentary flashing, or a special error color**
  - **The message should be non-judgmental i.e The message should never place blame on the user**

An effective error message philosophy can do much to improve the quality of an interactive system and will significantly reduce user frustration when problems do occur

User Interface Design

– **Menu and command labeling**

- **The typed command was once the most common mode of interaction between user and system software and was commonly used for applications of every type.**
- **Today, the use of window-oriented, point-and pick interfaces has reduced reliance on typed commands, but some power-users continue to prefer a command-oriented mode of interaction.**

- **A number of design issues arise when typed commands or menu labels are provided as a mode of interaction:**
  - **Will every menu option have a corresponding command?**
  - **What form will a command take? A control sequence? A function key? A typed word?**
  - **How difficult will it be to learn and remember the commands?**
  - **What can be done if a command is forgotten?**
  - **Can commands be customized or abbreviated by the user?**
  - **Are menu labels self-explanatory within the context of the interface?**
  - **Are submenus consistent with the function implied by a master menu item?**

COMPONENT- LEVEL DESIGN

- **What Is a Component?**
  - An Object-Oriented View
  - The Traditional View
  - A Process-Related View
- **Designing Class-Based Components**
  - Basic Design Principles
  - Component-Level Design Guidelines
  - Cohesion
  - Coupling
- **Designing Traditional Components**
  - Graphical Design Notation
  - Tabular Design Notation
  - Program Design Language

Component Definitions

- **What Is a Component?**
  - An Object-Oriented View
  - The Traditional View
  - A Process-Related View
- **A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.**
- **A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third-parties.**
- **A modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces**

Component-level Design

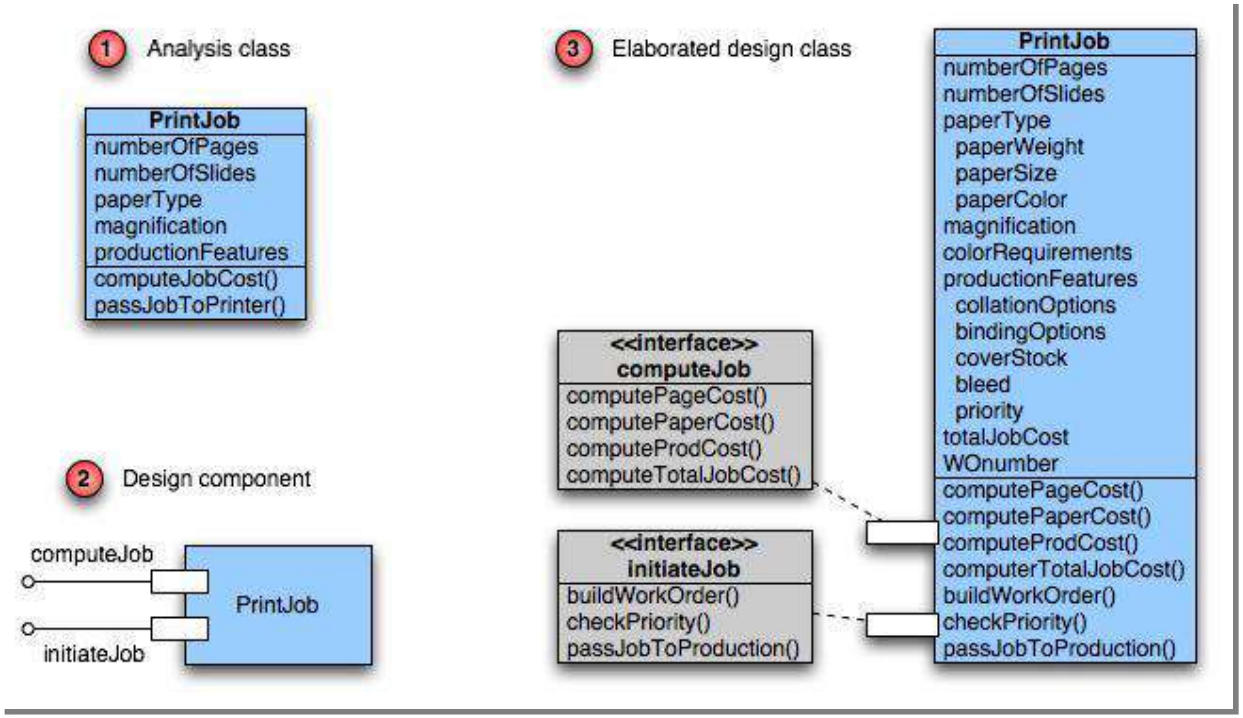
- A complete set of software components is defined during architectural design
- But the internal data structures and processing details of each component are not represented at a level of abstraction that is close to code
- Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each component

Component Views

- **An Object-Oriented View**
  - A component is a set of collaborating classes.
  - Each class within a component is fully elaborated to include all attributes and operations that are relevant to its implementation
  - As part of the design elaboration, all interfaces that enable the classes to communicate and collaborate with other design classes must also be defined
  - Begins with analysis model and elaborates analysis classes (for components that relate to the problem domain) and infrastructure classes (for components that provide support services for the problem domain)

Class Elaboration

- **Object Oriented View**



Component Views

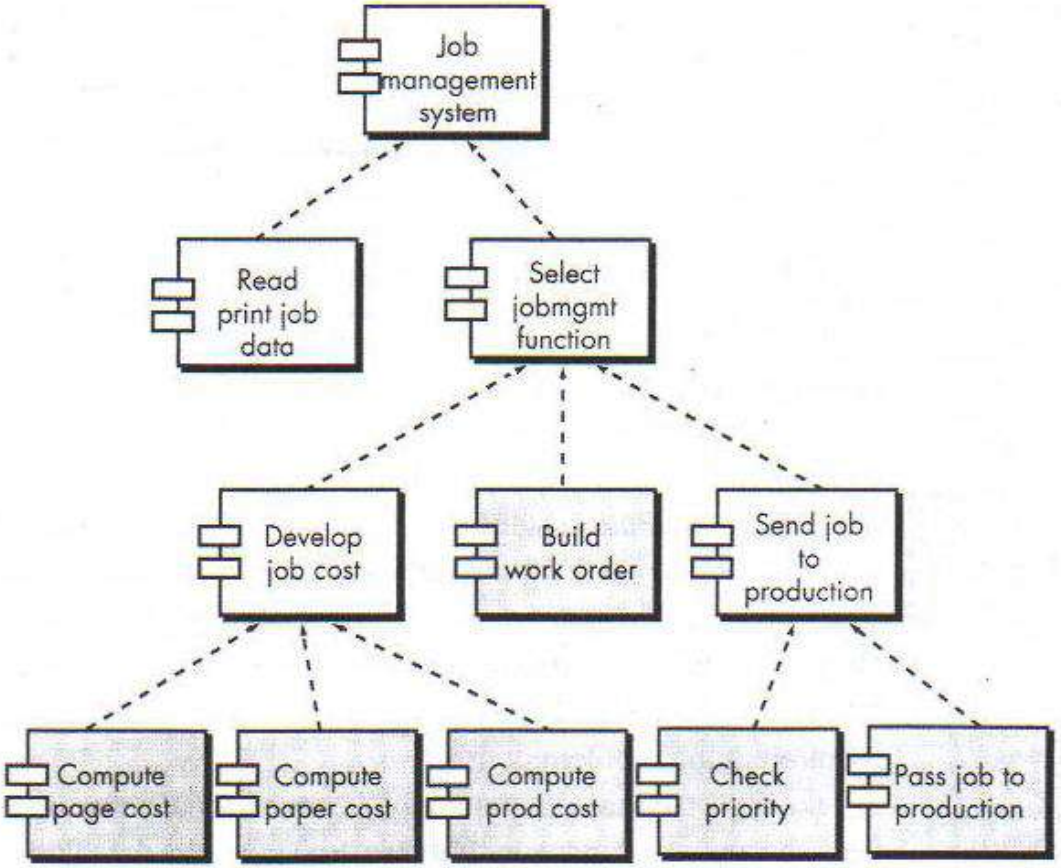
- **The Traditional View – A component is a functional element of a program that incorporates processing logic, the internal data structures required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it.**

Component Views

- **The Traditional View**
- **A conventional component called as module, resides within the software architecture and serves one of the three important roles**
  - **A control component that coordinates the invocation of all other problem domain components**
  - **A problem domain component that implements complete or partial function that is required by the customer**
  - **An infrastructure component that is responsible for functions that support the processing required in the problem domain.**

- **A Process-Related View**
- **Over the past two decades, the software engineering community has emphasized the need to build systems that make use of existing of software components or design patterns.**
- **As the software architecture is developed, you choose components or design patterns from the catalog and use them to populate the architecture.**

Component Views



Designing Class Based Components

Designing Class Based Components

1. Basic Design Principles
2. Component-Level Design Guidelines
3. Cohesion
4. Coupling

- Component-level design focuses on the elaboration of analysis classes (problem domain specific classes) and definition and refinement of infrastructure classes
- Purpose of using design principles is to create designs that are more amenable to change and to reduce propagation of side effects when changes do occur

Basic Design Principles

- SOLID Principles

S	SRP	Single responsibility principle an object should have only a single responsibility.
O	OCP	Open/closed principle "software entities … should be open for extension, but closed for modification".
L	LSP	Liskov substitution principle "objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program".
I	ISP	Interface segregation principle "many client specific interfaces are better than one general purpose interface
D	DIP	Dependency inversion principle one should "Depend upon Abstractions. Do not depend upon concretions

Basic Design Principles

- Single Responsibility Principle
- Open-Closed Principle
- Liskov Substitution Principle
- Dependency inversion Principle
- Interface segregation Principle

Single Responsibility Principle

- **Single responsibility principle** states that every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility.
- Classes that keeps track of a lot of information and have several responsibilities.
- One code change will most likely affect other parts of the class and therefore indirectly all other classes that uses it.
- That in turn leads to an even bigger maintenance mess since no one dares to do any changes other than adding new functionality to it.

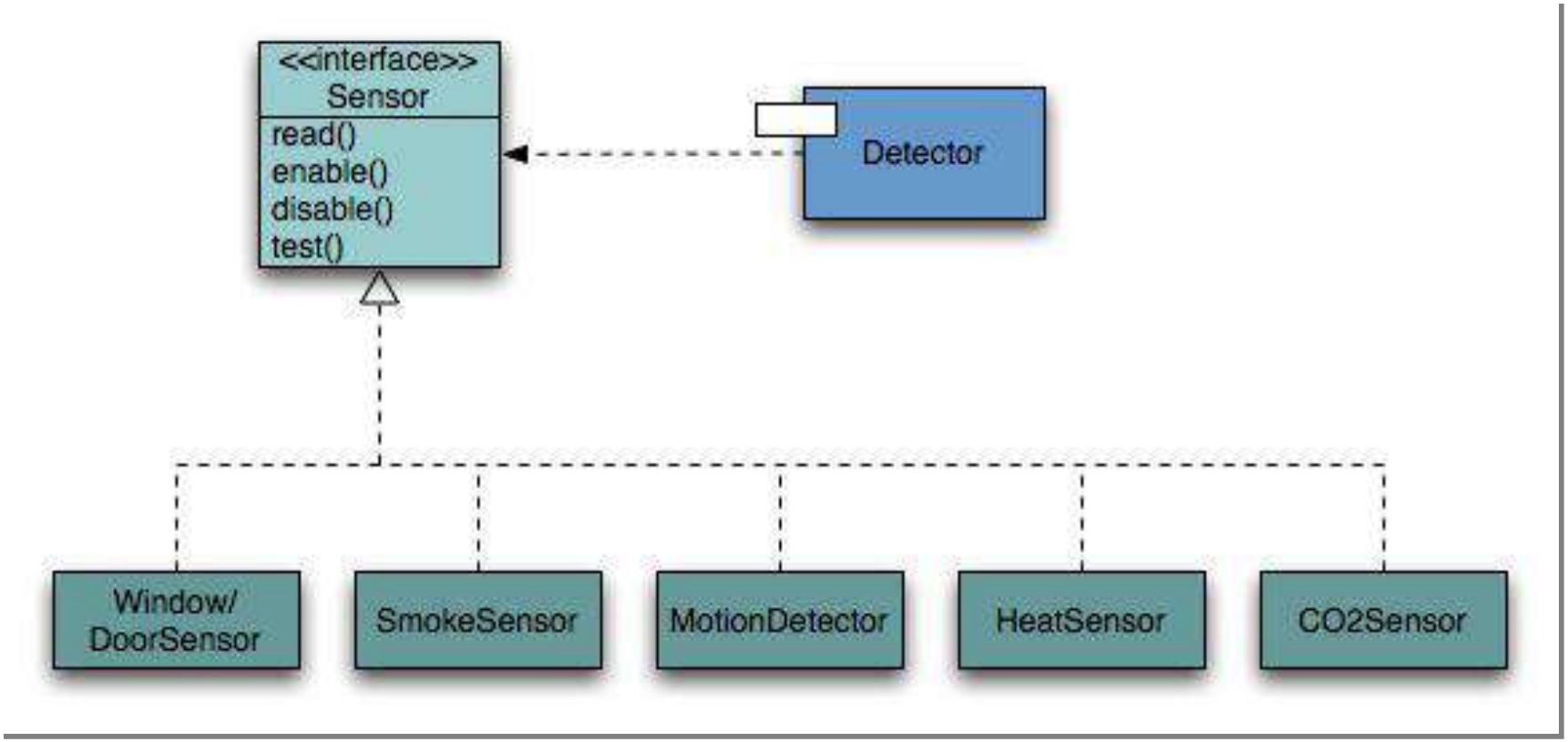
Single Responsibility Principle

- Just because you can, does not mean should

Open-Closed Principle

- A module should be open for extension but closed for modification
- The designer should specify the component in a way that it allows to be extended (within the functional domain that it addresses) without the need to make internal modifications (code or logic level) to the component
- Abstractions are created that serve as a buffer between the functionality that is likely to be extended and the design class

Open-Closed Principle



Open-Closed Principle

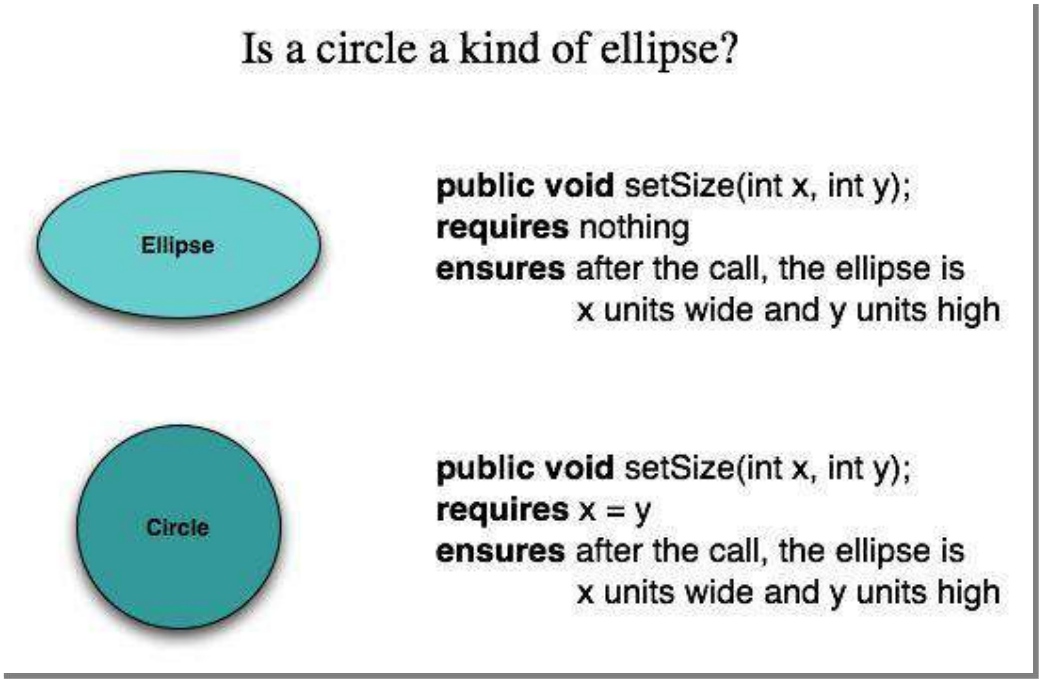
- Open chest surgery is not needed when putting on a coat.

The Liskov Substitution Principle (LSP).

- **“Subclasses should be substitutable for their base classes”.**
- **A component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead**
- **Any class derived from the base class must honour any implied contract between the base class and the components that uses it**
- **A contract is a precondition that must be true before the component uses a base class and a post condition that should be true after the component uses the base class**
- **When a designer creates derived classes, they must also conform to pre and post conditions**

The Liskov Substitution Principle (LSP).

- **Subclasses should be substitutable for base classes**



The Liskov Substitution Principle (LSP).

- 

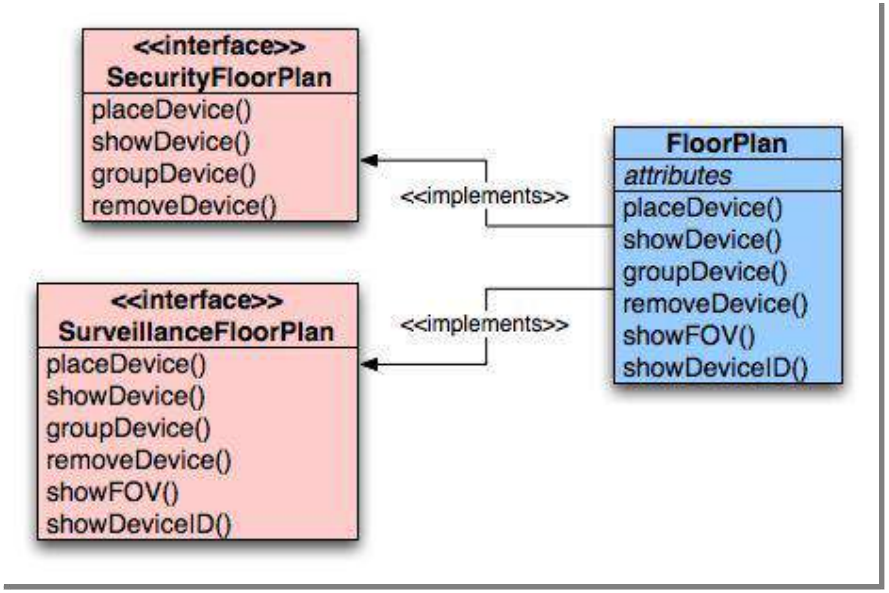
- If it looks like a duck, quacks like a duck, but needs batteries – you probably have the wrong abstraction

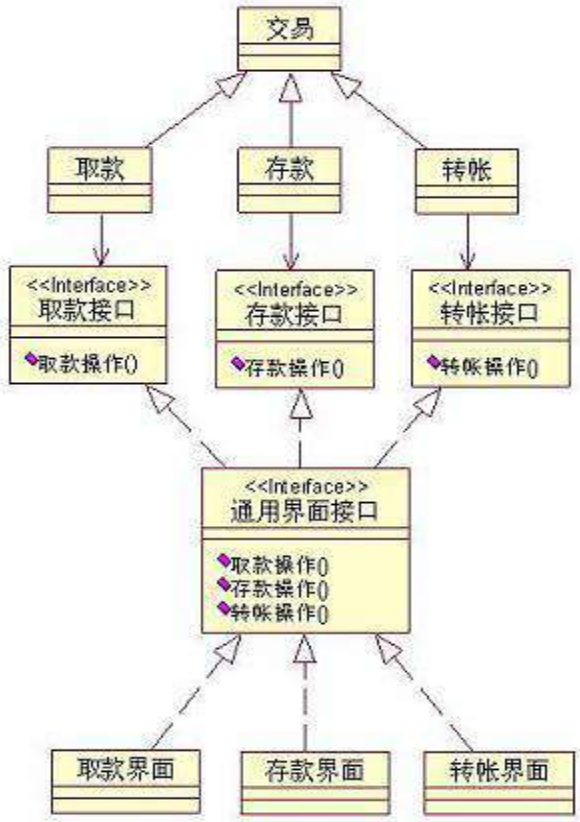
The Interface Segregation Principle (ISP).

- **"Many client-specific interfaces are better than one general purpose interface"**
- **Many instances in which multiple client components use the operations provided by a server class.**
- **ISP suggests that you should create a specialized interface to serve each major category of clients.**
- **Only those operations that are relevant to a particular category of clients should be specified in the interface for that client.**
- **If multiple clients require the same operations, it should be specified in each of the specialized interfaces.**

The Interface Segregation Principle (ISP).

- **Many client-specific interfaces are better than one general purpose interface.**





ISP图2

The Interface Segregation Principle (ISP).

- You want me to plug this in, where?

Dependency Inversion Principle (DIP)

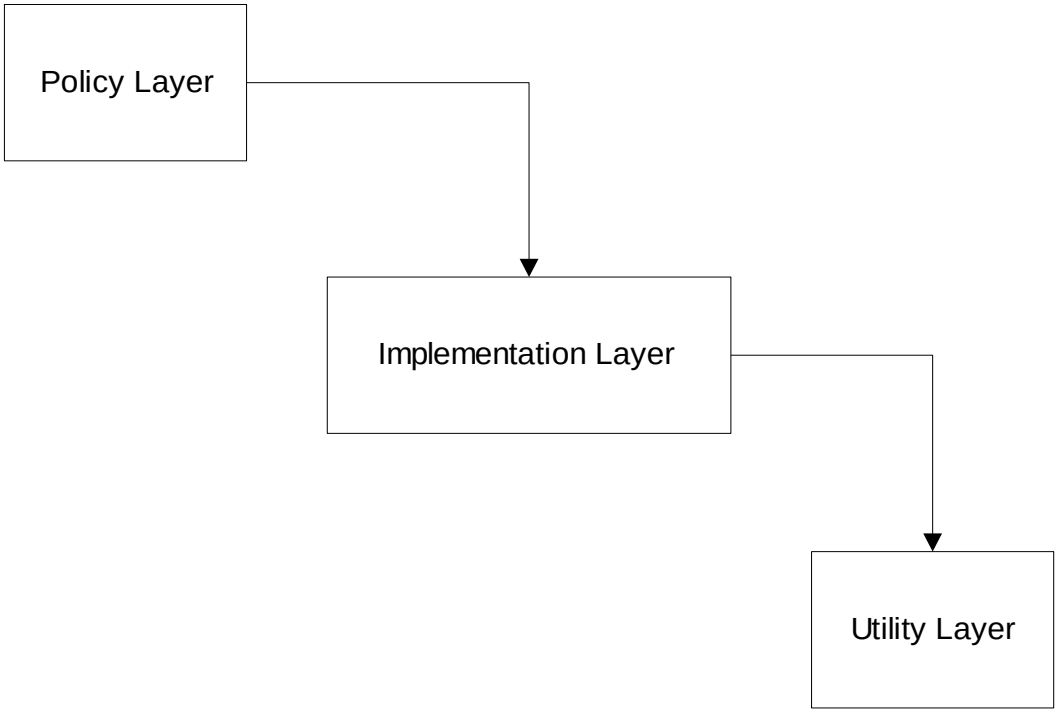
- **“Depend on abstractions. Do not depend on concretions”**
- **High level modules should not depend upon low level modules. Both should depend upon abstractions.**
- **Abstractions should not depend upon details. Details should depend upon abstractions**

Dependency Inversion Principle (DIP)

- **Dependency Inversion Principle:**
  - **High level components should not depend upon low level components. Instead, both should depend on abstractions.**
  - **Abstractions should not depend upon details. Details should depend upon the abstractions.**

- **We all can agree that complex systems need to be structured into layers. But if that is not done carefully the top levels tend to depend on the lower levels.**
  - **On the next page we show a "standard" architecture that appears to be practical and useful.**
  - **Unfortunately it has the ugly property that policy layer depends on implementation layer which depends on utility layer, e.g., dependencies all the way down.**

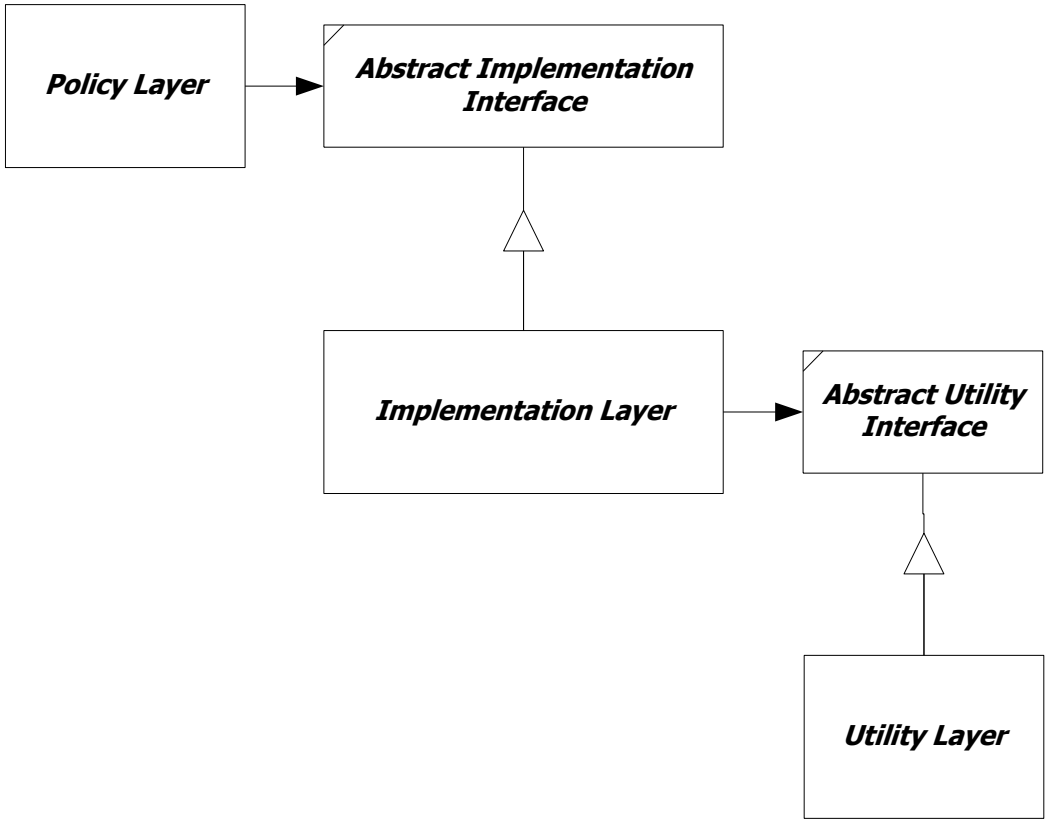
Dependency Inversion Principle (DIP)



Dependency Inversion Principle (DIP)

- The diagram on the next page shows a “better” model, e.g., less rigid, less fragile, more mobile.
  - Each layer is separated by an abstract interface.
    - policy depends, not on the implementation, but only on its abstract interface.
    - implementation depends only on its interface and on the interface defined by utility
    - utility depends only on its published interface
  - Policy is unaffected by any changes to implementation and utility and implementation is unaffected by changes to utility.
    - as long as we transport the interface along with its component each of the three components is reusable and robust

Dependency Inversion Principle (DIP)



Dependency Inversion Principle (DIP)

- 

- Would you solder a lamp directly to the electrical wiring in a wall?

Component-Level Design Guidelines

- In addition to the principles discussed, a set of pragmatic design guidelines can be applied as component-level design proceeds.
- These guidelines apply to components, their interfaces, and the dependencies and inheritance and inheritance characteristics that have an impact on the resultant design.
- Components: Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model.
- Architectural component names should be drawn from the problem domain and should have meaning to all stakeholders who view the architectural model.
- For example, the class name FloorPlan is meaningful to everyone reading it regardless of technical background.

Component-Level Design Guidelines

- Interfaces: Interfaces provide important information about communication and collaboration
- **Ambler recommends that**

  (1) lollipop representation of an interface should be used in lieu of the more formal UML box and dashed arrow approach, when diagrams grow complex;

  (2) for consistency, interfaces should flow from the left-hand side of the component box;

  (3)only those interfaces that are relevant to the component under consideration should be shown, even if other interfaces are available.
- Dependencies and Inheritance:
  - For improved readability, it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).

Cohesion

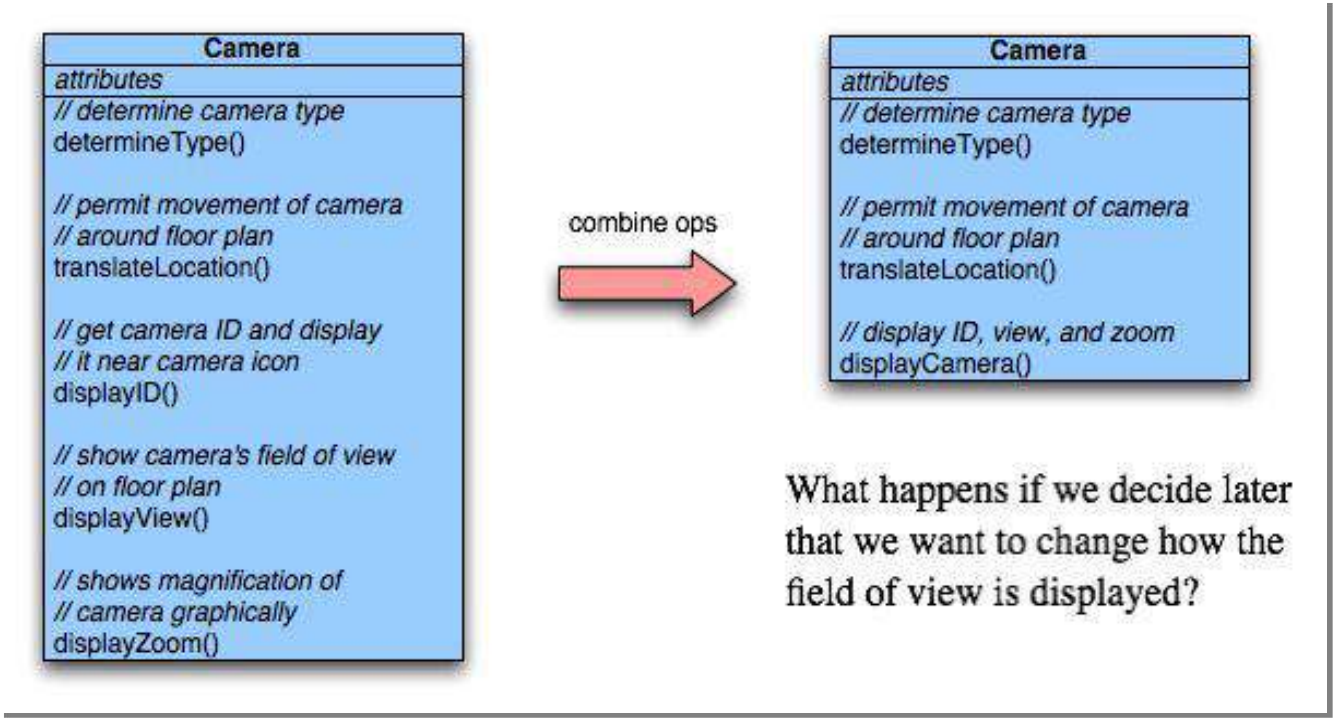
- The "single-mindedness" of a component.
- Cohesion implies that a single component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself.
- Types of cohesion
  - Functional
  - Layer
  - Communicational

Functional Cohesion

- **A form of cohesion in which modules together perform a function (a computation) that returns a result and has no side effects) are kept together, and everything *else* is kept out**
- **Has advantages**
  - **It is easier to understand a module when you know all it does is generate one specific output and has no side effects**
  - **It is easier to replace a functionally cohesive module with another that performs the same computation**
  - **Functionally cohesive module is much more likely to be reusable**

Functional Cohesion

- **Typically applies to operations. Occurs when a module performs one and only one computation and then returns a result.**

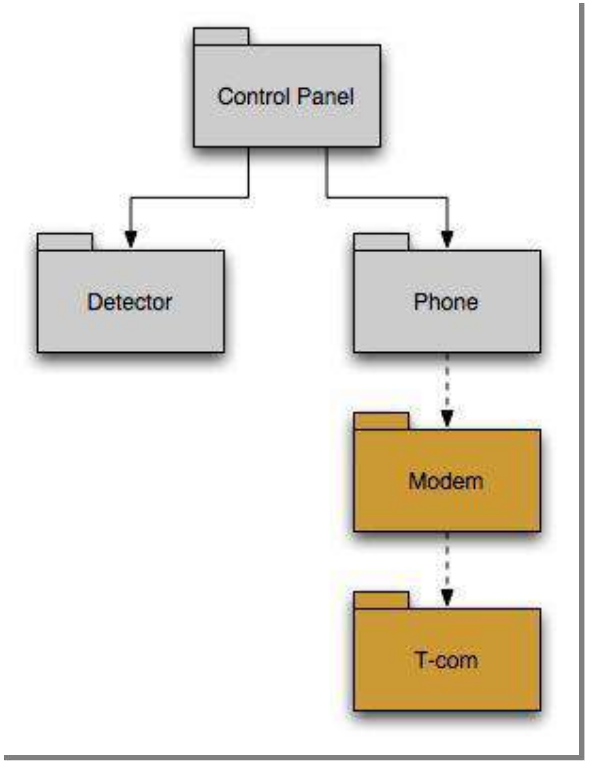


Layer Cohesion

- **Form of cohesion in which the facilities *for* providing or accessing a set of services through an API or hardware interface are kept together**
- **There must also be a strict hierarchy in which higher level layers can access only lower-level layers. In other words, the system is effectively divided into layers**
- **The set of related services which could form a layer might include:**
  - **Services *for computation***
  - **Services *for* transmission of messages or data**
  - **Services *for* storage of data**
  - **Services *for* managing security**
  - **Services *for* interacting with users**
  - **Services provided by an operating system**
  - **Services provided directly by the hardware**

Layer Cohesion

- **Applies to packages, components, and classes. Occurs when a higher layer can access a lower layer, but lower layers do not access higher layers.**



Communicational Cohesion

- All operations that access the same data are defined within one class.
- In general, such classes focus solely on the data in question, accessing and storing it.
- Example: A StudentRecord class that adds, removes, updates, and accesses various fields of a student record for client components.

Other Types of Cohesion

- **Procedural Cohesion**
  - **Components or operations are grouped in a manner that allows one to be invoked immediately after the preceding one was invoked, even when there is no data passed between them**
- **Sequential Cohesion**
  - **Components or operations are grouped in a manner that allows the first to provide input to the next and so on. The intent is to implement sequence of operations**
- **Temporal Cohesion**
  - **Operations that are performed to reflect a specific behaviour or state e.g an operation performed at start up or all operations performed when an error is detected**

Coupling

- **Coupling or Dependency is the degree to which each program module relies on each one of the other modules.**
- **Coupling is usually contrasted with cohesion. Low coupling often correlates with high cohesion, and vice versa**
- **Low coupling is often a sign of a well-structured computer system and a good design, and when combined with high cohesion, supports the general goals of high readability and maintainability.**

Coupling

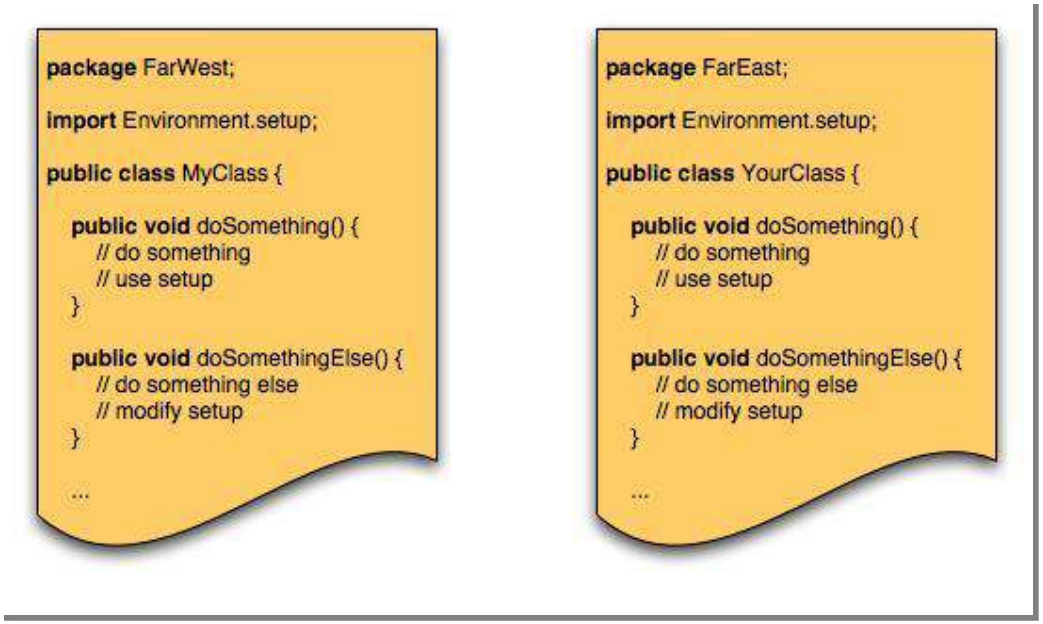
- Coupling can be "low" (also "loose" and "weak") or "high" (also "tight" and "strong"). Types of coupling, are as follows:
- Content coupling (high)
  - Content coupling (also known as Pathological coupling) is when one module modifies or relies on the internal workings of another module (e.g., accessing local data of another module).
  - Therefore changing the way the second module produces data (location, type, timing) will lead to changing the dependent module.
  - Violates information hiding

Coupling

- **Content coupling (high)**
  - Example
    - **module a modifies statements of module b**
    - **module a refers to local data of module *b* in terms of some numerical displacement within *b***
    - **module *a* branches into local label of module *b***
  - **Why is this bad?**
    - **almost any change to *b* requires changes to *a***

Coupling

- **Common coupling**
  - **Common coupling (also known as Global coupling) is when two modules share the same global data (e.g., a global variable)**
  - **Changing the shared resource implies changing all the modules using it**



```
package FarWest;

import Environment.setup;

public class MyClass {

   public void doSomething() {
      // do something
      // use setup
   }

   public void doSomethingElse() {
      // do something else
      // modify setup
   }

   ...
```

```
package FarEast;

import Environment.setup;

public class YourClass {

   public void doSomething() {
      // do something
      // use setup
   }

   public void doSomethingElse() {
      // do something else
      // modify setup
   }

   ...
```

Coupling

- **External coupling**
  - **External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface. This is basically related to the communication to external tools and devices.**
  - **Occurs when a component communicates or collaborates with infrastructure components (operating systems, data base capability, telecommunication functions).**
  - **Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system**

Coupling

- **Control coupling**
  - **Control coupling is one module controlling the flow of another, by passing it information on what to do (e.g., passing a what-to-do flag).**
  - **Occurs when *operation A*() invokes *operation B*() and passes a control flag. The control flag then "directs" logical flow within *B*.**
  - **Problem with this form of coupling is that an unrelated change in B can result in the necessity to change the meaning of the control flag that A passes**

Coupling

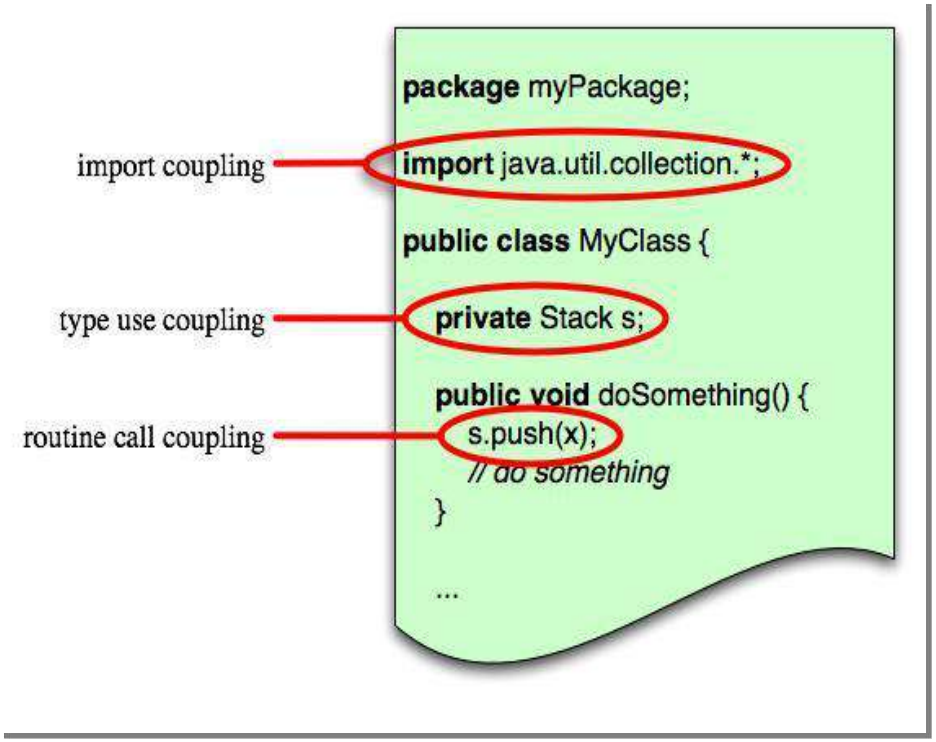
- **Stamp coupling (Data-structured coupling)**
  - **Stamp coupling is when modules share a composite data structure and use only a part of it, possibly a different part (e.g., passing a whole record to a function that only needs one field of it).**
  - **This may lead to changing the way a module reads a record because a field that the module doesn't need has been modified.**
  - **Occurs when Class B is declared as a type for an argument of an operation of Class A. Because Class B is now part of the definition of Class A, modifying the system becomes more complex**

Coupling

- **Data coupling**
  - **Data coupling is when modules share data through, for example, parameters. Each datum is an elementary piece, and these are the only data shared (e.g., passing an integer to a function that computes a square root).**
  - **Occurs when operations pass long strings of data arguments. The bandwidth of communication between classes and components grows and the complexity of the interface increases. Testing and maintenance are more difficult**

Coupling

- **Routine call Coupling**
  - **Certain types of coupling occur routinely in object-oriented programming.**



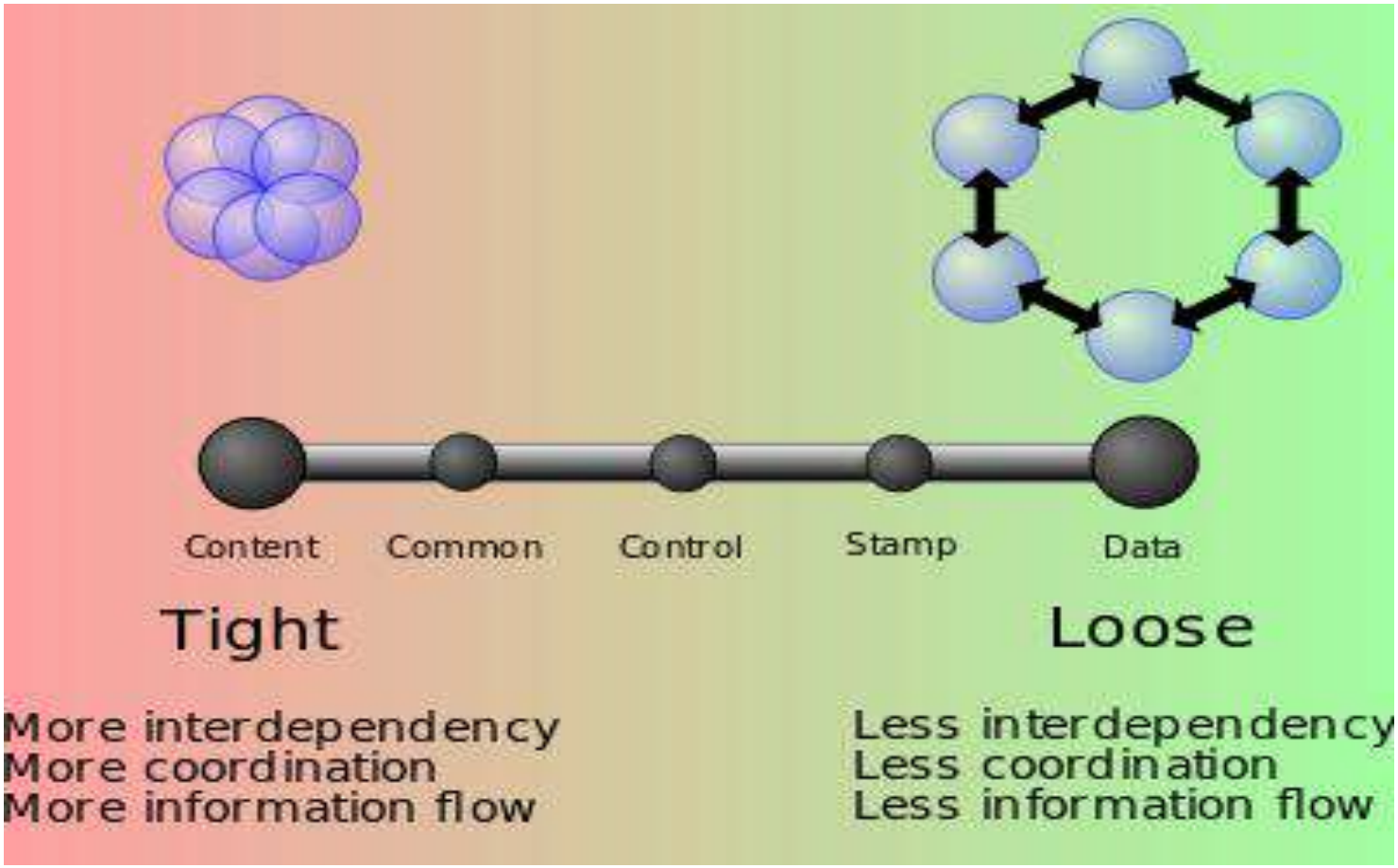
Coupling

- **Message coupling (low)**
  - **This is the loosest type of coupling. Component communication is done via parameters or message passing**
- **Software must communicate internally and externally and hence coupling is essential. However the designer should work to reduce coupling wherever possible and understand the ramifications of high coupling when it cannot be avoided**

Coupling

- Avoid
  - Content coupling
- Use caution
  - Common coupling
- Be aware
  - Routine call coupling
  - Type use coupling
  - Inclusion or import coupling

Coupling


Content
Common
Control
Stamp
Data
Tight
Loose
More interdependency
More coordination
More information flow
Less interdependency
Less coordination
Less information flow

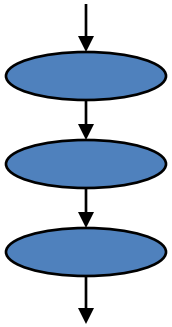
Designing Traditional Components

- **Conventional design constructs emphasize the maintainability of a functional/procedural program**
  - **Sequence, condition, and repetition**
  - **The constructs are sequence, condition, and repetition.**
  - ***Sequence* implements processing steps that are essential in the specification of any algorithm.**
  - **Condition provides the facility for selected processing based on some logical occurrence.**
  - **repetition allows for looping.**
  - **These three constructs are fundamental to structured programming—an important component-level design technique.**
- **Each construct has a predictable logical structure where control enters at the top and exits at the bottom, enabling a maintainer to easily follow the procedural flow**
- **Various notations depict the use of these constructs**
  - **Graphical design notation**
    - **Sequence, if-then-else, selection, repetition**
  - **Tabular design notation**
  - **Program design language**

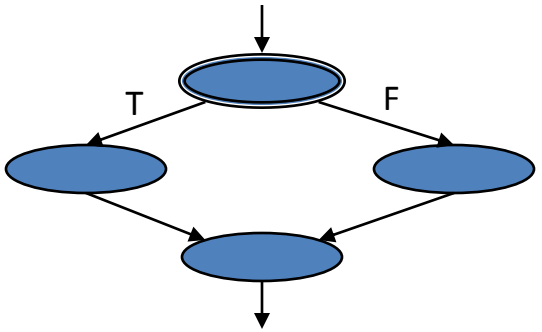
Graphical Design Notation

- ”A picture is worth a thousand words,” but it’s rather important to know which picture and which 1000 words.
- There is no question that graphical tools, such as the UML activity diagram or the flowchart, provide useful pictorial patterns that readily depict procedural detail.
- However, if graphical tools are misused, the wrong picture may lead to the wrong software.
- The activity diagram allows you to represent sequence, condition, and repetition—
- all elements of structured programming—and is a descendent of an earlier pictorial design representation (still used widely) called a **flowchart**.
- A flowchart, like an activity diagram, is quite simple pictorially. A **box** is used to indicate a processing step. A **diamond** represents a logical condition, and **arrows** show the flow of control. The **sequence** is represented as two processing boxes connected by a line (arrow) of control.

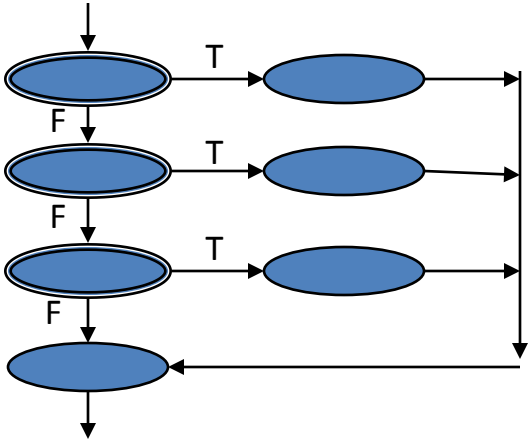
Graphical Design Notation



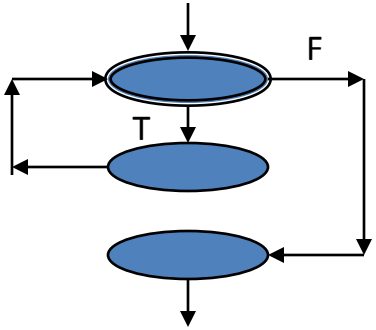
Sequence



If-then-else



Selection



Repetition

Tabular Design Notation

- **In many software applications, a module may be required to evaluate a complex combination of conditions and select appropriate actions based on these conditions.**
- **Decision tables provide a notation that translates actions and conditions (described in a processing narrative or a use case) into a tabular form.**
- **The following steps are applied to develop a decision table:**

1. **List all actions that can be associated with a specific procedure (or module)**
2. **List all conditions (or decisions made) during execution of the procedure**
3. **Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions**
4. **Define rules by indicating what action(s) occurs for a set of conditions.**

Tabular Design Notation (continued)

	Rules					
Conditions	1	2	3	4	5	6
Regular customer	T	T				
Silver customer			T	T		
Gold customer					T	T
Special discount	F	T	F	T	F	T
Actions						
No discount	✓					
Apply 8 percent discount			✓	✓		
Apply 15 percent discount					✓	✓
Apply additional *x* percent discount		✓		✓		✓

Program Design Language

- **Program design language (PDL), also called structured English or pseudocode, incorporates the logical structure of a programming language with the free-form expressive ability of a natural language (e.g., English).**
- **Narrative text (e.g., English) is embedded within a programming language-like syntax. Automated tools (e.g., [Cai03]) can be used to enhance the application of PDL.**
- **A basic PDL syntax should include constructs for component definition, interface description, data declaration, block structuring, condition constructs, repetition constructs, and input-output (I/O) constructs.**
- **PDL can be extended to include keywords for multitasking and/or concurrent processing, interrupt handling, interprocess synchronization, and many other features**

Program Design Language

```
component alarmManagement:
The intent of this component is to manage control panel switches and input from sensors by
type and to act on any alarm condition that is encountered.
    set default values for systemStatus (returned value), all data items
    initialize all system ports and reset all hardware
    check controlPanelSwitches (cps)
        if cps = "test" then invoke alarm set to "on"
        if cps = "alarmOff" then invoke alarm set to "off"
        •
        •
        •
        default for cps = none
    reset all signalValues and switches
    do for all sensors
        invoke checkSensor procedure returning signalValue
        if signalValue > bound [alarmType]
            then phone.message = message [alarmType]
                set alarmBell to "on" for alarmTimeSeconds
```

UNIT- IV
TESTING AND IMPLEMENTATION

Software testing fundamentals: Internal and external views of testing, white box testing, basis path testing, control structure testing, black box testing, regression testing, unit testing, integration testing, validation testing, system testing and debugging; Software implementation techniques: Coding practices, refactoring.

Software testing fundamentals

- The goal of testing is to find errors, and a good test is one that has a high probability of finding an error.
- Therefore, you should design and implement a computer based system or a product with “testability” in mind.
- The tests themselves must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum of effort.
- Testability. James Bach1 provides the following definition for testability: “Software testability is simply how easily [a computer program] can be tested.” The following characteristics lead to testable software.
- Operability. “The better it works, the more efficiently it can be tested.” If a system is designed and implemented with quality in mind, relatively few bugs will block the execution of tests, allowing testing to progress without fits and starts.
- Observability. “What you see is what you test.” Inputs provided as part of testing produce distinct outputs. System states and variables are visible or queriable during execution. Incorrect output is easily identified. Internal errors are automatically detected and reported. Source code is accessible.

Software testing fundamentals

- Controllability. “The better we can control the software, the more the testing can be automated and optimized.” All possible outputs can be generated through some combination of input, and I/O formats are consistent and structured.
- Decomposability. “By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting.” The software system is built from independent modules that can be tested independently.
- Simplicity. “The less there is to test, the more quickly we can test it.” The program should exhibit functional simplicity (e.g., the feature set is the minimum necessary to meet requirements); structural simplicity (e.g., architecture is modularized to limit the propagation of faults), and code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance).
- Stability. “The fewer the changes, the fewer the disruptions to testing.” Changes to the software are infrequent, controlled when they do occur, and do not invalidate existing tests. The software recovers well from failures.
- Understandability. “The more information we have, the smarter we will test.” The architectural design and the dependencies between internal, external, and shared components are well understood. Technical documentation is instantly accessible, well organized, specific and detailed, and accurate. Changes to the design are communicated to testers.

Software testing fundamentals

- What are good Test Characteristics (or) What is a “good” test?
- *A good test has a high probability of finding an error*. To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail. Ideally, the classes of failure are probed.
- *A good test is not redundant.* Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different).
- *A good test should be “best of breed”* . In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.
- *A good test should be neither too simple nor too complex*. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

Internal and External Views of Testing

- Any engineered product (and most other things) can be tested in one of two ways:

- (1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function.

- (2) Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised.
- The first test approach takes an external view and is called black-box testing. The second requires an internal view and is termed white-box testing.

Internal and External Views of Testing

- Black-box testing alludes to tests that are conducted at the software interface. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software.
- White-box testing of software is predicated on close examination of procedural detail. Logical paths through the software and collaborations between components are tested by exercising specific sets of conditions and/or loops.
- White-box testing would lead to “100 percent correct programs.” need do is define all logical paths, develop test cases to exercise them, and evaluate results, i.e, generate test cases to exercise program logic exhaustively.
- A limited number of important logical paths can be selected and exercised. Important data structures can be probed for validity.

White-Box Testing

- White-box testing is the detailed investigation of internal logic and structure of the code.
- White-box testing, sometimes called glass-box testing or open-box testing.
- It is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases.
- The tester needs to have a look inside the source code and find out which unit/chunk of the code is behaving inappropriately.
- Using White-box testing methods, you can derive test cases that

  (1) guarantee that all independent paths within a module have been exercised at least once,

  (2) exercise all logical decisions on their true and false sides,

  (3) execute all loops at their boundaries and within their operational bounds, and

  (4) exercise internal data structures to ensure their validity.

White-Box Testing

Advantages	Disadvantages
As the tester has knowledge of the source code, it becomes very easy to find out which type of data can help in testing the application effectively	Due to the fact that a skilled tester is needed to perform white-box testing, the costs are increased.
It helps in optimizing the code.	Sometimes it is impossible to look into every nook and corner to find out hidden errors that may create problems, as many paths will go untested.
Extra lines of code can be removed which can bring in hidden defects.	It is difficult to maintain white-box testing, as it requires specialized tools like code analyzers and debugging tools.
Due to the tester's knowledge about the code, maximum coverage is attained during test scenario writing.	

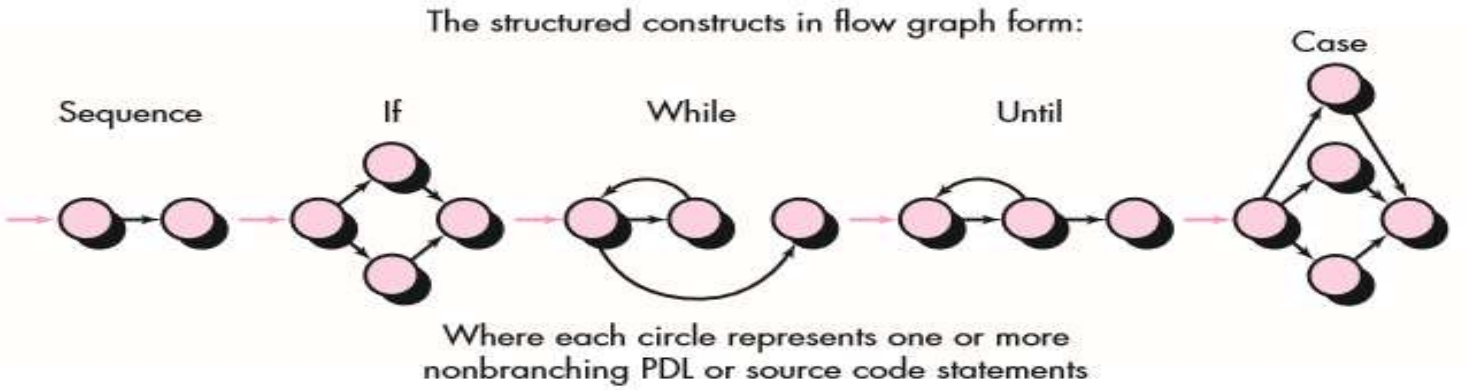
Basis Path Testing

- Basis path testing is a white-box testing technique.
- The basis path method enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.
- Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

- Flow Graph Notation
- Independent Program Paths
- Deriving Test Cases
- Graph Matrices

Flow Graph Notation

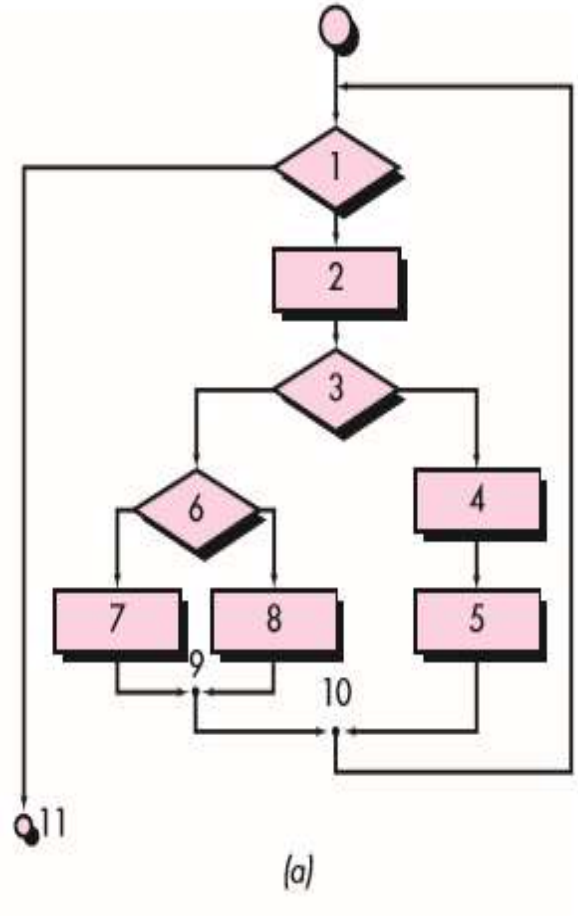
- A simple notation for the representation of control flow, called a flow graph (or program graph).
- The flow graph depicts logical control flow using the notation in the following figure.
- Arrows called edges represent flow of control
- Circles called nodes represent one or more actions.
- Areas bounded by edges and nodes called regions.
- A predicate node is a node containing a condition.
- Any procedural design can be translated into a flow graph.
- Note that compound Boolean expressions at tests generate at least two predicate node and additional arcs.

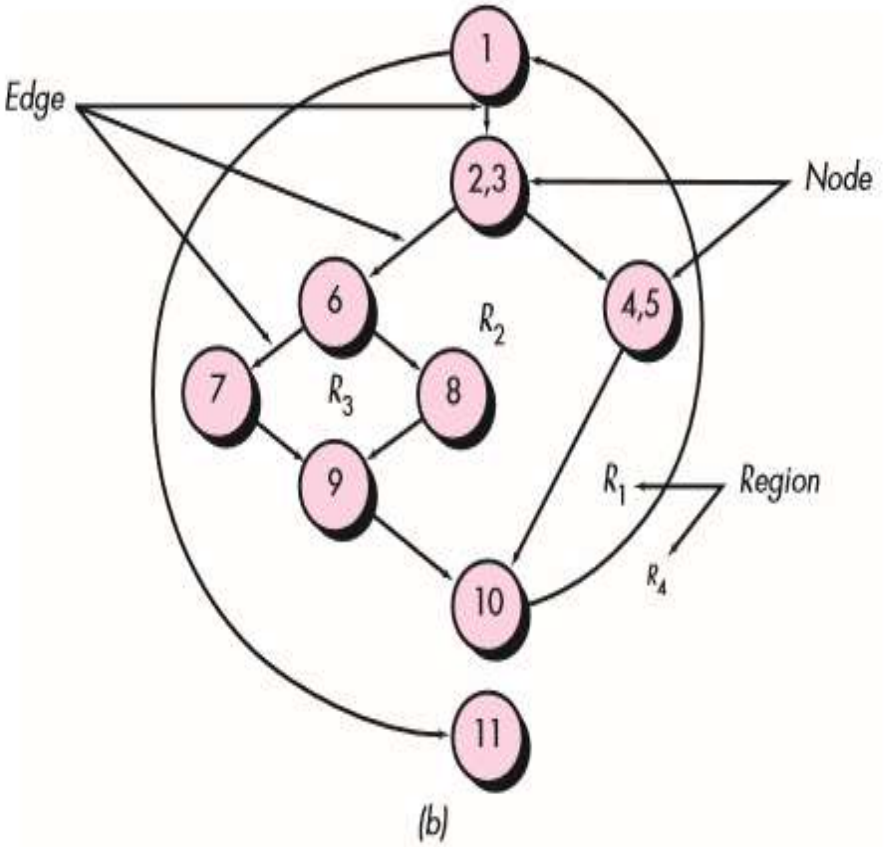


Flow Graph Notation

- To illustrate the use of a flow graph, consider the procedural design representation in Figure.
- Here, Figure (a) flow chart is used to depict program control structure.
- Figure(b) maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart).
- Figure(b), each circle, called a flow graph node, represents one or more procedural statements.
- A sequence of process boxes and a decision diamond can map into a single node.
- The arrows on the flow graph, called edges or links, represent flow of control and are analogous to flowchart arrows.
- An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the flow graph symbol for the if-then-else construct).
- Areas bounded by edges and nodes are called regions. When counting regions.

Flow Graph Notation





Independent Program Paths

- An independent path is any path through the program that introduces at least one new set of processing statements or a new condition.
- Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity if a program.
- When used in the context of the basis path testing method, the value computed for Cyclomatic complexity defines the number of independent paths in the basis set of a program and provides you with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.
- Cyclomatic complexity has a foundation in graph theory and provides you with an extremely useful software metric. Complexity is computed in one of three ways:

1. The number of regions of the flow graph corresponds to the Cyclomatic complexity.
2. Cyclomatic complexity V(G) for a flow graph G is defined as

   V(G) =E – N + 2 where E is the number of flow graph edges and N is the number of flow graph nodes.
3. Cyclomatic complexity V(G) for a flow graph G is also defined as V(G) = P+ 1 where P is the number of predicate nodes contained in the flow graph G.

Deriving Test Cases

The following steps can be applied to derive the basis set:

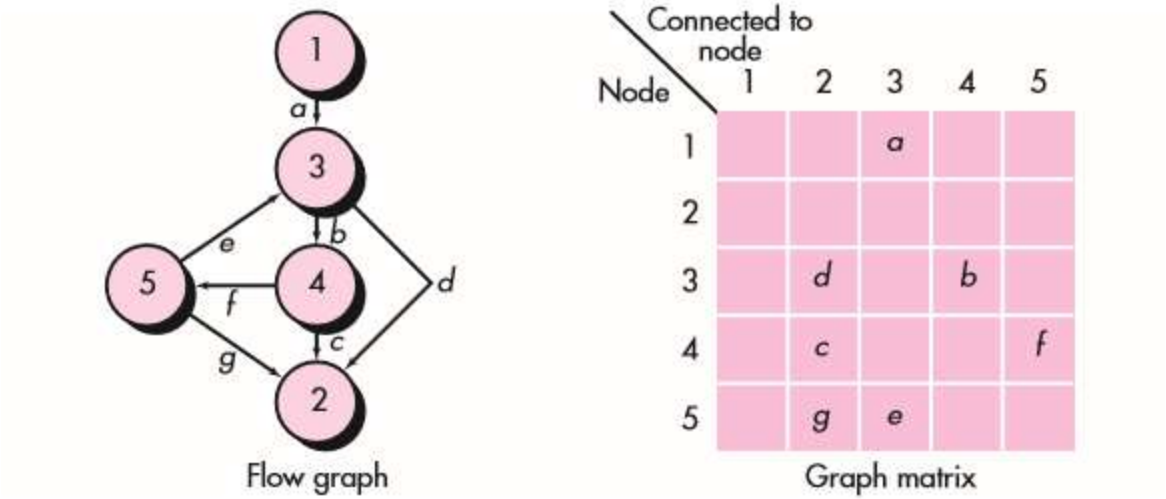
1. Using the design or code as a foundation, draw a corresponding flow graph.
2. Determine the Cyclomatic complexity of the resultant flow graph.
3. Determine a basis set of linearly independent paths.
4. Prepare test cases that will force execution of each path in the basis set.

Graph Matrices:

- A data structure, called a graph matrix, can be quite useful for developing a software tool that assists in basis path testing.
- A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph.
- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.

Graph Matrices:

- A simple example of a flow graph and its corresponding graph matrix is shown in Figure.



- Referring to the figure, each node on the flow graph is identified by numbers, while each edge is identified by letters.
- A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge b.

Graph Matrices:

- The graph matrix is nothing more than a tabular representation of a flow graph.
- By adding a link weight to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing.
- The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist).

Control Structure Testing

- Although basis path testing is simple and highly effective, it is not sufficient in itself.
- Other variations on control structure testing necessary. These broaden testing coverage and improve the quality of white-box testing.

1. Condition testing:

- Condition testing is a test-case design method that exercises the logical conditions contained in a program module.
- A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator.
- A relational expression takes the form

E1<relational-operator> E2

- Where E1 and E2 are arithmetic expressions and <relational-operator> is one of the following: $<, \leq, =, \neq$ (nonequality), $>$, or $\geq$.
- A compound condition is composed of two or more simple conditions, Boolean operators, and parentheses.
- The condition testing method focuses on testing each condition in the program to ensure that it does not contain errors.

Control Structure Testing

2. Data Flow Testing

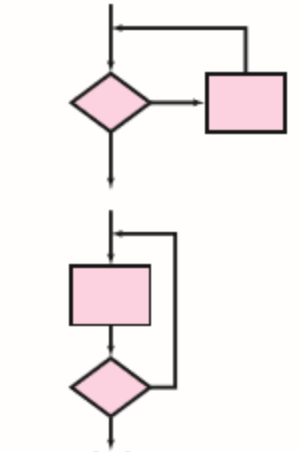
- The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program. T
- To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables.
- For a statement with S as its statement number,
  - DEF(S)= {X | statement S contains a definition of X}
  - USE(S) = {X | statement S contains a use of X}
- If statement S is an if or loop statement, its DEF set is empty and its USE set is based on the condition of statement S. The definition of variable X at statement S is said to be live at statement S' if there exists a path from statement S to statement S' that contains no other definition of X.

Control Structure Testing

3. Loop Testing

- Loops are the cornerstone for the vast majority of all algorithms implemented in software.
- Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs.
- Four different classes of loops can be defined:

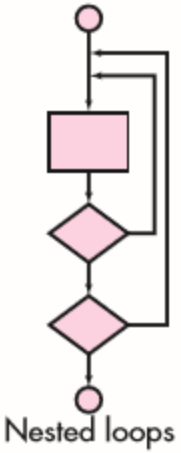
1. **Simple loops**: The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.
   1. Skip the loop entirely.
   2. Only one pass through the loop.
   3. Two passes through the loop.
   4. m passes through the loop where m < n.
   5. n - 1, n, n + 1 passes through the loop.


Simple loops

Control Structure Testing

2. **Nested loops:** If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases.

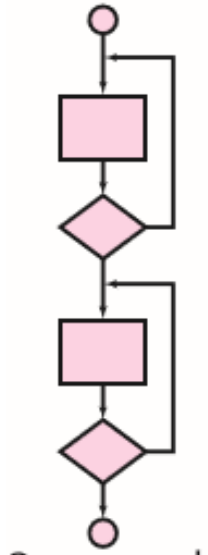
- Beizer suggests an approach that will help to reduce the number of tests:
  1. Start at the innermost loop. Set all other loops to minimum values.
  2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
  3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
  4. Continue until all loops have been tested.


Nested loops

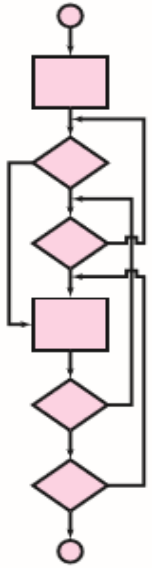
Control Structure Testing

3. Concatenated loops: In the concatenated loops, if two loops are independent of each other then they are tested using simple loops or else test them as nested loops. However if the loop counter for one loop is used as the initial value for the others, then it will not be considered as an independent loops.

4. Unstructured loops: Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.



Concatenated loops



Unstructured loops

Black-Box Testing

- Black-box testing, also called behavioral testing.
- It focuses on the functional requirements of the software.
- Black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program.
- Black-box testing attempts to find errors in the following categories:

  (1) incorrect or missing functions

  (2) interface errors

  (3) errors in data structures or external database access

  (4) behavior or performance errors

  (5) initialization and termination errors.

Black – Box Testing Techniques

1. Graph-Based Testing Methods **2. Equivalence Partitioning**

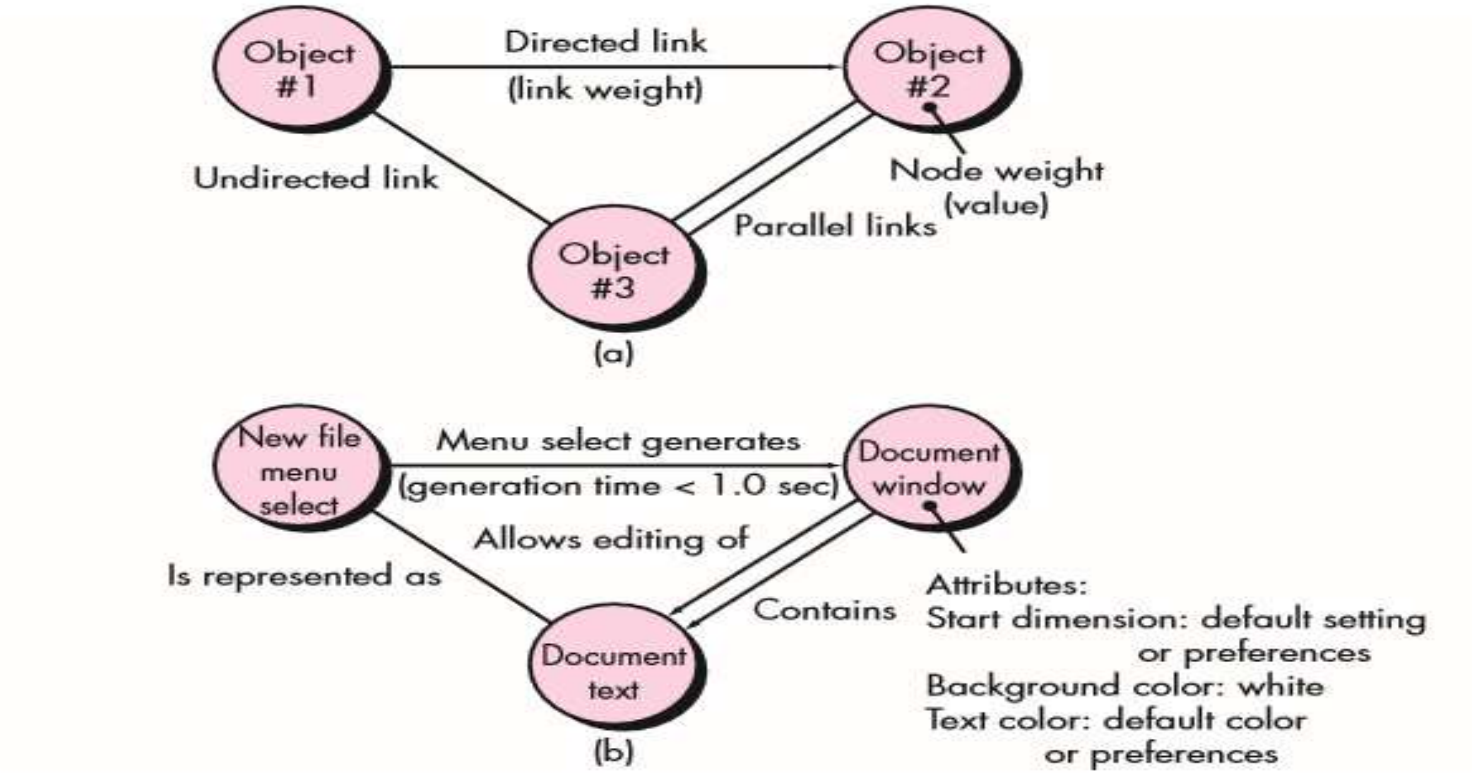
3. Boundary Value Analysis **4. Orthogonal Array Testing**

Black-Box Testing

1. Graph-Based Testing Methods

- The first step in black-box testing is to understand the objects5 that are modeled in software and the relationships that connect these objects.
- Next step is to define a series of tests that verify "all objects have the expected relationship to one another.
- To accomplish these steps, create a **graph**—a collection of nodes that represent objects, **links** that represent the relationships between objects, **node weights** that describe the properties of a node (e.g., a specific data value or state behavior), and **link weights** that describe some characteristic of a link.
- The symbolic representation of a graph is shown in below Figure.
- **Nodes** are represented as circles connected by links that take a number of different forms.
- **A directed link** (represented by an arrow) indicates that a relationship moves in only one direction.
- **A bidirectional link**, also called a symmetric link, implies that the relationship applies in both directions.
- **Parallel links** are used when a number of different relationships are established between graph nodes

Black-Box Testing



Black-Box Testing

2. Equivalence Partitioning

- Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived.
- Test-case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition.
- Equivalence classes may be defined according to the following guidelines:
  - ❖ If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
  - ❖ If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
  - ❖ If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
  - ❖ If an input condition is Boolean, one valid and one invalid class are defined.

Black-Box Testing

3.Boundary Value Analysis

- A greater number of errors occurs at the boundaries of the input domain rather than in the “center of input domain.
- For this reason that boundary value analysis (BVA) has been developed as a testing technique
- Boundary value analysis leads to a selection of test cases that exercise bounding values.
- BVA leads to the selection of test cases at the “edges” of the class. Rather than focusing solely on input conditions.
- BVA derives test cases from the output domain also.
- Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

1. If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.

Black-Box Testing

3. Apply guidelines 1 and 2 to out put conditions. For example, assume that a temperature versus pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum(and minimum)allowable number of table entries.
4. If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Most software engineers intuitively perform BVA to some degree. By applying these guidelines, boundary testing will be more complete, thereby having a higher likelihood for error detection.

Black-Box Testing

4. Orthogonal Array Testing

- Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing.
- The orthogonal array testing method is particularly useful in finding **region faults—**an error category associated with faulty logic within a software compone
- **For example**, when a train ticket has to be verified, the factors such as - the number of passengers, ticket number, seat numbers and the train numbers has to be tested, which becomes difficult when a tester verifies input one by one. Hence, it will be more efficient when he combines more inputs together and does testing. Here, use the Orthogonal Array testing method.
- When orthogonal array testing occurs, an **L9 orthogonal array** of test cases is created.
- The L9 orthogonal array has a “balancing property”.
- That is, test cases (represented by dark dots in the figure) are “dispersed uniformly throughout the test domain,” as illustrated in the right-hand cube in Figure.

Black-Box Testing

- To illustrate the use of the L9 orthogonal array, consider the send function for a fax application.
- Four parameters, P1, P2, P3, and P4, are passed to the send function. Each takes on three discrete values. For example, P1 takes on values:
- P1 = 1, send it now : P1 = 2, send it one hour later : P1 = 3, send it after midnight
- P2, P3, and P4 would also take on values of 1, 2, and 3, signifying other send functions.
- If a "one input item at a time" testing strategy were chosen, the following sequence of tests (P1,P2,P3,P4) would be specified: (1,1,1,1),(2,1,1,1),(3,1,1,1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), and (1, 1, 1, 3).
- The orthogonal array testing approach enables you to provide good test coverage with far fewer test cases than the exhaustive strategy. An L9 orthogonal array for the fax send function is illustrated in Figure

Test case	Test parameters			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Regression testing

- When any modification or changes are done to the application or even when any small change is done to the code then it can bring unexpected issues. Along with the new changes it becomes very important to test whether the existing functionality is intact or not. This can be achieved by doing the **regression testing.**
- The purpose of the regression testing is to find the bugs which may get introduced accidentally because of the new changes or modification.
- During confirmation testing the defect got fixed and that part of the application started working as intended. But there might be a possibility that the fix may have introduced or uncovered a different defect elsewhere in the software. The way to detect these '**unexpected side-effects**' of fixes is to do regression testing.
- This also ensures that the bugs found earlier are NOT creatable.
- Usually the regression testing is done by automation tools because in order to fix the defect the same test is carried out again and again and it will be very tedious and time consuming to do it manually.
- During regression testing the test cases are prioritized depending upon the changes done to the feature or module in the application. The feature or module where the changes or modification is done that entire feature is taken into priority for testing.

Regression testing

- This testing becomes very important when there are continuous modifications or enhancements done in the application or product. These changes or enhancements should NOT introduce new issues in the existing tested code.
- This helps in maintaining the quality of the product along with the new changes in the application.
- **Example:**
- Let's assume that there is an application which maintains the details of all the students in school. This application has four buttons Add, Save, Delete and Refresh. All the buttons functionalities are working as expected. Recently a new button 'Update' is added in the application. This 'Update' button functionality is tested and confirmed that it's working as expected. But at the same time it becomes very important to know that the introduction of this new button should not impact the other existing buttons functionality. Along with the 'Update' button all the other buttons functionality are tested in order to find any new issues in the existing code. This process is known as regression testing.

Regression testing

When to use Regression testing it:

1. Any new feature is added
2. Any enhancement is done
3. Any bug is fixed
4. Any performance related issue is fixed

Advantages of Regression testing:

- It helps us to make sure that any changes like bug fixes or any enhancements to the module or application have not impacted the existing tested code.
- It ensures that the bugs found earlier are NOT creatable.
- Regression testing can be done by using the automation tools
- It helps in improving the quality of the product.

Disadvantages of Regression testing:

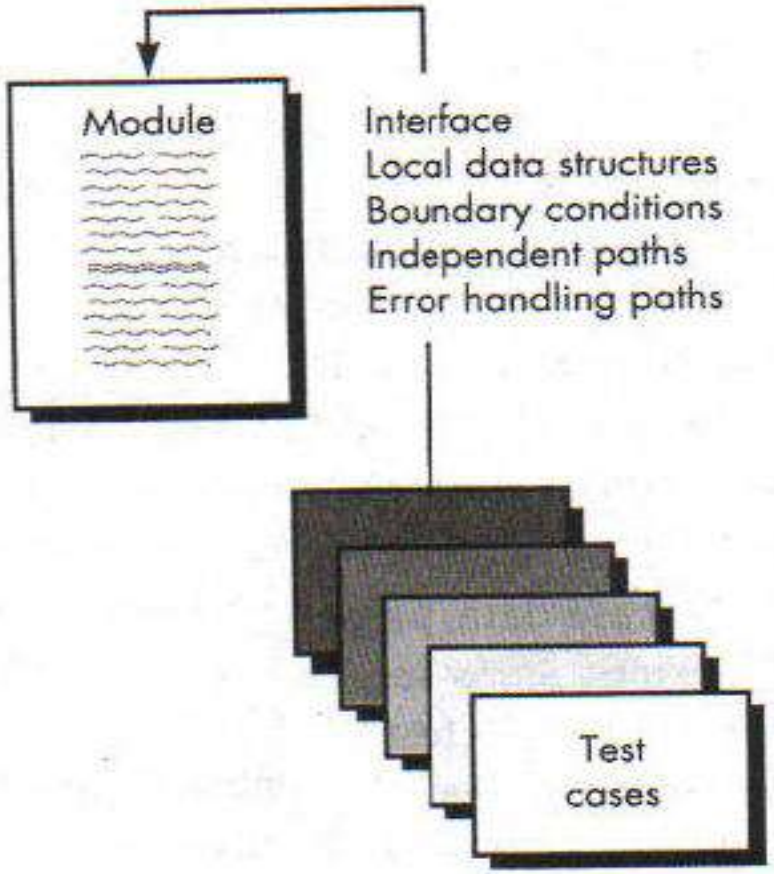
- If regression testing is done without using automated tools then it can be very tedious and time consuming because here we execute the same set of test cases again and again.
- Regression test is required even when a very small change is done in the code because this small modification can bring unexpected issues in the existing functionality.

Unit Testing

- Unit testing focuses verification effort on the smallest unit of software design—the software component or module
- **Targets for Unit Test Cases**
  - **Module interface**
    - Ensure that information flows properly into and out of the module
  - **Local data structures**
    - Ensure that data stored temporarily maintains its integrity during all steps in an algorithm execution
  - **Boundary conditions**
    - Ensure that the module operates properly at boundary values established to limit or restrict processing
  - **Independent paths (basis paths)**
    - Paths are exercised to ensure that all statements in a module have been executed at least once
  - **Error handling paths**
    - Ensure that the algorithms respond correctly to specific error conditions

Unit Testing

- **Targets for Unit Test Cases**
  -



Unit Testing

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

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

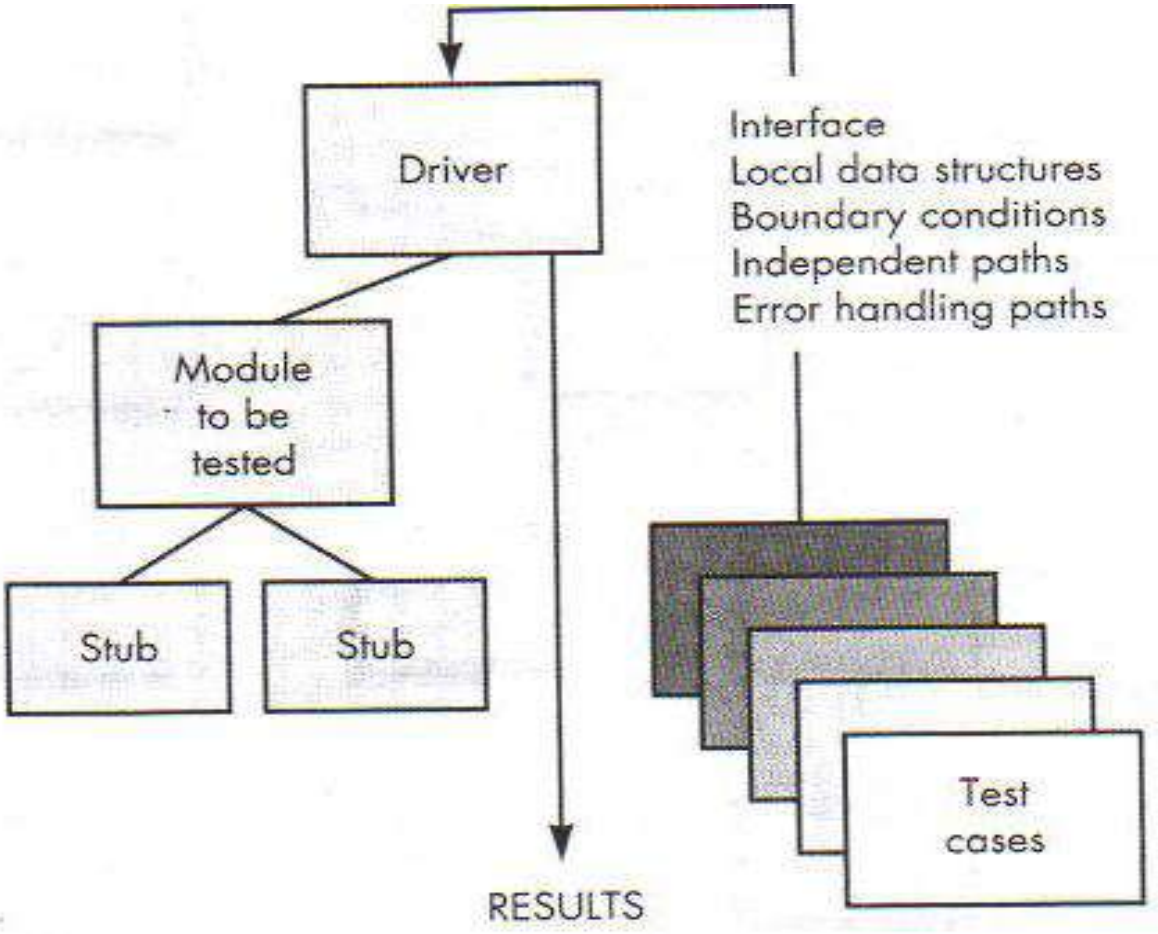
Unit Testing

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

Unit Testing

- **Unit test procedures**
  - **Because a component is not a stand-alone program, driver and / or stub software must be developed for each unit test.**
- **Driver**
  - **A simple main program that accepts test case data, passes such data to the component being tested, and prints the returned results**
- **Stubs**
  - **Serve to replace modules that are subordinate to (called by) the component to be tested**
  - **It uses the module's exact interface, may do minimal data manipulation, provides verification of entry, and returns control to the module undergoing testing**
- **Drivers and stubs both represent overhead**
  - **Both must be written but don't constitute part of the installed software product**

Unit Testing



Integration Testing

Integration testing is systematic technique for constructing the software architecture while at the same time conducting tests to cover to uncover errors associated with interfacing

- **Objective is to take unit tested modules and build a program structure based on the prescribed design**
- **Two Approaches**
  - **Non-incremental Integration Testing**
  - **Incremental Integration Testing**

Integration Testing

- **Non-incremental Integration Testing**
  - Commonly called the “Big Bang” approach
  - All components are combined in advance
  - The entire program is tested as a whole
  - Disadvantages
    - Chaos results
    - Many seemingly-unrelated errors are encountered
    - Correction is difficult because isolation of causes is complicated
    - Once a set of errors are corrected, more errors occur, and testing appears to enter an endless loop
- **Incremental Integration Testing**
  - Three kinds
    - Top-down integration
    - Bottom-up integration
    - Sandwich integration
  - The program is constructed and tested in small increments
  - Errors are easier to isolate and correct
  - Interfaces are more likely to be tested completely
  - A systematic test approach is applied

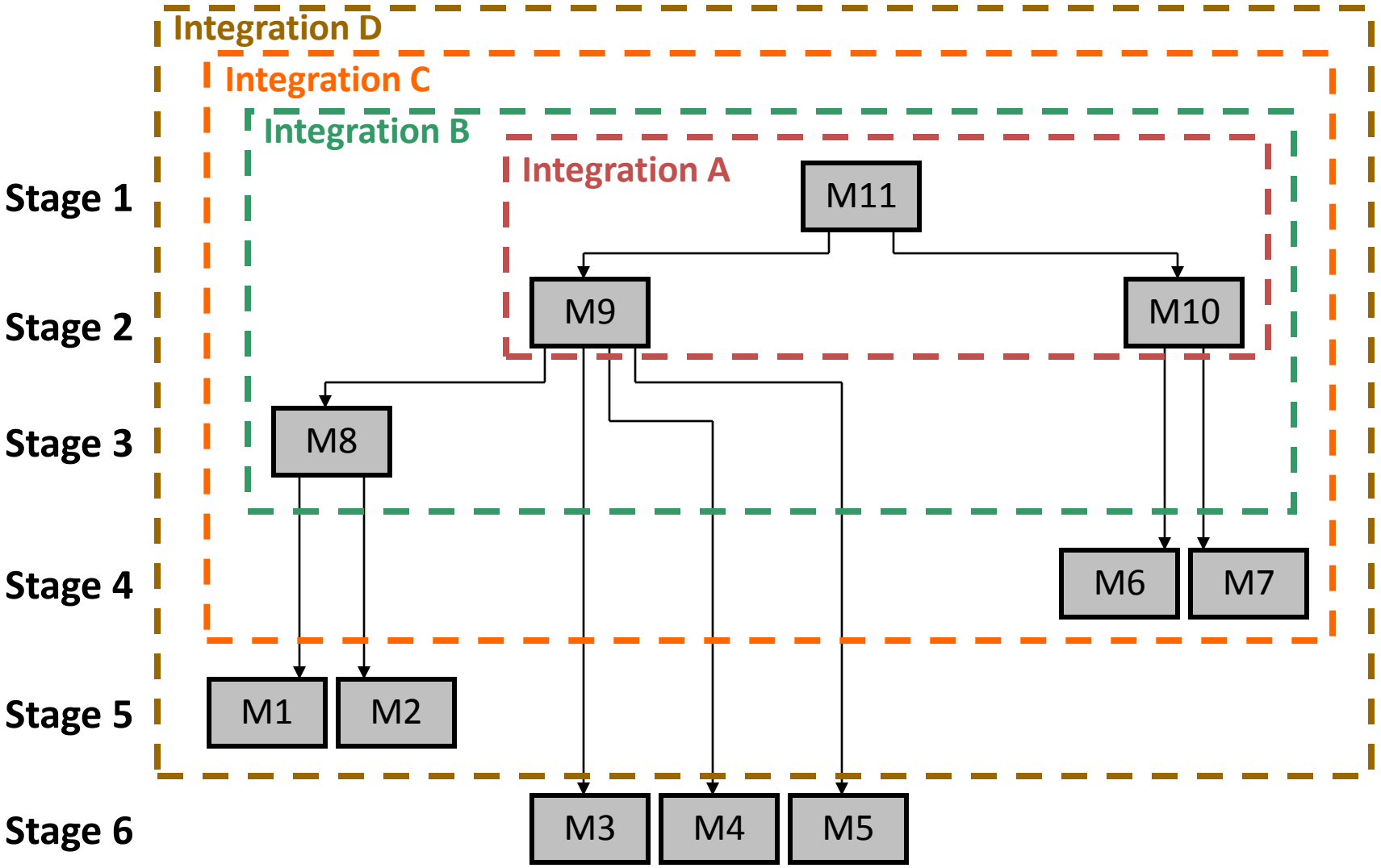
Integration Testing

- **Top-down Integration**
  - Modules are integrated by moving downward through the control hierarchy, beginning with the main module
  - Subordinate modules are incorporated in either a depth-first or breadth-first fashion
    - DF: All modules on a major control path are integrated
    - BF: All modules directly subordinate at each level are integrated
  - The main control module is used as a test driver, and stubs are substituted for all components directly subordinate to the main control module
  - Depending on the integration approach selected subordinate stubs are replaced one at a time with actual components
  - Tests are conducted as each component is integrated
  - On completion of each set of tests, another stub is replaced with real component

Integration Testing

- **Top-down Integration**
  - **Advantages**
    - **This approach verifies major control or decision points early in the test process**
  - **Disadvantages**
    - **Stubs need to be created to substitute for modules that have not been built or tested yet; this code is later discarded**
    - **Because stubs are used to replace lower level modules, no significant data flow can occur until much later in the integration/testing process**

Top-down Integration



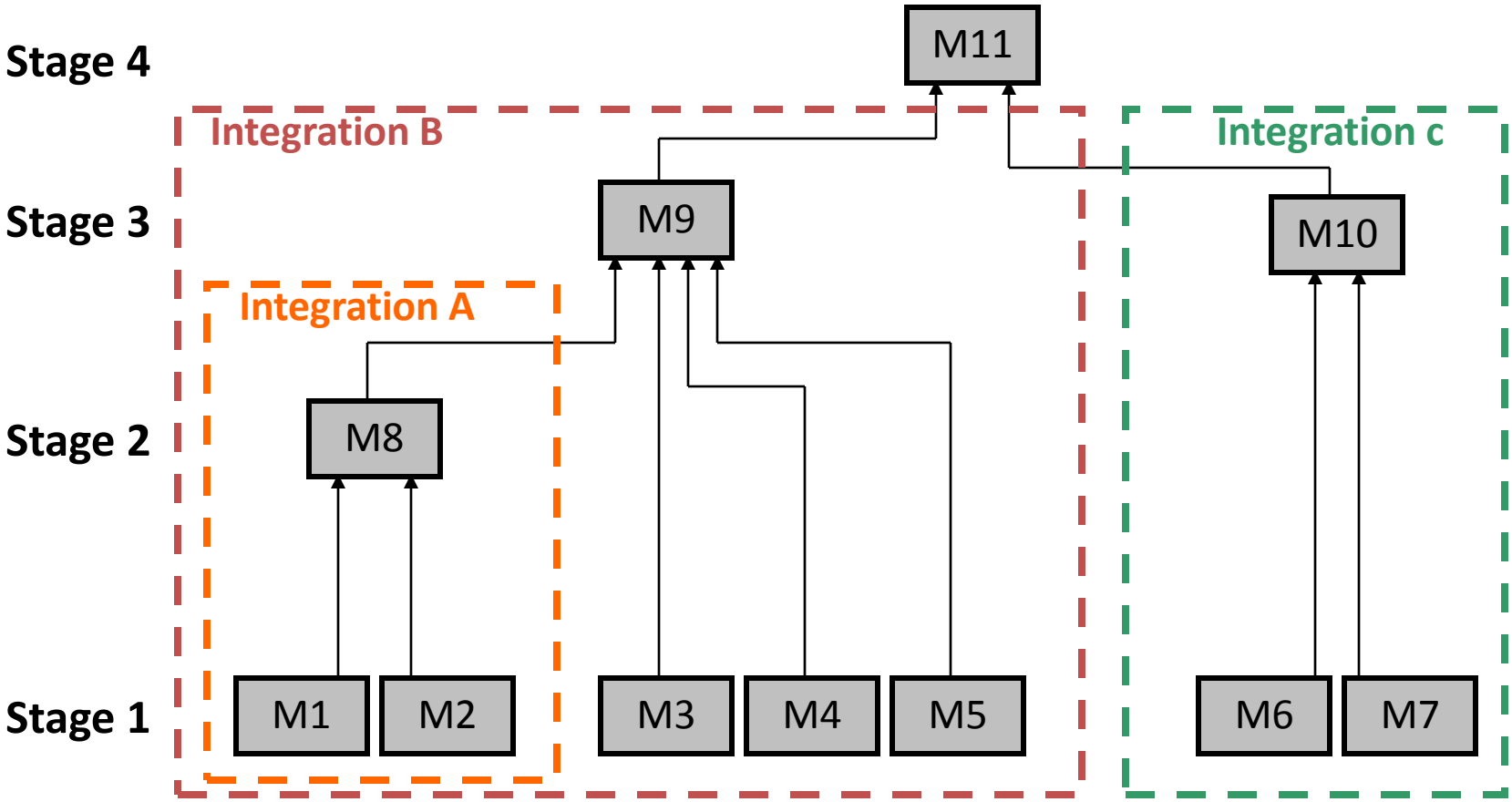
Integration Testing

- **Bottom-up Integration**
  - Integration and testing starts with the most atomic modules (i.e., components at the lowest levels in the program structure) in the control hierarchy
  - Begins construction and testing with atomic modules. As components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated
  - Low-level components are combined into clusters that perform a specific software sub-function
  - A driver is written to coordinate test case input and output
  - The cluster is tested
  - Drivers are removed and clusters are combined moving upward in the program structure
  - As integration moves upward, the need for separate test drivers lessens. If the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified

Integration Testing

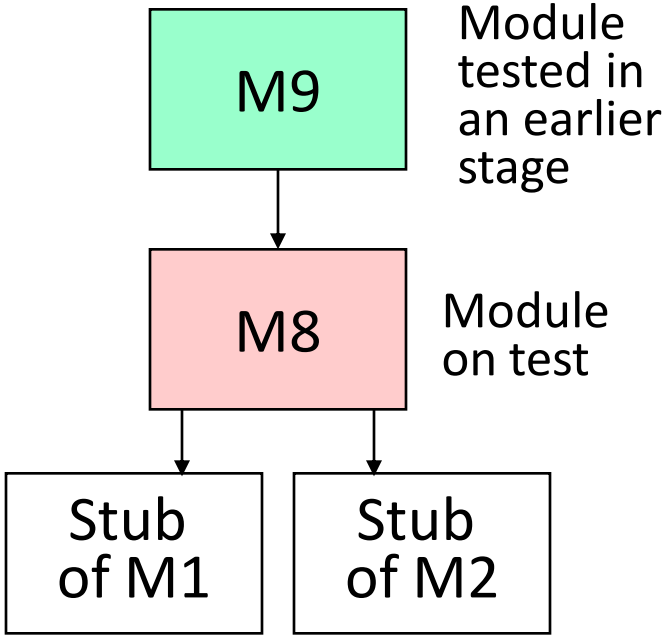
- **Bottom-up Integration**
  - **Advantages**
    - **This approach verifies low-level data processing early in the testing process**
    - **Need for stubs is eliminated**
  - **Disadvantages**
    - **Driver modules need to be built to test the lower-level modules; this code is later discarded or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

Bottom-up Integration

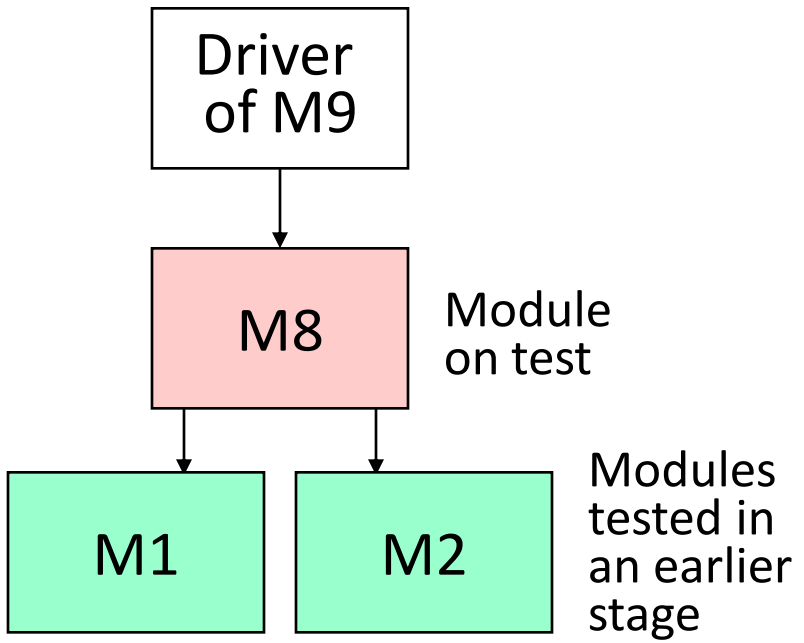


Use of stubs and drivers for incremental testing

Top-down testing of module M8



Bottom-up testing of module M8



Sandwich Integration

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

Regression Testing

- Each time a new module is added as part of integration testing, the software changes.
- New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly.
- Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects
- The regression test suite contains

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

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

• Test that focus on the software components that have been changed

Smoke Testing

- Taken from the world of hardware
  - Power is applied and a technician checks for sparks, smoke, or other dramatic signs of fundamental failure
- Designed as a pacing mechanism for time-critical projects
  - Allows the software team to assess its project on a frequent basis
- Includes the following activities
  - The software is compiled and linked into a build
  - A series of breadth tests is designed to expose errors that will keep the build from properly performing its function

Smoke Testing

- The goal is to uncover “show stopper” errors that have the highest likelihood of throwing the software project behind schedule
  - The build is integrated with other builds and the entire product is smoke tested daily
- Daily testing gives managers and practitioners a realistic assessment of the progress of the integration testing
  - After a smoke test is completed, detailed test scripts are executed

Smoke Testing

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

Validation Testing

- Validation testing follows integration testing
  - The distinction between conventional and object-oriented software disappears
  - Focuses on user-visible actions and user-recognizable output from the system
  - Demonstrates conformity with requirements

1. Validation-Test Criteria

- Designed to ensure that
  - All functional requirements are satisfied
  - All behavioral characteristics are achieved
  - All performance requirements are attained
  - Documentation is correct
  - Usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability)

Validation Testing

- After each validation test
  - The function or performance characteristic conforms to specification and is accepted
  - A deviation from specification is uncovered and a deficiency list is created
- Deviation or error discovered at this stage in a project can rarely be corrected prior to scheduled delivery

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

Validation Testing

- It is virtually impossible for a software developer to foresee how the customer will really use a program.
- Instructions for use may be misinterpreted; strange combinations of data may be regularly used; output that seemed clear to the tester may be unintelligible to a user in the field.
- When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements. Conducted by the end user rather than software engineers, an acceptance test can range from an informal "test drive" to a planned and systematically executed series of tests.
- In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.
- If software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one. Most software product builders use a process called alpha and beta testing to uncover errors that only the end user seems able to find.

Validation Testing

3. Alpha and Beta Testing

- Alpha Testing
  - Conducted at the developer's site by end users
  - Software is used in a natural setting with developers watching intently
  - Testing is conducted in a controlled environment
- Beta Testing
  - Conducted at end-user sites
  - Developer is generally not present
  - It serves as a live application of the software in an environment that cannot be controlled by the developer Alpha and Beta Testing
- Beta Testing
  - The end-user records all problems that are encountered and reports these to the developers at regular intervals

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

System Testing

- System testing is a series of different test whose primary purpose is to fully exercise the computer-based system.
- Each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.

1. Recovery testing

- Tests for recovery from system faults
- Forces the software to fail in a variety of ways and verifies that recovery is properly performed
- Tests reinitialization, checkpointing mechanisms, data recovery, and restart for correctness
- If recovery is automatic, reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness
- If recovery requires human intervention, the mean-time-to-repair is evaluated to determine whether it is within acceptable limits

System Testing

2. Security testing

- Verifies that protection mechanisms built into a system will, in fact, protect it from improper access
- During security testing, the tester plays the role of the individual who desires to penetrate the system.
- Anything goes! The tester may attempt to acquire passwords through external clerical means.
- may attack the system with custom software designed to break down any defenses that have been constructed

3. Stress testing

- Executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.
- Stress tests are designed to confront programs with abnormal situations. In essence, the tester who performs stress testing asks: “How high can we crank this up before it fails?”
- A variation of stress testing is a technique called sensitivity testing.

System Testing

- A very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation.
- Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

4. Performance Testing

- Performance testing is designed to test the run-time performance of software within the context of an integrated system.
- Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as tests are conducted
- Performance tests are often coupled with stress testing and usually requires both hardware and software instrumentation.
- That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion

System Testing

5. Deployment Testing

- **software must execute on a variety of platforms and under more than one operating system environment.**
- **Deployment testing, sometimes called configuration testing, exercises the software in each environment in which it is to operate.**
- **Deployment testing examines all installation procedures and specialized installation software (e.g., “installers”) that will be used by customers, and all documentation that will be used to introduce the software to end users.**

The Art of Debugging

- **Debugging occurs as a consequence of successful testing. When a test case uncovers an error, debugging is an action that results in the removal of the error.**

1. **The debugging process**

   The debugging process attempts to match symptom with cause, thereby leading to error correction.

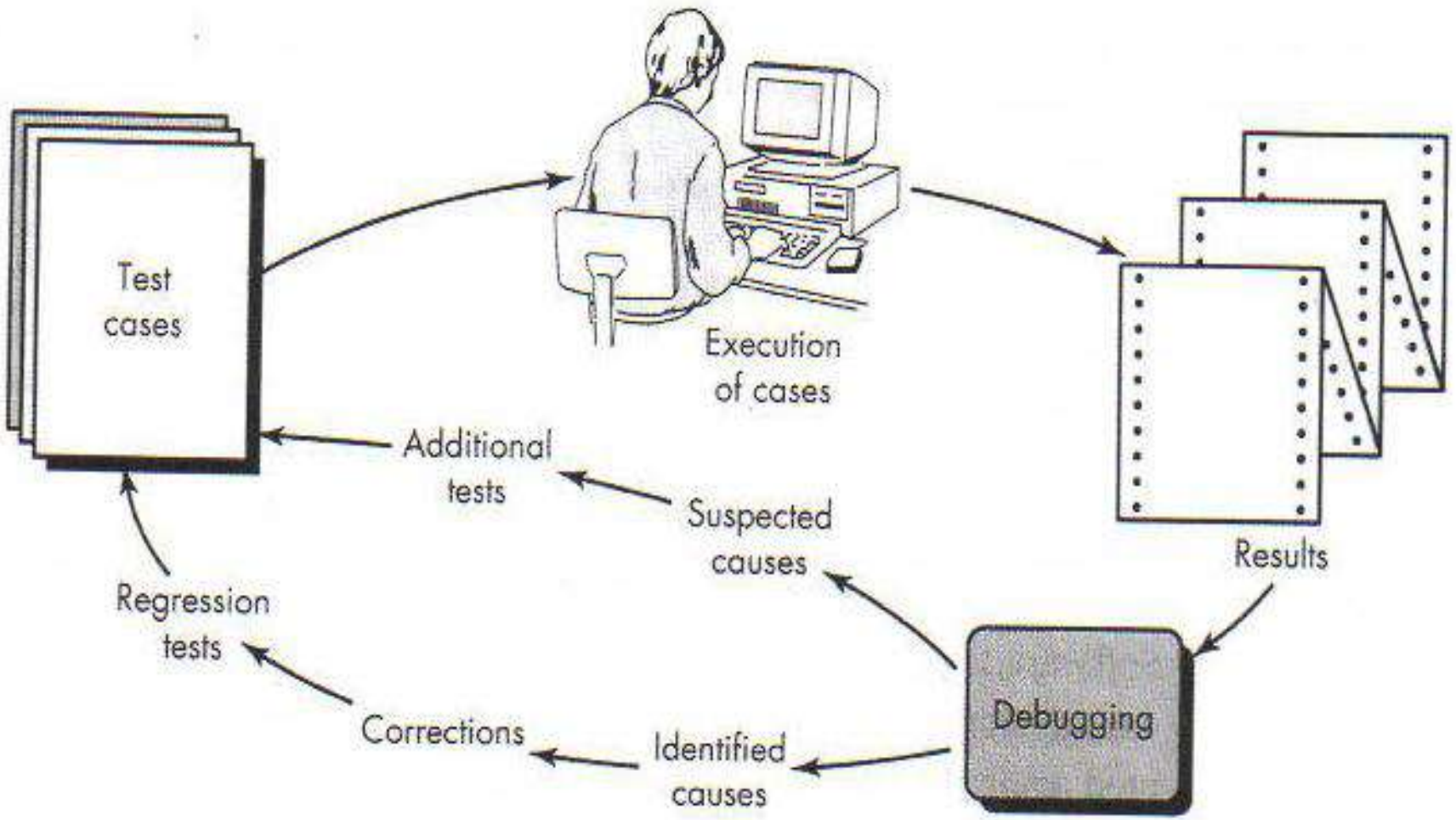
The debugging process will usually have one of two outcomes:

(1) the cause will be found and corrected or

(2) the cause will not be found. In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

- **Debugging process beings with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is observed**
- **Debugging attempts to match symptom with cause , thereby leading to error correction .**

The Art of Debugging



The Art of Debugging

- **Characteristics of bugs**
  - **The symptom and the cause may be geographically remote.**
  - **The symptom may disappear (temporarily) when another error is corrected**
  - **The symptom may actually be caused by non-errors**
  - **The symptom may be caused by human error that is not easily traced**
  - **The symptom my be a result of timing problems, rather than processing problems**
  - **It may be difficult to accurately reproduce input conditions**
  - **The symptom may be intermittent.**
  - **The symptom may be due to causes that are distributed across a number of tasks running on different processors**

The Art of Debugging

2. Psychological Considerations

- Unfortunately, there appears to be some evidence that debugging prowess is an innate human trait. Some people are good at it and others aren't.
- Although experimental evidence on debugging is open to many interpretations, large variances in debugging ability have been reported for programmers with the same education and experience.

3. Debugging Strategies

- Objective of debugging is to find and correct the cause of a software error or defect.
- Bugs are found by a combination of systematic evaluation, intuition, and luck.
- Debugging methods and tools are not a substitute for careful evaluation based on a complete design model and clear source code
- There are three main debugging strategies

1. Brute force
2. Backtracking
3. Cause elimination

The Art of Debugging

- Brute Force
  - Most commonly used and least efficient method for isolating the cause of a software error
  - Used when all else fails
  - Involves the use of memory dumps, run-time traces, and output statements
  - Leads many times to wasted effort and time
- Backtracking
  - Can be used successfully in small programs
  - The method starts at the location where a symptom has been uncovered
  - The source code is then traced backward (manually) until the location of the cause is found
  - In large programs, the number of potential backward paths may become unmanageably large

The Art of Debugging

- **Cause Elimination**
  - **Involves the use of induction or deduction and introduces the concept of binary partitioning**
    - **Induction (specific to general): Prove that a specific starting value is true; then prove the general case is true**
    - **Deduction (general to specific): Show that a specific conclusion follows from a set of general premises**
  - **Data related to the error occurrence are organized to isolate potential causes**
  - **A cause hypothesis is devised, and the aforementioned data are used to prove or disprove the hypothesis**
  - **Alternatively, a list of all possible causes is developed, and tests are conducted to eliminate each cause**
  - **If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug**

The Art of Debugging

4. Correcting the Error

- Once a bug has been found, it must be corrected.
- But the correction of a bug can introduce other errors and therefore do more harm than good.
- Van Vleck suggests three simple questions that you should ask before making the "correction" that removes the cause of a bug.
- Three Questions to ask Before Correcting the Error
  - Is the cause of the bug reproduced in another part of the program?
    - Similar errors may be occurring in other parts of the program
  - What next bug might be introduced by the fix that I'm about to make?
    - The source code (and even the design) should be studied to assess the coupling of logic and data structures related to the fix
  - What could we have done to prevent this bug in the first place?
    - This is the first step toward software quality assurance
    - By correcting the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs

Software Implementation Techniques

Coding Practices:

- **Best coding practices are a set of informal rules that the software development community has learned over time which can help improve the quality of software**
- **Many computer programs remain in use for far longer than the original authors ever envisaged (sometimes 40 years or more) so any rules need to facilitate both initial development and subsequent maintenance and enhancement by people other than the original authors.**
- **In Ninety-ninety rule, Tim Cargill is credited with this explanation as to why programming projects often run late: "The first 90% of the code accounts for the first 90% of the development time. The remaining 10% of the code accounts for the other 90% of the development time." Any guidance which can redress this lack of foresight is worth considering.**
- **The size of a project or program has a significant effect on error rates, programmer productivity, and the amount of management needed**

  a) Maintainability b) Dependability (c) Efficiency

  d) Usability.

Software Implementation Techniques

Refactoring:

- **Refactoring is usually motivated by noticing a code smell.**
- **For example the method at hand may be very long, or it may be a near duplicate of another nearby method.**
- **Once recognized, such problems can be addressed by *refactoring* the source code, or transforming it into a new form that behaves the same as before but that no longer "smells".**

There are two general categories of benefits to the activity of refactoring.

Maintainability. It is easier to fix bugs because the source code is easy to read and the intent of its author is easy to grasp. This might be achieved by reducing large monolithic routines into a set of individually concise, well-named, single-purpose methods. It might be achieved by moving a method to a more appropriate class, or by removing misleading comments.

Extensibility. It is easier to extend the capabilities of the application if it uses recognizable design patterns, and it provides some flexibility where none before may have existed.

- **Before applying a refactoring to a section of code, a solid set of automatic unit tests is needed. The tests are used to demonstrate that the behavior of the module is correct before the refactoring.**
- **The tests can never prove that there are no bugs, but the important point is that this process can be cost-effective: good unit tests can catch enough errors to make them worthwhile and to make refactoring safe enough.**

UNIT- V
PROJECT MANAGEMENT

Project Management: Estimation: FP based, LOC based, make/buy decision; COCOMO II: Planning, project plan, planning process, RFP risk management, identification, projection; RMMM: Scheduling and tracking, relationship between people and effort, task set and network, scheduling; EVA: Process and project metrics.

Estimation

- Software cost and effort estimation will never be an exact science. Too many variables—human, technical, environmental, political—can affect the ultimate cost of software and effort applied to develop it.

- **To achieve reliable cost and effort estimates, a number of options arise:**
  1. Delay estimation until late in the project (obviously, we can achieve 100 percent accurate estimates after the project is complete!).
  2. Base estimates on similar projects that have already been completed.
  3. Use relatively simple decomposition techniques to generate project cost and effort estimates.
  4. Use one or more empirical models for software cost and effort estimation.

Unfortunately, the first option, however attractive, is not practical. Cost estimates must be provided up-front. However, recognize that the longer you wait, the more you know, and the more you know, the less likely you are to make serious errors in your estimates.

The second option can work reasonably well, if the current project is quite similar to past efforts and other project influences (e.g., the customer, business conditions, the software engineering environment, deadlines) are roughly equivalent. Unfortunately, past experience has not always been a good indicator of future results.

Estimation

- A model is based on experience (historical data) and takes the form

$$d = f(vi)$$

- where d is one of a number of estimated values (e.g., effort, cost, project duration) and vi are selected independent parameters (e.g., estimated LOC or FP).

Function Point based Estimation :

- A **Function Point** (FP) is a unit of measurement to express the amount of business functionality, an information system (as a product) provides to a user. FPs measure software size. They are widely accepted as an industry standard for functional sizing.
- Function point analysis is a method of quantifying the size and complexity of a software system in terms of the functions that the system delivers to the user
- It is independent of the computer language, development methodology, technology or capability of the project team used to develop the application
- Function point analysis is designed to measure business applications (not scientific applications)

Estimation

- Scientific applications generally deal with complex algorithms that the function point method is not designed to handle
- Function points are independent of the language, tools, or methodologies used for implementation (ex. Do not take into consideration programming languages, DBMS, or processing hardware)
- Function points can be estimated early in analysis and design
- **Uses of Function Point:**
- Measure productivity (ex. Number of function points achieved per work hour expended)
- Estimate development and support (cost benefit analysis, staffing estimation)
- Monitor outsourcing agreements (Ensure that the outsourcing entity delivers the level of support and productivity gains that they promise)
- Drive IS related business decisions (Allow decisions regarding the retaining, retiring and redesign of applications to be made)
- Normalize other measures (Other measures, such as defects, frequently require the size in function points)

Estimation

- **LOC based estimation**
- **Source lines of code** (**SLOC**), also known as **lines of code** (**LOC**), is a software metric used to measure the size of a computer program by counting the number of lines in the text of the program's source code.
- SLOC is typically used to predict the amount of effort that will be required to develop a program, as well as to estimate programming productivity or maintainability once the software is produced.
- Lines used for commenting the code and header file are ignored.
- **Two major types of LOC:**

1. Physical LOC

- Physical LOC is the count of lines in the text of the program's source code including comment lines.
- Blank lines are also included unless the lines of code in a section consists of more than 25% blank lines.

2. Logical LOC

- Logical LOC attempts to measure the number of executable statements, but their specific definitions are tied to specific computer languages.
- Ex: Logical LOC measure for C-like programming languages is the number of statement-terminating semicolons(;)

Estimation

LOC-based Estimation

The problems of lines of code (LOC)

- Different languages lead to different lengths of code
- It is not clear how to count lines of code
- A report, screen, or GUI generator can generate thousands of lines of code in minutes
- Depending on the application, the complexity of code is different.

make/buy decision

- In many software application areas, it is often more cost effective to acquire rather than develop computer software.
- Software engineering managers are faced with a make/ buy decision that can be further complicated by a number of acquisition options.
  (1) Software may be purchased (or licensed) off-the-shelf
  (2) "full-experience" or "partial-experience" software components may be acquired and then modified and integrated to meet specific needs.
  (3) Software may be custom built by an outside contractor to meet the purchaser's specifications.
- In the final analysis the make/buy decision is made based on the following conditions:
  (1) Will the delivery date of the software product be sooner than that for internally developed software?
  (2) Will the cost of acquisition plus the cost of customization be less than the cost of developing the software internally?
  (3) Will the cost of outside support (e.g., a maintenance contract) be less than the cost of internal support?

make/buy decision

Creating a Decision Tree :

- The steps just described can be augmented using statistical techniques such as decision tree analysis.
- For example, considered the figure below it depicts a decision tree for a software based system X. In this case, the software engineering organization can

  (1) build system X from scratch

  (2) reuse existing partial-experience components to construct the system

  (3) buy an available software product and modify it to meet local needs, or
  (4) contract the software development to an outside vendor.

If the system is to be built from scratch, there is a 70 percent probability that the job will be difficult.

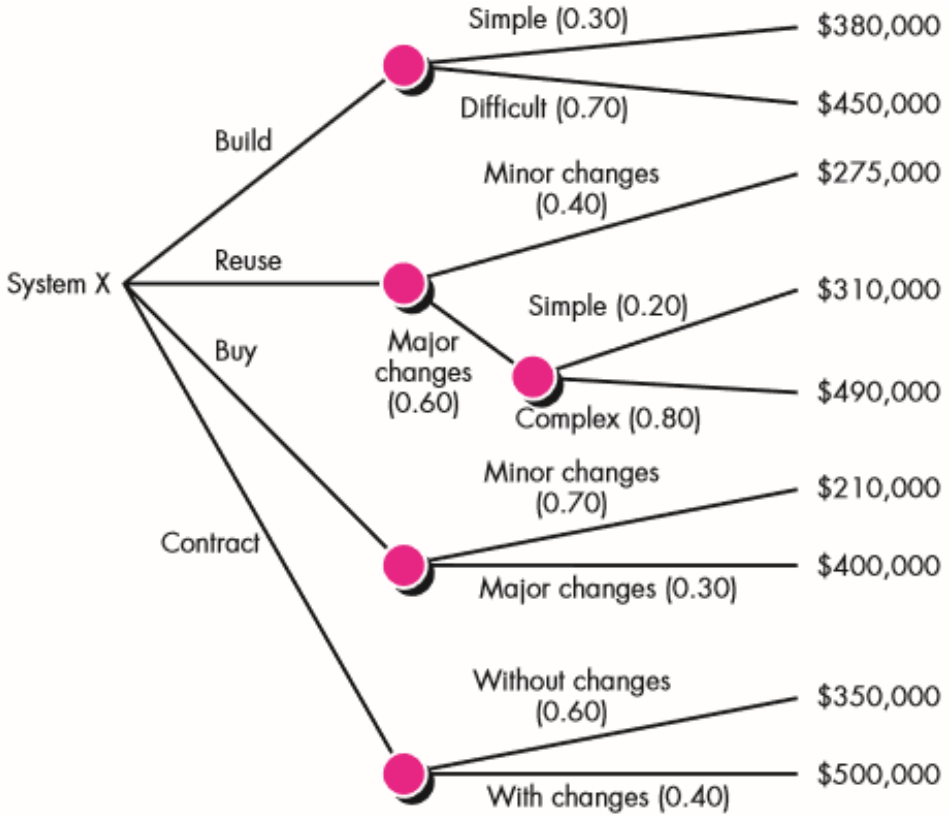
The expected value for cost, computed along any branch of the decision tree, is:

$$\text{Expected cost} = \Sigma\ (\text{path probability})_i \times (\text{estimated path cost})_i$$

where i is the decision tree path. For the build path.

make/buy decision

- It is important to note, however, that many criteria —not just cost—must be considered during the decision-making process. Availability, experience of the developer/ vendor/contractor, conformance to requirements, local "politics," and the likelihood of change are but a few of the criteria that may affect the ultimate decision to build, reuse, buy, or contract.



make/buy decision

Outsourcing

- Sooner or later, every company that develops computer software asks a fundamental question: “Is there a way that we can get the software and systems we need at a lower price?”
- The answer to this question is not a simple one, and the emotional discussions that occur in response to the question always lead to a single word: **outsourcing**. Regardless of the breadth of focus, the outsourcing decision is often a financial one.
- Outsourcing is extremely simple. Software engineering activities are contracted to a third party who does the work at lower cost and, hopefully, higher quality.
- The decision to outsource can be either **strategic** or **tactical**.
- At the strategic level, business managers consider whether a significant portion of all software work can be contracted to others.
- At the tactical level, a project manager determines whether part or all of a project can be best accomplished by subcontracting the software work.
- On the positive side, cost savings can usually be achieved by reducing the number of software people and the facilities (e.g., computers, infrastructure) that support them.
- On the negative side, a company loses some control over the software that it needs.

COCOMO II

- Barry Boehm [Boe81] introduced a hierarchy of software estimation models bearing the name COCOMO, for Constructive Cost MOdel. The original COCOMO model became one of the most widely used and discussed software cost estimation models in the industry. It has evolved into a more comprehensive estimation model, called COCOMOII.
- COCOMOII is actually a hierarchy of estimation models that address the following areas:
- **Application composition model**. Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.
- **Early design stage model**. Used once requirements have been stabilized and basic software architecture has been established.
- **Post-architecture-stage model**. Used during the construction of the software.
- The COCOMO II models require sizing information.
- Three different sizing options are available as part of the model hierarchy: **object points, function points, and lines of source code**.

COCOMO II

- The COCOMO II application composition model uses object points :
- The object point is an indirect software measure that is computed using counts of the number of

  (1) screens (at the user interface),

  (2) reports

  (3) components likely to be required to build the application.
- Each object instance (e.g., a screen or report) is classified into one of three complexity levels (i.e., simple, medium, or difficult).
- Once complexity is determined, the number of screens, reports, and components are weighted according to the table given below

Object type	Complexity weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL component			10

COCOMO II

- When component-based development or general software reuse is to be applied, the percent of reuse (%reuse) is estimated and the object point count is adjusted:

$$\text{NOP} = (\text{object points}) \times [(100 - \%\text{reuse})/100]$$

where NOP is defined as new object points.

- To derive an estimate of effort based on the computed NOP value, a "productivity rate" must be derived.

$$\text{PROD} = \frac{\text{NOP}}{\text{person-month}}$$

- Once the productivity rate has been determined, an estimate of project effort is computed using,

$$\text{Estimated effort} = \frac{\text{NOP}}{\text{PROD}}$$

Developer's experience/capability	Very low	Low	Nominal	High	Very high
Environment maturity/capability	Very low	Low	Nominal	High	Very high
PROD	4	7	13	25	50

The Project Planning Process

- The **Project Planning Phase** is the second phase in the *project life cycle*. It involves creating of a set of plans to help guide your team through the execution and closure phases of the project.
- The plans created during this phase will help you to manage time, cost, quality, change, risk and issues. They will also help you manage staff and external suppliers, to ensure that you deliver the project on time and within budget.
- The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule.
- In addition, estimates should attempt to define best-case and worst-case scenarios so that project outcomes can be bounded.
- Although there is an inherent degree of uncertainty, the software team embarks on a plan that has been established as a consequence of these tasks.
- Therefore, the plan must be adapted and updated as the project proceeds.
- The **Project Planning Phase** is often the most challenging phase for a Project Manager, as you need to make an educated guess of the staff, resources and equipment needed to complete your project. You may also need to plan your communications and procurement activities, as well as contract any 3rd party suppliers.

Risk Management

- A Hazard is

  Any real or potential condition that can cause injury, illness, or death to personnel; damage to or loss of a system, equipment or property; or damage to the environment. Simpler .... A threat of harm. A hazard can lead to one or several consequences.

- Risk is

  The expectation of a loss or damage (consequence)

  The combined severity and probability of a loss

  The long term rate of loss

  A potential problem (leading to a loss) that may - or may not occur in the future.

- Risk Management is A set of practices and support tools to identify, analyze, and treat risks explicitly.
- Treating a risk means understanding it better, avoiding or reducing it (risk mitigation), or preparing for the risk to materialize.
- Risk management tries to reduce the probability of a risk to occur and the impact (loss) caused by risks.

Risk Management

- Reactive versus Proactive Risk Strategies
- Software risks

➢Reactive versus Proactive Risk Strategies

- The majority of software teams rely solely on reactive risk strategies. At best, a reactive strategy monitors the project for likely risks. Resources are set aside to deal with them, should they become actual problems.
- The software team does nothing about risks until something goes wrong. Then, the team flies into action in an attempt to correct the problem rapidly. This is often called a fire-fighting mode.
- A considerably more intelligent strategy for risk management is to be proactive.
- A proactive strategy begins long before technical work is initiated. Potential risks are identified, their probability and impact are assessed, and they are ranked by importance. Then,
- The software team establishes a plan for managing risk. The primary objective is to avoid risk, but because not all risks can be avoided, the team works to develop a contingency plan that will enable it to respond in a controlled and effective manner.

Software Risks

Risk always involves two characteristics:

- Risk always involves two characteristics: uncertainty—the risk may or may not happen; that is, there are no 100 percent probable risks—and loss—if the risk becomes a reality, unwanted consequences or losses will occur.
- When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk.
- Different categories of risks are follows:

1. *Project risks*
   - ❖ Threaten the project plan. That is, if project risks become real, it is likely that the project schedule will slip and that costs will increase.
   - ❖ Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, stakeholder, and requirements problems and their impact on a software project.

Software Risks

2. *Technical risks*
   - ❖ Threaten the quality and timeliness of the software to be produced.
   - ❖ If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems.
   - ❖ In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors. Technical risks occur because the problem is harder to solve than you thought it would be.
3. *Business risks*
   - ❖ Business risks threaten the viability of the software to be built and often jeopardize the project or the product.
   - ❖ Candidates for the top five business risks are
     (1) building an excellent product or system that no one really wants (market risk)
     (2) building a product that no longer fits into the overall business strategy for the company (strategic risk)
     (3) building a product that the sales force doesn't understand how to sell (sales risk)
     (4) losing the support of senior management due to a change in focus or a change in people (management risk)
     (5) losing budgetary or personnel commitment (budget risks).

Software Risks

Another general categorization of risks has been proposed by Charette.

1. *Known risks* are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment).
2. *Predictable* risks are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced).
3. *Unpredictable risks* are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

Risk Identification

- Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.).
- By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.
- There are two distinct types of risks: **generic risks** and **product-specific risks**.
- **Generic risks** are a potential threat to every software project.
- **Product-specific risks** can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built.
- To identify **product-specific risks**, the project plan and the software statement of scope are examined, and an answer to the following question is developed: “What special characteristics of this product may threaten our project plan?”

Risk Identification

- One method for identifying risks is to create a risk item checklist.
- The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:
- Product size—risks associated with the overall size of the software to be built or modified.
- Business impact—risks associated with constraints imposed by management or the marketplace.
- Stakeholder characteristics—risks associated with the sophistication of the stakeholders and the developer's ability to communicate with stakeholders in a timely manner.
- Process definition—risks associated with the degree to which the software process has been defined and is followed by the development organization. • Development environment—risks associated with the availability and quality of the tools to be used to build the product.
- Technology to be built—risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
- Staff size and experience—risks associated with the overall technical and project experience of the software engineers who will do the work.

Risk Identification

Assessing Overall Project Risk

The following questions have been derived from risk data obtained by surveying experienced software project managers in different parts of the world.

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

Risk Identification

- The project manager identify the risk drivers that affect software risk components— performance, cost, support, and schedule.
- The risk components are defined in the following manner:
- Performance risk—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.
- Cost risk—the degree of uncertainty that the project budget will be maintained.
- Support risk—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.
- Schedule risk—the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

- The impact of each risk driver on the risk component is divided into one of four impact categories—negligible, marginal, critical, or catastrophic.

Risk Projection

- Risk projection, also called risk estimation, attempts to rate each risk in two ways.

  (1) The likelihood or probability that the risk is real and

  (2) The consequences of the problems associated with the risk, should it occur

Managers and technical staff to perform four risk projection steps:

1. Establish a scale that reflects the perceived likelihood of a risk.
2. Delineate the consequences of the risk.
3. Estimate the impact of the risk on the project and the product.
4. Assess the overall accuracy of the risk projection so that there will be no misunderstandings.

The intent of these steps is to consider risks in a manner that leads to prioritization. No software team has the resources to address every possible risk with the same degree of rigor.

By prioritizing risks, you can allocate resources where they will have the most impact.

Risk Projection

1. Developing a Risk Table

- A risk table provides you with a simple technique for risk projection. A sample risk table is illustrated in Figure.
- List all the risks (no matter how remote) in the first column of the table.
- Each risk is categorized in the second column (e.g., PS implies a project size risk, BU implies a business risk).
- The probability of occurrence of each risk is entered in the next column of the table. The probability value for each risk can be estimated by team members individually.
- Next, the impact of each risk is assessed. Each risk component is assessed, and an impact category is determined.
- The categories for each of the four risk components—performance, support, cost, and schedule—are averaged to determine an overall impact value.
- Once the first four columns of the risk table have been completed, the table is sorted by probability and by impact.
- High-probability, high-impact risks percolate to the top of the table, and low-probability risks drop to the bottom.

Risk Projection

Sample Risk table prior to sorting

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

Impact values:
1—catastrophic
2—critical
3—marginal
4—negligible

Risk Projection

2. Assessing Risk Impact

- Three factors affect the consequences that are likely if a risk does occur:

  its nature, its scope, and its timing.
- The **nature of the risk** indicates the problems that are likely if it occurs. For example, a poorly defined external interface to customer hardware (a technical risk) will preclude early design and testing and will likely lead to system integration problems late in a project.
- The **scope of a risk** combines the severity (just how serious is it?) with its overall distribution (how much of the project will be affected or how many stakeholders are harmed?).
- The **timing of a risk** considers when and for how long the impact will be felt. In most cases, you want the "bad news" to occur as soon as possible, but in some cases, the longer the delay, the better.
- The overall risk exposure RE is determined using the following relationship

  $$RE = P * C$$

  where P is the probability of occurrence for a risk, and C is the cost to the project should the risk occur.

Risk Mitigation, Monitoring, and Management

- An effective strategy for dealing with risk must consider three issues
  (Note: these are not mutually exclusive)
  - Risk mitigation
  - Risk monitoring
  - Risk management and contingency planning
- Risk mitigation - is the primary strategy and is achieved through a plan
  - Example: Risk of high staff turnover
- Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market)
- Mitigate those causes that are under our control before the project starts
- Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave

Risk Mitigation, Monitoring, and Management

- Organize project teams so that information about each development activity is widely dispersed
- Define documentation standards and establish mechanisms to ensure that documents are developed in a timely manner
- Conduct peer reviews of all work (so that more than one person is "up to speed")
- Assign a backup staff member for every critical technologist.
- During risk monitoring, the project manager monitors factors that may provide an indication of whether a risk is becoming more or less likely
- Risk management and contingency planning assume that mitigation efforts have failed and that the risk has become a reality

Risk Mitigation, Monitoring, and Management

- RMMM steps incur additional project cost
  - Large projects may have identified 30 – 40 risks
- Risk is not limited to the software project itself
  - Risks can occur after the software has been delivered to the user
- Software safety and hazard analysis
  - These are software quality assurance activities that focus on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail
  - If hazards can be identified early in the software process, software design features can be specified that will either eliminate or control potential hazards.
- It is important to note that risk mitigation, monitoring, and management (RMMM) steps incur additional project cost

The RMMM Plan

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

What is PROJECT SCHEDULING?

- In the late 1960s, a bright-eyed young engineer was chosen to "write" a computer program for an automated manufacturing application. The reason for his selection was simple. He was the only person in his technical group who had attended a computer programming seminar. He knew the ins and outs of assembly language and FORTRAN but nothing about software engineering and even less about project scheduling and tracking. His boss gave him the appropriate manuals and a verbal description of what had to be done. He was informed that the project must be completed in two months. He read the manuals, considered his approach, and began writing code. After two weeks, the boss called him into his office and asked how things were going. "Really great," said the young engineer with youthful enthusiasm. "This was much simpler than I thought. I'm probably close to 75 percent finished."

What is PROJECT SCHEDULING?

- The boss smiled and encouraged the young engineer to keep up the good work. They planned to meet again in a week's time. A week later the boss called the engineer into his office and asked, "Where arewe?" "Everything's going well," said the youngster, "but I've run into a few small snags. I'll get them ironed out and be back on track soon." "How does the deadline look?" the boss asked. "No problem," said the engineer. "I'm close to 90 percent complete." If you've been working in the software world for more than a few years, you can finish the story. It'll come as no surprise that the young engineer1 stayed 90 percent complete for the entire project duration and finished (with the help of others) only one month late. This story has been repeated tens of thousands of times by software developers during the past five decades. The big question is why?

What is PROJECT SCHEDULING?

- ✓ You've selected an appropriate process model.
- ✓ You've identified the software engineering tasks that have to be performed.
- ✓ You estimated the amount of work and the number of people, you know the deadline, you've even considered the risks.
- ✓ Now it's time to connect the dots. That is, you have to create a network of software engineering tasks that will enable you to get the job done on time.
- ✓ Once the network is created, you have to assign responsibility for each task, make sure it gets done, and adapt the network as risks become reality.

What is PROJECT SCHEDULING?

- **Why it's Important?**
  - ✓ In order to build a complex system, many software engineering tasks occur in parallel.
  - ✓ The result of work performed during one task may have a profound effect on work to be conducted in another task.
  - ✓ These interdependencies are very difficult to understand without a schedule.
  - ✓ It's also virtually impossible to assess progress on a moderate or large software project without a detailed schedule

- **What are the steps?**
  - ✓ The software engineering tasks dictated by the software process model are refined for the functionality to be built.
  - ✓ Effort and duration are allocated to each task and a task network (also called an "activity network") is created in a manner that enables the software team to meet the delivery deadline established.

What is PROJECT SCHEDULING?

Basic Concept of Project Scheduling

- ✓ An unrealistic deadline established by someone outside the software development group and forced on managers and practitioner's within the group.
- ✓ Changing customer requirements that are not reflected in schedule changes.
- ✓ An honest underestimate of the amount of effort and/or the number of resources that will be required to do the job.
- ✓ Predictable and/or unpredictable risks that were not considered when the project commenced.
- ✓ Technical difficulties that could not have been foreseen in advance.

▪**Why should we do when the management demands that we make a dead line I impossible?**

- ✓ Perform a detailed estimate using historical data from past projects.
- ✓ Determine the estimated effort and duration for the project.
- ✓ Using an incremental process model, develop a software engineering strategy that will deliver critical functionality by the imposed deadline, but delay other functionality until later. Document the plan.
- ✓ Meet with the customer and (using the detailed estimate), explain why the imposed deadline is unrealistic.

Project Scheduling

- Project Scheduling
  - Basic Principles
  - The Relationship Between People and Effort
  - Effort Distribution

- Software project scheduling is an action that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks.
- During early stages of project planning, a macroscopic schedule is developed.
- As the project gets under way, each entry on the macroscopic schedule is refined into a detailed schedule.

Project Scheduling

- **Basic Principles of Project Scheduling.**

1. **Compartmentalization:** The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are refined.
2. **Interdependency:** The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence, while others can occur in parallel. Other activities can occur independently.
3. **Time allocation:** Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date. whether work will be conducted on a full-time or part-time basis.
4. **Effort validation:** Every project has a defined number of people on the software team. The project manager must ensure that no more than the allocated number of people have been scheduled at any given time.
5. **Defined responsibilities**. Every task that is scheduled should be assigned to a specific team member.

Project Scheduling

6. **Defined outcomes:** Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a component) or a part of a work product. Work products are often combined in deliverables.
7. **Defined milestones:** Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality and has been approved.

Each of these principles is applied as the project schedule evolves.

Project Scheduling

- The Relationship Between People and Effort

- In a small software development project a single person can analyze requirements, perform design, generate code, and conduct tests. As the size of a project increases, more people must become involved.
- There is a common myth that is still believed by many managers who are responsible for software development projects: “If we fall behind schedule, we can always add more programmers and catch up later in the project.”
- Unfortunately, adding people late in a project often has a disruptive effect on the project, causing schedules to slip even further. The people who are added must learn the system, and the people who teach them are the same people who were doing the work.
- While teaching, no work is done, and the project falls further behind. In addition to the time it takes to learn the system, more people.
- Although communication is absolutely essential to successful software development, every new communication path requires additional effort and therefore additional time.

Project Scheduling

Effort Distribution

- A recommended distribution of effort across the software process is often referred to as the 40–20–40 rule.
- Forty percent of all effort is allocated to frontend analysis and design. A similar percentage is applied to back-end testing. You can correctly infer that coding (20 percent of effort) is deemphasized.
- Work expended on project planning rarely accounts for more than 2 to 3 percent of effort, unless the plan commits an organization to large expenditures with high risk.
- Customer communication and requirements analysis may comprise 10 to 25 percent of project effort.
- Effort expended on analysis or prototyping should increase in direct proportion with project size and complexity.
- A range of 20 to 25 percent of effort is normally applied to software design. Time expended for design review and subsequent iteration must also be considered.
- Because of the effort applied to software design, code should follow with relatively little difficulty.
- A range of 15 to 20 percent of overall effort can be achieved. Testing and subsequent debugging can account for 30 to 40 percent of software development effort.
- The criticality of the software often dictates the amount of testing that is required. If software is human rated (i.e., software failure can result in loss of life), even higher percentages are typical.

Defining a Task Set for the Software Project

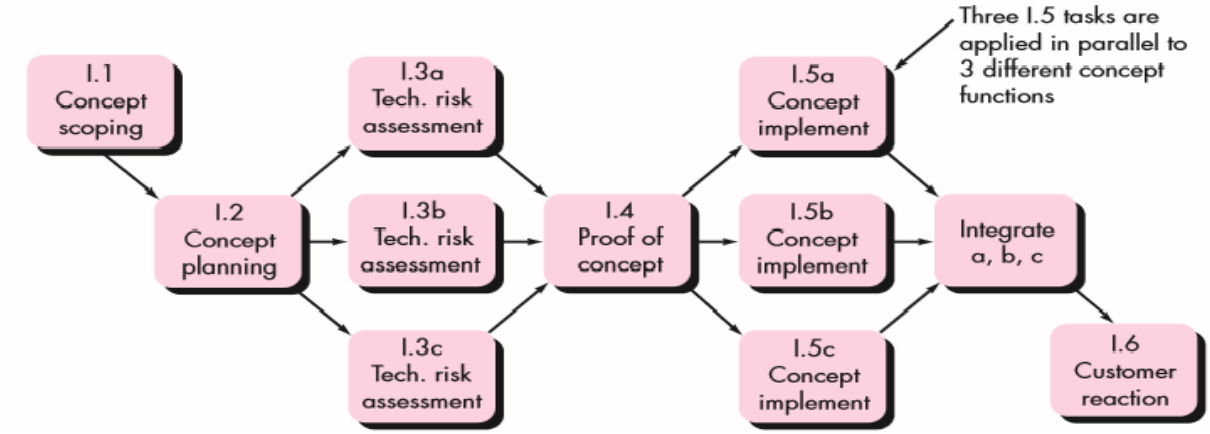
- A task set is a collection of software engineering work tasks, milestones, work products, and quality assurance filters that must be accomplished to complete a particular project.
- The task set must provide enough discipline to achieve high software quality. But, at the same time, it must not burden the project team with unnecessary work.
- Most software organizations encounter the following projects:
  1. Concept development projects that are initiated to explore some new business concept or application of some new technology.
  2. New application development projects that are undertaken as a consequence of a specific customer request.
  3. Application enhancement projects that occur when existing software undergoes major modifications to function, performance, or interfaces that are observable by the end user.
  4. Application maintenance projects that correct, adapt, or extend existing software in ways that may not be immediately obvious to the end user.
  5. Reengineering projects that are undertaken with the intent of rebuilding an existing (legacy) system in whole or in part.

Defining a Task Set for the Software Project

1. A Task Set Example
   - Concept development projects are initiated when the potential for some new technology must be explored. There is no certainty that the technology will be applicable, but a customer (e.g., marketing) believes that potential benefit exists.
2. Refinement of Software Engineering Actions
   - The software engineering actions are used to define a macroscopic schedule for a project.
   - The macroscopic schedule must be refined to create a detailed project schedule.
   - Refinement begins by taking each action and decomposing it into a set of tasks (with related work products and milestones).

Defining a Task Network

- A task network, also called an activity network, is a graphic representation of the task flow for a project.
- It is sometimes used as the mechanism through which task sequence and dependencies are input to an automated project scheduling tool.
- In its simplest form (used when creating a macroscopic schedule), the task network depicts major software engineering actions. Figure below shows a schematic task network for a concept development project.
- It is important to note that the task network shown in Figure 27.2 is macroscopic. In a detailed task network (a precursor to a detailed schedule), each action shown in the figure would be expanded.



Scheduling

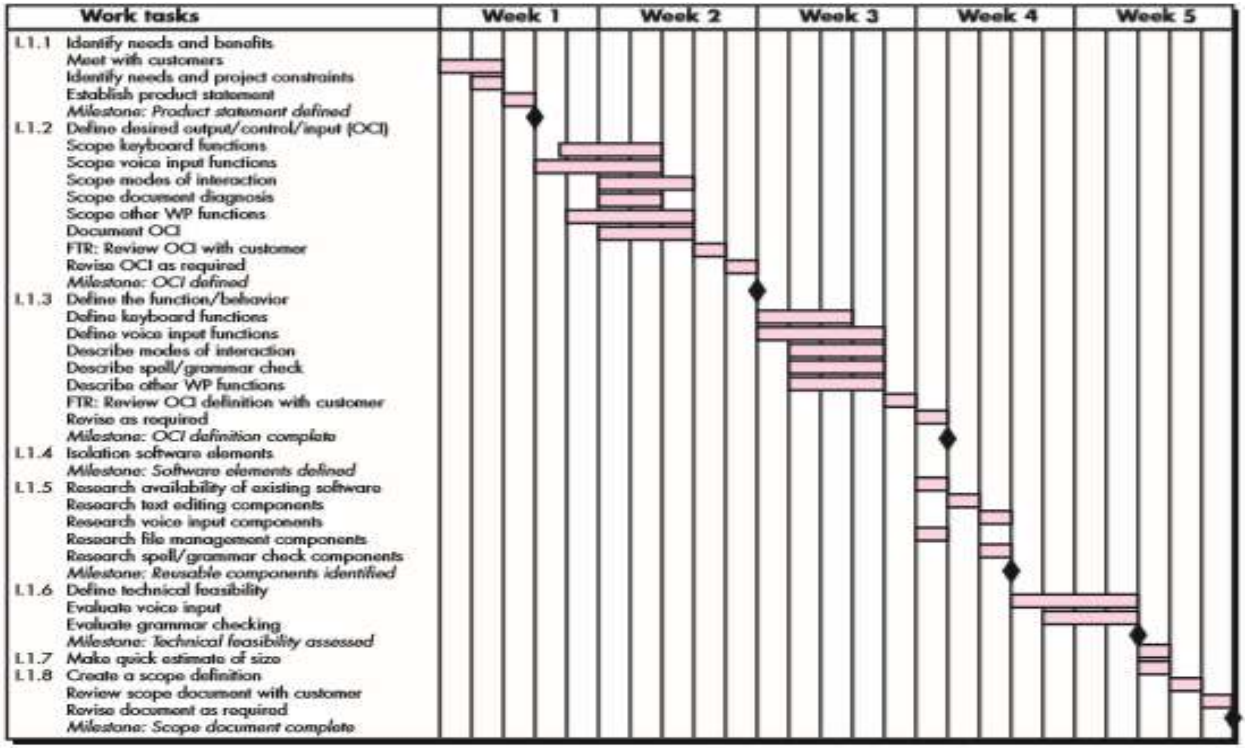
- Scheduling of a software project does not differ greatly from scheduling of any multitask engineering effort. Therefore, generalized project scheduling tools and techniques can be applied with little modification for software projects.
- Program evaluation and review technique (PERT) and the critical path method (CPM) are two project scheduling methods that can be applied to software development.

1. Time-Line Charts:

- When creating a software project schedule, begin with a set of tasks.
- If automated tools are used, the work breakdown is input as a task network or task outline. Effort, duration, and start date are then input for each task. In addition, tasks may be assigned to specific individuals.
- As a consequence of this input, a time-line chart, also called a Gantt chart, is generated.
- A time-line chart can be developed for the entire project. Alternatively, separate charts can be developed for each project function or for each individual working on the project.

Scheduling

- All project tasks (for concept scoping) are listed in the left hand column. The horizontal bars indicate the duration of each task. When multiple bars occur at the same time on the calendar, task concurrency is implied. The diamonds indicate milestones.
- Once the information necessary for the generation of a time-line chart has been input, the majority of software project scheduling tools produce project tables. —a tabular listing of all project tasks, their planned and actual start and end dates, and a variety of related information. Used in conjunction with the time-line chart, project tables enable you to track progress.



Scheduling

2. Tracking the Schedule

- If it has been properly developed, the project schedule becomes a road map that defines the tasks and milestones to be tracked and controlled as the project proceeds.
- Tracking can be accomplished in a number of different ways:
- Conducting periodic project status meetings in which each team member reports progress and problems.
- Evaluating the results of all reviews conducted throughout the software engineering process.
- Determining whether formal project milestones have been accomplished by the scheduled date.
- Comparing the actual start date to the planned start date for each project task listed in the resource table.
- Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon.
- Using earned value analysis to assess progress quantitatively.

In reality, all of these tracking techniques are used by experienced project managers.

Scheduling

3. Tracking Progress for an OO Project

Technical milestone: OO analysis complete

- o All hierarchy classes defined and reviewed
- o Class attributes and operations are defined and reviewed
- o Class relationships defined and reviewed
- o Behavioral model defined and reviewed
- o Reusable classed identified

Technical milestone: OO design complete

- o Subsystems defined and reviewed
- o Classes allocated to subsystems and reviewed
- o Task allocation has been established and reviewed
- o Responsibilities and collaborations have been identified
- o Attributes and operations have been designed and reviewed
- o Communication model has been created and reviewed

Scheduling

- **Technical milestone: OO programming complete**
  - o Each new design model class has been implemented
  - o Classes extracted from the reuse library have been implemented
  - o Prototype or increment has been built
- **Technical milestone: OO testing**
  - o The correctness and completeness of the OOA and OOD models has been reviewed
  - o Class-responsibility-collaboration network has been developed and reviewed
  - o Test cases are designed and class-level tests have been conducted for each class
  - o Test cases are designed, cluster testing is completed, and classes have been integrated
  - o System level tests are complete

Scheduling

- **Scheduling for WebApp Projects**
- WebApp project scheduling distributes estimated effort across the planned time line (duration) for building each WebApp increment.
- This is accomplished by allocating the effort to specific tasks.
- The overall WebApp schedule evolves over time.
- During the first iteration, a macroscopic schedule is developed.
- This type of schedule identifies all WebApp increments and projects the dates on which each will be deployed.
- As the development of an increment gets under way, the entry for the increment on the macroscopic schedule is refined into a detailed schedule.
- Here, specific development tasks (required to accomplish an activity) are identified and scheduled.

EARNED VALUE ANALYSIS

- It is reasonable to ask whether there is a quantitative technique for assessing progress as the software team progresses through the work tasks allocated to the project schedule.
- A Technique for performing quantitative analysis of progress does exist. It is called earned value analysis (EVA).
- To determine the earned value, the following steps are performed:
  1. The budgeted cost of work scheduled (BCWS) is determined for each work task represented in the schedule. During estimation, the work (in person-hours or person-days) of each software engineering task is planned. Hence, BCWSi is the effort planned for work task i. To determine progress at a given point along the project schedule, the value of BCWS is the sum of the BCWSi values for all work tasks that should have been completed by that point in time on the project schedule.
  2. The BCWS values for all work tasks are summed to derive the budget at completion (BAC). Hence, BAC (BCWSk) for all tasks k
  3. Next, the value for budgeted cost of work performed (BCWP) is computed. The value for BCWP is the sum of the BCWS values for all work tasks that have actually been completed by a point in time on the project schedule.

EARNED VALUE ANALYSIS

- Given values for BCWS, BAC, and BCWP, important progress indicators can be computed:

  Schedule performance index, SPI = BCWP / BCWS

  Schedule variance, SV = BCWP – BCWS

- SPI is an indication of the efficiency with which the project is utilizing scheduled resources. An SPI value close to 1.0 indicates efficient execution of the project schedule. SV is simply an absolute indication of variance from the planned schedule.
- Percent scheduled for completion = BCWS / BAC

  provides an indication of the percentage of work that should have been completed by time t.
- Percent complete = BCWP / BAC

  provides a quantitative indication of the percent of completeness of the project at a given point in time t. It is also possible to compute the actual cost of work performed (ACWP). The value for ACWP is the sum of the effort actually expended on work tasks that have been completed by a point in time on the project schedule. It is then possible to compute

EARNED VALUE ANALYSIS

Cost performance index, CPI = BCWP /ACWP

Cost variance, CV = BCWP - ACWP

A CPI value close to 1.0 provides a strong indication that the project is within its defined budget. CV is an absolute indication of cost savings (against planned costs) or shortfall at a particular stage of a project.

Process and Project Metrics

What are Metrics?

- Software process and project metrics are quantitative measures
- They are a management tool
- They offer insight into the effectiveness of the software process and the projects that are conducted using the process as a framework
- Basic quality and productivity data are collected
- These data are analyzed, compared against past averages, and assessed
- The goal is to determine whether quality and productivity improvements have occurred
- The data can also be used to pinpoint problem areas
- Remedies can then be developed and the software process can be improved

Process and Project Metrics

Reasons to Measure

- To characterize in order to
  - Gain an understanding of processes, products, resources, and environments
  - Establish baselines for comparisons with future

assessments

- To evaluate in order to
  - Determine status with respect to plans
- To predict in order to
  - Gain understanding of relationships among processes and products
  - Build models of these relationships
- To improve in order to
  - Identify roadblocks, root causes, inefficiencies, and other opportunities for improving product quality and process performance

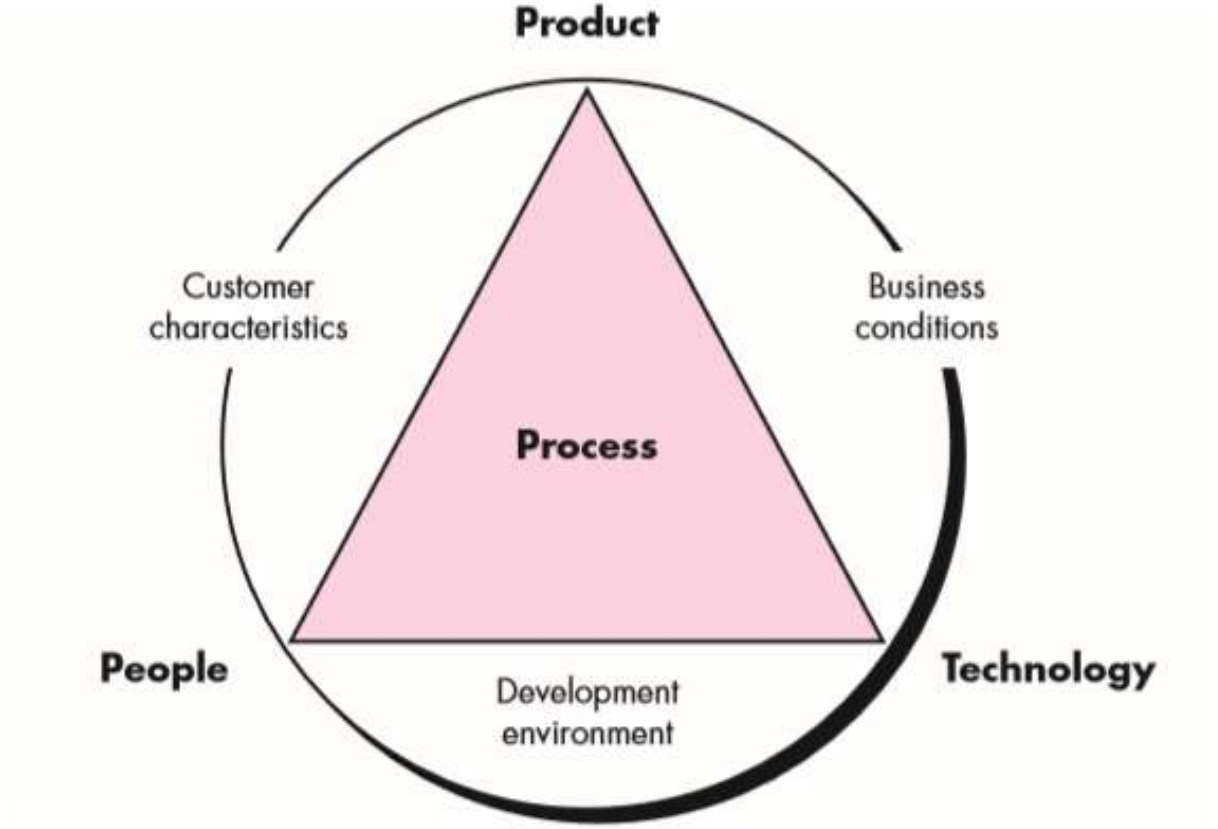
Metrics In The Process and Project Domains

- *Process metrics are collected across all projects and over long periods of time.*
- Their intent is to provide a set of process indicators that lead to long-term software process improvement.
- *Project metrics enable a software project manager to*
- assess the status of an ongoing project,
- track potential risks,
- uncover problem areas before they go “critical,”
- adjust work flow or tasks,
- evaluate the project team’s ability to control quality of software work products

Process Metrics and Software Process Improvement:-

- Software process improvement, it is important to note that process is only one of a number of "controllable factors in improving software quality and organizational performance".
- Process sits at the center of a triangle connecting three factors that have a profound influence on software quality and organizational performance.
- The skill and motivation of people has been shown to be the single most influential factor in quality and performance.
- The complexity of the product can have a substantial impact on quality and team performance.
- The technology (i.e., the software engineering methods and tools) that populates the process also has an impact.
- In addition, the process triangle exists within a circle of environmental conditions that include the development environment (e.g., integrated software tools), business conditions (e.g., deadlines, business rules), and customer characteristics (e.g., ease of communication and collaboration).

Determinants for software quality and organizational effectiveness.



- Measure the effectiveness of a process by deriving a set of metrics based on outcomes of the process such as
  - Errors uncovered before release of the software
  - Defects delivered to and reported by the end users
  - Work products delivered
  - Human effort expended
  - Calendar time expended
  - Conformance to the schedule
  - Time and effort to complete each generic activity.
- **Etiquette(good manners) of Process Metrics**:
- Use common sense and organizational sensitivity when interpreting metrics data
- Provide regular feedback to the individuals and teams who collect measures and metrics
- Don't use metrics to evaluate individuals
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them.
- Never use metrics to pressure individuals or teams.
- Metrics data that indicate a problem should not be considered "negative

Project Metrics:-

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

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

during the project is also reduced

  - As rework goes down, the overall project cost is reduced

Software Measurement

- Measurements in the physical world can be categorized in two ways: direct measures and indirect measures.
- Direct measures of the software process include cost and effort applied.
- Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time.
- Indirect measures of the product include functionality, quality, complexity, efficiency, reliability, maintainability.
- Project metrics can be consolidated to create process metrics for an organization.

Software Measurement

Size-Oriented Metrics

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

Function-Oriented Metrics:-

- Function-oriented metrics use a measure of the functionality delivered by the application as a normalization value
- Most widely used metric of this type is the function point
- Computation of the function point is based on characteristics of the software's information domain and complexity.

Software Measurement

- Function Point Controversy
- Like the KLOC measure, function point use also has proponents and opponents

- Proponents claim that
  - FP is programming language independent

  - FP is based on data that are more likely to be known in the early stages of a project, making it more attractive as an estimation approach

- Opponents claim that
  - FP requires some “sleight of hand” because the computation is based on subjective data

  - Counts of the information domain can be difficult to collect after the fact

  - FP has no direct physical meaning…it’s just a number

Reconciling LOC and FP Metrics:-

- Relationship between LOC and FP depends upon
  - The programming language that is used to implement the software
  - The quality of the design

- FP and LOC have been found to be relatively accurate predictors of software development effort and cost
  - However, a historical baseline of information must first be established.

- LOC and FP can be used to estimate object-oriented software projects
  - However, they do not provide enough granularity for the schedule and effort adjustments required in the iterations of an evolutionary or incremental process
- The table on the next slide provides a rough estimate of the average LOC to one FP in various programming languages.

LOC Per Function Point

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

Object-oriented Metrics:-

- Following set of metrics for OO projects:
- **Number of scenario scripts:-** A scenario script is a detailed sequence of steps that describe the interaction between the user and the application.
- Each script is organized into triplets of the form

{**initiator,** ***action, participant}***

- where initiator is the object that requests some service, *action is the result of the request, and participant is the server object that satisfies* the request.

Number of key classes.:- Key classes are the "highly independent components " that are defined early in object-oriented analysis

- Because key classes are central to the problem domain, the number of such classes is an indication of the amount of effort required to develop the software.
- Also an indication of the potential amount of reuse to be applied during system development.

Number of support classes:- Support classes are required to implement the system but are not immediately related to the problem domain.

- The number of support classes is an indication of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during system development.

- **Number of subsystems**
  - A subsystem is an aggregation of classes that support a function that is visible to the end user of a system.

Average number of support classes per key class

- Key classes are identified early in a project (e.g., at requirements analysis)
- Estimation of the number of support classes can be made from the number of key classes
- GUI applications have between two and three times more support classes as key classes
- Non-GUI applications have between one and two times more support classes as key classes

Use-Case–Oriented Metrics:-

- Use cases describe user-visible functions and features that are basic requirements for a system.
- The number of use cases is directly proportional to the size of the application in LOC and to the number of test cases that will have to be designed to fully exercise the application.

WebApp Project Metrics:-

- The objective of all WebApp projects is to deliver a combination of content and functionality to the end user.
- The measures that can be collected are:
- Number of static Web pages.
- Number of dynamic Web pages.
- Number of internal page links.:-Internal page links are pointers that provide a hyperlink to some other Web page within the WebApp

Number of persistent data objects.

- Number of external systems interfaced:- WebApps must often interface with “backroom” business applications.
- Number of static content objects:-Static content objects encompass static text-based, graphical, video, animation, and audio information that are incorporated within the WebApp.
- Number of dynamic content objects.
- Number of executable functions

N_{sp} = number of Static Web pages

N_{dp} = number of Dynamic Web pages

Then,

$$\text{Customization index, } C = \frac{N_{dp}}{N_{dp} + N_{sp}}$$

The value of C ranges from 0 to 1. As C grows larger, the level of WebApp customization becomes a significant technical issue.

Metrics For Software Quality

- The overriding goal of software engineering is to produce a high-quality system, application, or product within a time frame that satisfies a market need.
- The quality of a system, application, or product is only as good as the requirements that describe the problem, the design that models the solution, the code that leads to an executable program, and the tests that exercise the software to uncover errors.

Measuring Quality

- There are many measures of software quality,8 correctness, maintainability, integrity, and usability provide useful indicators for the project team

Correctness:

- Correctness is the degree to which the software performs its required function.
- The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements.
- Defects are those problems reported by a user of the program after the program has been released for general use.

Maintainability:

- Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements.
- *Mean -time-to-change (MTTC), the time it takes to analyze the change* request, design an appropriate modification, implement the change, test it, and distribute the change to all users.

- **Integrity:**
- Software integrity has become increasingly important in the age of cyber terrorists and hackers.
- Attacks can be made on all three components of software: programs, data, and documentation.
- To measure integrity, two attributes must be defined:
- threat and security.

- **Usability:**
- If a program is not easy to use, it is often doomed to failure, even if the functions that it performs are valuable

Defect Removal Efficiency:

- Defect removal efficiency provides benefits at both the project and process level
- It is a measure of the filtering ability of quality assurances activities as they are applied throughout all process framework activities
  - It indicates the percentage of software errors found before software release
- It is defined as DRE = E / (E + D)
  - E is the number of errors found before delivery of the software to the end user
  - D is the number of defects found after delivery
- As D increases, DRE decreases (i.e., becomes a smaller and smaller fraction)

- The ideal value of DRE is 1, which means no defects are found after delivery

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

Integrating Metrics Within The Software Process

Arguments for Software Metrics:-

- Most software developers do not measure, and most have little desire to begin
- Establishing a successful company-wide software metrics program can be a multi-year effort
- But if we do not measure, there is no real way of determining whether we are improving
- Measurement is used to establish a process baseline from which improvements can be assessed
- Software metrics help people to develop better project estimates, produce higher-quality systems, and get products out the door on time.
- The collection of quality metrics enables an organization to “tune” its software process to remove the “vital few” causes of defects that have the greatest impact on software development

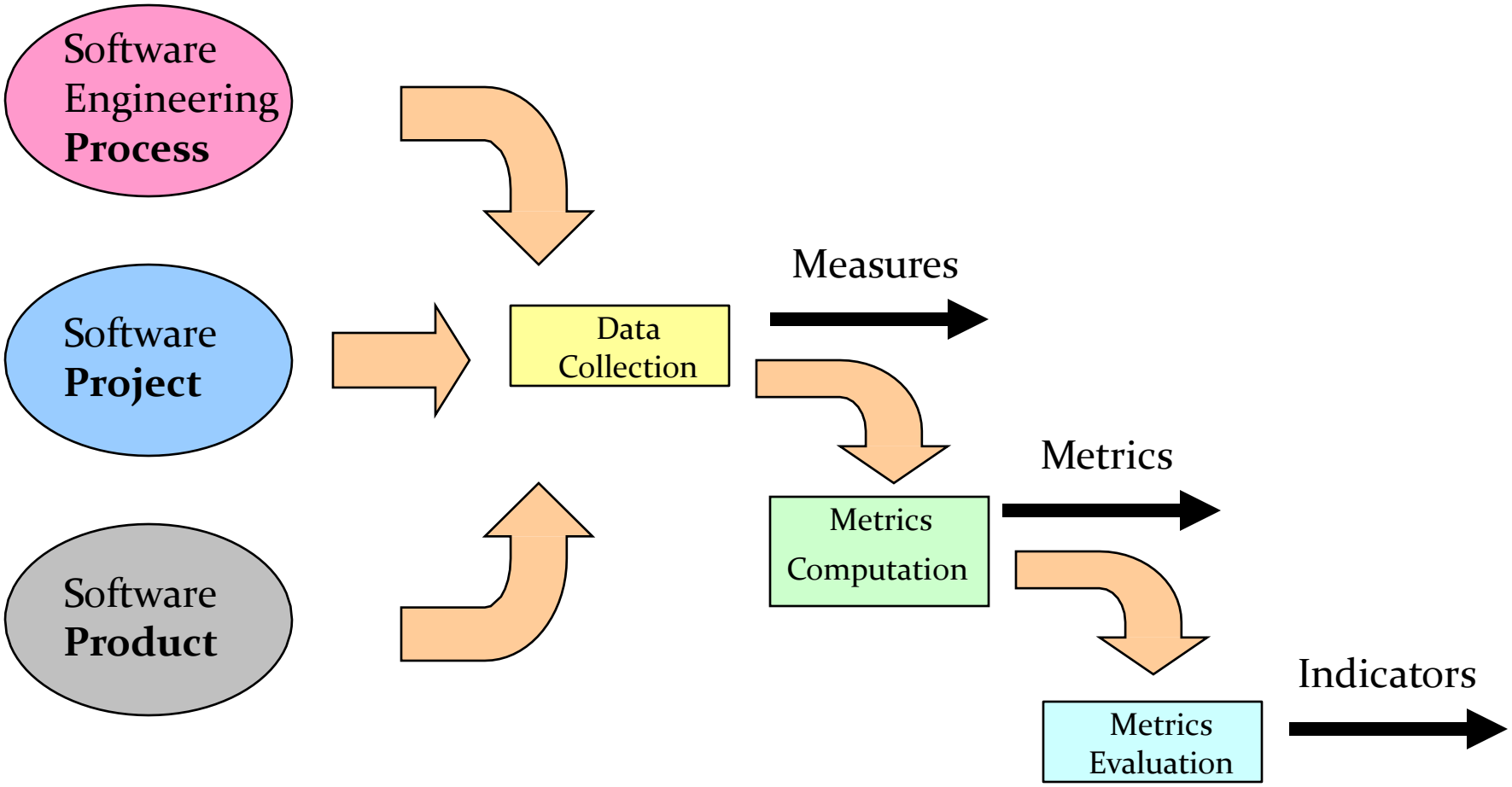
Establishing a Baseline:-

- By establishing a metrics baseline, benefits can be obtained at the software process, product, and project levels
- The same metrics can serve many masters
- The baseline consists of data collected from past software development projects.
- Baseline data must have the following attributes
  - Data must be reasonably accurate (guesses should be avoided)
  - Data should be collected for as many projects as possible
  - Measures must be consistent (e.g., a line of code must be interpreted consistently across all projects)
  - Past applications should be similar to the work that is to be estimated.

Metrics Collection, Computation, and Evaluation

- Data collection requires an historical investigation of past projects to reconstruct required data
- After data is collected and metrics are computed, the metrics should be evaluated and applied during estimation, technical work, project control, and process improvement.

Software Metrics Baseline Process



Metrics For Small Organizations

- Most software organizations have fewer than 20 software engineers.
- It is reasonable to suggest that software organizations of all sizes measure and then use the resultant metrics to help improve their local software process and the quality and timeliness of the products they produce.
- A commonsense approach to the implementation of any software process-related activity is: keep it simple, customize to meet local needs, and be sure it adds value.

A small organization might select the following set of easily collected measures:

- Time (hours or days) elapsed from the time a request is made until evaluation is complete, *tqueue.*
- Effort (person-hours) to perform the evaluation, *Weval.*
- Time (hours or days) elapsed from completion of evaluation to assignment of change order to personnel, *teval.*

- Effort (person-hours) required to make thechange, *Wchange.*
- Time required (hours or days) to make the change, *tchange.*
- Errorsuncovered during work to make change, *Echange.*
- Defects uncovered after change is released to the customer base, *Dchange.*
- The defect removal efficiency can be computed as

$$\text{DRE} = \frac{E_{\text{change}}}{E_{\text{change}} + D_{\text{change}}}$$

DRE can be compared to elapsed time and total effort to determine the impact of quality assurance activities on the time and effort required to make a change.

Establishing a software metrics program

- The Software Engineering Institute has developed a comprehensive guidebook for establishing a “goal-driven” software metrics program.

The guidebook suggests the following steps:

- Identify business goal
- Identify what you want to know
- Identify subgoals
- Identify subgoal entities and attributes
- Formalize measurement goals
- Identify quantifiable questions and indicators related to subgoals
- Identify data elements needed to be collected to construct the indicators
- Define measures to be used and create operational definitions for them
- Identify actions needed to implement the measures
- Prepare a plan to implement the measures

Establishing a software metrics program

- For example, consider the SafeHome product. Working as a team, software engineering and business managers develop a list of prioritized business goals:
  1. Improve our customers' satisfaction with our products.
  2. Make our products easier to use.
  3. Reduce the time it takes us to get a new product to market.
  4. Make support for our products easier.
  5. Improve our overall profitability.