SE UNIT 2

Chapter 5 Requirements Engineering

- The step we generally know as analysis is also known as Software Requirements Engineering.
- It is a systematic and disciplined approach to deal with requirements.
- It helps software engineers to better understand the problem they will work to solve.
- It consists of tasks that lead to an understanding of what the business impact of software will be, what the customer wants, and how the end-users will interact with the software.
- The software engineer and customer play an active role in this activity and the software engineer acts as interrogator, consultant, problem solver and negotiator.
- The activity may appear to be simple but it is not since the communication content is very high.

Requirements Engineering Tasks

- The software requirements activity is accomplished through the execution of following seven distinct functions:

1. Inception
2. Elicitation
3. Elaboration
4. Negotiation
5. Specification
6. Validation
7. Management

1. <u>Inception</u> : (Start, beginning). How does a software project get started?
   - Sometimes a casual discussion results into a major software development project. But, in most of the cases the software development projects begin when a business need is identified or a potential new market or service is discovered.
   - Analyst establish
     - A basic understanding of problem
     - People who want a solution
     - Nature of the solution
     - Effectiveness of communication and collaboration required with stakeholders

2. <u>Elicitation</u> : (drawing out, extracting, obtaining)

- <u>Requirements elicitation</u> is the process of <u>getting requirements</u> from the customer.
- It appears to be simple and easy but it is not.
- Difficulties faced are
  - problems of <u>scope</u> (boundary is ill-defined or customers give unnecessary details),
  - problems of <u>understanding</u> (customers not clear about the capability of computing environment, having problem in communicating, omit information that is obvious, give conflicting requirements etc.) and
  - problems of <u>volatility</u> (change).

3. <u>Elaboration</u> : (Expansion, amplification, explanation)

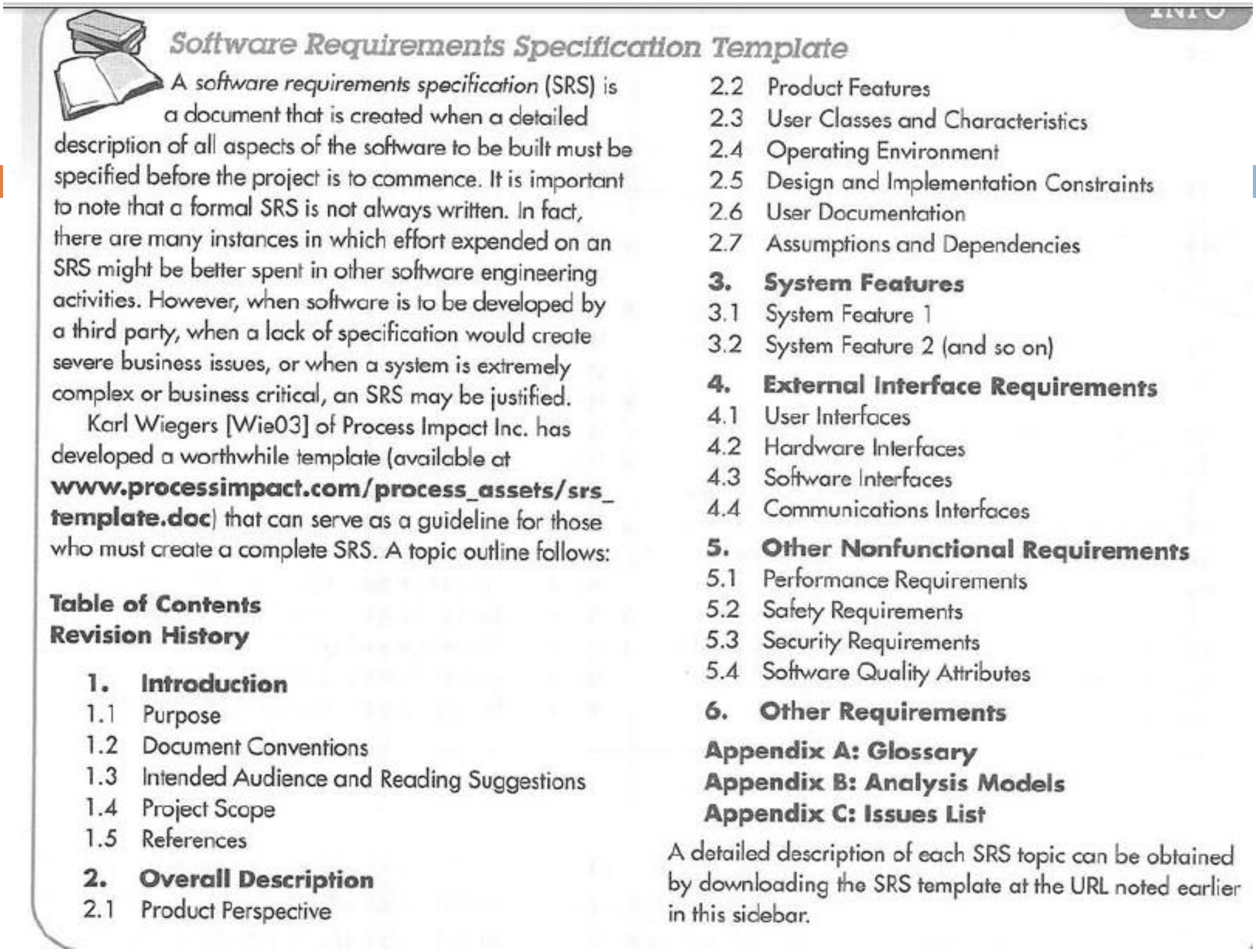
- The information obtained from the customer during inception and elicitation is expanded and refined during elaboration.
- It focuses on developing a refined technical model of software functions, features and constraints.
- It is basically an analysis modeling action i.e. you make analysis models like ERD, DFD, STD.

4. <u>Negotiation</u> :

- Lot of times customers / end users ask for more than what can be given with the resources available (time, money, manpower).
- There may be conflicting requirements from different users also. There is a need for the <u>negotiation process</u> in such cases <u>to reconcile (resolve, settle) the conflicts</u>.
- Using an iterative process the requirements are eliminated, combined, and / or modified so that each party achieves some measure of satisfaction.

5. <u>Specification</u> : Whatever requirements the requirement engineer has elicited, elaborated and finalized after negotiations so far have to be <u>put in a form which can be used by others</u> to take the work ahead.

- That process is called specification process.
- The output of that process is called Specification. It can be a written document, a formal mathematical model, a collection of usage scenarios, a prototype or any combination of these.
- It serves as the foundation for subsequent software engineering activities like design, development, testing etc.



Software Requirements Specification Template

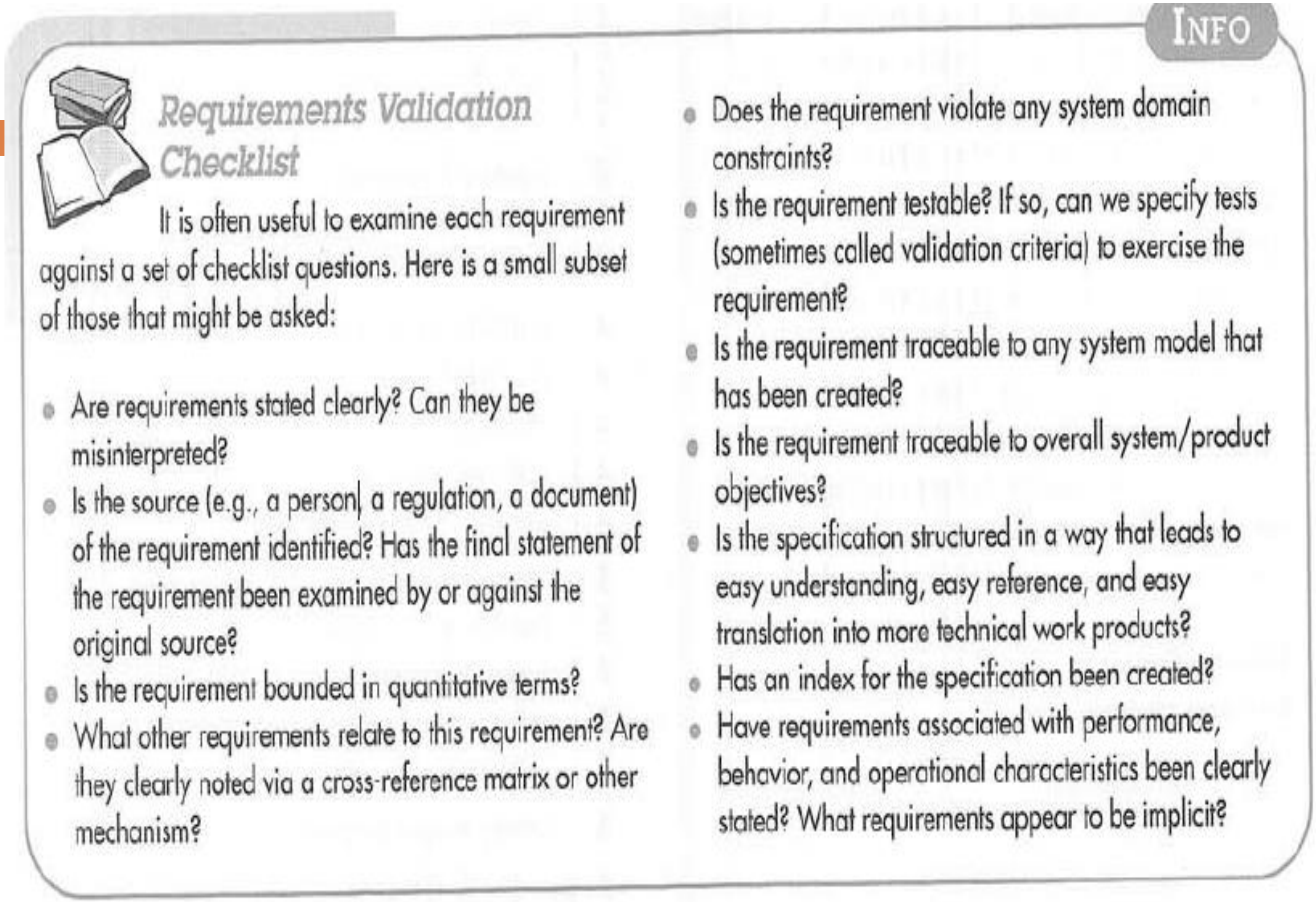
A *software requirements specification* (SRS) is a document that is created when a detailed description of all aspects of the software to be built must be specified before the project is to commence. It is important to note that a formal SRS is not always written. In fact, there are many instances in which effort expended on an SRS might be better spent in other software engineering activities. However, when software is to be developed by a third party, when a lack of specification would create severe business issues, or when a system is extremely complex or business critical, an SRS may be justified.

Karl Wiegers [Wie03] of Process Impact Inc. has developed a worthwhile template (available at **www.processimpact.com/process_assets/srs_template.doc**) that can serve as a guideline for those who must create a complete SRS. A topic outline follows:

A detailed description of each SRS topic can be obtained by downloading the SRS template at the URL noted earlier in this sidebar.

6. <u>Validation</u> : (Confirmation)

- Lot of work products are produced as a consequence of requirements engineering.
- It is necessary to assess the quality of these work products.
- Their <u>quality is assessed</u> through the step of <u>validation</u>.
- It examines the specification to ensure that all requirements have been stated unambiguously (clearly), that inconsistencies, errors and omissions are detected and resolved, and that work products conform to the standards.
- Validation is done through FTR (formal technical review).



INFO

Requirements Validation Checklist

It is often useful to examine each requirement against a set of checklist questions. Here is a small subset of those that might be asked:

- Are requirements stated clearly? Can they be misinterpreted?
- Is the source (e.g., a person, a regulation, a document) of the requirement identified? Has the final statement of the requirement been examined by or against the original source?
- Is the requirement bounded in quantitative terms?
- What other requirements relate to this requirement? Are they clearly noted via a cross-reference matrix or other mechanism?
- Does the requirement violate any system domain constraints?
- Is the requirement testable? If so, can we specify tests (sometimes called validation criteria) to exercise the requirement?
- Is the requirement traceable to any system model that has been created?
- Is the requirement traceable to overall system/product objectives?
- Is the specification structured in a way that leads to easy understanding, easy reference, and easy translation into more technical work products?
- Has an index for the specification been created?
- Have requirements associated with performance, behavior, and operational characteristics been clearly stated? What requirements appear to be implicit?

7. <u>Requirement Management :</u>

- Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds.
- Each requirement is assigned a <u>unique identifier</u>

Establishing the ground work (Inception)

- Starting the requirements engineering process and successfully completing it is very difficult.
- In an ideal setting, customers and software engineers work together on the same team.
- In such cases, requirements engineering is simply a matter of conducting meaningful conversations with colleagues who are well-known members of the team.
- But reality is often quite different. Customers may be located in a different city or country, may have only a unclear idea of what is required, may have conflicting opinions about the system to be built, may have limited technical knowledge and limited time to interact with the requirements engineer.
- None of these things are desirable but they do exist and the engineer has to deal with it. How do you start the process in such a situation?

- Identify the stakeholders – anyone who benefits in a direct or indirect way from the system. Keep expanding this list and ensure that you get contribution from all of them since all of them will have different views and requirements.
- Recognize Multiple view points. The multiple view points in lot of cases will be conflicting with each other. So categorize these conflicting, multiple view points to arrive at some decision.
- Work toward collaboration. Try to work as a team. A team tries to achieve the main objective by following the philosophy of give and take rather than insisting on sticking to one's own ideas.

- <u>First, start with context-free questions</u> to “break the ice”.
- These questions will lead to basic understanding of the problem, the people who want a solution, the nature of solution that is desired and the effectiveness of the first encounter itself.
- Some of these questions could be
  - Who is behind the request for this work?
  - Who will use the solution?
  - What will be the economic benefit?
  - Are you the right person to answer these questions? Are your answers official?
  - Are my questions relevant to the problem you have?
  - Can anyone else provide additional information?
  - Should I be asking you anything else?

Eliciting Requirements

- The style of collecting information discussed above is not very effective.
- There is always a <u>feeling of "us and them</u>" in this kind of meetings.
- So this technique is OK for the first meeting but then one can go for <u>team oriented</u> approach called <u>Collaborative Requirements Gathering</u>.
- In this approach a joint team of customers and developers is formed to identify the problem, propose elements of the solution, negotiate different approaches and specify a preliminary set of solution requirements.

- Conducted at a neutral site.
- Rules for preparation and participation are established.
- Agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A facilitator (a customer, a developer or an outsider) controls the meeting.
- A definition mechanism is used.(work sheets, flip charts etc)
- What should happen and how it should happen before and during a meeting is also decided.

- <u>Before a meeting</u>
  - A meeting place, time and date are selected and a facilitator is chosen.
  - Attendees from the development team and customer organizations are invited to attend and the product request is distributed to all attendees before the meeting date.
  - Each attendee is requested to make a list of objects, services, constraints and performance criteria.
    - Objects : That are part of the environment in which the system operates, that are to be produced and that are used by the system.
    - Services : Processes or functions that interact with the objects.
    - Constraints : cost , size, business rules.
    - Performance :Speed , accuracy etc.

- **<u>During meeting</u>**
  - First point is need and justification of the product.
  - Each participant presents his/her list for discussion. Critique and debate are strictly prohibited at this stage.
  - A combined list is created (no deletion is allowed).
  - Discussion starts and a consensus lists (for objects, services, constraints, performance) are created.
  - Team is divided into smaller sub teams and each works to develop mini-specifications for each item on the consensus list.
  - Mini-specs are then presented to the whole team and discussed.
  - A list of validation criteria is prepared.
  - One person is assigned to write the complete draft specification.

Quality Function Deployment (QFD)

- After the meeting one can do Quality Function Deployment (QFD) for requirements.
- It is a Quality management technique.
- It emphasizes an understanding of what is valuable to the customer.
- If we give it to the customer the customer will be happy i.e. better quality.
- QFD identifies 3 types of requirements
  - Normal requirements: Explicitly stated requirements.
  - Expected requirements: Implicit requirements. Absence will be cause for dissatisfaction.
  - Exciting requirements: they go beyond the customer's expectations and prove to be very satisfying.

- <u>Elicitation work products</u>
- At the end of the elicitation step there will be few work products depending on the size of the project. Some of these work products are:
  - A statement of need and feasibility
  - A bounded statement of scope for the system (how many guests, how many items and what type of service?)
  - A list of customers, users and other stakeholders who participated in requirements elicitation.
  - A description of system’s technical environment
  - A list of requirements and constraints that apply
  - A set of usage scenarios (withdrawal of money from ATM, opening of a new SB account etc)
  - Any prototypes developed to better define requirements

Part of Elaboration : Use-cases

- A system will have number of functionalities.
- Those functionalities will be useful in different situations.
- Use-cases describe how the system will be used in a given situation.
- The system will be used by people or devices. They are called actors.
- Basically use-cases are written narratives that describe the role of an actor.
- **How to write a use case**
  - Define a set of actors. Anything that communicates with the system.
  - Every actor has one or more goals when using system.
  - It is an iterative process so not all actors are defined during the first iteration.
  - Primary actors during first iteration and secondary actors during later iterations.
  - Secondary actors support the system so that primary actors can do their work.

- Questions that should be answered by a use case are as follows :
  - Who is the primary actor, the secondary actor (s)?
  - What are the actor’s goals?
  - What preconditions should exist before the story begins? Withdraw money – card valid, pin right.
  - What main tasks or functions are performed by the actor?
  - What extensions might be considered as the story is described? Balance not enough, time out.
  - What variations in the actor’s interaction are possible?
  - What system information will the actor acquire, produce, or change?
  - Will the actor have to inform the system about changes in the external environment?
  - What information does the actor desire from the system?
  - Does the actor wish to be informed about unexpected changes?

- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
- Do any requirements conflict with other requirements?
- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?
- Does the requirements model properly reflect the information, function, and behavior of the system to be built?
- Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?
- Have requirements patterns been used to simplify the requirements model?
- Have all patterns been properly validated? Are all patterns consistent with customer requirements?

Typical Contents of a Software Requirements Specification

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

Chapter 7: REQUIREMENTS MODELING

- There are two views of requirements modeling:
  - **Structured analysis or conventional view:** considers data and the processes that transform the data as separate entities.
  - Data objects are modeled in a way that defines their attributes and relationships.
  - Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system.
  - **Object-oriented analysis,** focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements

Flow Oriented Modeling

- Although data flow-oriented modeling is perceived as an outdated technique by some software engineers, it continues to be one of the most widely used requirements analysis notations in use today.
- Although the *data flow diagram (DFD) and* related diagrams and information are not a formal part of UML, they can be used to complement UML diagrams and provide additional insight into system requirements and flow.
- The DFD takes an input-process-output view of a system.
- Data objects flow into the software, are transformed by processing elements, and resultant data objects flow out of the software.
- Data objects are represented by labeled arrows, and transformations are represented by circles (also called bubbles).
- The DFD is presented in a hierarchical fashion. That is, the first data flow model (sometimes called a level 0 DFD or *context diagram) represents the system as a whole. Subsequent data* flow diagrams refine the context diagram, providing increasing detail with each subsequent level.

- Guidelines for DFD
  - (1) the level 0 data flow diagram should depict the software,/system as a single bubble
  - (2) primary input and output should be carefully noted
  - (3) refinement should begin by isolating candidate processes, data objects, and data stores to be represented at the next level
  - (4) all arrows and bubbles should be labeled with meaningful names
  - (5) information flow continuity must be maintained from level 1 to level 2
  - (6) one bubble at a time should be refined.

Chapter 8 Design Concepts

- Once the step of analysis is completed (i.e. requirements engineering is done) we have to perform the design activity.
- Design is a meaningful <u>engineering representation</u> of something that is to be built.
- It is the place where <u>creativity rules</u>.
- It is where customer requirements, business needs and technical considerations all come together in the formulation of a product or a system.
- **<u>Design</u> is the <u>first technical activity</u> (other two are <u>coding</u> and <u>testing</u>) <u>required to</u> <u>build and verify</u> the software. It sits at the technical kernel of software engineering.**

Design Process and Design Quality

- Design activity is extremely important in software engineering as the quality of software (and hence ultimately customer satisfaction) depends to a large extent on this activity.
- If quality of design is good then the quality of software will be good.
- Software design is an iterative process. Initially the design is represented at a high level of abstraction (generalization, concept, idea). As design iterations occur, subsequent refinement leads to lower level of abstraction.
- Since design has direct impact on quality of the final software we have to continuously assess the quality through FTRs or walkthroughs.
- We need some guidelines or characteristics to evaluate the design. The characteristics you can look for evaluation are:

1. Whether the design takes care of all explicit and implicit requirements or not.
2. Whether it is readable and understandable to those who are going to generate code based on this, test the software and support the software.
3. Whether it gives complete picture of the software (i.e. addresses data, functional and behavioral domains) from an implementation point of view.

Quality Attributes

- You can also check the design model for following attributes suggested by Hewlett Packard to evaluate the quality of the design model. They are known by their acronym FURPS.
- Functionality, Usability, Reliability, Performance, Supportability.
  - Functionality – feature set and program capabilities
  - Usability – human factors (aesthetics, consistency, documentation)
  - Reliability – frequency and severity of failure
  - Performance – processing speed, response time, throughput, efficiency
  - Supportability – maintainability (extensibility, adaptability, serviceability), testability, compatibility, configurability
- Not all of these attributes are equally important for all applications. Their importance will depend on the type of application.

Design concepts

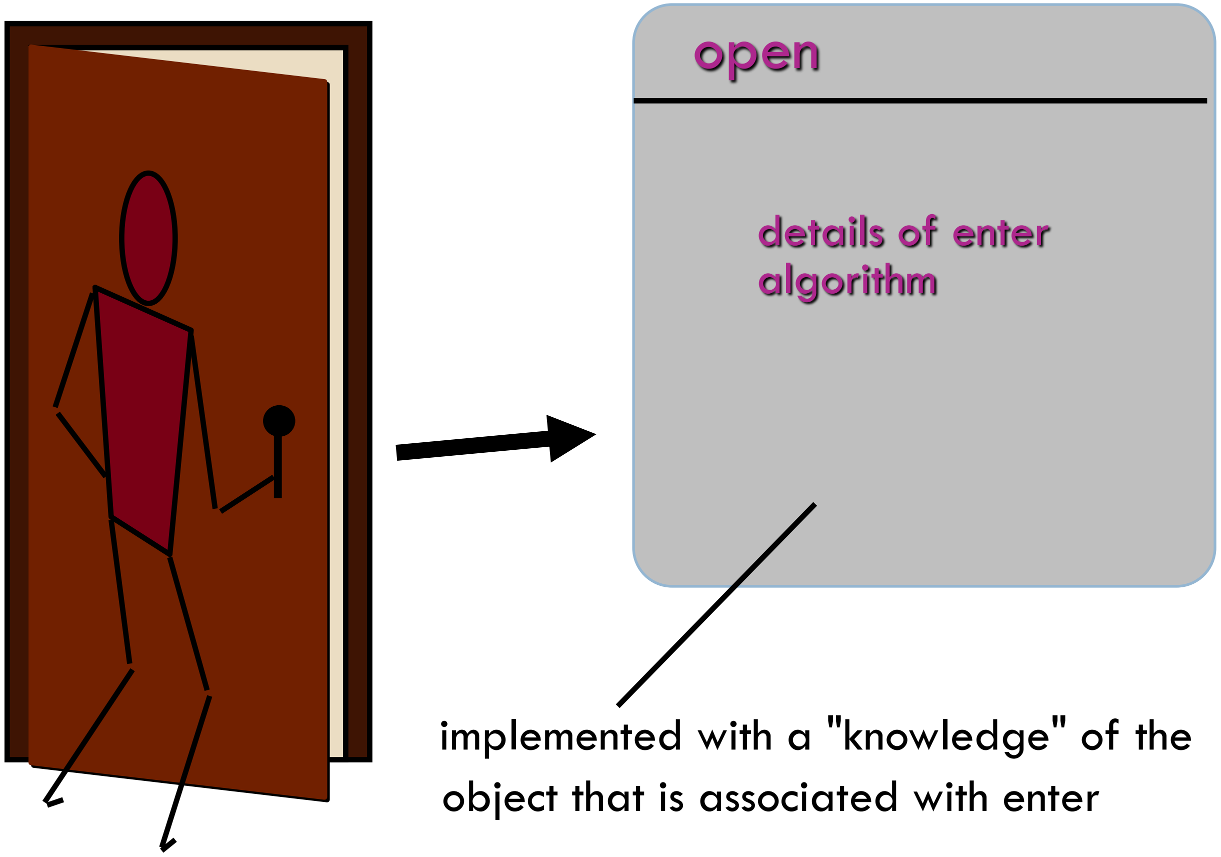
- Some of the important design concepts that we need to know and understand to produce a good design are as follows:
- abstraction—data, procedure, control
- architecture—the overall structure of the software
- patterns—”conveys the essence” of a proven design solution
- modularity—compartmentalization of data and function
- Information hiding—controlled interfaces
- Functional independence—single-minded function and low coupling
- refinement—elaboration of detail for all abstractions
- Refactoring—a reorganization technique that simplifies the design

Abstraction

- Abstraction is one of the fundamental ways that we as humans cope up with complexity.
- It is the idea of looking at a problem and the solution at the highest level using general terminology and then moving down slowly towards the implementation details.
- The different levels at which we see the problem and solution are called different levels of abstractions.
- We go to different levels of abstraction by the process of refinement.
- As we move through different levels of abstraction we work to create procedural and data abstractions.

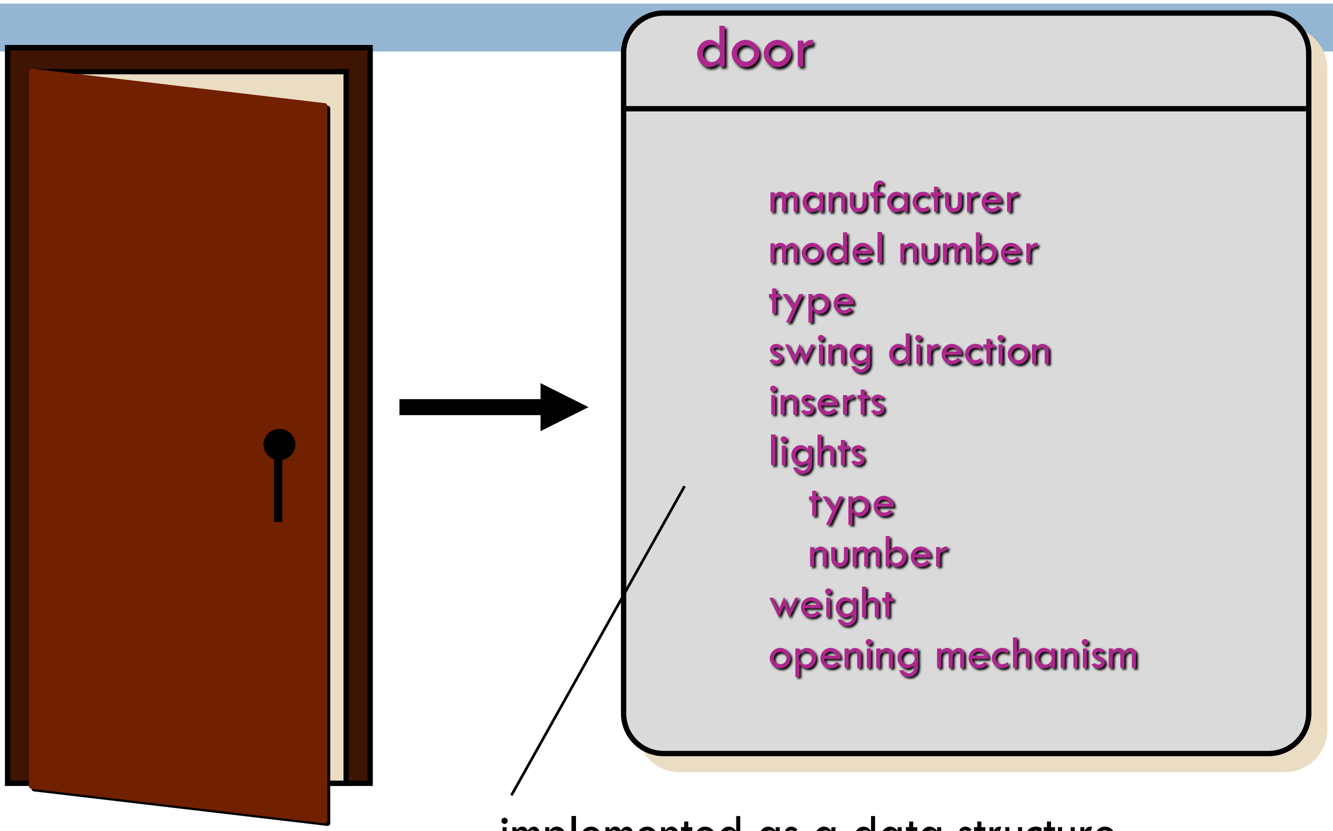
- A procedural abstraction is a named sequence of instructions that has a specific and limited function.
  - For example <u>Calculate salary</u> can be refined into get attendance data, calculate net basic, calculate net DA, calculate net HRA, calculate gross, calculate PF, calculate professional tax, calculate loan deduction, calculate total deduction, calculate net salary etc. to get a lower level of abstraction.
  - Then each of them can (for example how to calculate net basic) be refined further to get one more lower level of abstraction.
- Data abstraction is a named collection of data that describes a data object.
  - By refining the data object we get to the lower level of data abstraction.
  - For example when you refine a data object called door you get its attributes like door type (revolving, sliding, normal), swing direction (in ,out, both) , weight, dimensions (LWH) etc. i.e. you are going to lower level of abstraction.

Procedural Abstraction



implemented with a "knowledge" of the object that is associated with enter

Data Abstraction



implemented as a data structure

Architecture

- Architecture is:

  - structure of program components

  - the manner in which these components interact

  - structure of data that are used by the components

Design Patterns

- A pattern is a named piece of insight which conveys the essence of a proven solution to a recurring problem within a certain context
- A design pattern describes a design structure that solves a particular design problem within a specific context
- The intents are to determine :

  - whether the pattern is applicable to the current work

  - whether the pattern can be reused

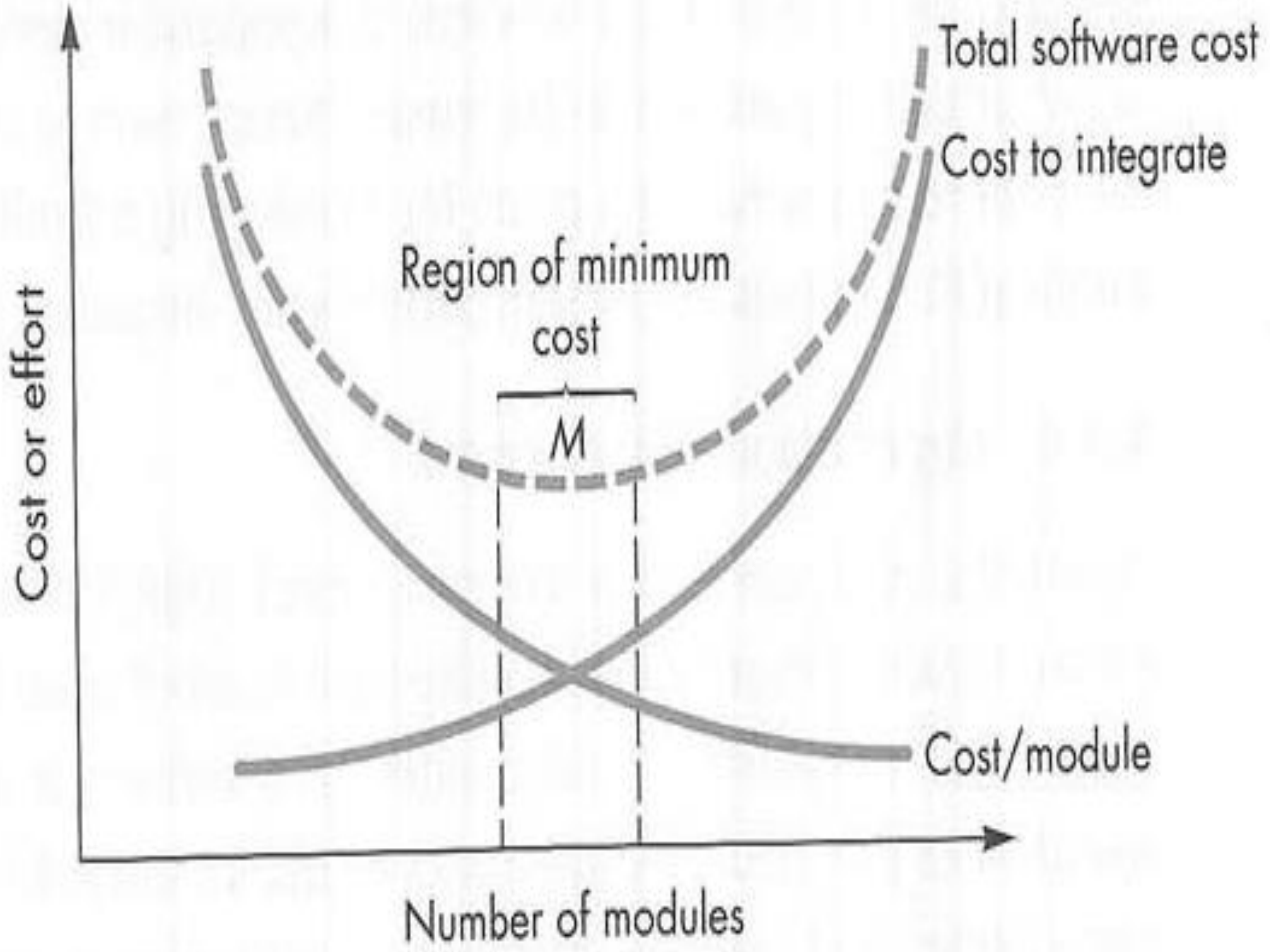
  - whether the pattern can serve as a guide for developing a similar , but functionality or structurally different pattern

Separation of concerns

- **Break up a large problem in smaller parts.**
- Any complex problem is solvable by subdividing it into pieces that can be solved independently.
- The perceived complexity of two problems when they are combined is often greater than the sum of perceived complexity when each is taken separately.
- This leads to divide and conquer strategy.
- If there are two problems p1 and p2 with complexity of p1 greater than that of p2 [i.e. $c(p1) > c(p2)$] then the effort required to solve the problem p1 will be more than that of p2 [i.e. $E(p1) > E(p2)$].
- If these two problems are combined to make one problem p1 + p2 then people feel that the complexity of combined problem is more than the sum of individual complexities i.e. $c(p1 + p2) > c(p1) + c(p2)$.
- Hence the same rule is applied for effort requirement. $E(p1 + p2) > E(p1) + E(p2)$.

Modularity

- Modularity is the concept of <u>breaking up a problem into separately named and addressable components</u> (modules) that can be integrated to satisfy the problem requirement.
- It is always easier to solve smaller components of a problem and then integrate the components to get the solution to the problem.
- So people like to divide the problem and then conquer it. But how much division should be there?
- Breaking up the problem into smaller and smaller parts will make the effort required for every part very small but the <u>effort required to integrate</u> the parts will <u>start increasing as number of parts grows</u>.
- There is a total cost or effort curve that can help us in deciding the ideal number of modules. Figure 8.2 page 226. X axis – no of modules. Y axis cost or effort of development and integration.


Total software cost
Cost to integrate
Region of minimum cost
M
Cost or effort
Cost/module
Number of modules

Information Hiding

- Modules should be specified and designed so that information (procedure and data) contained within one module is inaccessible to other modules that have no need for such information.
- Abstraction helps to define the procedural entities that make up the software.
- (Partitioning of books in the library should be such that books required for only BCA are hidden from the view of MCA students and vice versa.)
- The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance.
- Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

<u>Functional Independence</u>

- Everybody understands the importance of modularity in general and hence tries to solve every problem by breaking it up into number of modules.
- A modular design reduces complexity, facilitates change and results in easier implementation by encouraging parallel development of different parts of the system.
- When a system is modularized the <u>modules should be functionally independent</u>.
- This functional independence can be achieved by developing modules with "<u>single minded"</u> function and dislike <u>to excessive interaction</u> with other modules.
- In other words each module should address a specific sub-function of requirements and should have a simple interface (flow of information between modules should be less).

- Independent modules are easier to develop/maintain/test and error propagation is reduced.
- Functional independence is a key to good design and design is the key to software quality.
- How do you measure functional independence? There are two qualitative criteria cohesion and coupling.

COHESION - the degree to which a module performs one and only one function.

COUPLING - the degree to which a module is "connected" to other modules in the system.

- Cohesion
  - Cohesion (unity, hanging together) is a natural extension of the information hiding concept.
  - A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program.
  - A cohesive module should ideally do just one thing.
  - (Teaching team for MCA should be cohesive i.e. should not teach BCA subjects or teach at other institutions or do something else. MCA teaching team is hidden from BCA students.)

- <u>Coupling</u>
  - Coupling is a measure of interconnection among modules in a software structure.
  - In software design we try for <u>lowest possible coupling</u>.
  - Simple connectivity among modules results in software that is easier to understand and less prone to a “ripple effect”, caused when errors occur at one location and propagate through a system.

Refactoring

- It is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior.
- Another definition is refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the component yet improves its internal working.
- 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.
- Breaking up of a non-cohesive component into 2 or 3 cohesive parts.

Object-oriented design concepts

- OO concepts:
  - classes
  - objects
  - inheritance
  - message passing
  - polymorphism

Chapter 9 Architectural Design

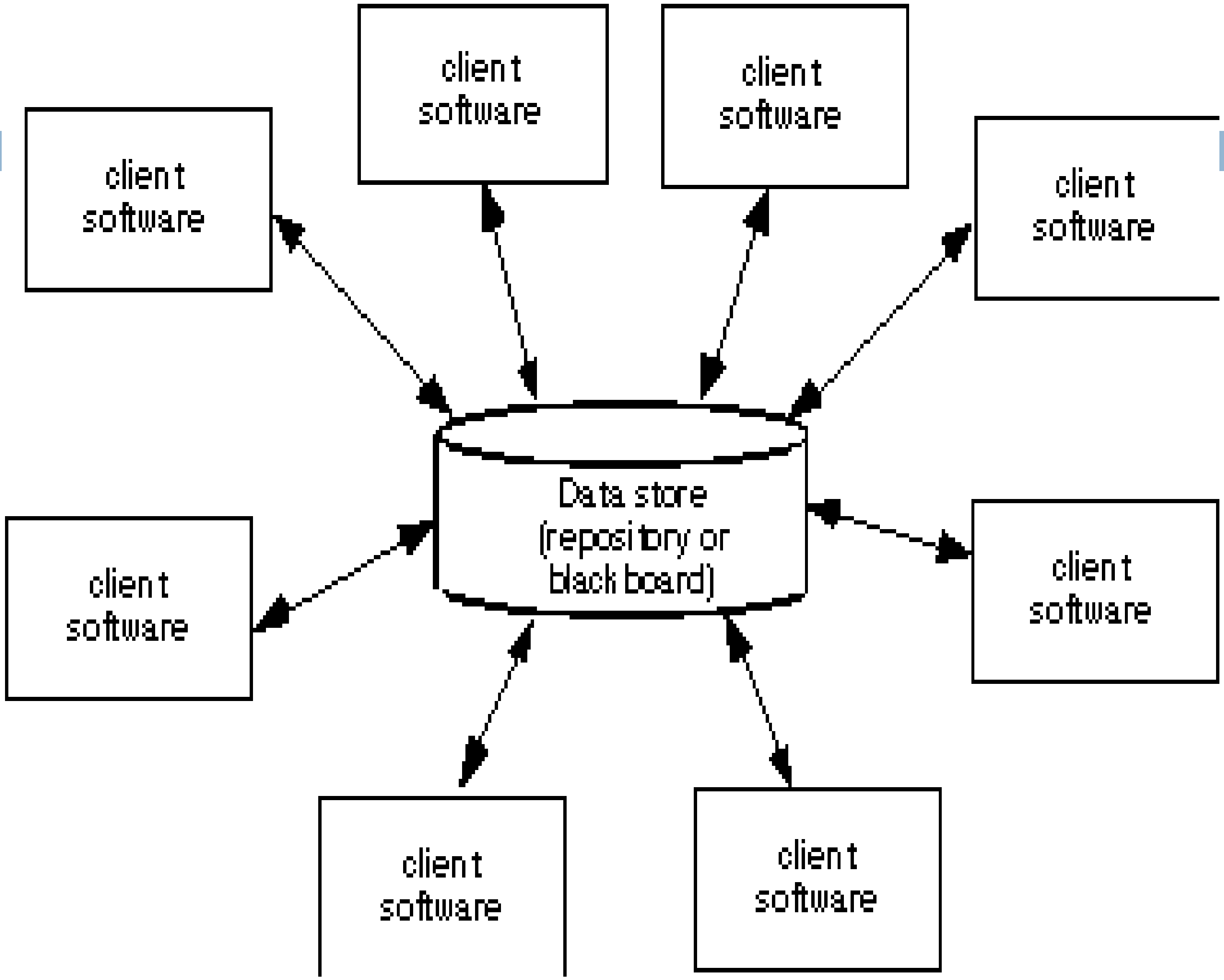
- **What is software Architecture of a computing system?**
  - It is
    - the structure of the system consisting of its various components,
    - the externally visible properties of those components and
    - the relationships among them.
  - At the architectural level internal properties like details of algorithms are not specified. They come at component level.
  - The architecture is a representation that enables you 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 s/w.

Architectural styles

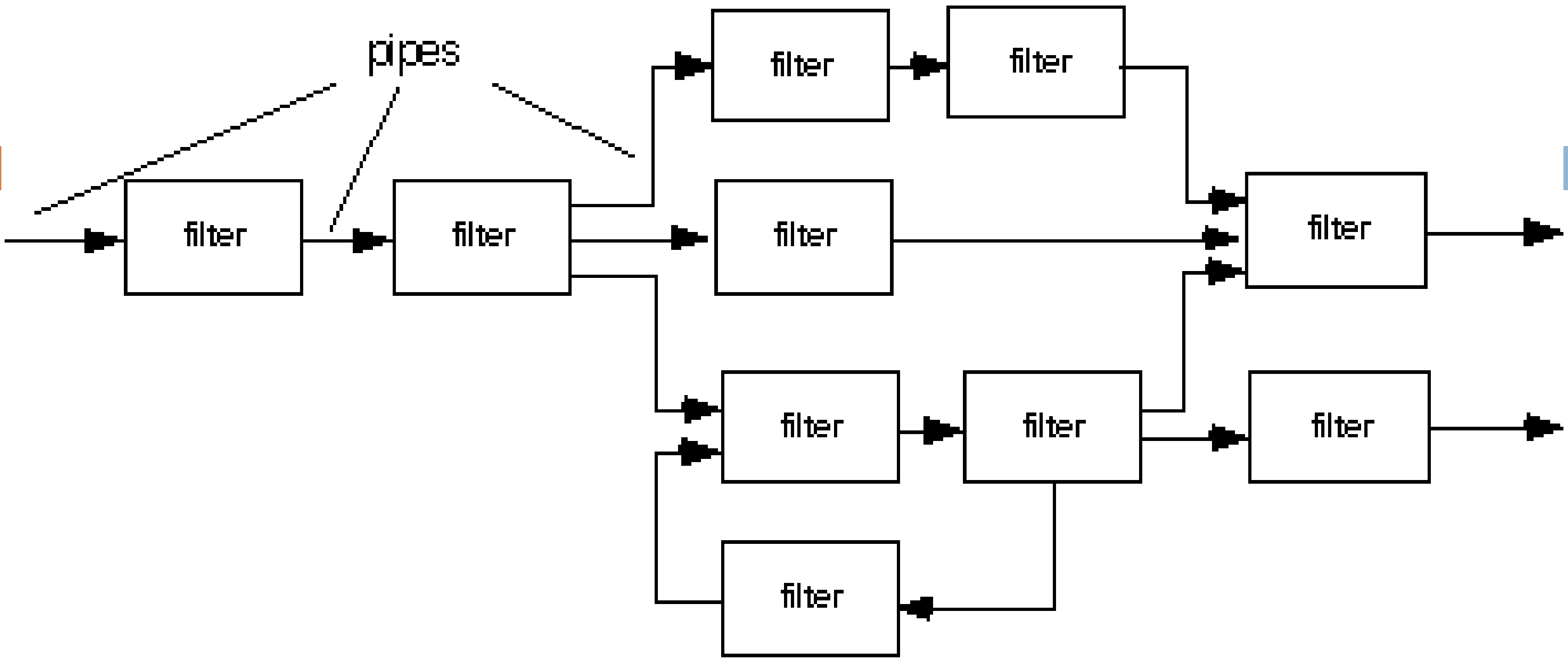
- Each style describes a system category that encompasses:
  1. a set of components (e.g., a database, computational modules) that perform a function required by a system,
  2. a set of connectors that enable "communication, coordination and cooperation" among components
  3. constraints that define how components can be integrated to form the system, and
  4. semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.
- An architectural style is a transformation that is imposed on the design of an entire system.
- Out of millions of computer-based systems developed so far in the last 50 years the vast majority can be categorized into one of a relatively small number of architectural styles. They are as follows:

Taxonomy of architectural styles

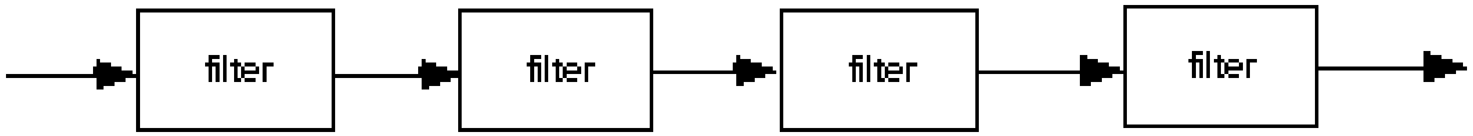
- **Data-centered architecture:**
- A data-store like a file or a database resides at the center of this architecture and is accessed frequently by other components that update, delete, or otherwise modify data within the store.
- The means of communication distinguishes the two subtypes:
  - repository and blackboard
- Repository: a client sends a request to the system to perform a Necessary action (eg insert date)
- Blackboard: The system sends notification and data to subscribers when data of interest changes, and is thus active
- This style promotes integrability i.e. existing components can be changed and new client components can be added to the architecture without concern about other clients since the client components operate independently.


client software
client software
client software
client software
client software
client software
client software
client software
Data store (repository or black board)

- **<u>Data-flow architecture</u>**
- This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data.
- This style can be described as a pipe and filter pattern. <u>Components are called filters</u>.
- They transform data and transmit them through the pipes to the next filter.
- Each filter works independently of those components upstream and downstream.
- It expects data input in a certain form and generates output of a specified form.
- It does not require knowledge of the working of its neighboring filters. (Seems like assembly line. Result processing software for B Sc ?).
- If the data flow degenerates into a single line of transforms, it is termed batch sequential. This pattern accepts a batch of data and then applies a series of sequential components (filters) to transform it. (Result processing for lower standards. All students study the same subjects.)
- (There is no data flow in data-centered architecture. Data stay in the center they do not flow.)

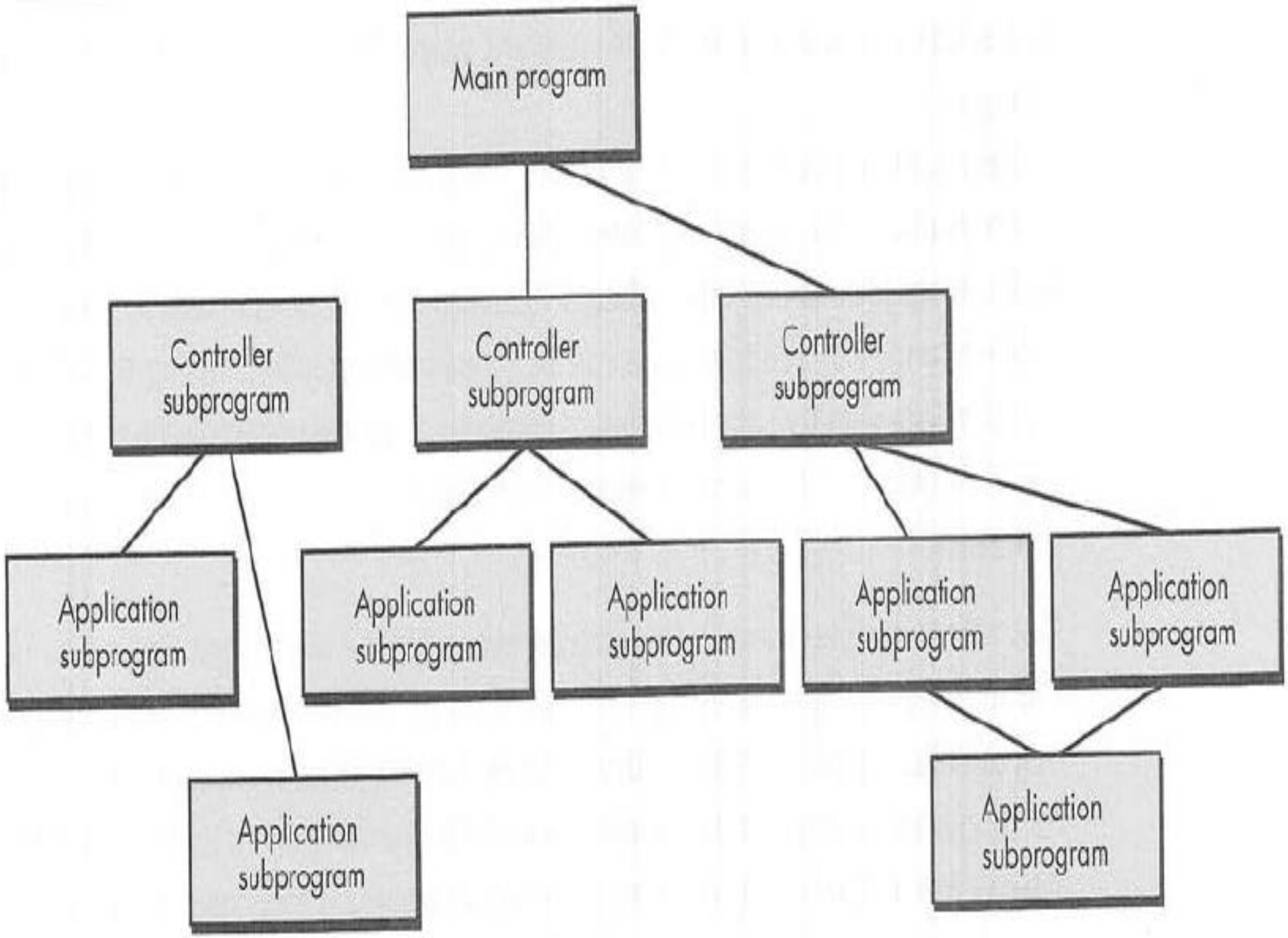


(a) pipes and filters



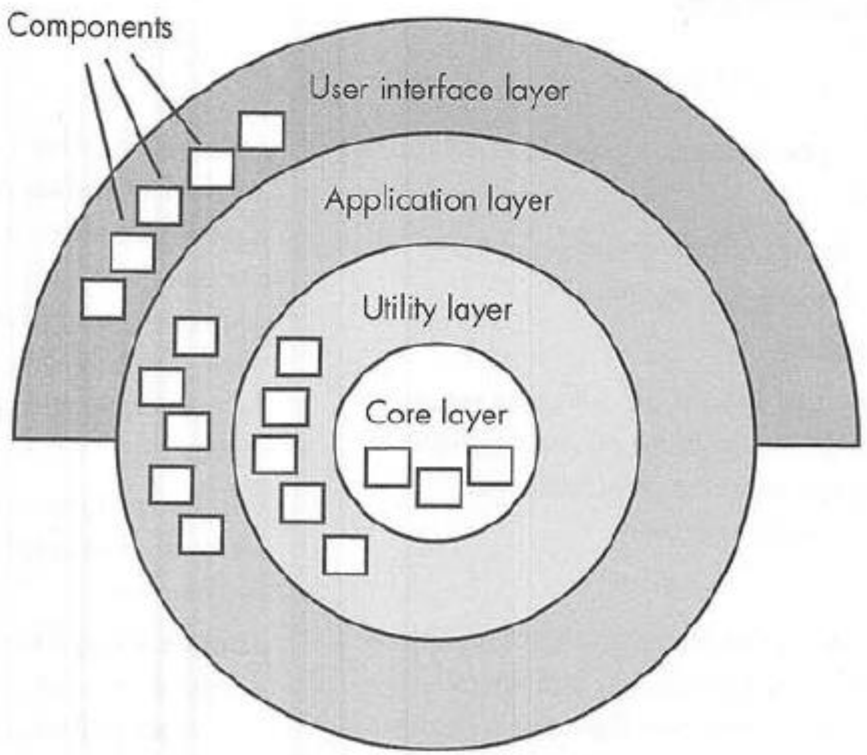
(b) batch sequential

- **<u>Call and return architecture</u>**
- When one requires flexibility to modify the program structure one can use this type of architecture. There are number of sub-styles in this type of architecture like
- Main program / subprogram architectures where the program structure is decomposed into a control hierarchy i.e. a main program invokes a number of program components which in turn may invoke still other components. MS Word or Excel?
- Remote procedure call architectures where the components of the main program / subprogram are <u>distributed across multiple computers on a network</u>.


Main program
Controller subprogram
Controller subprogram
Controller subprogram
Application subprogram
Application subprogram
Application subprogram
Application subprogram
Application subprogram
Application subprogram
Application subprogram

- **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 is accomplished by message passing.

- <u>**Layered architectures**</u>
- In this architecture a number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
- At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions. (OS?)


Components
User interface layer
Application layer
Utility layer
Core layer