

SOFTWARE ENGINEERING

Introduction to Software Engineering

Software is

- Instructions (computer programs) that when executed provide desired features, function, and performance**
- Data structures that enable the programs to adequately manipulate information**
- Documents that describe the operation and use of the programs**
- Computer programs and associated documentation such as requirements, design models and user manuals.**

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 is about creating quality. Better quality leads to reduced rework. Reduced rework results in faster delivery times

Software Affliction

**The word “affliction “ is defined as anything causing pain or distress”
“Chronic affliction” is “lasting a long time or recurring often;
continuing indefinitely”.**

Affliction refers to problems that are encountered in the development of computer software. The problems are not limited to software that “doesn’t function properly”, rather, the affliction encompasses problems associated with how we develop software how we maintain a growing volume of existing software and how we can expect to keep pace with a growing demand for more software.

Software Affliction

Common software problems

- Software can cost hundreds or thousands of dollars (Rupees) per line**
- Lifetime maintenance costs are high**
- Software is late or fails**
- Software is not performant (too slow)**
- Software is incomprehensible**
- Software is more trouble to use than it is worth**
- Millions are spent for an incomprehensible tool that comes late just to cause trouble and we don't have answers**

Software Affliction

Past Approaches to solutions

- **Use more people**
- **Create better programming languages**
- **Design before writing**
- **Start by base lining requirements**
- **Train people better**

Create more chaos

**Bad programs can be
written in any language**

**Are you designing the
right program**

But they change

To do what?

Software Affliction

Famous Software Failures

- **AT&T long distance Service fails for 9 hours (wrong BREAK statement in C code – 1990)**
- **Mars Climate Orbiter (September 23rd 1999) , the 125 million dollar Mars Climate Orbiter lost by officials at NASA. The failure attributed to a failure in NASA's system engineering process. The process did not specify the system of measurement to be used on the project. As a result one of the development teams used imperial measurement while the other used metric system of measurement. When the parameters are passed from one module to the other during the orbit navigation, no conversion performed resulting in the loss of the craft**

Software Affliction

Famous Software Failures

- **E-mail buffer overflow (1998)- Several e-mail systems suffer from “ buffer overflow error” when extremely long e-mail addresses are received. The internal buffers receiving the addresses do not check for length and allow the buffers to overflow causing the applications to crash. Hostile hackers use this fault to trick the computer into running malicious program in its place**

Software Engineering

The economies of ALL developed nations are dependent on software.

More and more systems are software controlled

Expenditure on software represents a significant fraction of the National Income in all developed countries.

Software costs often dominate computer system costs.

The costs of software on a PC are often greater than the hardware cost.

Software costs more to maintain than it does to develop. For systems with a long life, maintenance costs may be several times development costs.

Software Engineering

The field of software engineering was born in 1968 in response to chronic failures of large software projects to meet schedule and budget constraints

– Recognition of "the software crisis"

Software Engineering term became popular after NATO Conference in Garmisch Partenkirchen (Germany), 1968

Software Engineering

Engineering

- The application of scientific and mathematical principles to practical ends such as the design, manufacture, and operation of efficient and economical structures, machines, processes, and systems.
- (Business / Professions) the profession of applying scientific principles to the design, construction, and maintenance of engines, cars, machines, etc. (mechanical engineering), buildings, bridges, roads, etc. (civil engineering), electrical machines and communication systems (electrical engineering), chemical plant and machinery (chemical engineering), or aircraft (aeronautical engineering)

Software Engineering

Software Engineering

- Software Engineering is the establishment and use of sound engineering principles in order to produce software that is reliable and works efficiently on real machines**
- Software Engineering is the application of systematic , disciplined, quantifiable approach to the development, operation, and maintenance of software that is, the application of engineering to software**
- Software engineering is an engineering discipline that is concerned with all aspects of software production.**

Software Engineering

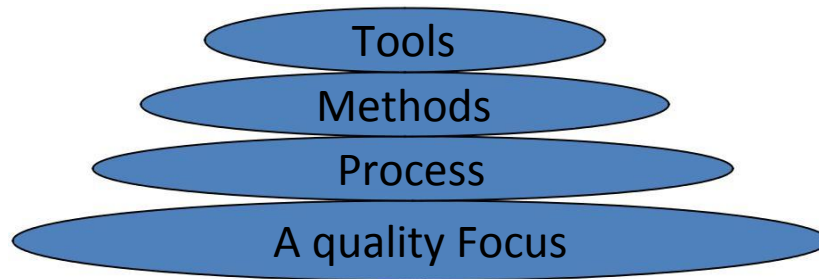
Software Engineering

- Software engineers should adopt a systematic and organised approach to their work and use appropriate tools and techniques depending on the problem to be solved, the development constraints and the resources available.**
- System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this process concerned with developing the software infrastructure, control, applications and databases in the system.**

Software Engineering – Layered Technology

Software Engineering

- **Software Engineering is a layered technology and rests on an organizational commitment to quality. The foundation for software engineering is the process layer. Software Engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software**



- **Software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products (models, documents, data, reports, forms etc) are produced, milestones are established, quality is ensured, and change is properly managed.**

Software Engineering –Generic View

Engineering is the analysis, design, construction, verification, and management of technical (or social) entities. Questions that are to be asked and answered:

What is the problem to be solved?

What characteristics of the entity are used to solve the problem?

How will the entity (and the solution) be realized?

How will the entity be constructed?

What approach will be used to uncover errors that were made in the design and construction of the entity?

How will the entity be supported over the long term, when corrections, adaptations, and enhancements are requested by users of the entity.

Software Engineering –Generic View

The work associated with software engineering can be categorized into three generic phases, regardless of application area, project size, or complexity

- The *definition phase* focuses on *what*. That is, during definition, the software engineer attempts to identify what information is to be processed, what function and performance are desired, what system behavior can be expected, what interfaces are to be established, what design constraints exist, and what validation criteria are required to define a successful system. The key requirements of the system and the software are identified.

Software Engineering –Generic View

- The *development phase* focuses on *how*. That is, during development a software engineer attempts to define how data are to be structured, how function is to be implemented within a software architecture, how procedural details are to be implemented, how interfaces are to be characterized, how the design will be translated into a programming language (or nonprocedural language), and how testing will be performed.

Software Engineering –Generic View

- The *support phase* focuses on *change* associated with error correction, adaptations required as the software's environment evolves, and changes due to enhancements brought about by changing customer requirements.
- Four types of change are encountered during the support phase
 - Correction.** Even with the best quality assurance activities, it is likely that the customer will uncover defects in the software. *Corrective maintenance* changes the software to correct defects.
 - Adaptation.** Over time, the original environment (e.g., CPU, operating system, business rules, external product characteristics) for which the software was developed is likely to change. *Adaptive maintenance* results in modification to the software to accommodate changes to its external environment

Software Engineering –Generic View

Enhancement. As software is used, the customer/user will recognize additional functions that will provide benefit. *Perfective maintenance* extends the software beyond its original functional requirements.

Prevention. Computer software deteriorates due to change, and because of this, *preventive maintenance*, makes changes to computer programs so that they can be more easily corrected, adapted, and enhanced

Software Engineering –Key Challenges

Coping with legacy systems, coping with increasing diversity and coping with demands for delivery times.

Legacy systems - old, valuable systems must be maintained and updated.

Heterogeneity - systems are distributed and includes a mix of hardware and software

Delivery - there is increasing pressure for faster delivery of software.

Software Standards

A software standard is a standard, protocol, or other common format of a document, file, or data transfer accepted and used by one or more software developers while working on one or more than one software programs. Software standards enable interoperability between different programs created by different developers.

Software standards consist of certain terms, concepts, data formats, document styles and techniques agreed upon by software creators so that their software can understand the files and data created by a different software program. To be considered a standard, a certain protocol needs to be accepted and incorporated by a group of developers who contribute to the definition and maintenance of the standard.

Software Standards

The protocols [HTML](#), [TCP/IP](#), [SMTP](#), [POP](#) and [FTP](#) are software standards that an application designer must understand and follow if their software expects to interface with these standards

In order for an email sent from [Microsoft Outlook](#) can be read from within the [Yahoo! Mail](#) application, the email will be sent using SMTP, which the different receiving program understands and can parse properly to display the email. Without a standardized technique to send an email, the two different programs would be unable to accurately share and display the delivered information.

Software Standards

Sources of Standards

- Standards organizations like WWW Consortium (W3C) and International Standards Organization (ISO) which consist of groups of larger software companies like Microsoft and Apple Inc. Representatives of these companies contribute their ideas about how to make a single, unified software standard to handle various problems they are facing and create standards for all parties to agree to a certain software standard that they all should use to make their software connect to each other

Open v. closed standards

- Standard can be a closed standard or an open standard. The documentation for an open standard is open to the public and anyone can create a software that implements and uses the standard. The documentation and specification for closed standards are not available to the public, enabling its developer to sell and license the code to manage their data format to other interested software developers

Process Framework

Process

- **A set of activities whose goal is the development or evolution of software.**
- **Generic activities in all software processes are:**
 - Specification - what the system should do and its development constraints**
 - Development - production of the software system**
 - Validation - checking that the software is what the customer wants**
 - Evolution - changing the software in response to changing demands.**

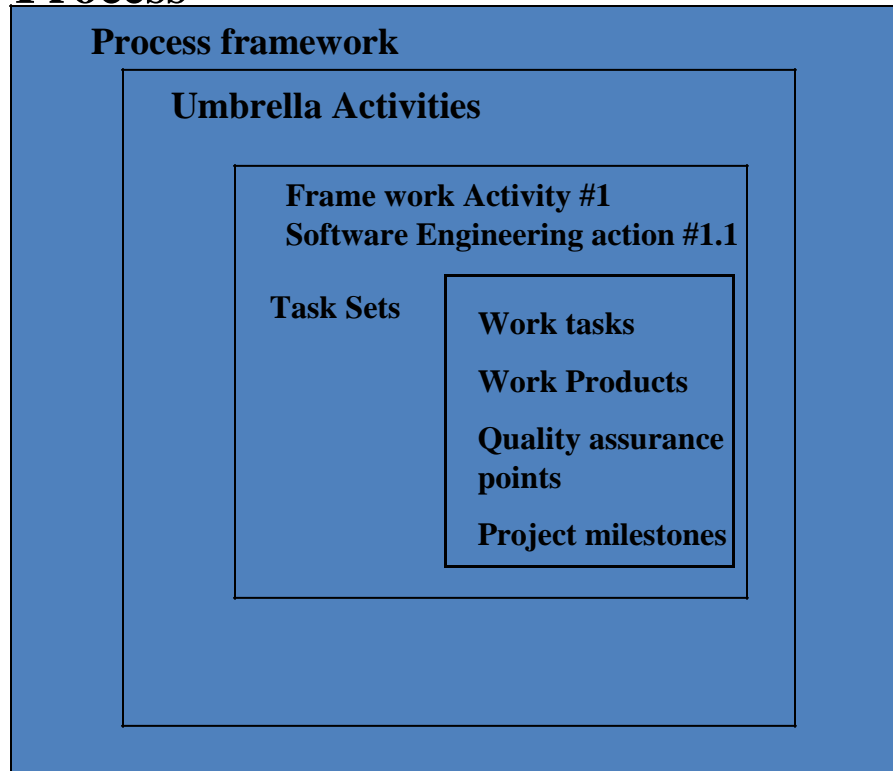
Process Framework

Process

- Software Engineering methods provide the technical “how to’s” for building software.**
- Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing and support**
- Software engineering tools provide automated or semi-automated support for the process and methods**

Process Framework

Software Process



Process Framework

Process framework

- A process framework establishes the foundation for a complete software process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size and complexity. It also contains a set of umbrella activities that are applicable across the entire software process**
- Each framework activity is populated by a set of software engineering actions – a collection of related tasks that produces a major software engineering work product. Each action is populated with individual work tasks that accomplish some part of the work implied by the action**

Process Framework

Process framework

– Generic framework activities

Communication involving communication and collaboration with the customer and encompasses requirements gathering and other related activities

Planning – this activity establishes a plan for the software engineering work that follows. It describes the technical task to be conducted, the risks that are likely, the resources that will be required, work products to be produced and a work schedule

Process Framework

Process framework

– Generic framework activities

Communication involving communication and collaboration with the customer and encompasses requirements gathering and other related activities

Planning – this activity establishes a plan for the software engineering work that follows. It describes the technical task to be conducted, the risks that are likely, the resources that will be required, work products to be produced and a work schedule

Process Framework

Process framework

– Generic framework activities

Modeling which encompasses the creation of models that allow the developer and customer to better understand software requirements and the design that will achieve those requirements

Construction activity combines code generation and the testing that is required to uncover errors in the code

Deployment involves delivery to the customer who evaluates the delivered product and provides feedback on the evaluation

Process Framework

Process framework

– Umbrella activities

Software Project Tracking and control – This allows the software team to assess progress against the project plan and take necessary action to maintain schedule

Risk management – assesses risks that may effect the outcome of the project or the quality of the product

Software Quality Assurance – defines and conducts the activities required to ensure software quality

Process Framework

Process framework

– Umbrella activities

Formal technical reviews – assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next action or activity

Measurement – defines and collects process, project and product measures that assist the team in delivering software that meets customers' needs

Software configuration management – manages the effects of change throughout the software process

Process Framework

Process framework

– Umbrella activities

Reusability management – defines criteria for work product reuse and establishes mechanisms to achieve reusable components

Work product preparation and production – Encompasses the activities required to create work products such as models, documents, logs, forms

– Umbrella activities are applied throughout the software process

Capability Maturity Model Integrated (CMMI)

Companies want to deliver products and services better, faster and cheaper. All the organizations have found themselves building increasingly complex products and services. A single company does not develop all the components that compose a product or service. Some components are built in-house and some are acquired, then all the components are integrated into the final product or service.

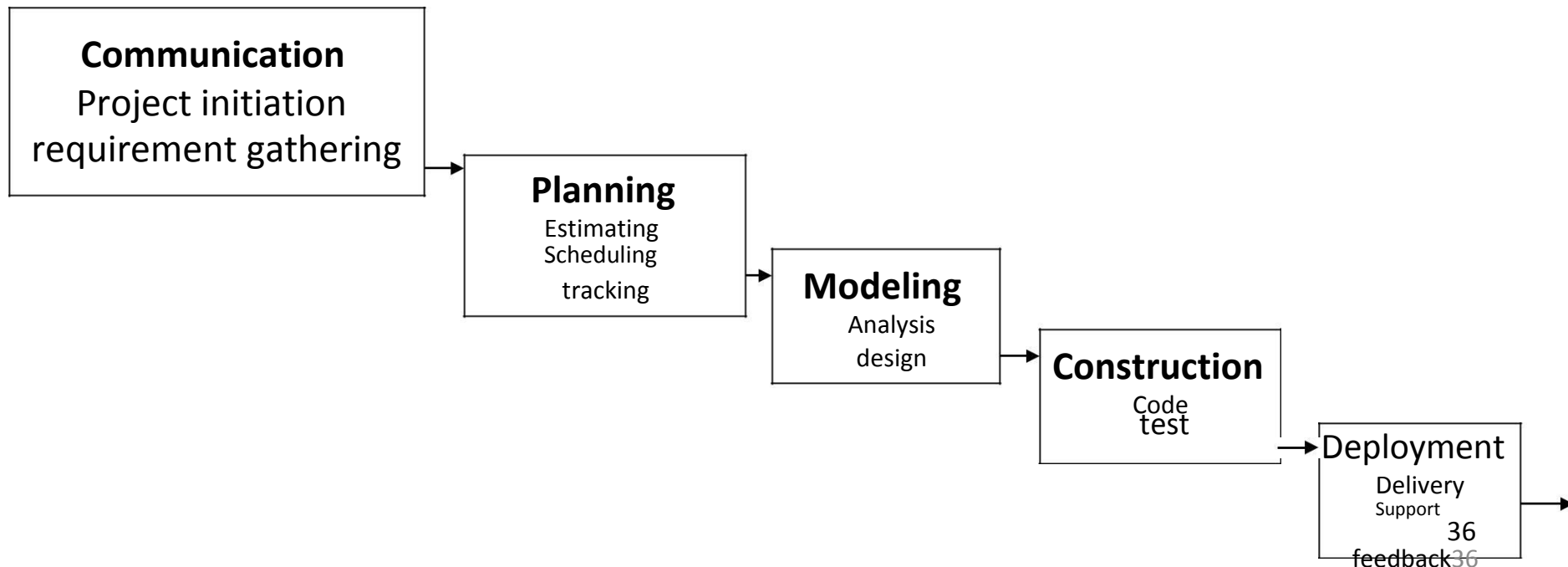
CMMI for development consists of best practices that address development and maintenance activities applied to products and services. It addresses practices that cover the product's lifecycle from conception through delivery and maintenance

Process Assessment

- **The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer needs or that it will exhibit the technical characteristics that will lead to long-term quality characteristics. Process should be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering. Different approaches of assessment are**
 - Standard CMMI Assessment Method for Process Improvement (SCAMPI)**
 - CMM-Based Appraisal for Internal Process Improvement (CBA IPI)**
 - ISO/IEC15504**
 - ISO 9001:2000 for software stresses the importance for an organization to identify, implement, manage, and continually improve the effectiveness of the processes that are necessary for the quality management system, and to manage the interactions of these processes in order to achieve the organization's objectives**

The Waterfall Model

This Model suggests a systematic, sequential approach to SW development that begins with customer specification of requirements and progresses through planning, modeling, construction and deployment, culminating in on-going support of the completed software



The Waterfall Model

Waterfall Assumptions

Requirements are known from the start, before design

Requirements are stable

The design can be done abstractly and speculatively i.e. it is possible to correctly guess in advance how to make it work

Everything will fit together when we start the integration

Pros and Cons

•Pro

Can be used for projects with well-defined requirements

Cons

Inflexible

Limited use of iteration, problems from earlier phases are hard to fix

Expects full requirements early

Working program is not available early

•High risk issues are not tackled early enough

THE INCREMENTAL PROCESS MODELS

Rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality.

User requirements are prioritised and the highest priority requirements are included in early increments.

Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve.

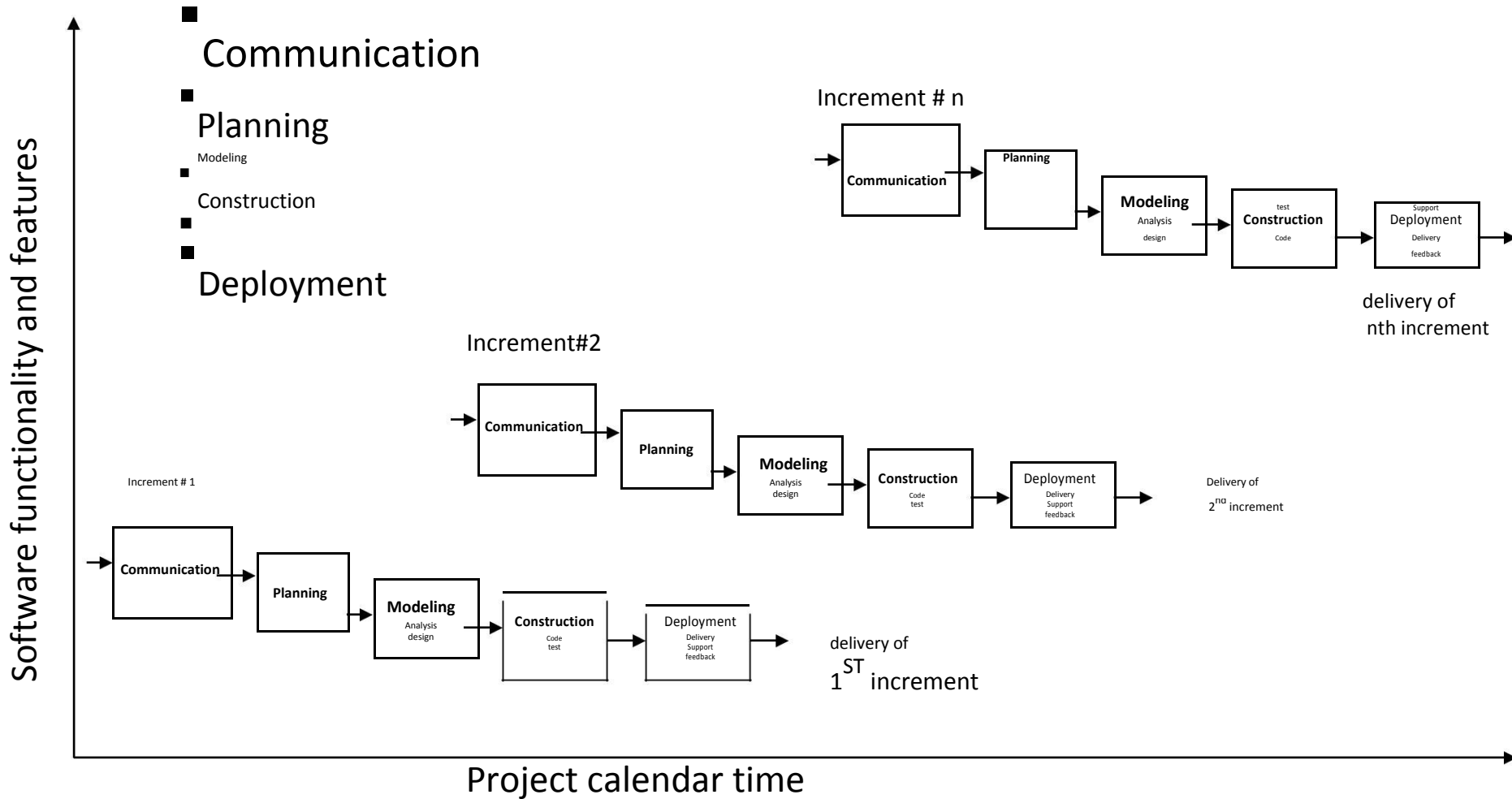
Customer value can be delivered with each increment so system functionality is available earlier.

Early increments act as a prototype to help elicit requirements for later increments.

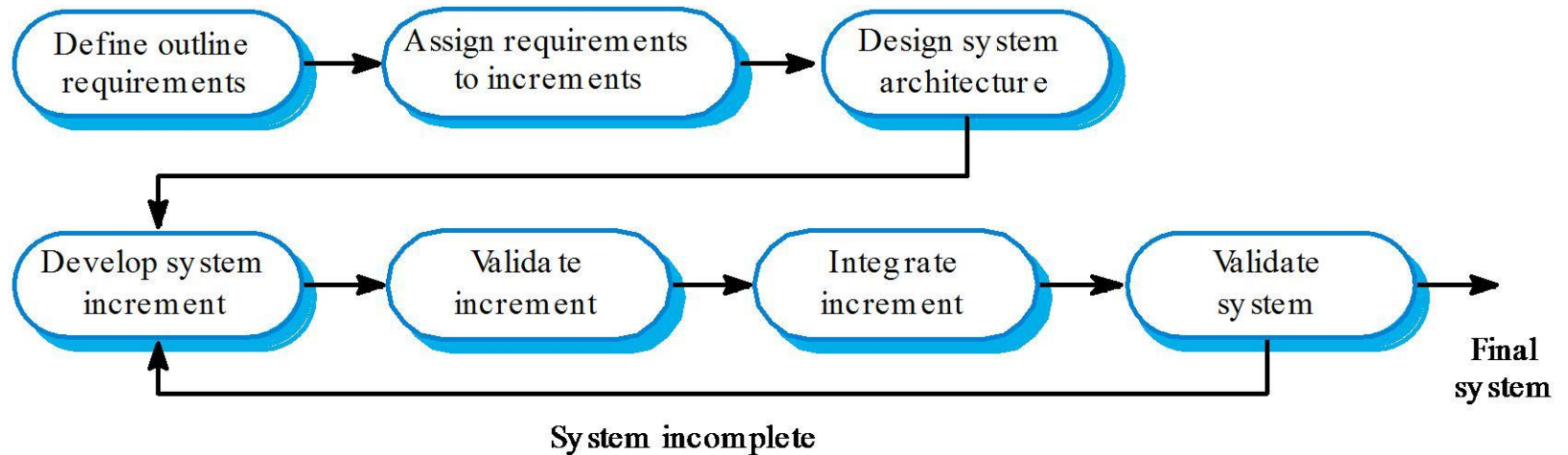
Lower risk of overall project failure.

The highest priority system services tend to receive the most testing.

The Incremental Model



The Incremental Model



The Incremental Model

Software releases in increments

1st increment constitutes Core product

Basic requirements are addressed

Core product undergoes detailed evaluation by the customer

As a result, plan is developed for the next increment

Plan addresses the modification of core product to better meet the needs of customer

Process is repeated until the complete product is produced

THE RAD MODEL

(Rapid Application Development)

An incremental software process model

Having a short development cycle

High-speed adoption of the waterfall model using a component based construction approach

Creates a fully functional system within a very short span time of 60 to 90 days

THE RAD MODEL

Multiple software teams work in parallel on different functions

Modeling encompasses three major phases: Business modeling, Data modeling and process modeling

Construction uses reusable components, automatic code generation and testing

Problems in RAD

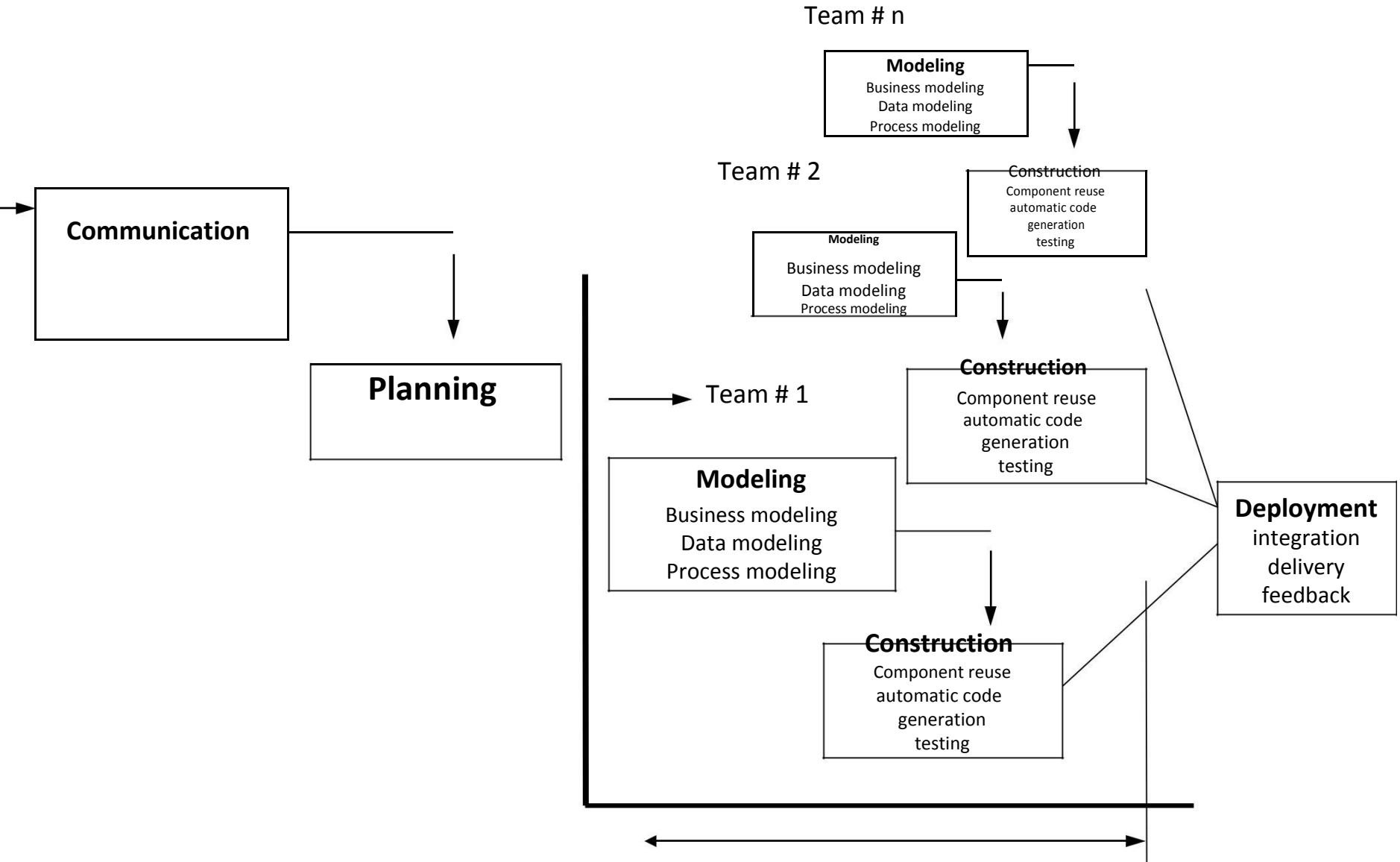
- Requires a number of RAD teams**

- Requires commitment from both developer and customer for rapid-fire completion of activities**

- Requires modularity**

- Not suited when technical risks are high**

The RAD Model



EVOLUTIONARY PROCESS MODEL

Software evolves over a period of time

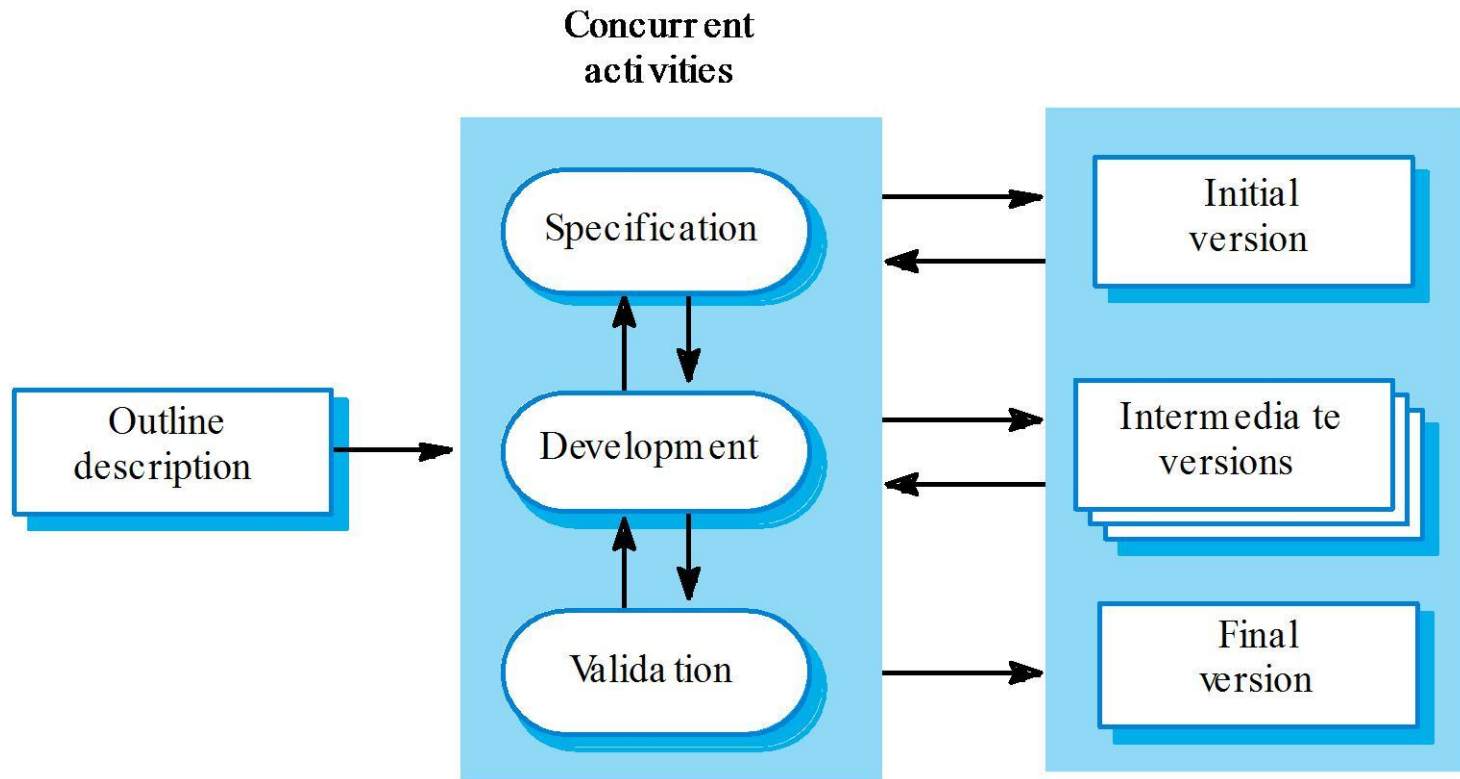
Business and product requirements often change as development proceeds making a straight-line path to an end product unrealistic

Evolutionary models are iterative and as such are applicable to modern day applications

Types of evolutionary models

- Prototyping
- Spiral model
- Concurrent development model

Evolutionary development



Evolutionary development

Problems

- Lack of process visibility;**
- Systems are often poorly structured;**
- Special skills (e.g. in languages for rapid prototyping) may be required.**

Applicability

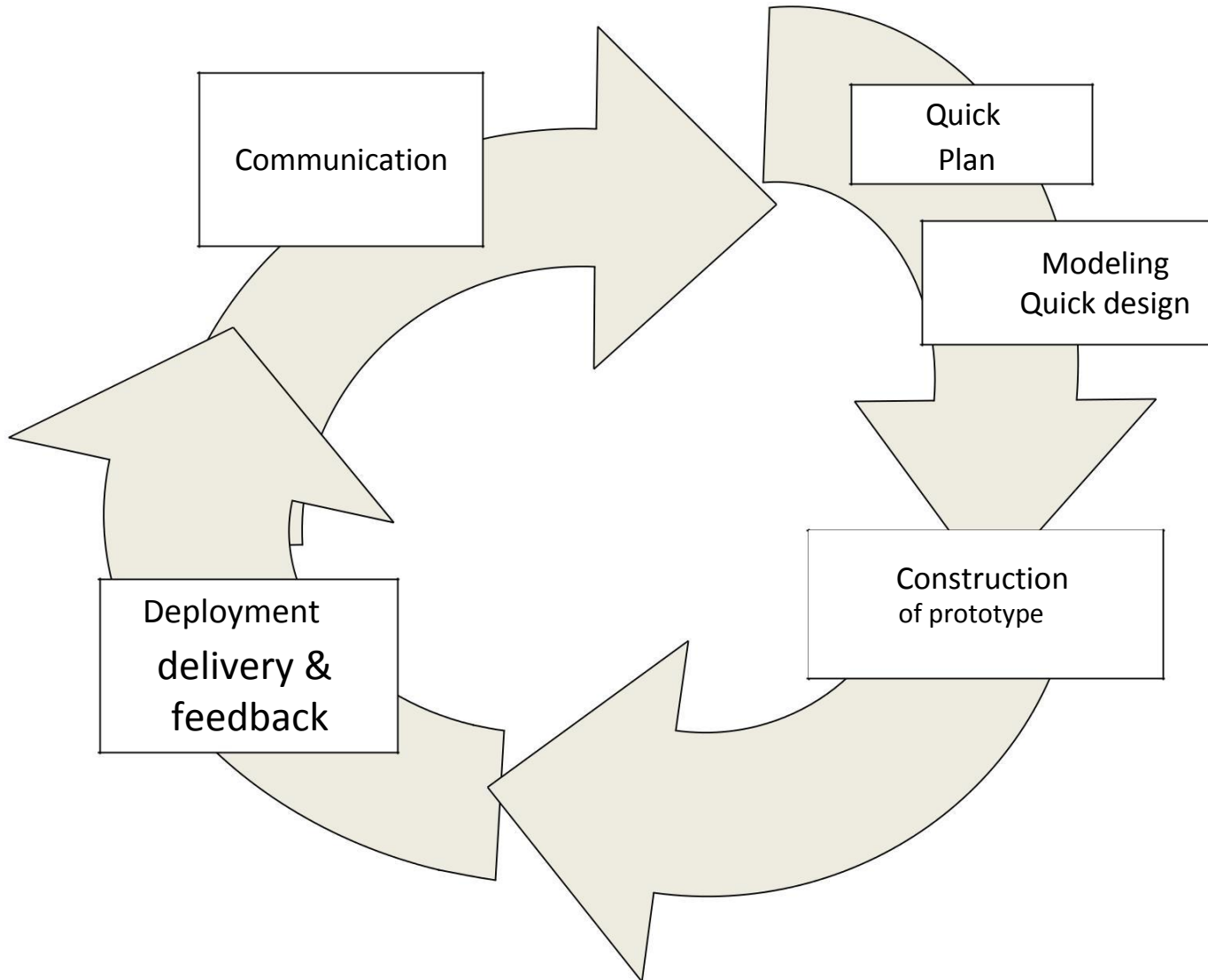
- For small or medium-size interactive systems;**
- For parts of large systems (e.g. the user interface);**
- For short-lifetime systems.**

Prototype

Prototyping

- **A prototype is a mock-up of the proposed system, which is released to the users during the requirements definition or early design stages**
- **Prototyping is frequently used to provide early feedback to customers and users to improve communication of requirements between the analysts and the users**
- **It acts as a vehicle for clarifying certain types of requirements vividly**
- **Specifications document built with the assistance of prototype tends to undergo fewer changes during and after the development**

Evolutionary Models: Prototype



Prototype

Prototyping

– Prototypes serve three major purposes

Clarify and complete the requirements- used as a requirement tool, the prototype is a preliminary implementation of a part of the system that is not well understood.

Explore design alternatives – Used as a design tool, a prototype lets stakeholders explore different user interaction techniques, optimize system usability, and evaluate potential technical approaches

Grow into the ultimate product – Used as a construction tool, a prototype is a functional implementation of an initial subset of the product, which can be elaborated into the complete product through a sequence of small-scale development cycles

Prototype

Prototyping

– Steps involved

Establish goals for the prototype – lay down and agree on the objectives of the prototype to be developed

Define prototype features and functionality – A detailed list of features and functions that are to be demonstrated in the prototype are documented, discussed and agreed to

Get the prototype ready – Prototype will be developed and reviewed against the list of features and functionality to ensure that all the required features and functionality are developed

Evaluate the prototype – Prototype is executed. Users' and developers' feedback is collected. The evaluation report is then used to elicit and analyze new requirements

Prototype

Prototyping

– Types of Prototypes

Horizontal Prototypes

- **A horizontal prototype is also called a behavioral prototype or a mock-up. It is called horizontal because it does not drive into all the layers of an architecture but primarily depicts a portion of the user interface**
- **Horizontal prototypes can demonstrate the functional options the user will have, the look and feel of the user interface and the information architecture**
- **Horizontal prototype does not perform any useful work, although it looks as if it should**

Prototype

Prototyping

– Types of Prototypes

Vertical prototypes

- **Also known as structural prototype or proof of concept implements a slice of application functionality from the user interface through the technical service layers.**
- **A vertical prototype works like the real system is supposed to work because it touches on all levels of the system implementation. Vertical prototypes are constructed using production tools in a production like environment.**
- **Vertical prototypes are used to explore critical interface and timing requirements and to reduce risk during design**

Prototype

Prototyping

– Throwing away prototype

A throwaway prototype is built to answer questions, resolve uncertainties, and improve requirements.

A throwaway prototype emphasizes quick implementation and modification over robustness, reliability, performance, and long-term maintainability

The throwaway prototype is most appropriate when the team faces uncertainty, ambiguity, incompleteness, or vagueness in the requirements

Prototype

Prototyping

– Evolutionary prototype

Evolutionary prototype provides a solid architectural foundation for building the product incrementally as the requirements become clear over time.

The tools, languages, platforms used to develop the prototype be the same as the actual development platform selected

To maximize the re-use of the prototype, important conventions for the design and programming be laid down before the development of the prototype begins

The prototype be reviewed, not only externally (behaviour) but also internally (design and code review)

Prototype

Prototyping

– Benefits of prototyping

Improved communication between the developers and the end-users

Increased user involvement

Ability to clarify hidden or ambiguous requirement

Quicker review of the Specifications

Better expectation setting of end-users

Prototype

Prototyping

– Drawbacks

A user may get disillusioned with a prototype since the prototype is not as robust as the system and does not contain the full functionality

A user may think that the system is “ready” by looking at the prototype and may think that the developers are taking him for a ride by asking for considerable amount of additional time

Developers may use “prototype” as an excuse to back the code, rather than to gather requirements

The cost of prototype is added to the project cost

Developers may start considering the prototype as substitute for the documented Specifications

The Spiral Model

The spiral model , also known as the spiral lifecycle model is a risk driven process model generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. The main features are

- Cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk
- A set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory solutions

This model of development combines the features of the prototyping model and the waterfall model. The spiral model is favored for large, expensive, and complicated projects.

The Spiral Model

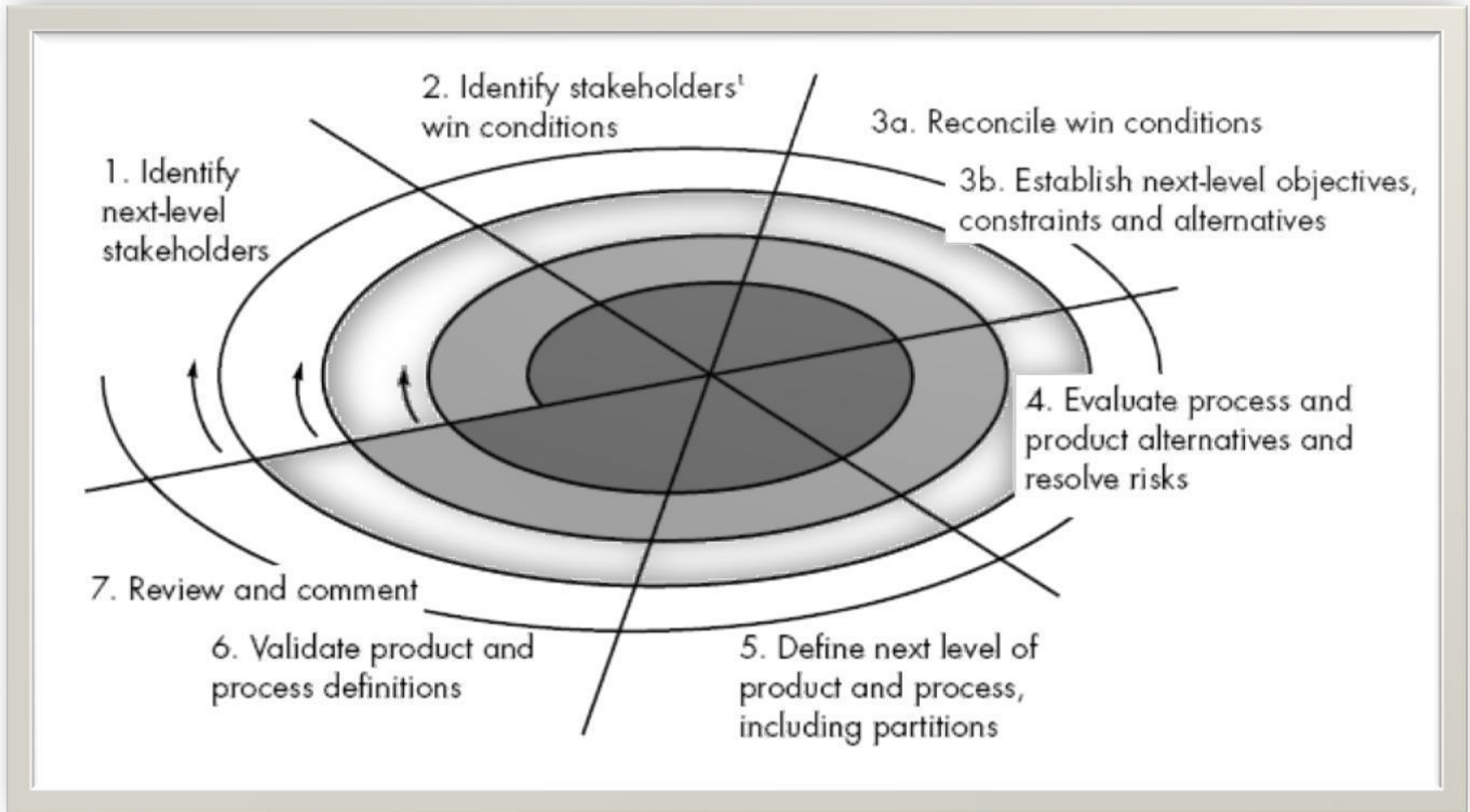
The steps in the spiral model can be generalized as follows:

- The new system requirements are defined in as much detail as possible. This usually involves interviewing a number of users representing all the external or internal users and other aspects of the existing system.
- A preliminary design is created for the new system.
- A first prototype of the new system is constructed from the preliminary design. This is usually a scaled-down system, and represents an approximation of the characteristics of the final product.
- A second prototype is evolved by a fourfold procedure:
 - Evaluating the first prototype in terms of its strengths, weaknesses, and risks
 - Defining the requirements of the second prototype
 - Planning and designing the second prototype
 - Constructing and testing the second prototype.

The Spiral Model

- At the customer's option, the entire project can be aborted if the risk is deemed too great. Risk factors might involve development cost overruns, operating-cost miscalculation, or any other factor that could, in the customer's judgment, result in a less-than-satisfactory final product.
- The existing prototype is evaluated in the same manner as was the previous prototype, and, if necessary, another prototype is developed from it according to the fourfold procedure outlined above.
- The preceding steps are iterated until the customer is satisfied that the refined prototype represents the final product desired.
- The final system is constructed, based on the refined prototype.
- The final system is thoroughly evaluated and tested. Routine maintenance is carried out on a continuing basis to prevent large-scale failures and to minimize downtime.

The Spiral Model



The Spiral Model

The spiral model is divided into a number of framework activities

COMMUNICATION

- *Tasks required are establish effective communication between developer

PLANNING

- *Estimation
- *Scheduling
- *Risk analysis

MODELING

- *Analysis
- *Design

CONSTRUCTION

- *Code
- *Test

DEPLOYMENT

- *Delivery
- *Feedback

Each region is populated by a series of work tasks.

The Spiral Model

Advantages:

- Software evolves as the process progresses

- The developer and customer better understand and react to risks at each evolutionary level

- Enables the developer to use prototyping approach at any stage in the evolution of the product

- It maintains the systematic stepwise approach of classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world

- The model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic

The Spiral Model

Disadvantages:

- It may be difficult to convince customers that the evolutionary approach is controllable

- It requires considerable risk assessment expertise and relies on this expertise for the success

- If a major risk is not uncovered and managed problems will occur

CONCURRENT DEVELOPMENT MODEL

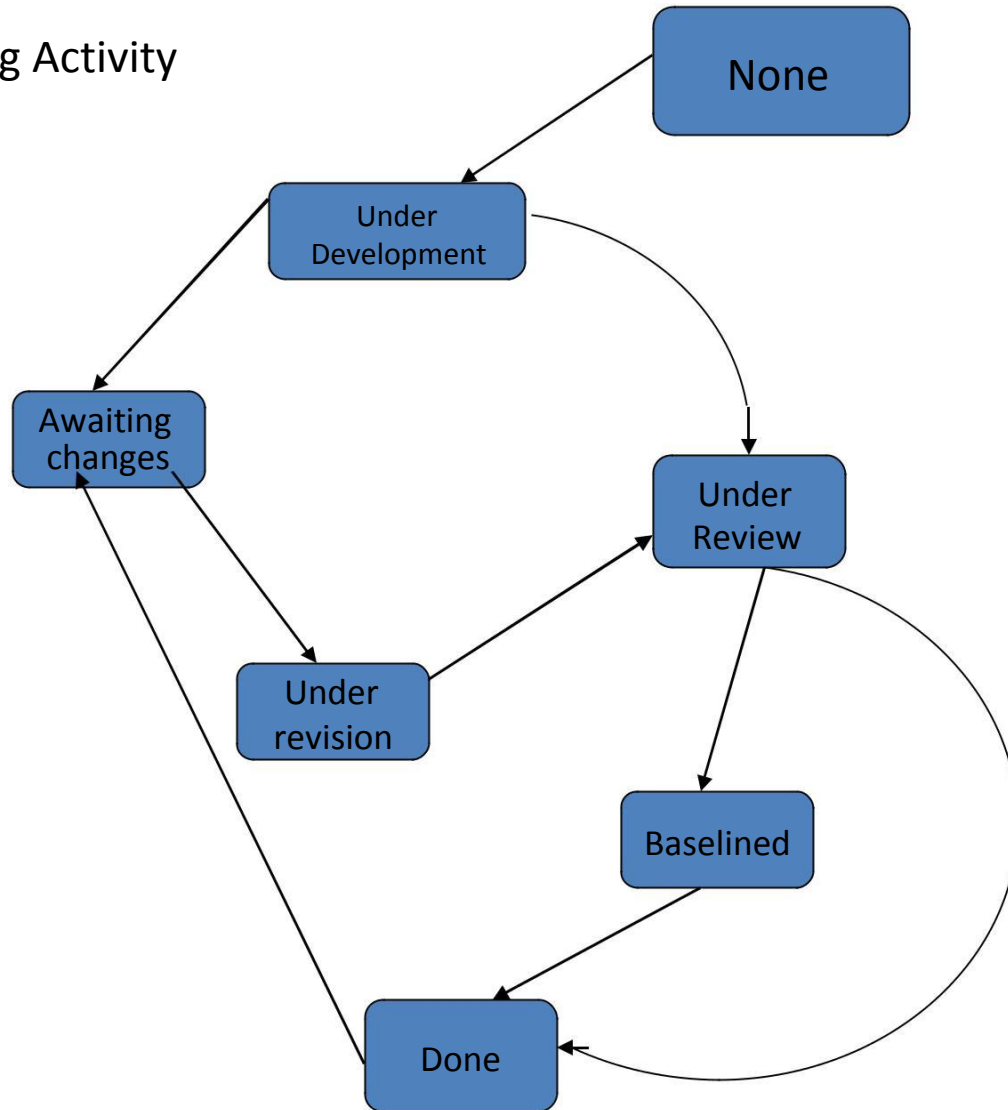
- Constitutes a series of events that will trigger transitions from state to state for each of the software engineering activities framework activities, actions or tasks

It can be represented as a series of framework activities, software engineering actions and tasks and their associated states

- All activities exist concurrently but reside in different states
- Applicable to all types of software development
- Event generated at one point in the process trigger transitions among the states

CONCURRENT DEVELOPMENT MODEL

Modeling Activity



CONCURRENT DEVELOPMENT MODEL

- The *modeling* activity which existed in the '*none*' state while initial communication was completed, now makes a transition into the '*under development*' state.
- If the customer indicates that changes in requirements must be made, the *modeling* activity moves from *under development* state into *awaiting changes* state

SPECIALIZED PROCESS MODELS

- Component-Based Development

Commercial off-the-shelf (COTS) software components, built by vendors can be used when software 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

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

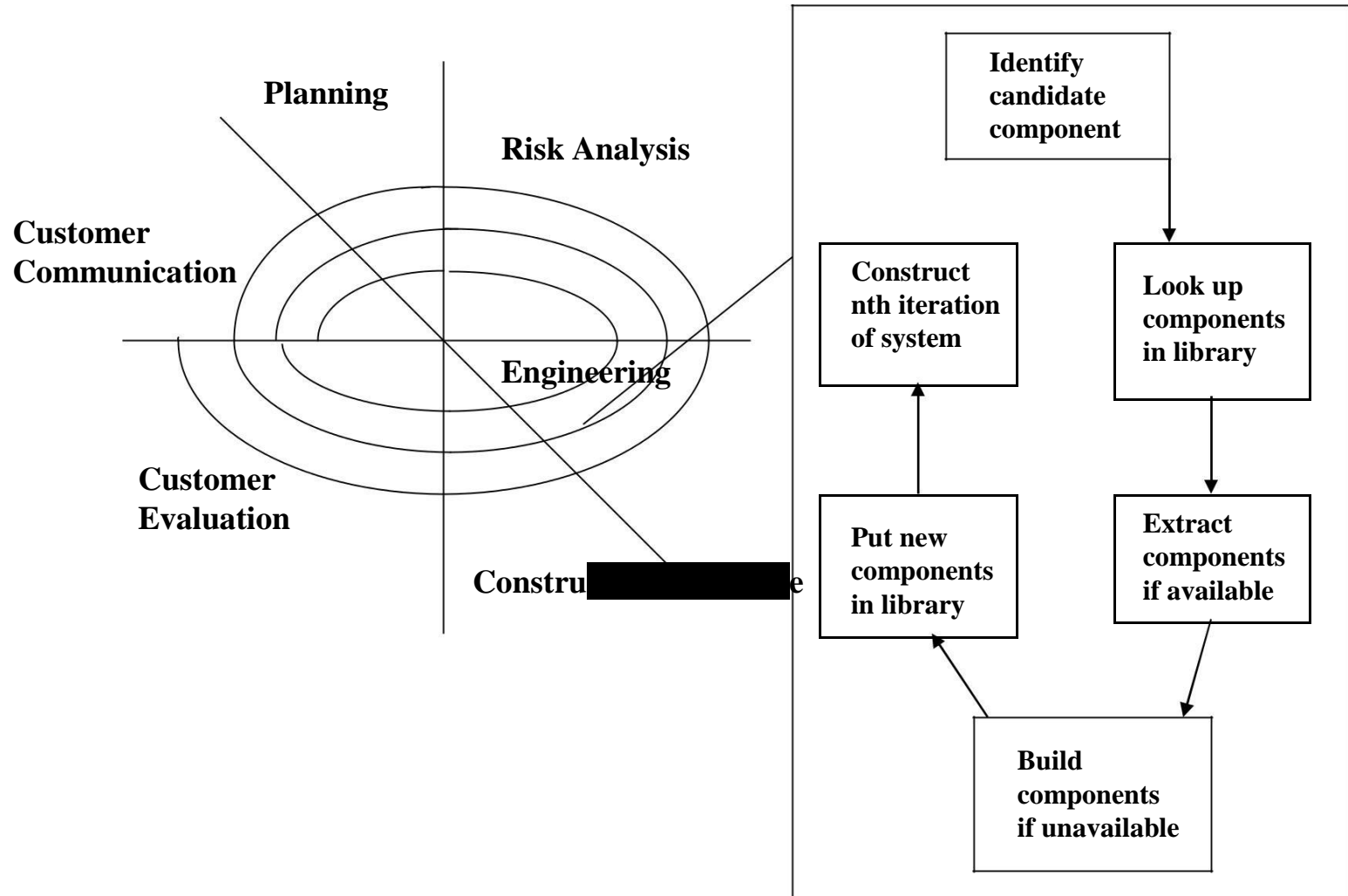
SPECIALIZED PROCESS MODELS

- 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

The Component Assembly Model



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

SOFTWARE REQUIREMENTS

IEEE defines Requirement as :

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

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

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 organization’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**

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.

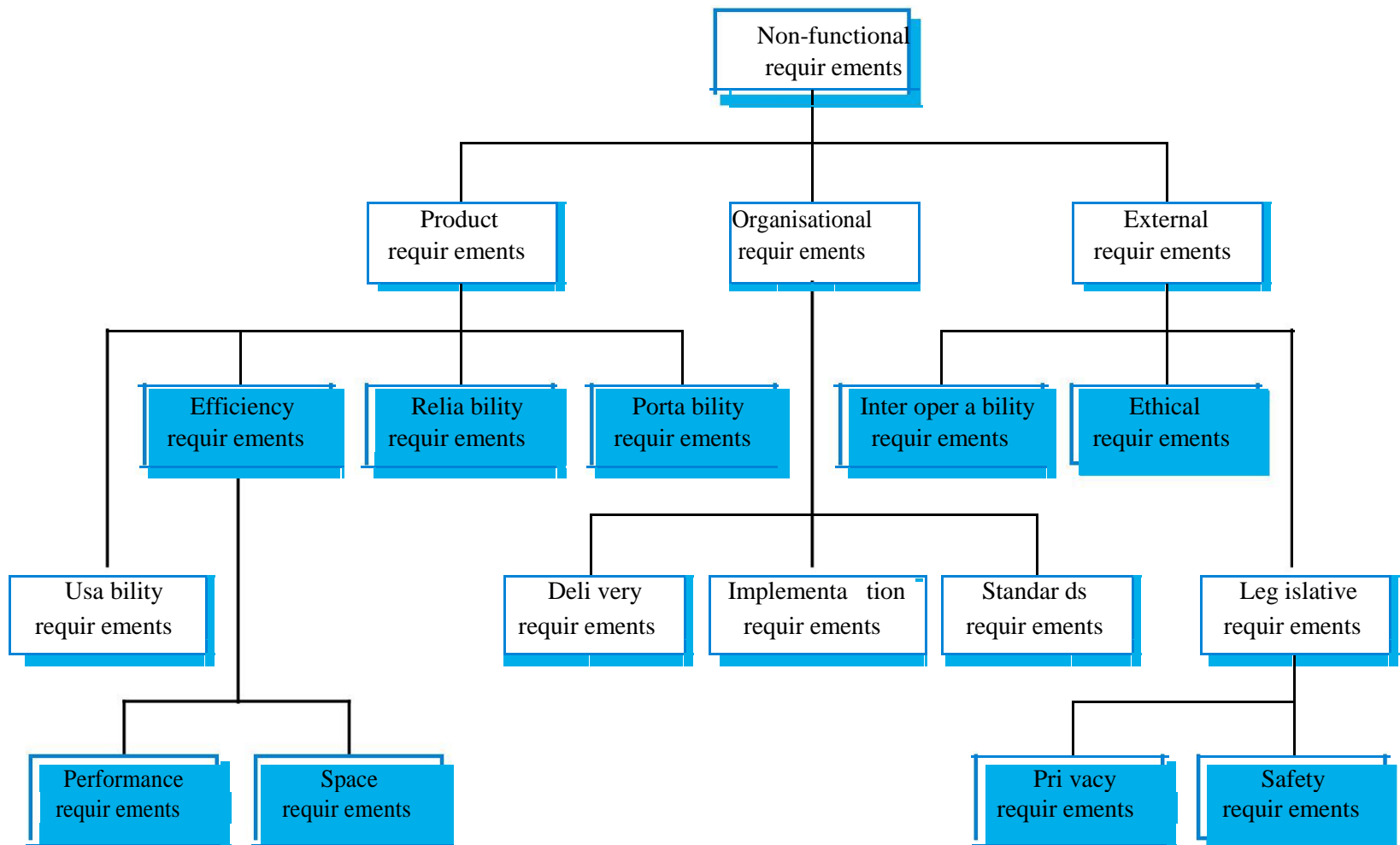
Organisational requirements

- 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.

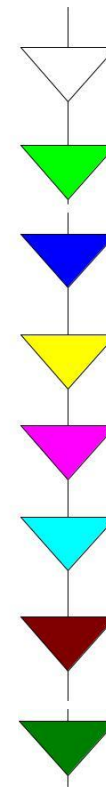
Implicitness

- 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.

Requirements imprecision

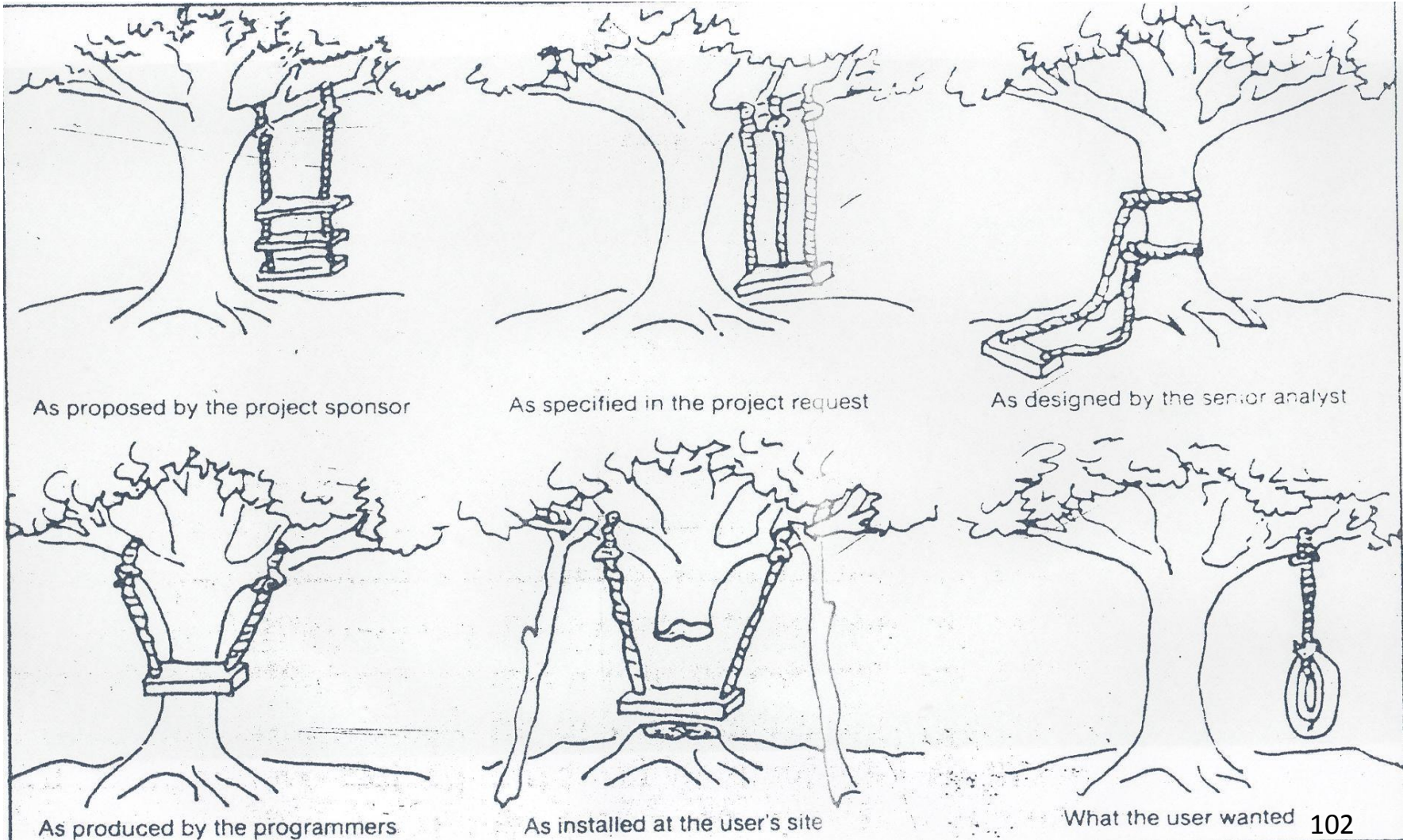
Problems arise when requirements are not precisely stated.

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



As proposed by the project sponsor

As specified in the project request

As designed by the senior analyst

As produced by the programmers

As installed at the user's site

What the user wanted

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

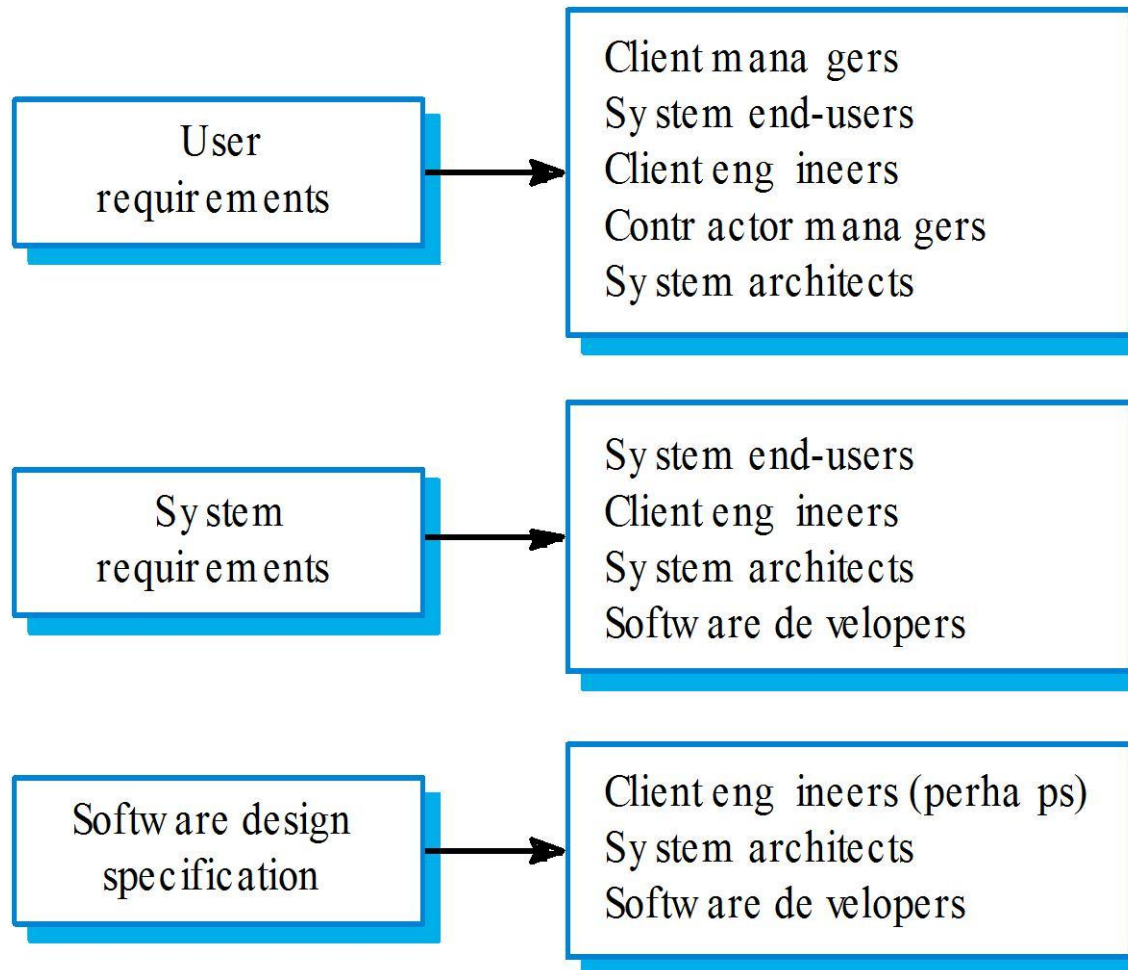
- Procedural interfaces;**
- Data structures that are exchanged;**
- Data representations.**

Formal notations are an effective technique for interface specification.

Example interface description

```
interface PrintServer {  
  
    // def ines 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 \tilde{G} the 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.

Tabular specification

Condition	Action
Sugar level falling ($r_2 < r_1$)	CompDose = 0
Sugar level stable ($r_2 = r_1$)	CompDose = 0
Sugar level increasing and rate of increase decreasing ($(r_2 - r_1) < (r_1 - r_0)$)	CompDose = 0
Sugar level increasing and rate of increase stable or increasing. ($(r_2 - r_1) \geq (r_1 - r_0)$)	CompDose = round $((r_2 - r_1) / 4)$ 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

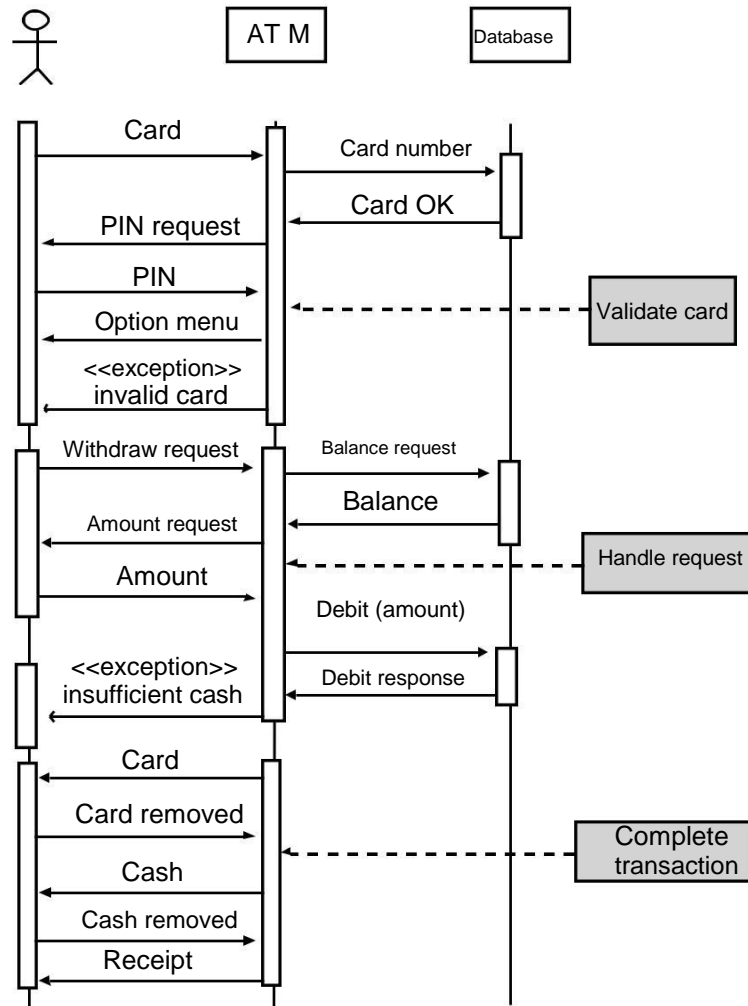
These show the sequence of events that take place during some user interaction with a system.

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



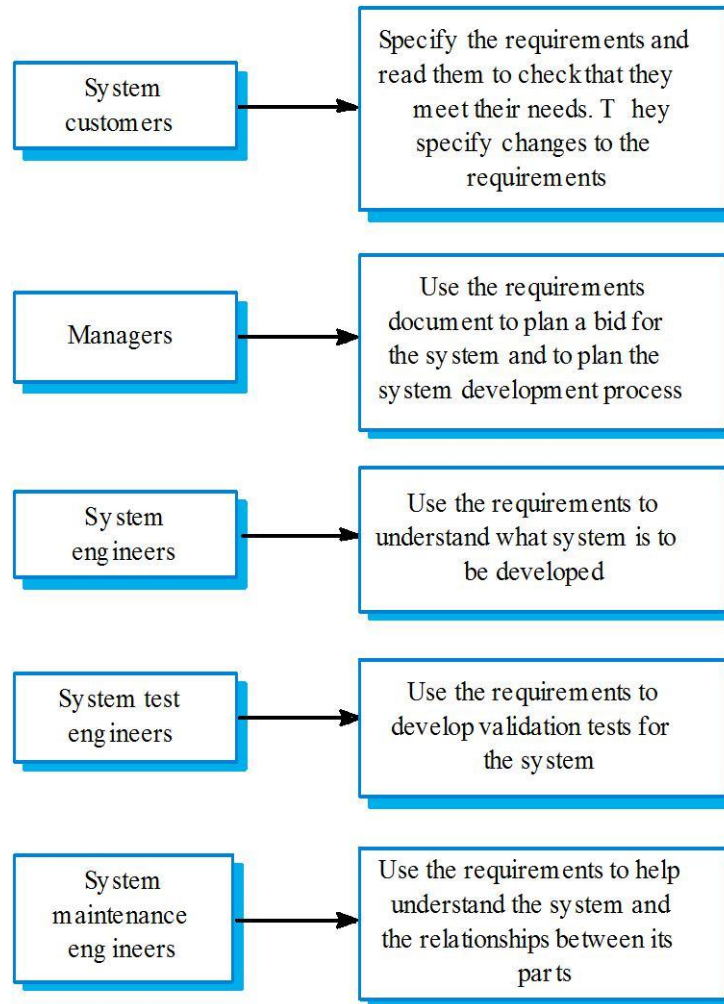
The Software Requirements Specifications (SRS) Document

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

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 are 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

SYLLABUS

Requirements engineering process : Feasibility studies, Requirements elicitation and analysis, Requirements validation, Requirements management.

System models : Context Models, Behavioral models, Data models, Object models, structured methods.

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:

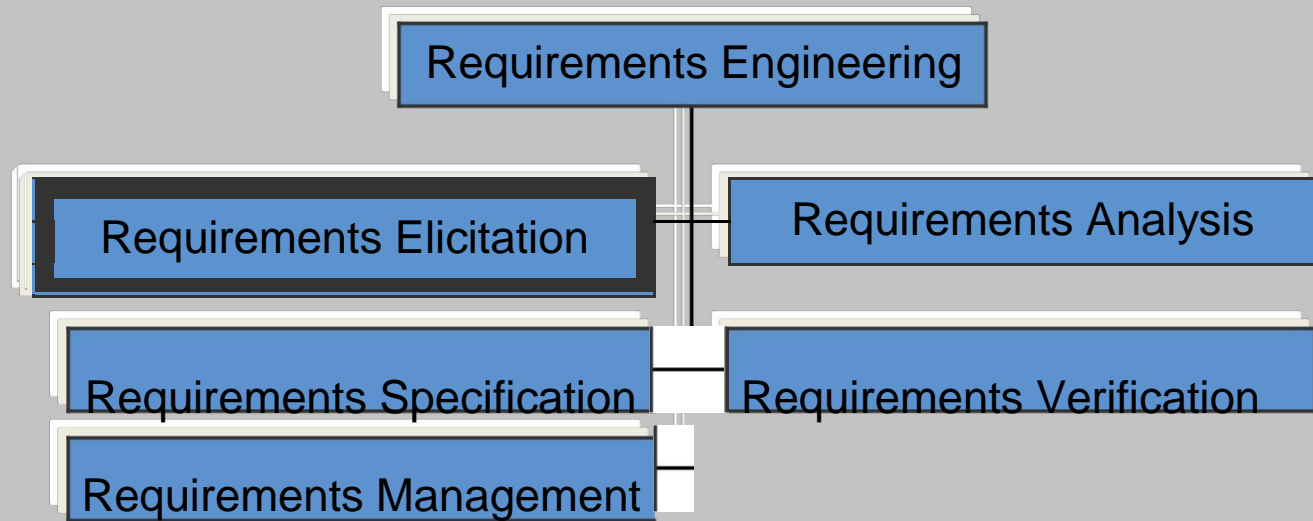
- Requirements elicitation - discovering requirements from system stakeholders
- Requirements analysis and negotiation - checking requirements and resolving stakeholder conflicts

Requirements Engineering Processes

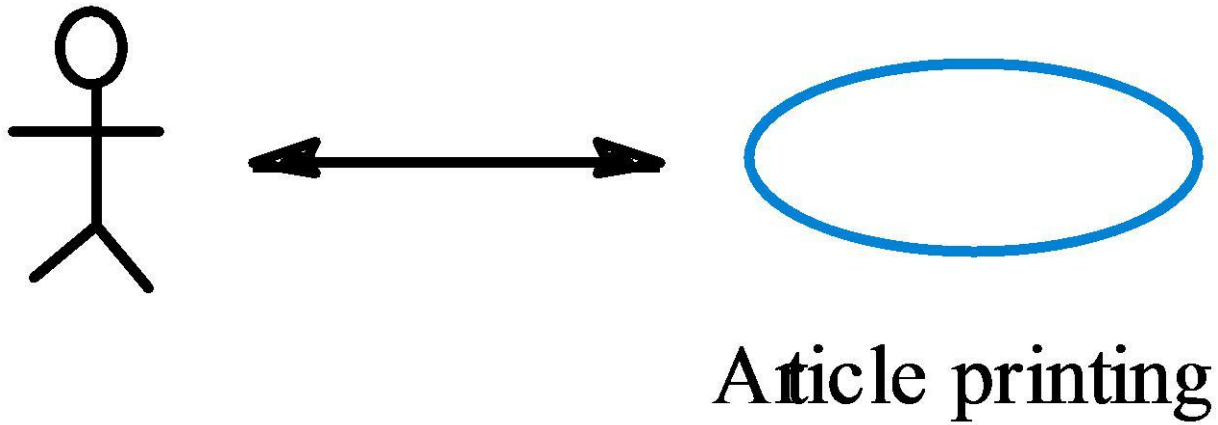
Activities within the RE process may include:

- 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

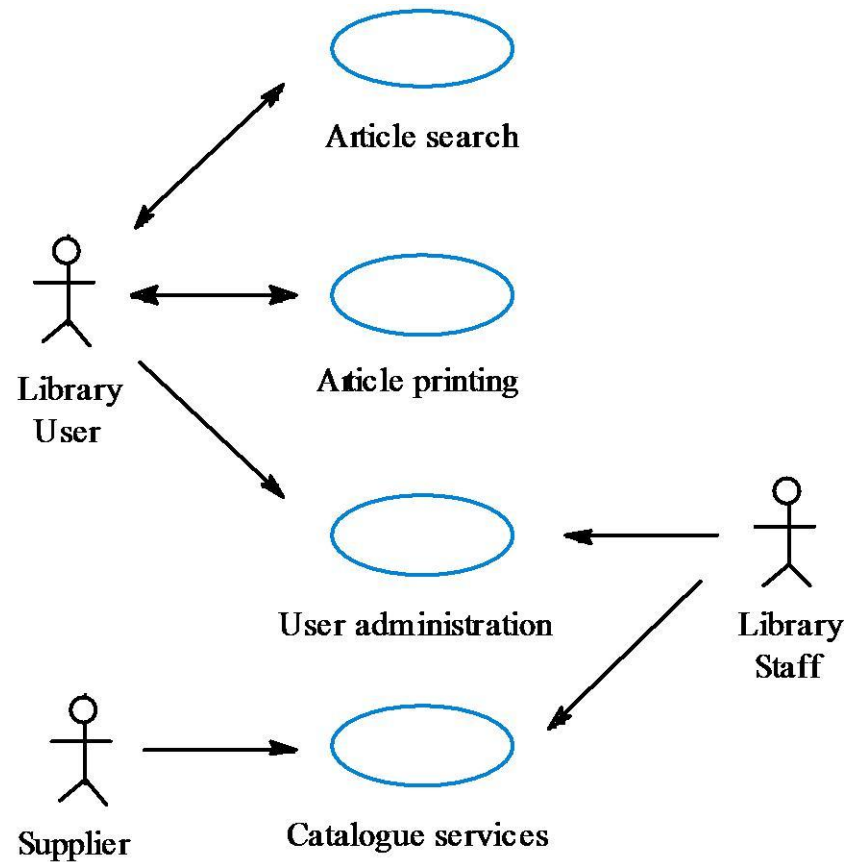
Requirements Engineering Processes



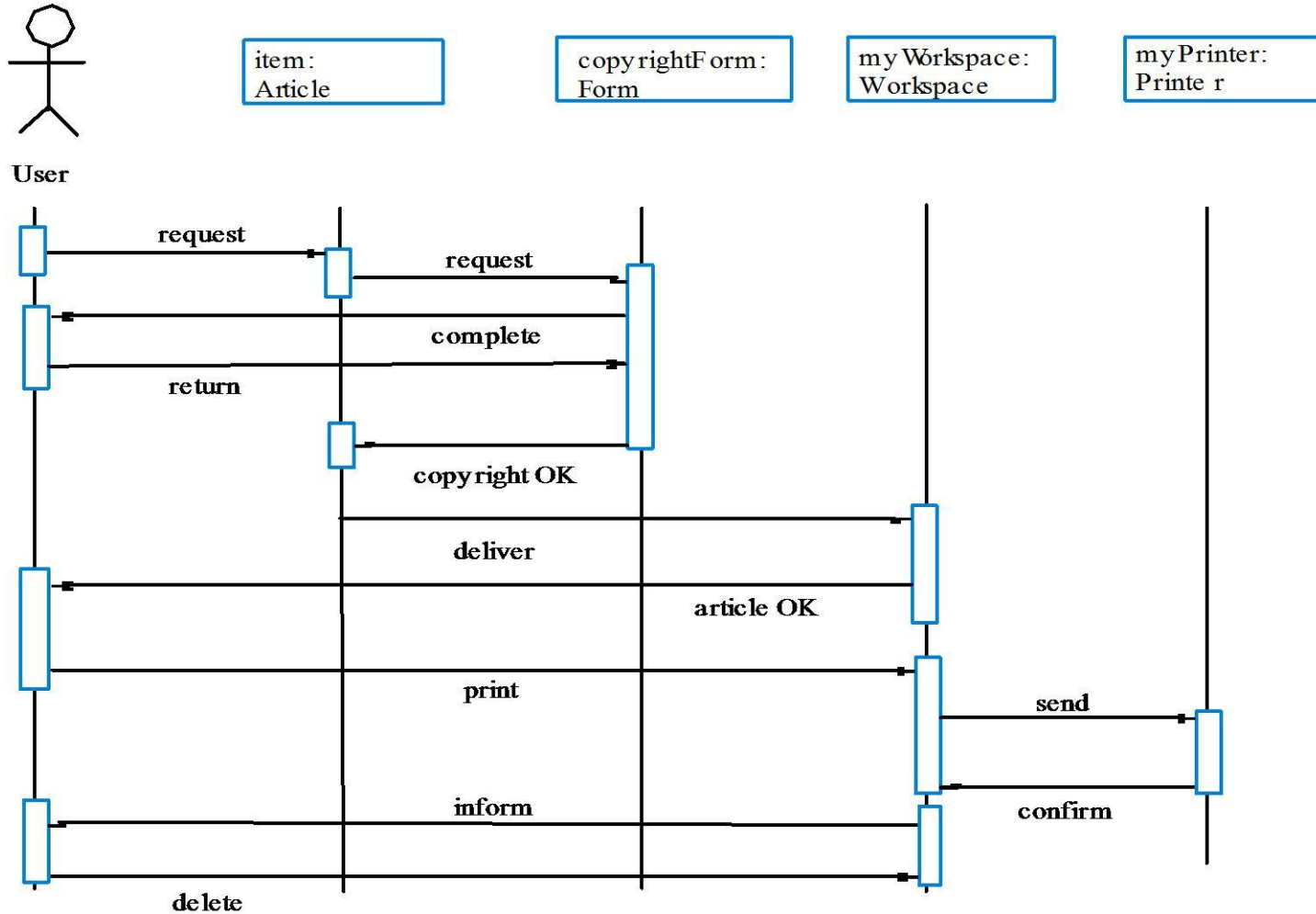
Article printing use-case



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 scientist 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

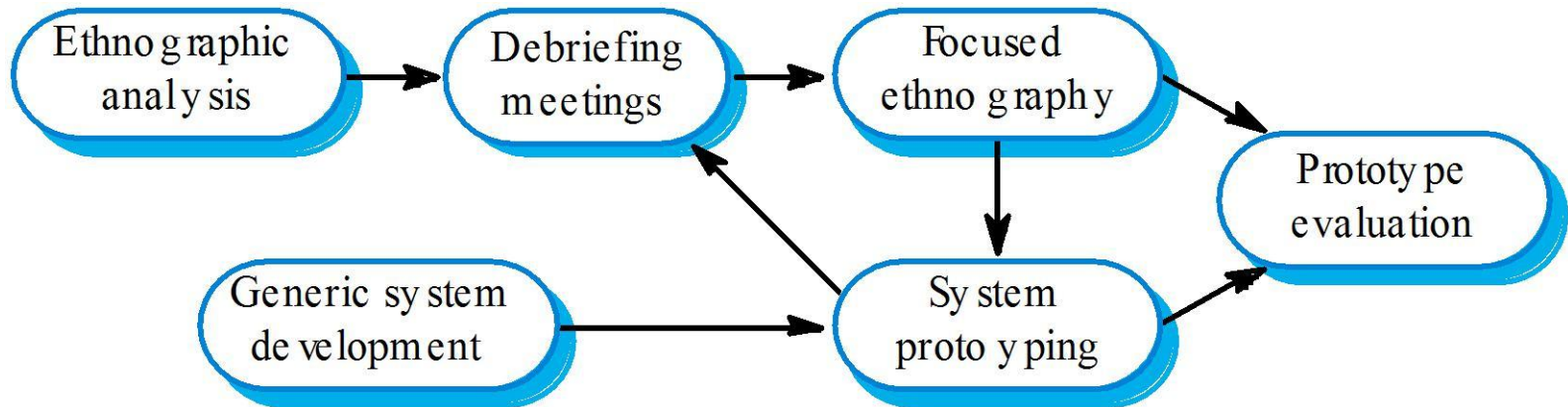
Developed in a project studying the air traffic control process

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 include¹

- 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.**

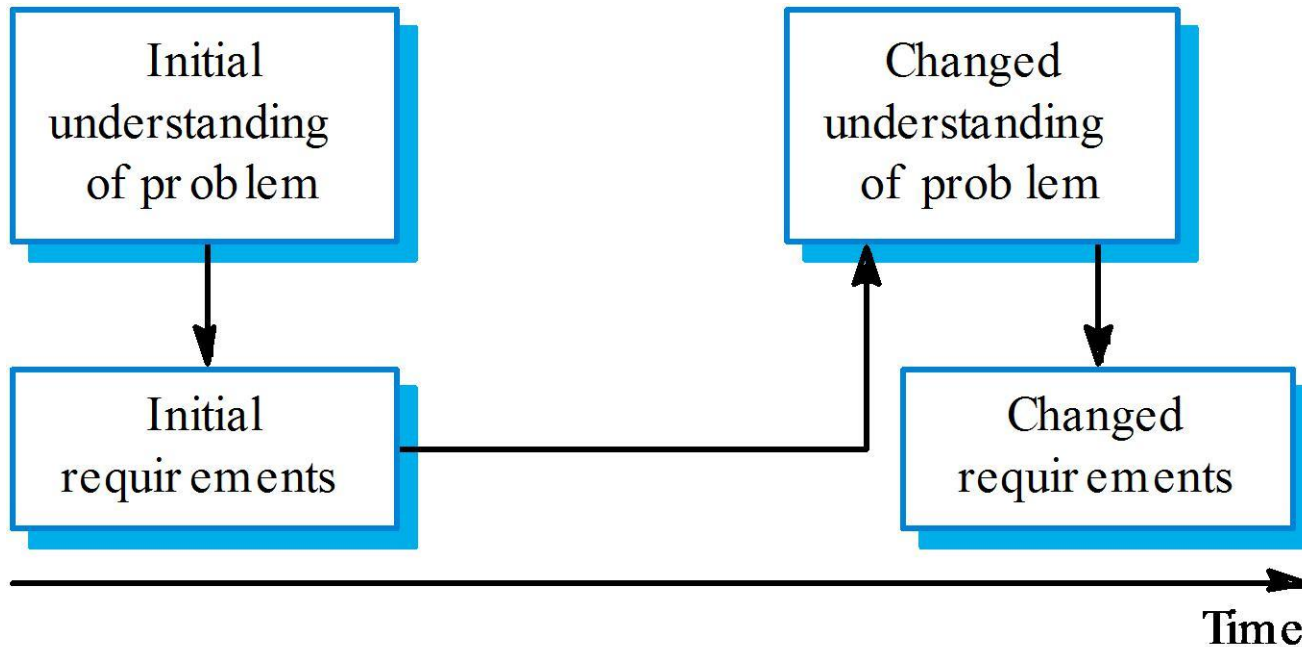
Requirements Change

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;**

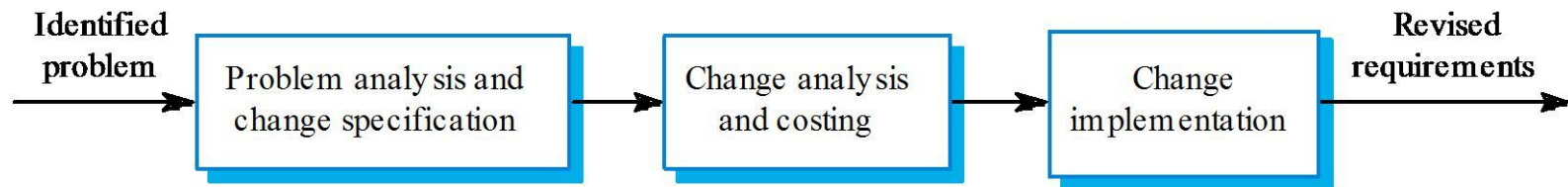
Requirements Change Management

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



System models

System modelling

System modelling helps the analyst to understand the functionality of the system and models are used to communicate with customers.

Different models present the system from different perspectives

- External perspective showing the system's context or environment;**
- Behavioural perspective showing the behaviour of the system;**
- Structural perspective showing the system or data architecture.**

Model types

Data processing model showing how the data is processed at different stages.

Composition model showing how entities are composed of other entities.

Architectural model showing principal sub-systems.

Classification model showing how entities have common characteristics.

Stimulus/response model showing the system's reaction to events.

Context models

System Context Diagrams are diagrams used in systems design to represent the more important external factors that interact with the system at hand.

Context diagrams are used early in a project to get agreement on the scope under investigation.

Context diagrams are typically included in a requirements document.

These diagrams must be read by all project stakeholders and thus should be written in plain language, so the stakeholders can understand items within the document.

Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries.

Context models

Context diagrams can be developed with the use of two types of building blocks:

- *Entities (Actors)*: labeled boxes; one in the center representing the system, and around it multiple boxes for each external actor**
- *Relationships*: labeled lines between the entities and system**

Social and organisational concerns may affect the decision on where to position system boundaries.

Context models

A Context Diagram provides no information about the timing, sequencing, or synchronization of processes such as which processes occur in sequence or in parallel. Therefore it should not be confused with a flowchart or process flow which can show these things.

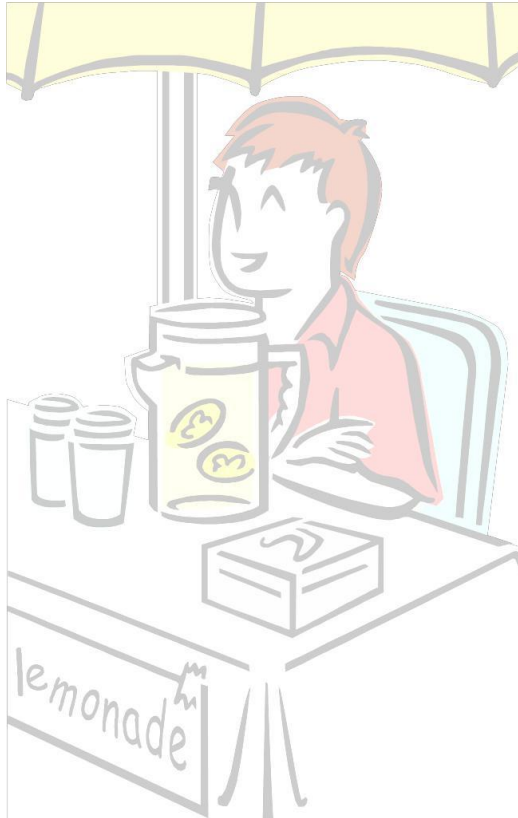
Some of the benefits of a Context Diagram are:

- Shows the scope and boundaries of a system at a glance including the other systems that interface with it**
- No technical knowledge is assumed or required to understand the diagram**
- Easy to draw and amend due to its limited notation**
- Easy to expand by adding different levels of Data Flow Diagrams**
- Can benefit a wide audience including stakeholders, business analyst, data analysts, developers**

Creating Context Diagram

Example

The operations of a simple lemonade stand will be used to for the creation of Context Diagrams



Steps:

Create a list of activities

Construct Context Level Diagram
(identifies sources and sink)

Creating Context Diagrams

Example

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



1. Create a list of activities

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

Creating Context Diagrams

Example

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



1. Create a list of activities

Customer Order
Serve Product
Collect Payment

Produce Product
Store Product

Order Raw Materials
Pay for Raw Materials

Pay for Labor

Creating Context Diagrams

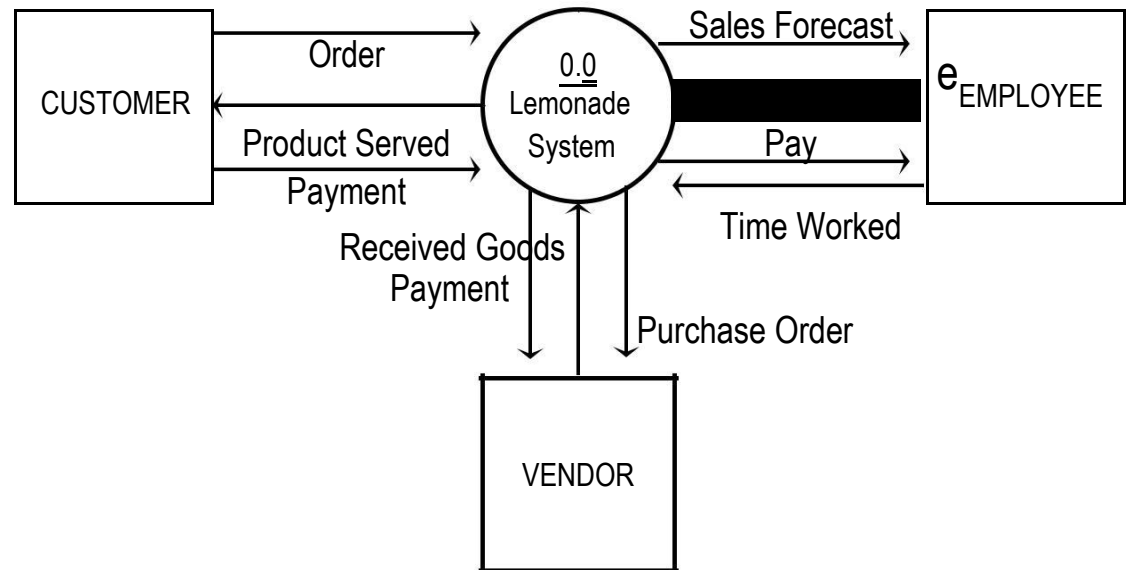
Example

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



Construct Context Level Diagram

Context Level Diagram



Context models

Context Diagram pitfalls to avoid

- **Examine context diagram to be sure that none of the following were inadvertently included:**

Internal actors who initiate data flows or processes (as mentioned above, these have no place in a context diagram)

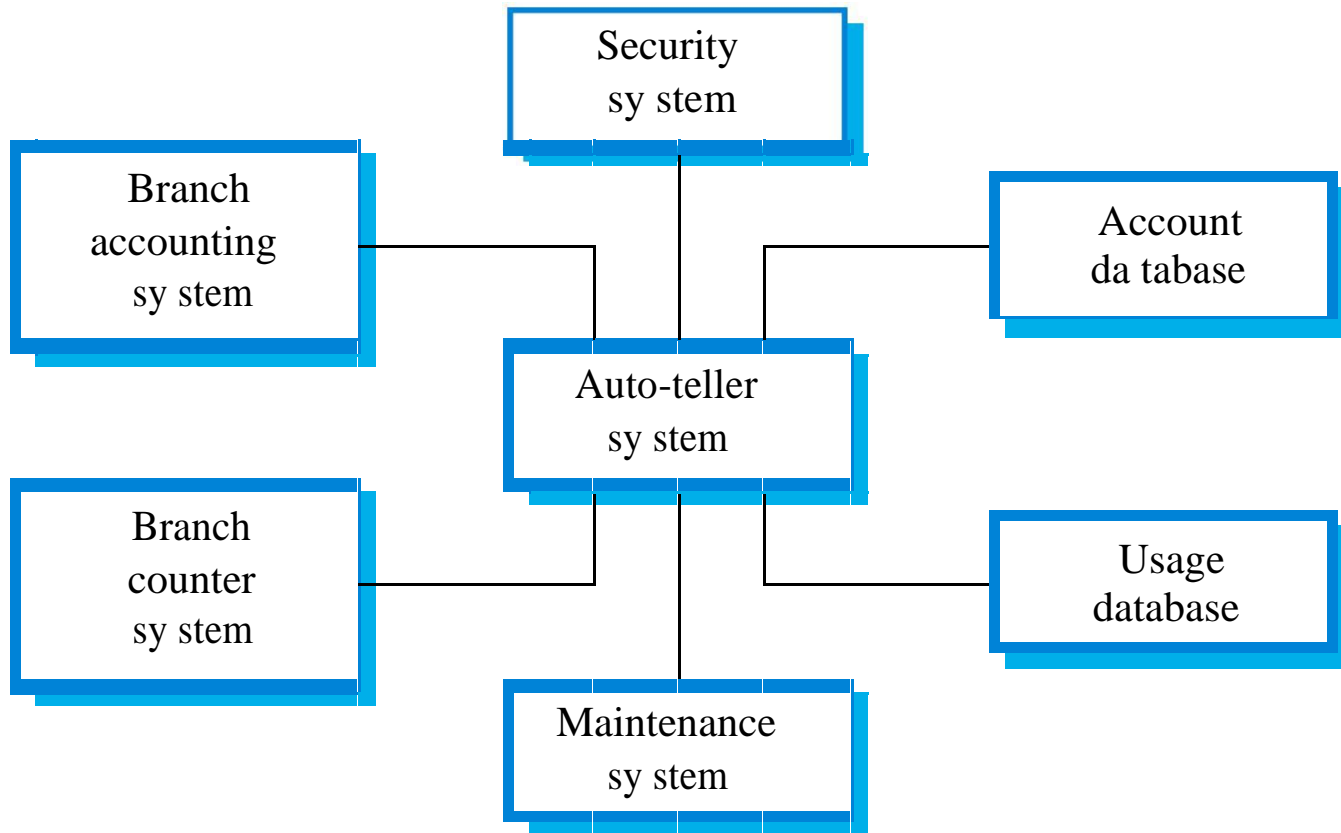
“Black holes,” meaning many inputs into the process are depicted but no outputs.

Or the converse, “miracles,” many outputs come out of the process, but nothing goes in

“Isolated entities,” meaning external entities are shown but not linked

Entity-to-entity data flows with no process in between

The context of an ATM system



Context models

Some of the benefits of a Context Diagram are:

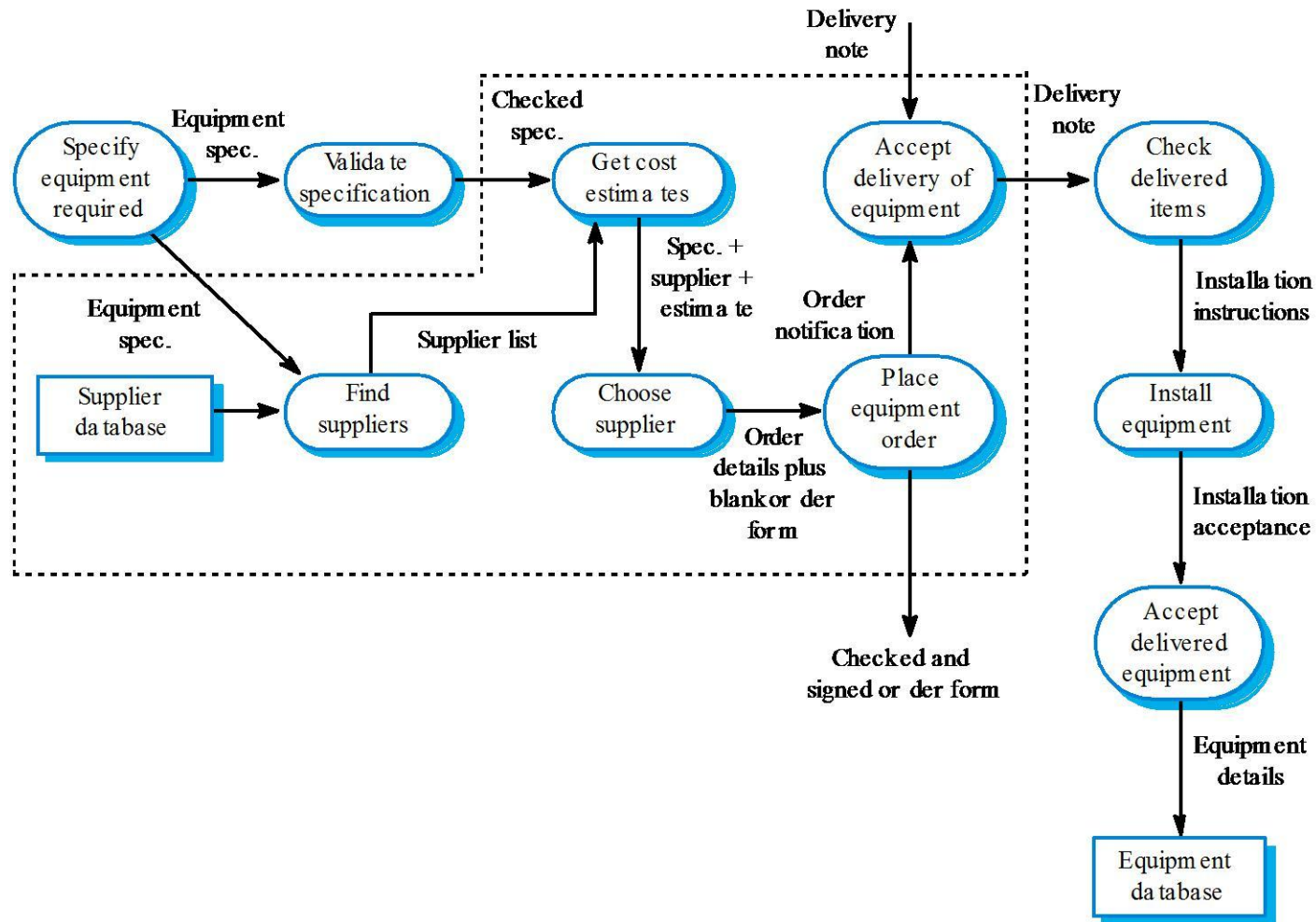
- Shows the scope and boundaries of a system at a glance including the other systems that interface with it**
- No technical knowledge is assumed or required to understand the diagram**
- Easy to draw and amend due to its limited notation**
- Easy to expand by adding different levels of Data Flow Diagrams**
- Can benefit a wide audience including stakeholders, business analyst, data analysts, developers**

Process models

Context (Architectural Models) can be further enhanced by Process models which show the overall process and the processes that are supported by the system.

Data flow models may be used to show the processes and the flow of information from one process to another.

Equipment procurement process



Behavioural models

Behavioural models are used to describe the overall behaviour of a system.

Two types of behavioural model are:

- Data processing models that show how data is processed as it moves through the system;**
- State machine models that show the systems response to events.**

These models show different perspectives so both of them are required to describe the system's behaviour.

Behavioural models

Most business systems are primarily driven by data. They are controlled by the data inputs to the system with relatively little external event processing. A Data Flow model represents the behaviour of these systems

Real-time systems are often event-driven with minimal data processing. A state machine model is the most effective way to represent their behaviour

Data processing models

Data flow diagrams (DFDs) may be used to model the system's data processing.

These show the processing steps as data flows through a system.

DFDs are an intrinsic part of many analysis methods.

Simple and intuitive notation that customers can understand.

Show end-to-end processing of data.

Data flow diagrams

A data flow diagram (DFD) is a graphical representation of the "flow" of data through an [information system](#), modeling its *process* aspects.

DFDs model the system from a functional perspective.

Tracking and documenting how the data associated with a process is helpful to develop an overall understanding of the system.

Data flow diagrams may also be used in showing the data exchange between a system and other systems in its environment.

The dataflow diagram is one of the most commonly used systems-modeling tools, particularly for operational systems in which the *functions* of the system are of paramount importance and more complex than the data that the system manipulates.

Data flow diagrams

Data Flow models are valuable because tracking and documenting how the data associated with a particular process moves through the system helps analysts understand what is going on

DFDs have the advantage that, unlike some other modelling notations, they are simple and intuitive

It is usually possible to explain them to potential system users who can then participate in validating the analysis

Data flow models should be “top-down” process.

Data flow diagrams

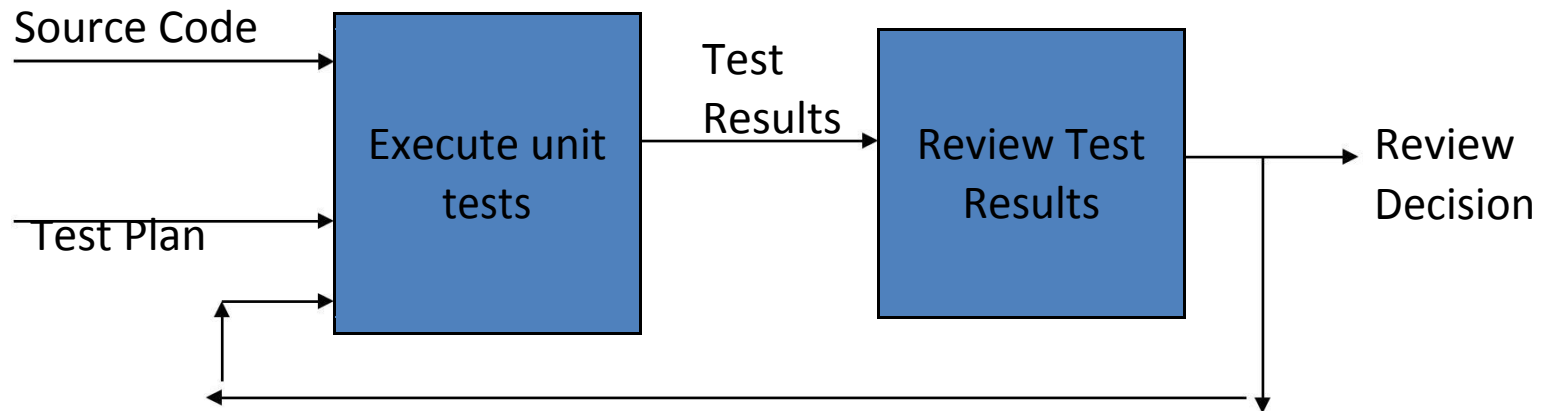
A Data Flow Diagram shows the flow of the data among a set of components. The components may be tasks, software components or even the abstraction of the functionality that will be included in the software system

Rules and interpretations for correct data flow diagrams

- Boxes are processes and must be verb phrases**
- Arcs represent data and must be labeled with noun phrases**
- Control is not shown. Some sequencing may be inferred from the ordering**
- A process may be a one-time activity, or it may imply a continuous processing**
- Two arcs coming out of a box may indicate that both outputs are produced or that one or the other is produced**

Data flow diagrams

E.g Data Flow for unit testing



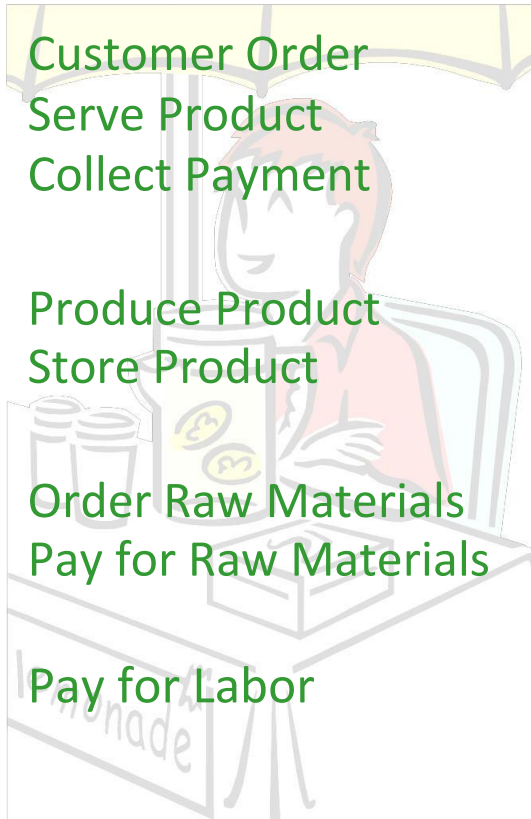
The phrases within the boxes are verb phrases. They represent actions. Each arrow/line is labeled with a noun phrase that represent some artifact.

The data flow diagram does not show decisions explicitly

Creating Data Flow Diagrams

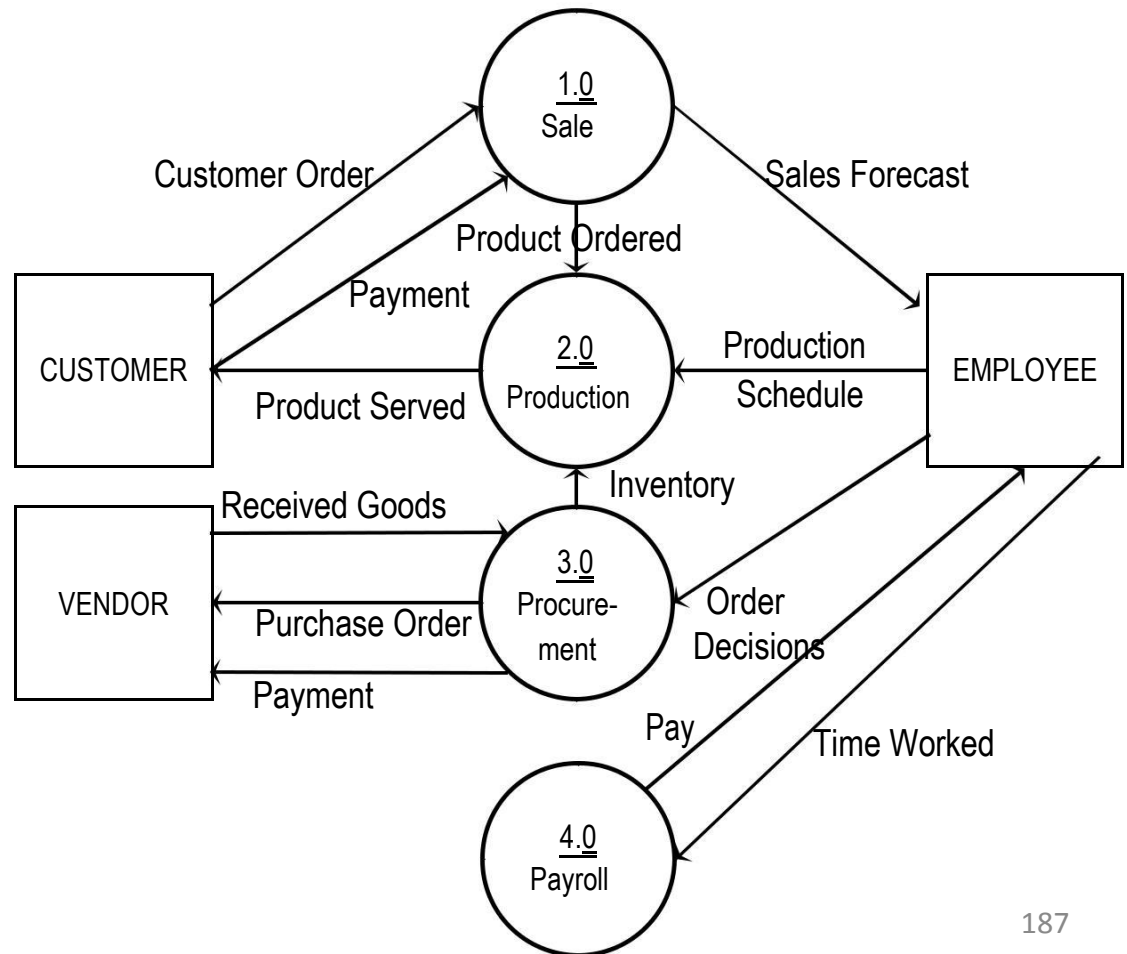
Example

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



Construct Level 0 DFD
(identifies manageable sub processes)

Level 0 DFD



Creating Data Flow Diagrams

Example

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

Customer Order
Serve Product
Collect Payment

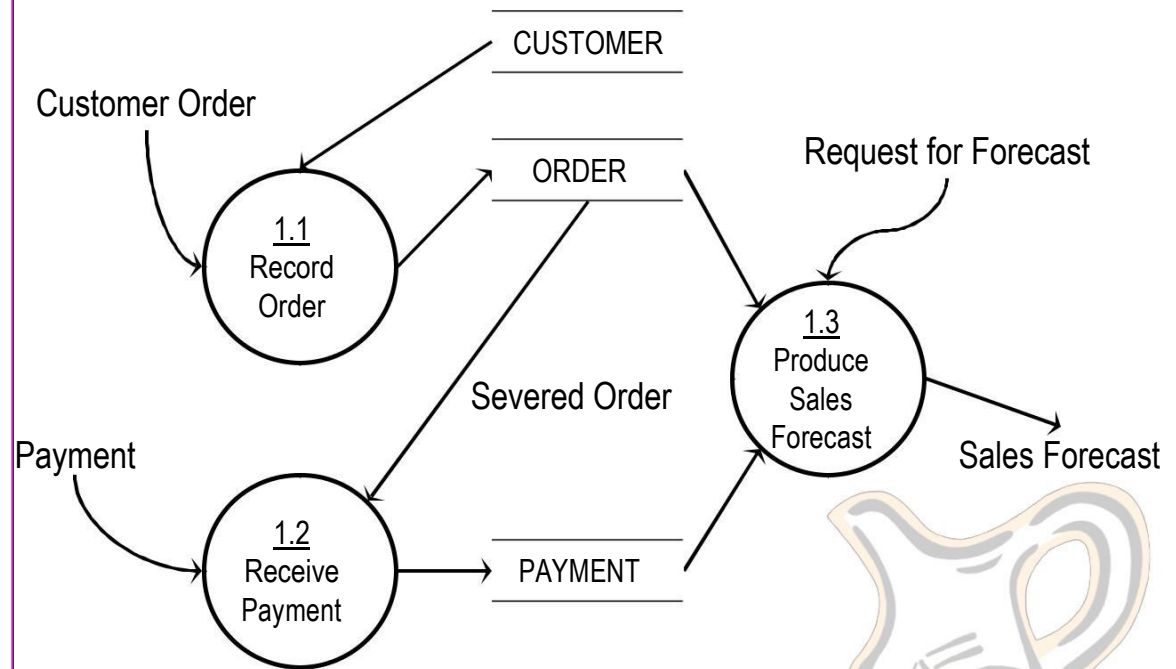
Produce Product
Store Product

Order Raw Materials
Pay for Raw Materials

Pay for Labor

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

Level 1 DFD



Creating Data Flow Diagrams

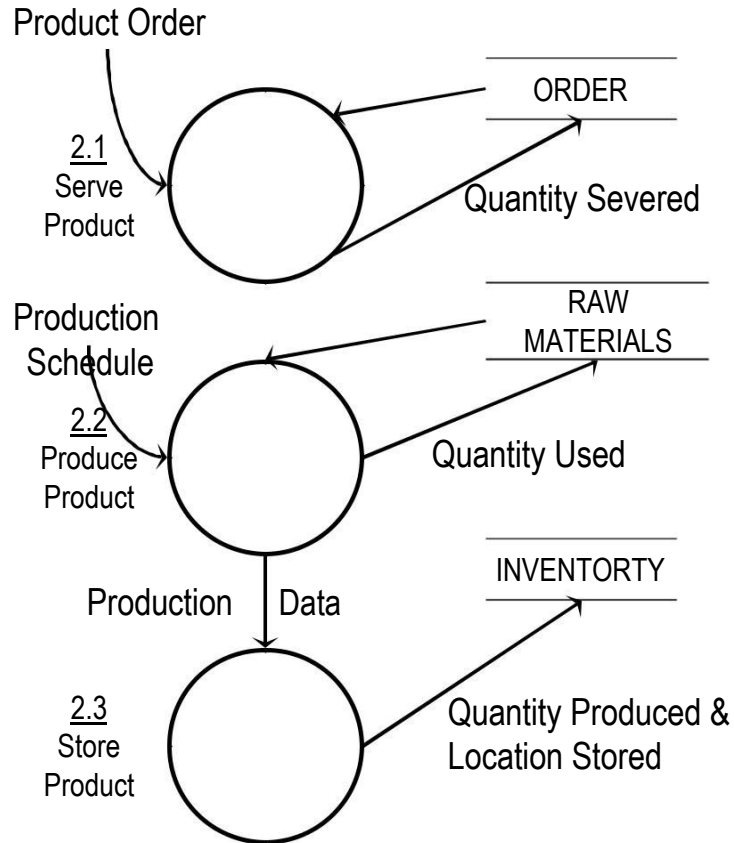
Example

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



Construct Level 1 (continued)

Level 1 DFD



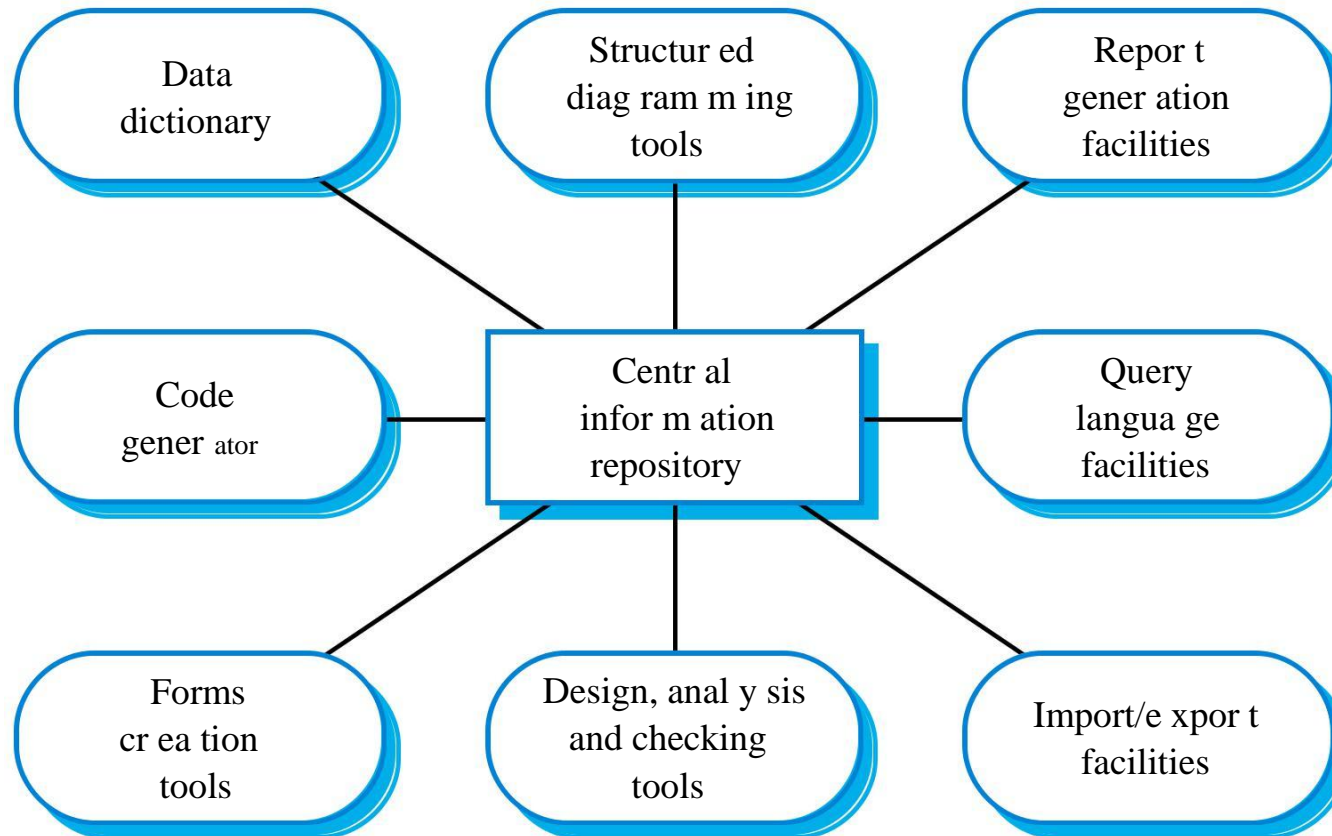
CASE workbenches

A coherent set of tools that is designed to support related software process activities such as analysis, design or testing.

Analysis and design workbenches support system modelling during both requirements engineering and system design.

These workbenches may support a specific design method or may provide support for a creating several different types of system model.

An analysis and design workbench



Analysis workbench components

Diagram editors

Model analysis and checking tools

Repository and associated query language

Data dictionary

Report definition and generation tools

Forms definition tools

Import/export translators

Code generation tools

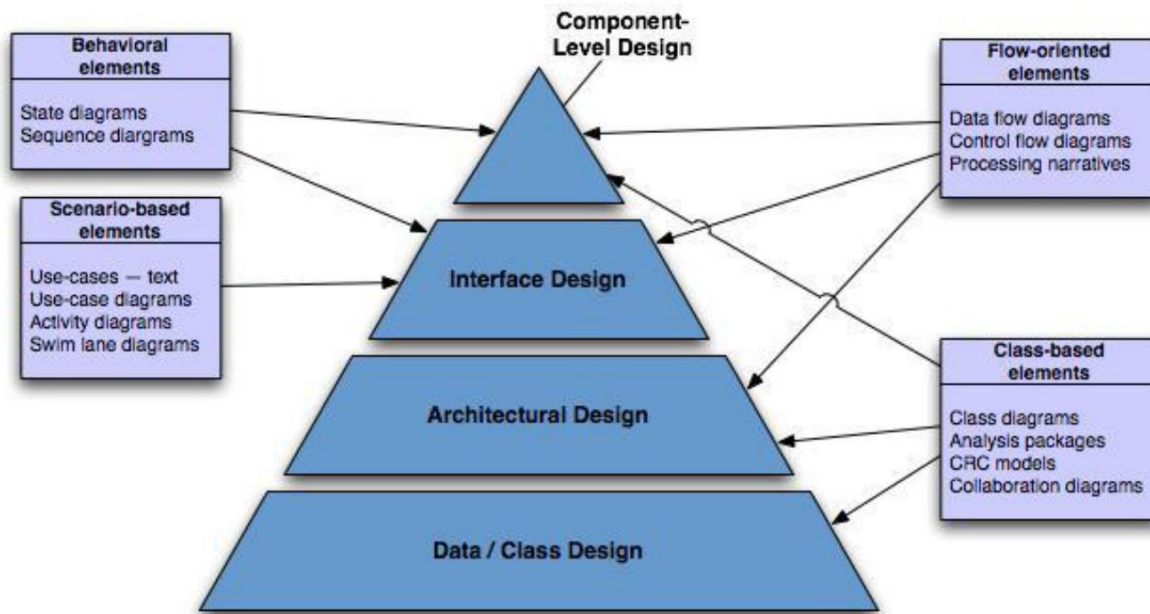
Design Engineering

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

Design Process and Design Quality

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

Design Process and Design Quality

Design Principles

The design process should not suffer from tunnel vision – A good designer should consider alternative approaches. Judging each based on the requirements of the problem, the resources available to do the job and any other constraints

The design should be traceable to the analysis model – because a single element of the design model often traces to multiple requirements, it is necessary to have a means of tracking how the requirements have been satisfied by the model

Design Process and Design Quality

Design Principles

The design should not reinvent the wheel – Systems are constructed using a set of design patterns, many of which may have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited! Design time should be invested in representing truly new ideas and integrating those patterns that already exist.

The design should minimise intellectual distance between the software and the problem as it exists in the real world – That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.

Design Process and Design Quality

Design Principles

The design should exhibit uniformity and integration – a design is uniform if it appears that one person developed the whole thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.

The design should be structured to degrade gently, even with bad data, events, or operating conditions are encountered – Well-designed software should never “bomb”. It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.

Design Process and Design Quality

Design Principles

The design should be reviewed to minimize conceptual (semantic) errors – there is sometimes the tendency to focus on minute details when the design is reviewed, missing the forest for the trees. The designer team should ensure that major conceptual elements of the design have been addressed before worrying about the syntax of the design model.

Design is not coding, coding is not design – Even when detailed designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made at the coding level address the small implementation details that enable the procedural design to be coded.

Design Process and Design Quality

Design Principles

The design should be structured to accommodate change

The design should be assessed for quality as it is being created

A design should be represented using a notation that effectively communicates its meaning

Design Process and Design Quality

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 from failure, and the predictability of the program**

Design Process and Design Quality

Quality attributes

- 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

Design Concepts

Fundamental software design concepts

- **Abstraction**
- **Architecture**
- **Patterns**
- **Modularity**
- **Information hiding**
- **Functional independence**
- **Refinement**
- **Refactoring**
- **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

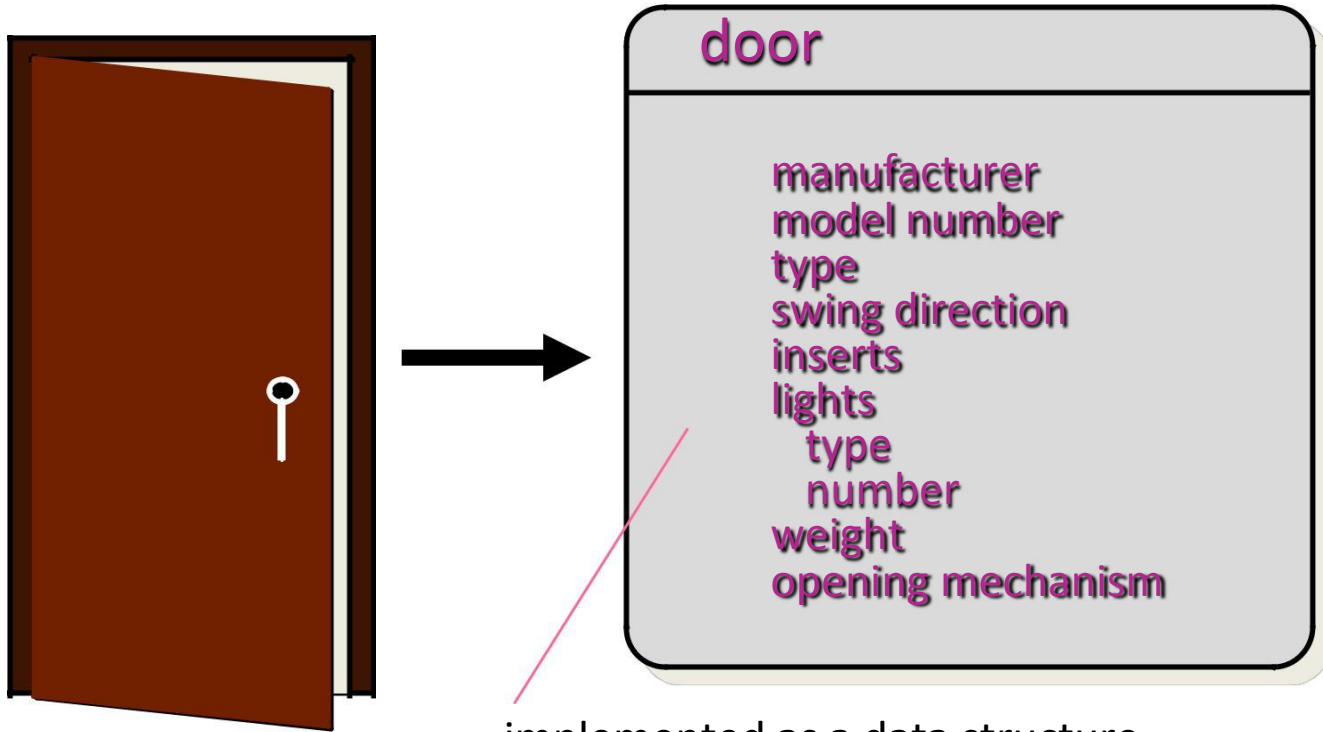
Abstraction

- 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



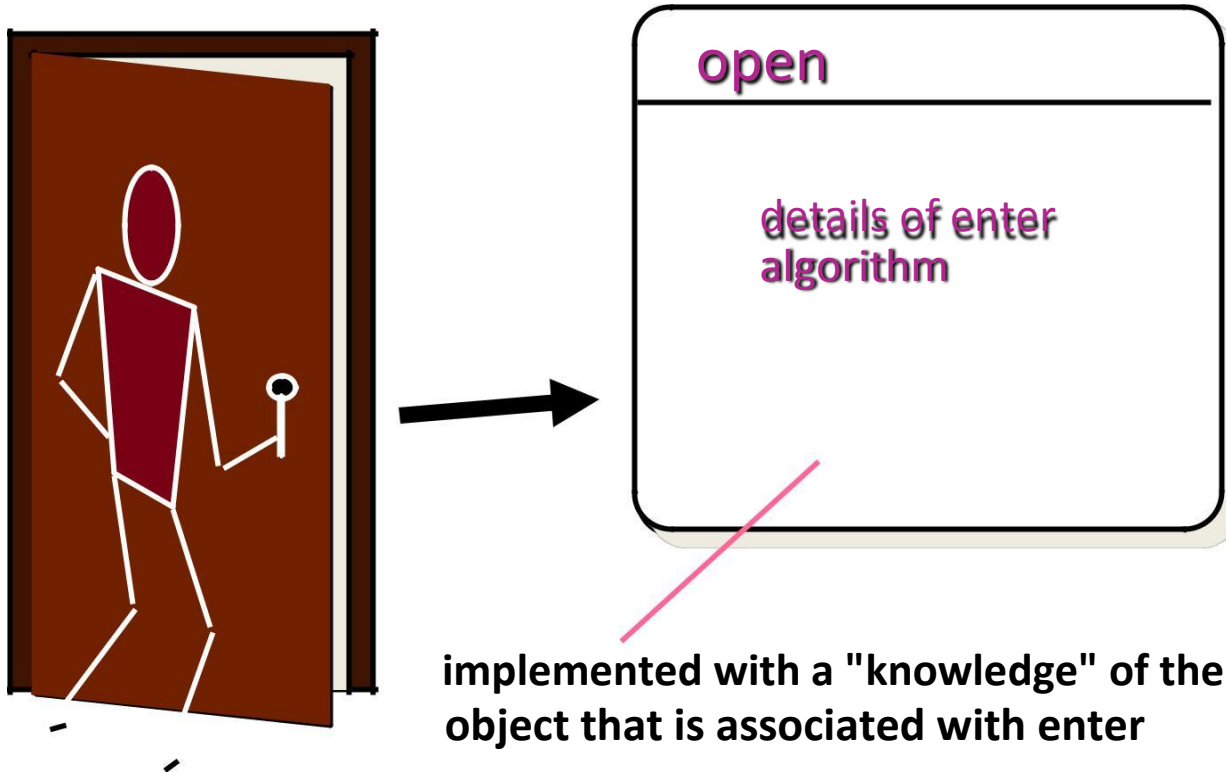
Design Concepts

Abstraction

Procedural abstraction refers to a sequence of instructions that have a 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

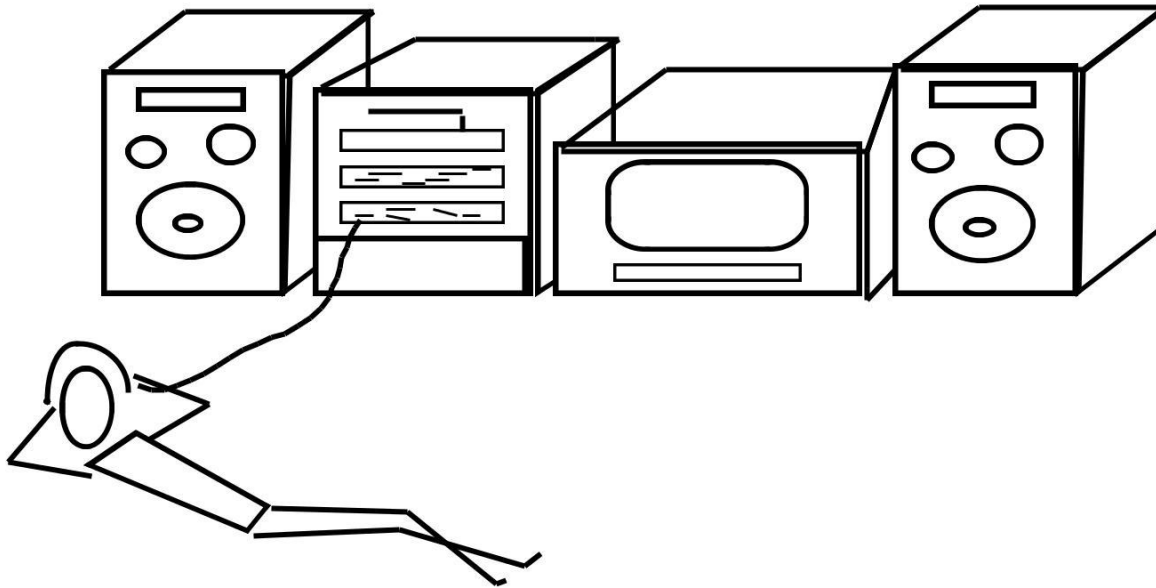
Design Concepts

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**

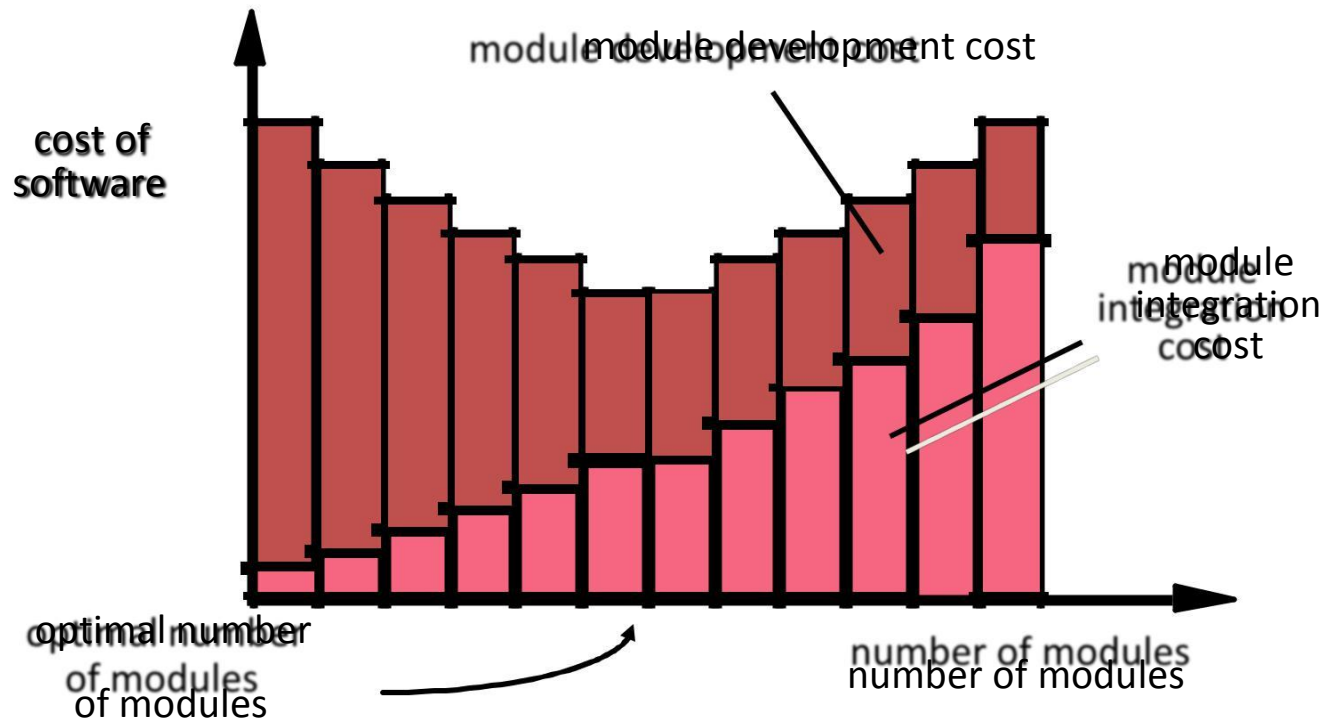
Modular Design

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**

Design Model

Architectural design elements

- The architectural model is derived from

Information about the application domain for the software to be built

Specific analysis 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

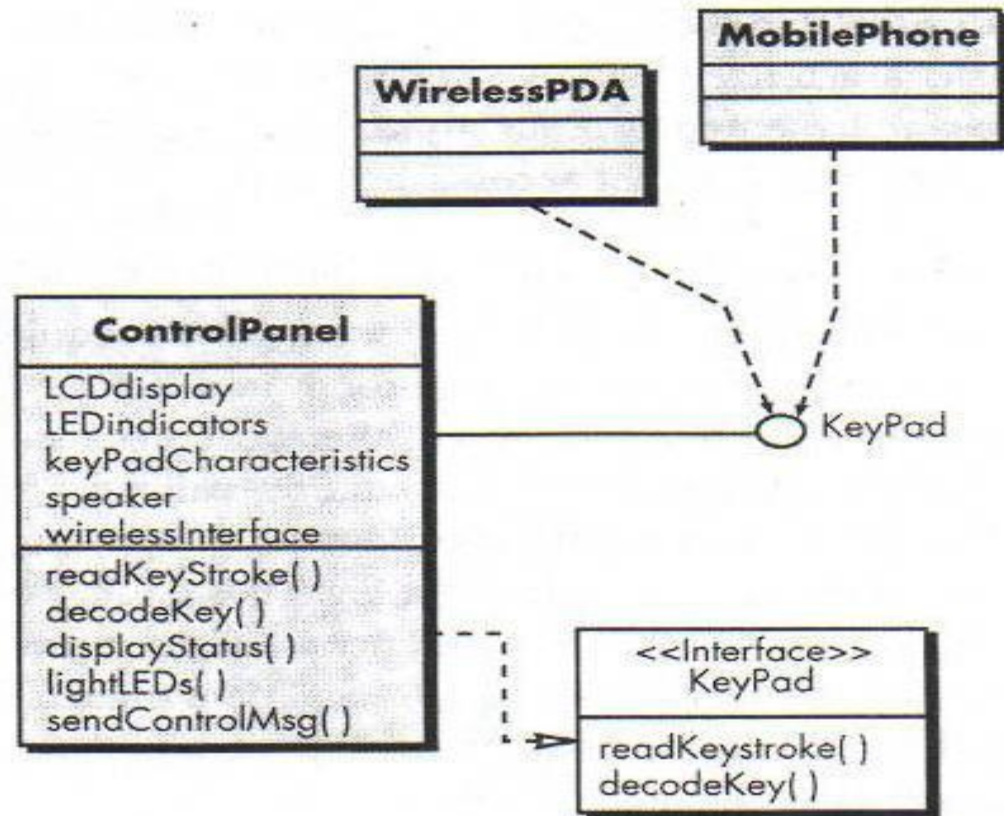
- Important elements of interface design

The user interface

External interfaces to other systems, devices, networks or other producers or consumers of information

Internal interfaces between various design components

Design Model - Interface Elements



Design Model

Component-level design elements

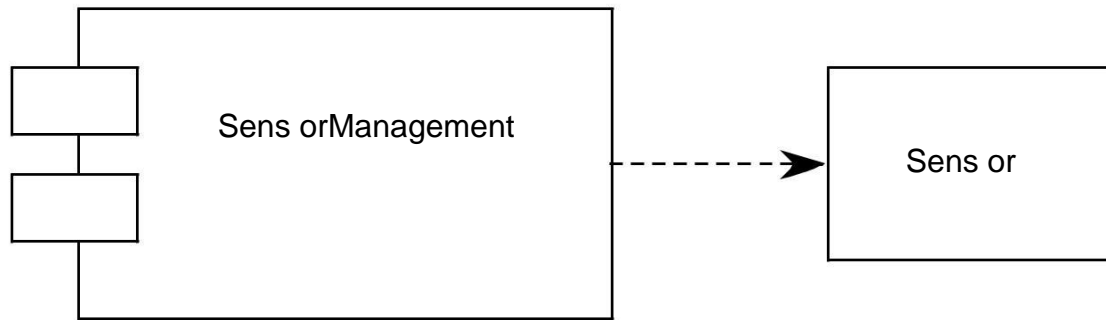
Component-level design for software fully describes the internal detail of each software component.

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 activity diagram can be used to represent processing logic. Detailed procedural flow for a component can be represented using either pseudocode or diagrammatic form

Component Elements

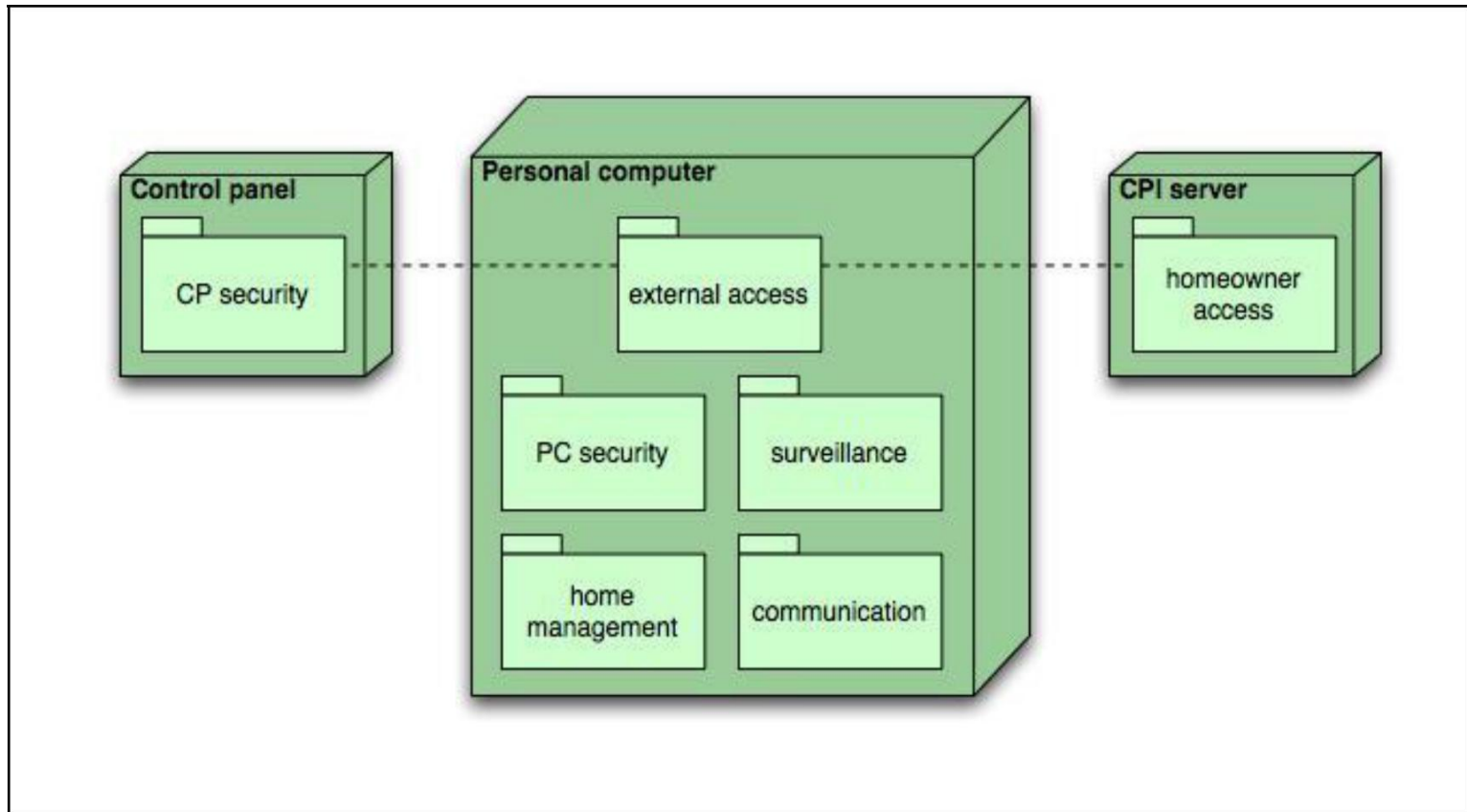


Design Model

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



226

Pattern-based software design

Pattern-based design is a technique that reuses design elements that have proven successful in the past.

Throughout the design process, designer should look for every opportunity to reuse existing design patterns rather than creating new ones

A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution.

The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate.

Pattern-based software design

Design Pattern Template

- **Pattern name**—describes the essence of the pattern in a short but expressive name
- **Intent**—describes the pattern and what it does
- **Also known as**—lists any synonyms for the pattern
- **Motivation**—provides an example of the problem
- **Applicability**—notes specific design situations in which the pattern is applicable
- **Structure**—describes the classes that are required to implement the pattern

Pattern-based software design

Design Pattern Template

- **Participants**—describes the responsibilities of the classes that are required to implement the pattern
- **Collaborations**—describes how the participants collaborate to carry out their responsibilities
- **Consequences**—describes the “design forces” that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented
- **Related patterns**—cross-references related design patterns

Pattern-based software design

Design patterns can be used throughout software design.

Various patterns that can be examined are

- Architectural patterns – these patterns define the overall structure of the software, indicate the relationships among subsystems and software components, and define the rules for specifying relationships among the elements of the architecture**
- Design patterns – these patterns address a specific element of the design such as an aggregate of components to solve some design problem, relationships among components, or the mechanisms for effecting component-to-component communication**

Pattern-based software design

- **Idioms – called coding patterns, these language-specific patterns generally implement an algorithmic element of a component, a specific interface protocol, or a mechanism for communication among components**

Frameworks

- **Frameworks are implementation-specific infrastructure for design work. The designer may select a ‘reusable mini-architecture’ that provides the generic structure and behaviour for a family of software abstractions along with a context, which specifies their collaboration and use within a given domain**

Creating an architectural design

Software Architecture

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.

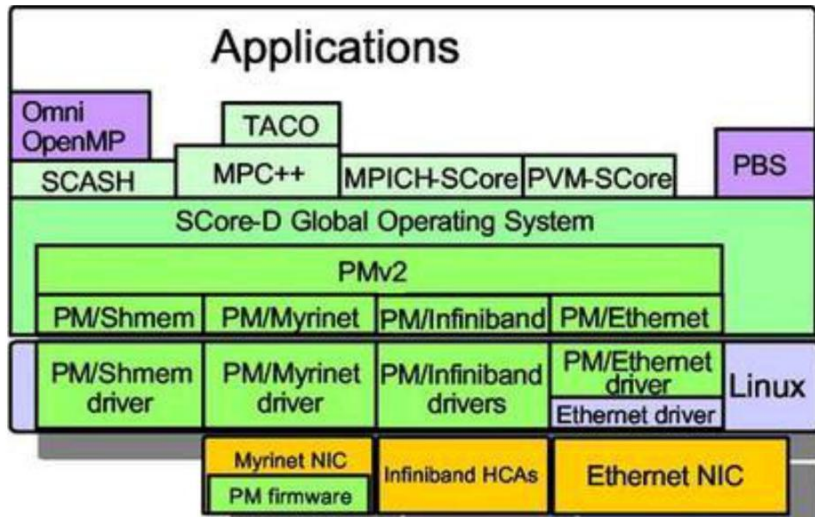
Architectural design focuses on the representation of the structure of software components, their properties, and interactions

Software Architecture

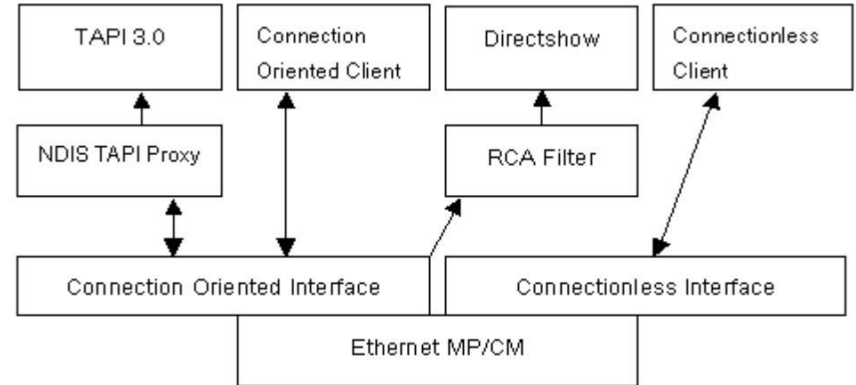
Importance of Architecture

- 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

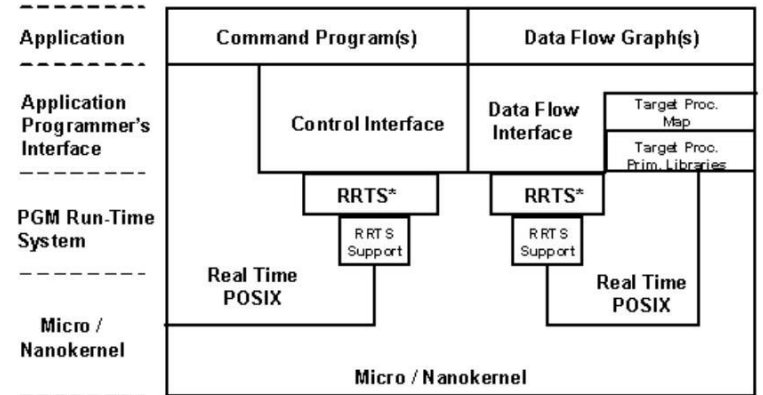
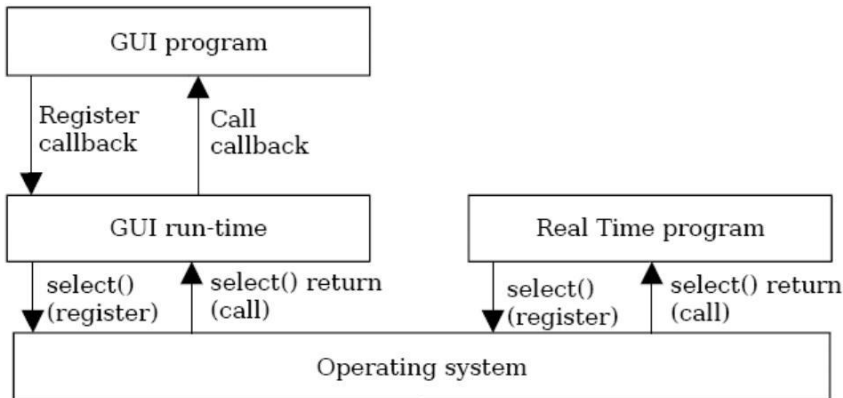
Example Software Architecture Diagrams



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



Software Architecture Diagram



*RRTS: RASPP Run-Time Support

Data Design

Purpose of Data Design

Data design translates data objects defined as part of the analysis model into

- Data structures at the software component level
- A possible database architecture at the application level

It focuses on the representation of data structures that are directly accessed by one or more software components

The challenge is to store and retrieve the data in such way that useful information can be extracted from the data environment

"Data quality is the difference between a data warehouse and a data garbage dump"

Data Design Principles

The systematic analysis principles that are applied to function and behavior should also be applied to data

All data structures and the operations to be performed on each one should be identified

A mechanism for defining the content of each data object should be established and used to define both data and the operations applied to it

Low-level data design decisions should be deferred until late in the design process

Data Design Principles

The representation of a data structure should be known only to those modules that must make direct use of the data contained within the structure

A library of useful data structures and the operations that may be applied to them should be developed

A software programming language should support the specification and realization of abstract data types

Software Architectural Styles

Common Architectural Styles of Homes



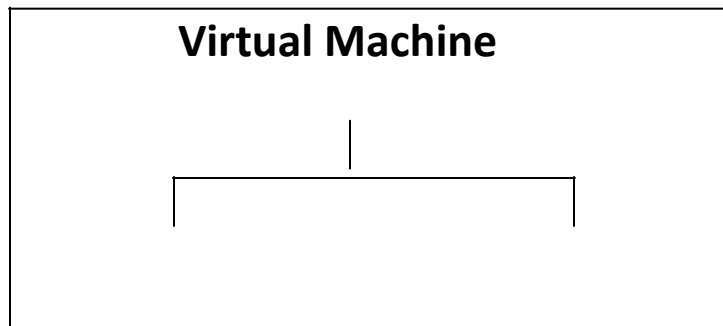
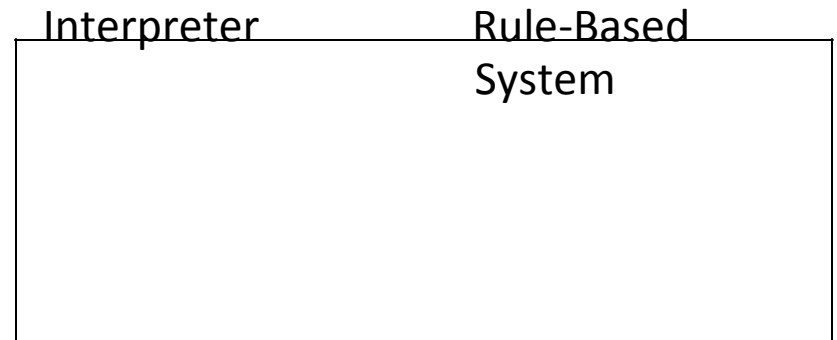
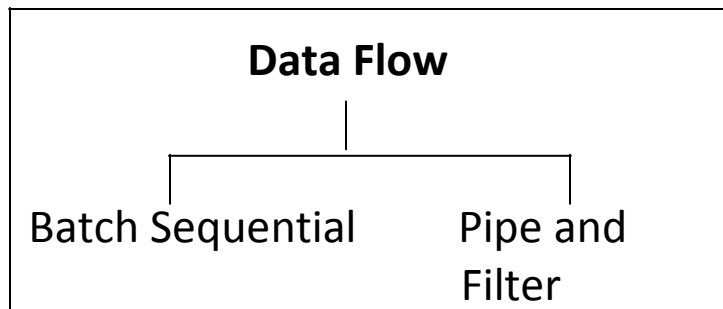
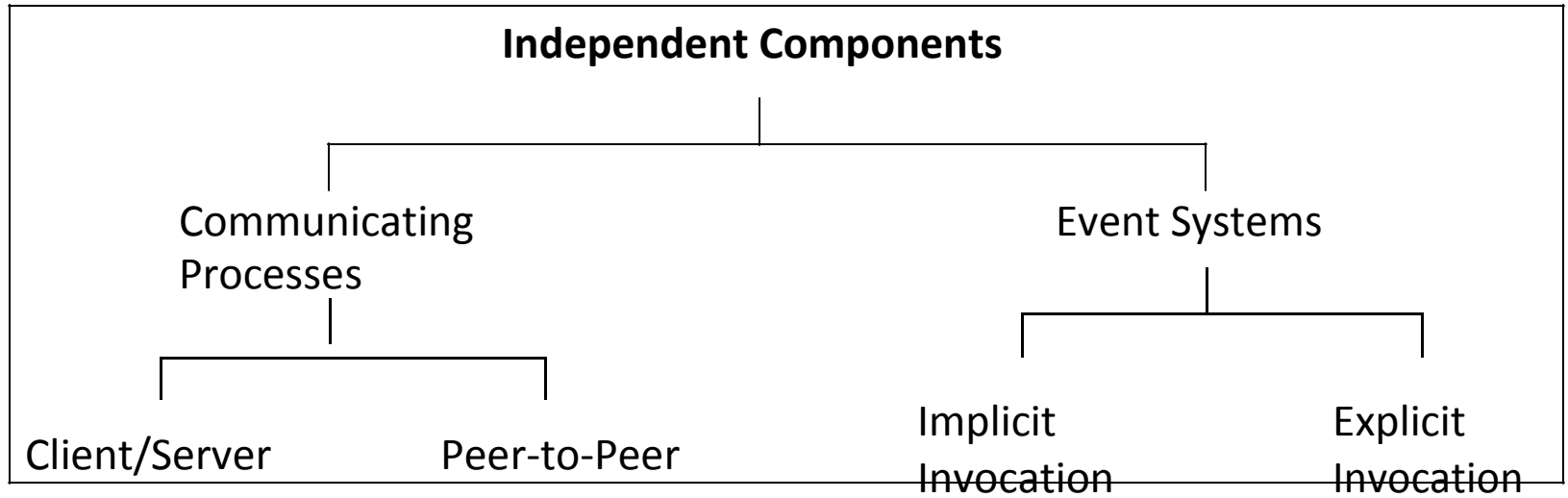
Software Architectural Style

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**
- Semantic constraints that define how components can be integrated to form the system**
- A topological layout of the components indicating their runtime interrelationships**

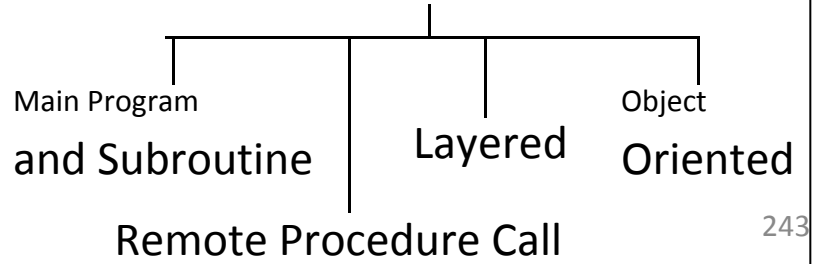
A Taxonomy of Architectural Styles



Data-Centered



Call and Return



Data Flow Style

Has the goal of modifiability

Characterized by viewing the system as a series of transformations on successive pieces of input data

Data enters the system and then flows through the components one at a time until they are assigned to output or a data store

Batch sequential style

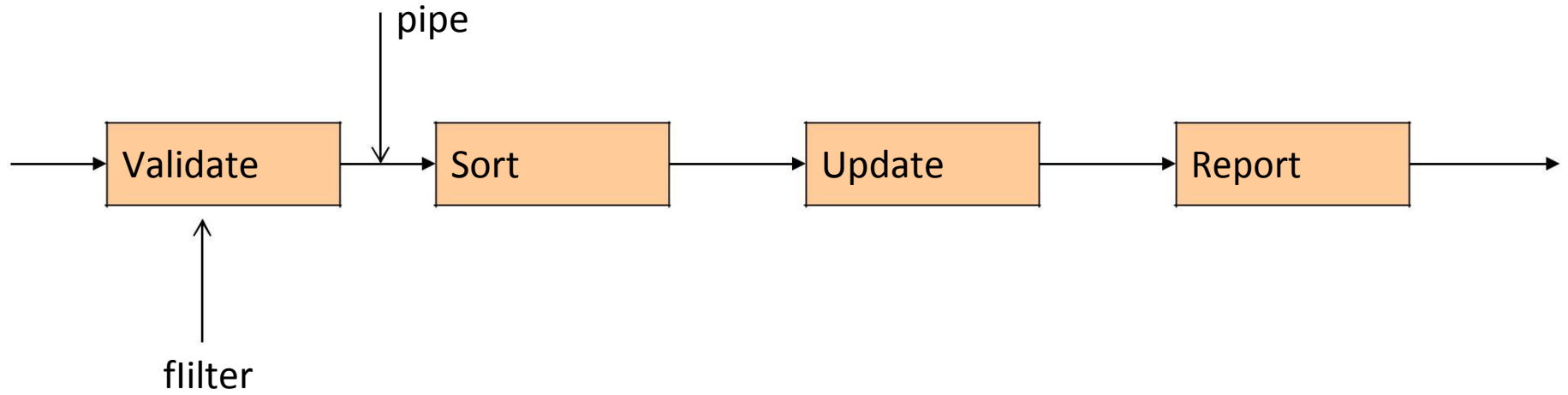
- The processing steps are independent components
- Each step runs to completion before the next step begins

Data Flow Style

Pipe-and-filter style

- **Emphasizes the incremental transformation of data by successive components**
- **The filters incrementally transform the data (entering and exiting via streams)**
- **The filters use little contextual information and retain no state between instantiations**
- **The pipes are stateless and simply exist to move data between filters**

Data Flow Style



Data Flow Style

Advantages

- Has a simplistic design in the limited ways in which the components interact with the environment
- Consists of no more and no less than the construction of its parts
- Simplifies reuse and maintenance
- Is easily made into a parallel or distributed execution in order to enhance system performance

Data Flow Style

Disadvantages

- Implicitly encourages a batch mentality so interactive applications are difficult to create in this style
- Ordering of filters can be difficult to maintain so the filters cannot cooperatively interact to solve a problem
- Exhibits poor performance

Filters typically force the least common denominator of data representation (usually ASCII stream)

Filter may need unlimited buffers if they cannot start producing output until they receive all of the input

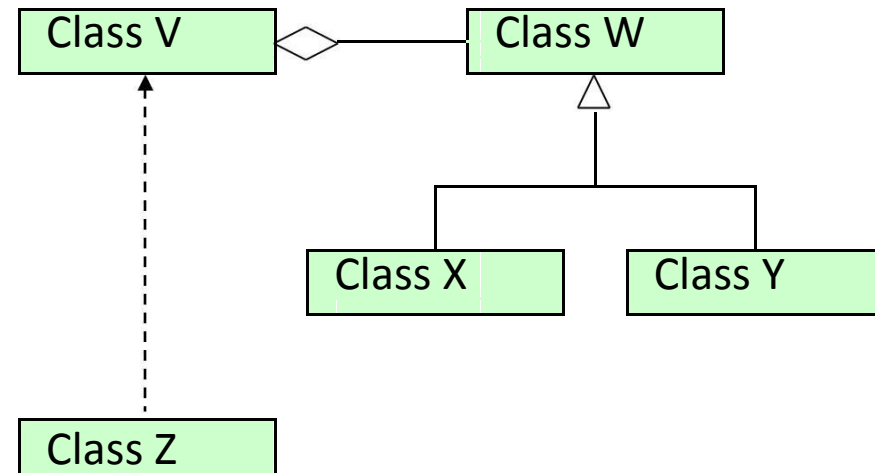
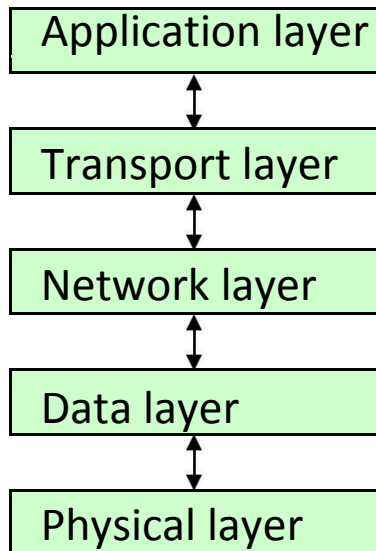
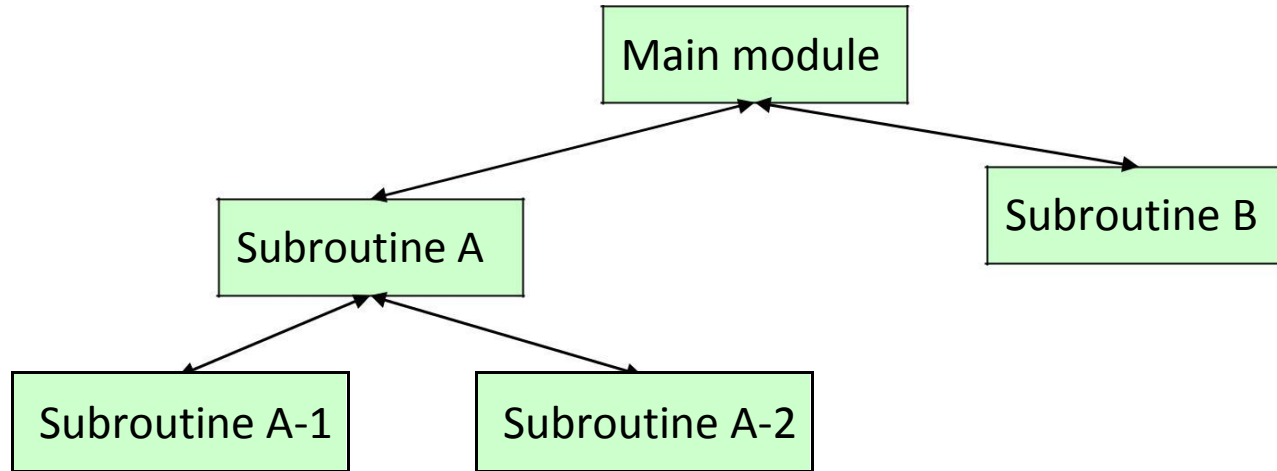
Each filter operates as a separate process or procedure call, thus incurring overhead in set-up and take-down time

Data Flow Style

Use this style when it makes sense to view your system as one that produces a well-defined easily identified output

The output should be a direct result of sequentially transforming a well-defined easily identified input in a time-independent fashion

Call-and-Return Style



Call-and-Return Style

Has the goal of modifiability and scalability

Has been the dominant architecture since the start of software development

Main program and subroutine style

- Decomposes a program hierarchically into small pieces (i.e., modules)
- Typically has a single thread of control that travels through various components in the hierarchy

Remote procedure call style

- Consists of main program and subroutine style of system that is decomposed into parts that are resident on computers connected via a network
- Strives to increase performance by distributing the computations and taking advantage of multiple processors
- Incurs a finite communication time between subroutine call and response

Call-and-Return Style

Object-oriented or abstract data type system

- Emphasizes the bundling of data and how to manipulate and access data
- Keeps the internal data representation hidden and allows access to the object only through provided operations
- Permits inheritance and polymorphism

Layered system

- Assigns components to layers in order to control inter-component interaction
- Only allows a layer to communicate with its immediate neighbor
- Assigns core functionality such as hardware interfacing or system kernel operations to the lowest layer

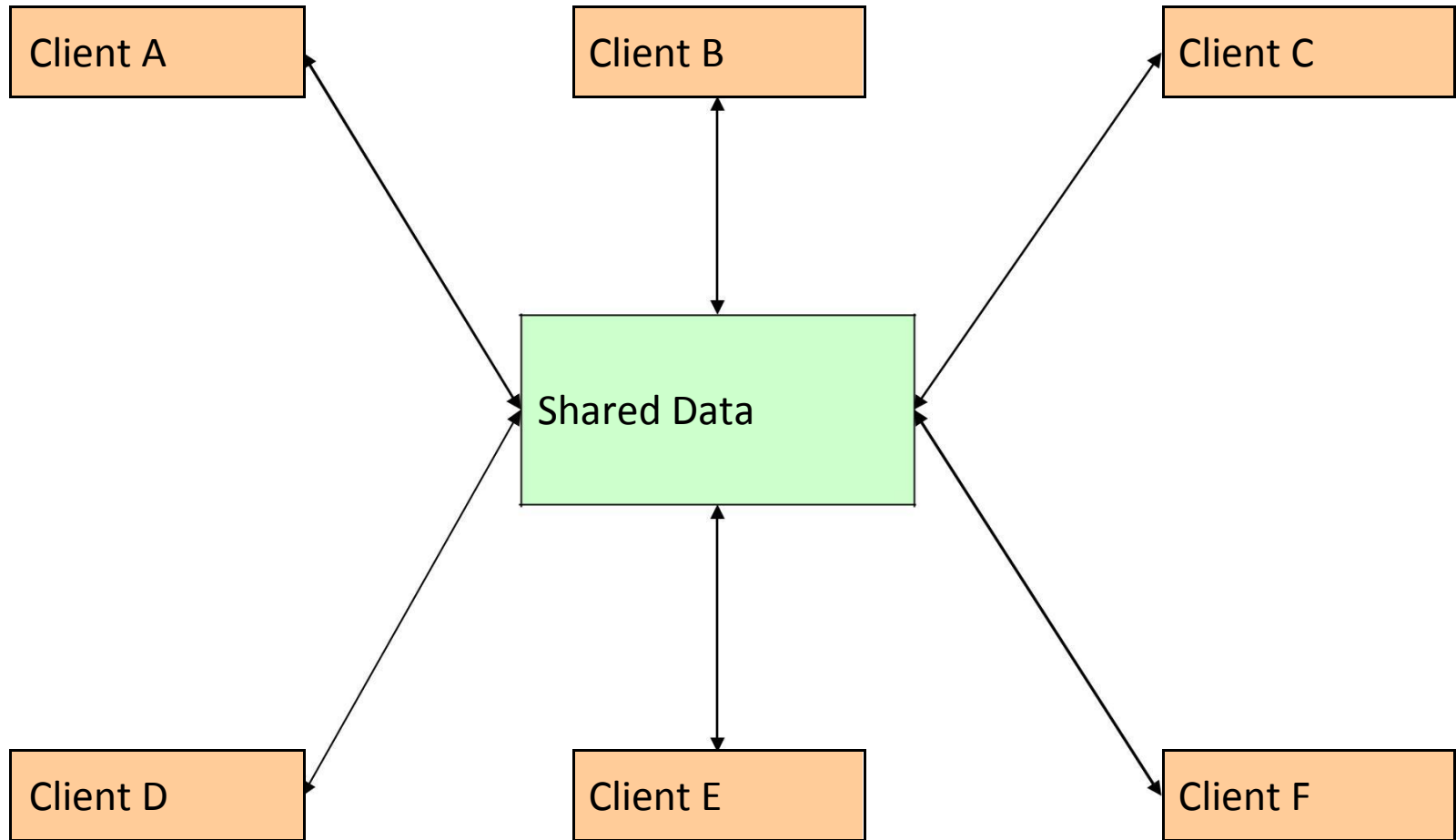
Call-and-Return Style

Layered system

- Builds each successive layer on its predecessor, hiding the lower layer and providing services for the upper layer
- Is compromised by layer bridging that skips one or more layers to improve runtime performance

Use this style when the order of computation is fixed, when interfaces are specific, and when components can make no useful progress while awaiting the results of request to other components

Data-Centered Style



Data-Centered Style

Has the goal of integrating the data

Refers to systems in which the access and update of a widely accessed data store occur

A client runs on an independent thread of control

The shared data may be a passive repository or an active blackboard

- A blackboard notifies subscriber clients when changes occur in data of interest

At its heart is a centralized data store that communicates with a number of clients

Clients are relatively independent of each other so they can be added, removed, or changed in functionality

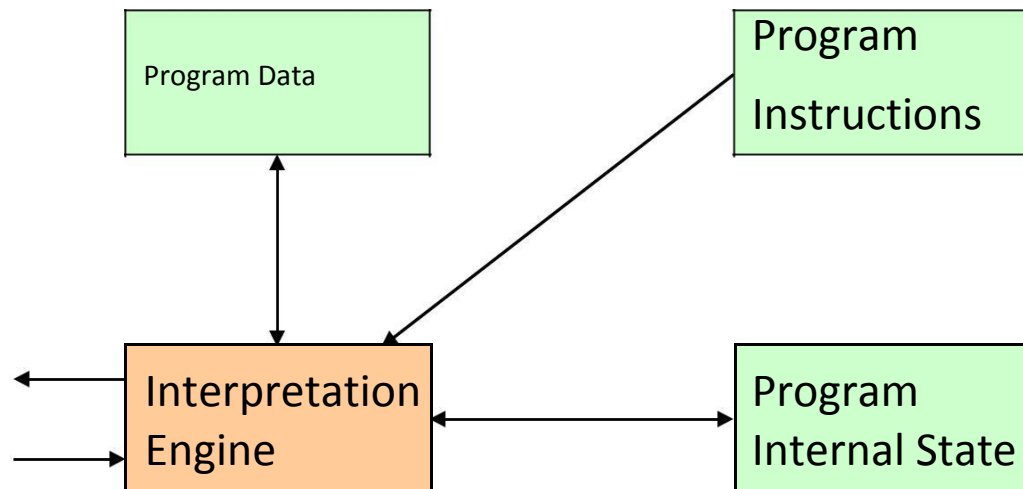
The data store is independent of the clients

Data-Centered Style

Use this style when a central issue is the storage, representation, management, and retrieval of a large amount of related persistent data

Note that this style becomes client/server if the clients are modeled as independent processes

Virtual Machine Style



Virtual Machine Style

Has the goal of portability

Software systems in this style simulate some functionality that is not native to the hardware and/or software on which it is implemented

- Can simulate and test hardware platforms that have not yet been built
- Can simulate "disaster modes" as in flight simulators or safety-critical systems that would be too complex, costly, or dangerous to test with the real system

Virtual Machine Style

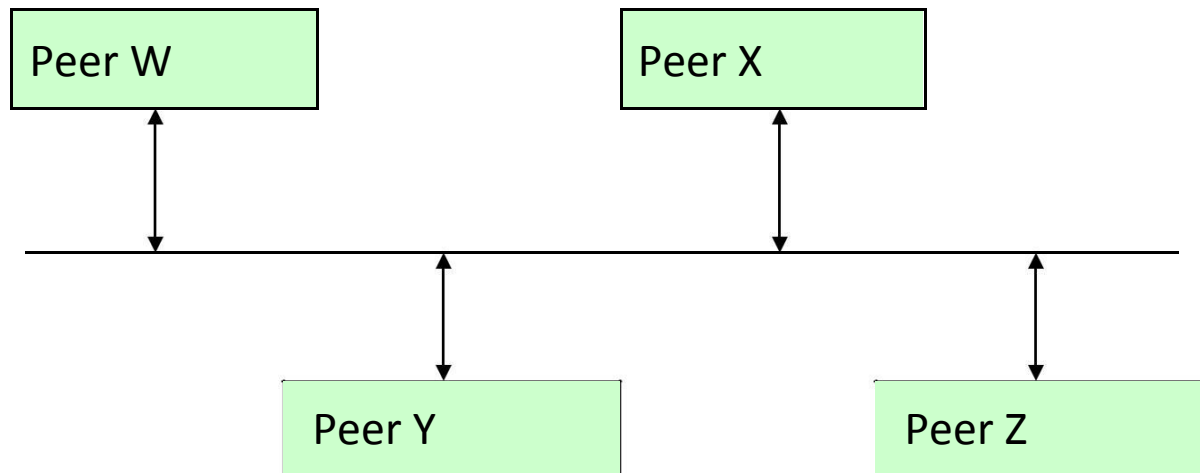
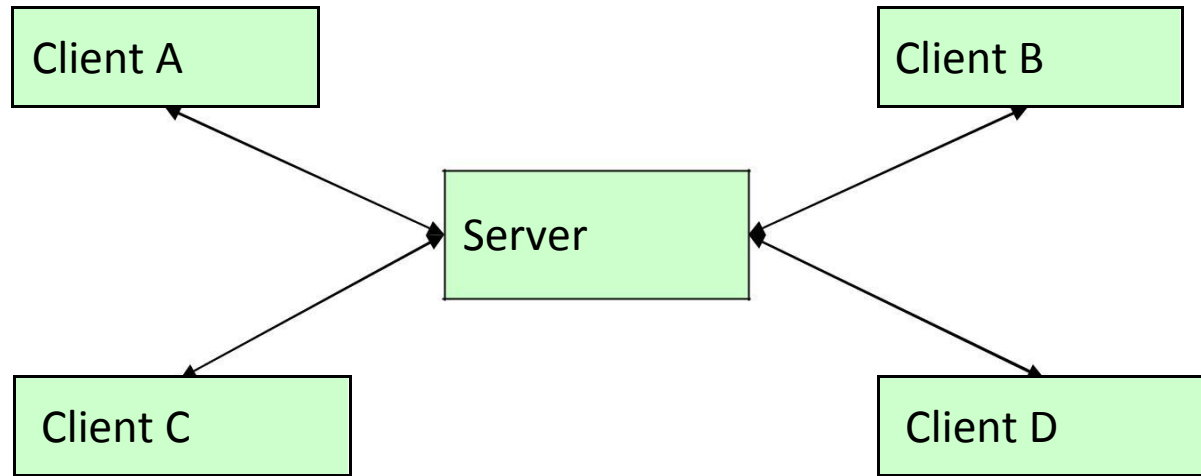
Examples include interpreters, rule-based systems, and command language processors

Interpreters

- Add flexibility through the ability to interrupt and query the program and introduce modifications at runtime**
- Incur a performance cost because of the additional computation involved in execution**

Use this style when you have developed a program or some form of computation but have no make of machine to directly run it on

Independent Component Style



Independent Component Style

Consists of a number of independent processes that communicate through messages

Has the goal of modifiability by decoupling various portions of the computation

Sends data between processes but the processes do not directly control each other

Event systems style

- Individual components announce data that they wish to share (publish) with their environment
- The other components may register an interest in this class of data (subscribe)

Independent Component Style

- Makes use of a message component that manages communication among the other components
- Components publish information by sending it to the message manager
- When the data appears, the subscriber is invoked and receives the data
- Decouples component implementation from knowing the names and locations of other components

Independent Component Style

Communicating processes style

- These are classic multi-processing systems
- Well-known subtypes are client/server and peer-to-peer
- The goal is to achieve scalability
- A server exists to provide data and/or services to one or more clients
- The client originates a call to the server which services the request

Independent Component Style

Use this style when

- Your system has a graphical user interface
- Your system runs on a multiprocessor platform
- Your system can be structured as a set of loosely coupled components
- Performance tuning by reallocating work among processes is important
- Message passing is sufficient as an interaction mechanism among components

Heterogeneous Styles

Systems are seldom built from a single architectural style

Three kinds of heterogeneity

– Locationally heterogeneous

The drawing of the architecture reveals different styles in different areas (e.g., a branch of a call-and-return system may have a shared repository)

– Hierarchically heterogeneous

A component of one style, when decomposed, is structured according to the rules of a different style

– Simultaneously heterogeneous

Two or more architectural styles may both be appropriate descriptions for the style used by a computer-based system

Architectural Patterns

- **An architectural patterns for software define a specific approach for handling some behavioural characteristic of the system. Some patterns can be**

Concurrency - many applications must handle multiple tasks in a manner that simulates parallelism

Persistence – data persists if it survives past the execution of the process that created it. Persistent data are stored in a database or file and may be read or modified by other processes at a later time

Distribution – the distribution problem addresses the manner in which system or components within systems communicate with one another in a distributed environment. Most common architectural pattern established to address the distribution problem is the ‘broker’ pattern

Architectural Design Process

Architectural Design Steps

Represent the system in context

Define archetypes

Refine the architecture into components

Describe instantiations of the system

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

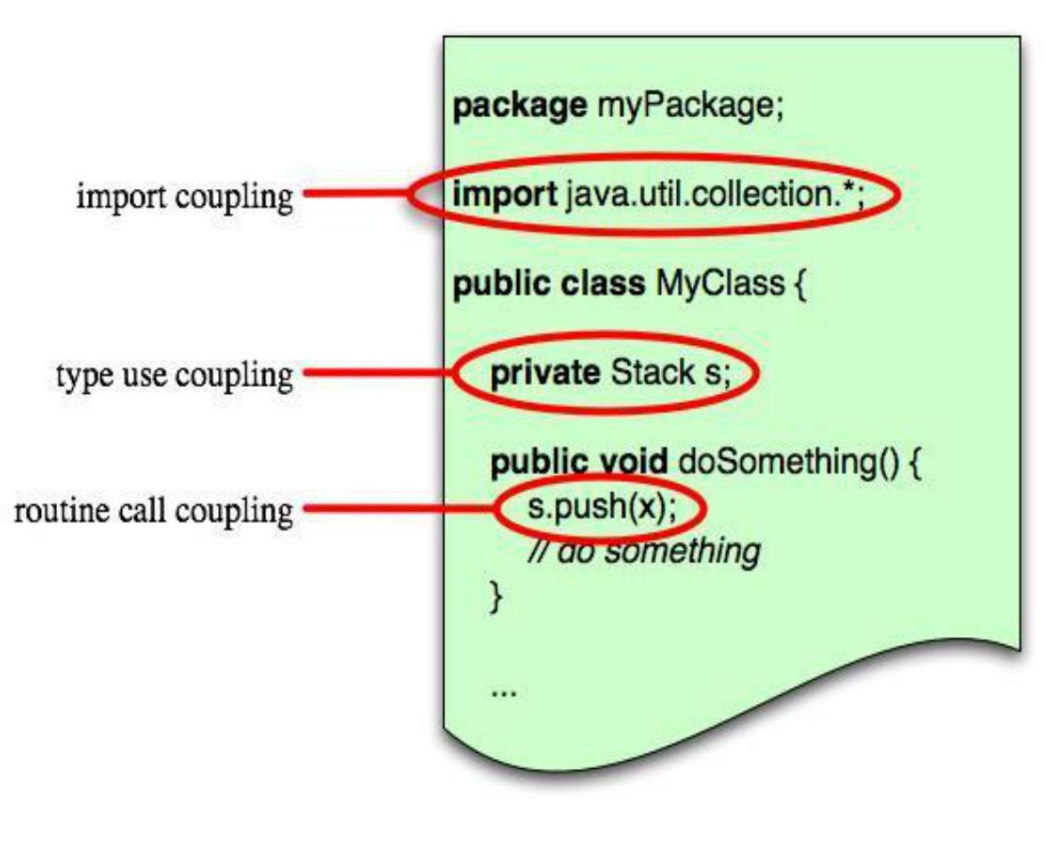
Data coupling

- 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 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

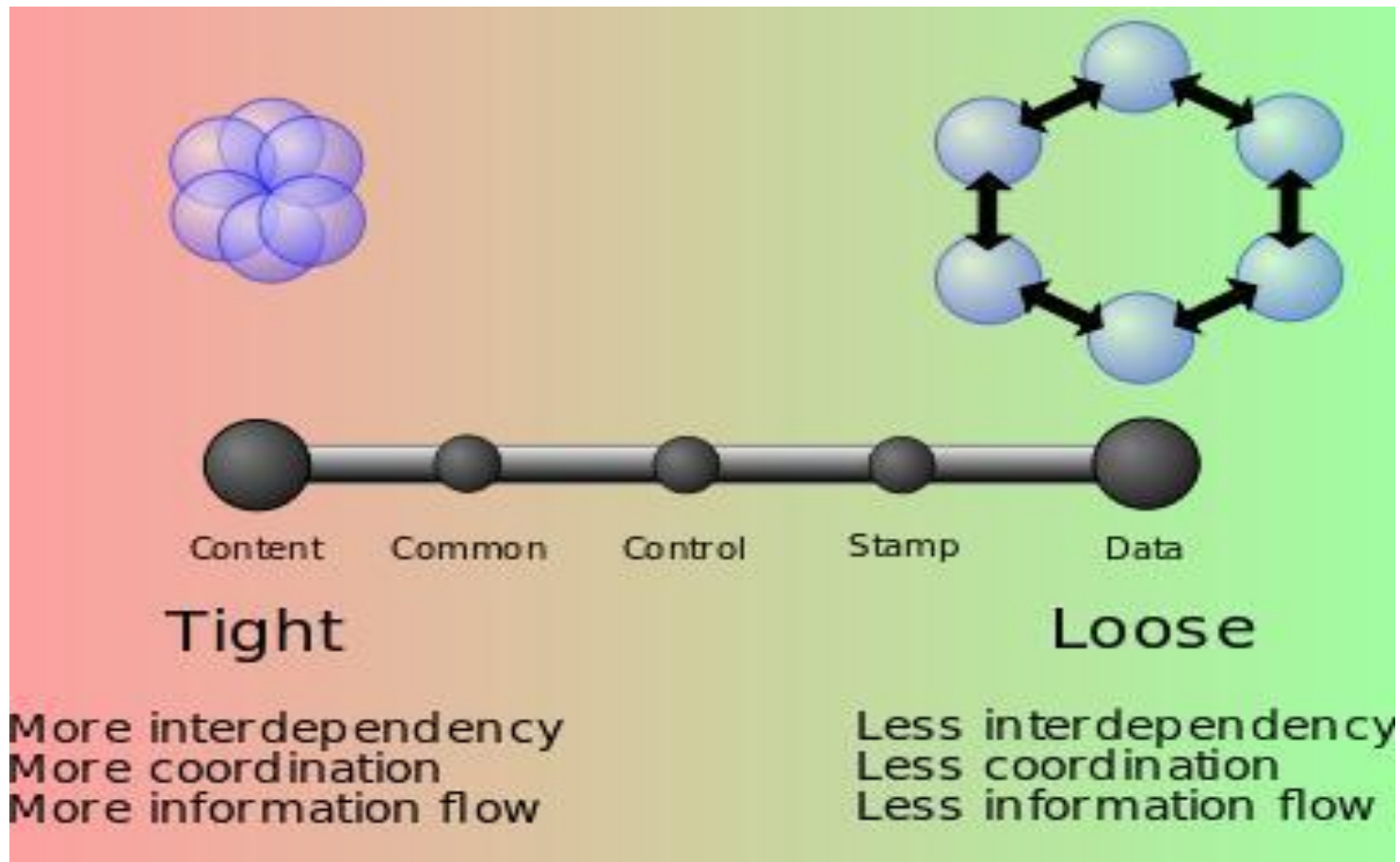
Use caution

- Common coupling

Be aware

- Routine call coupling
- Type use coupling
- Inclusion or import coupling

Coupling



Conducting Component-level Design

The designer has to transform information from analysis and architectural models into a design representation that provides sufficient detail to guide the construction activity. Typical steps for a component level design are

Identify design classes in problem domain

Identify infrastructure design classes

Elaborate design classes

Describe persistent data sources

Elaborate behavioral representations

Elaborate deployment diagrams

Refactor design and consider alternatives

Steps 1 & 2 – Identify Classes

Most classes from the problem domain are analysis classes created as part of the analysis model

The infrastructure design classes are introduced as components during architectural design

Step 3 – Class Elaboration

Specify message details when classes or components collaborate

Identify appropriate interfaces for each component

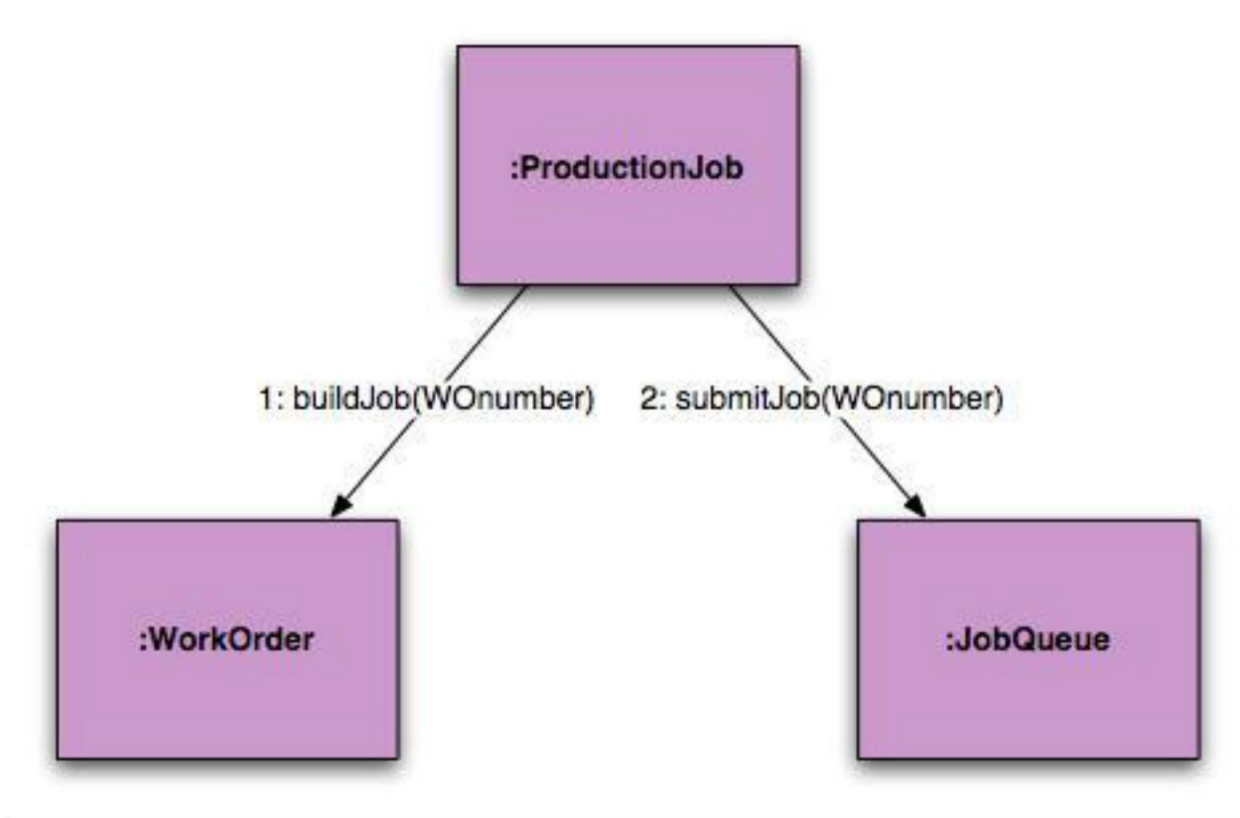
Elaborate attributes and define data structures required to implement them

Describe processing flow within each operation in detail

3a. Collaboration Details

Messages can be elaborated by expanding their syntax in the following manner:

- [guard condition] sequence expression (return value) :=
message name (argument list)



3a. Collaboration Details

[*guard condition*] is written in a Constraint Language (OCL) and specifies any set of conditions that must be met before the message can be sent

sequence expression is an integer value that indicates the sequence order in which the message is sent

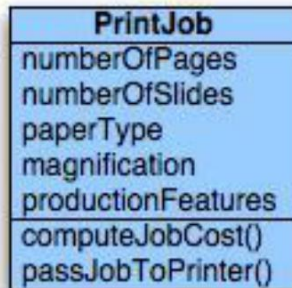
(return value) is the name of the information that is returned by the operation invoked by the message

message name identifies the operation to be invoked

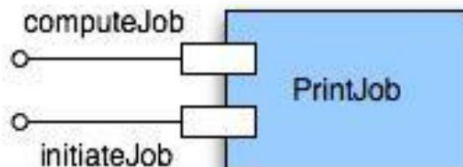
(argument list) list of attributes that are passed to the operation

3b. Appropriate Interfaces

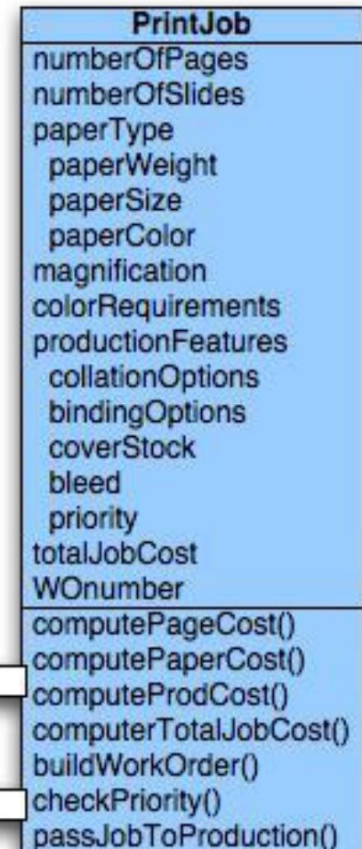
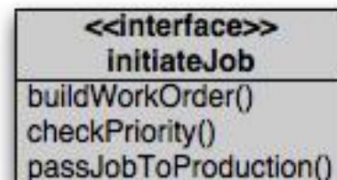
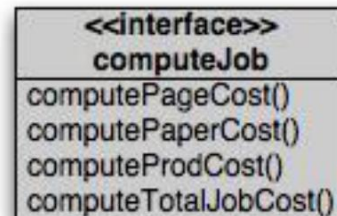
1 Analysis class



2 Design component



3 Elaborated design class



3b. Appropriate Interfaces

The `PrintJob` interface “`initiateJob`” does not exhibit sufficient cohesion because it performs three different sub functions *building a workorder* , *checking job priority* and passing job to production

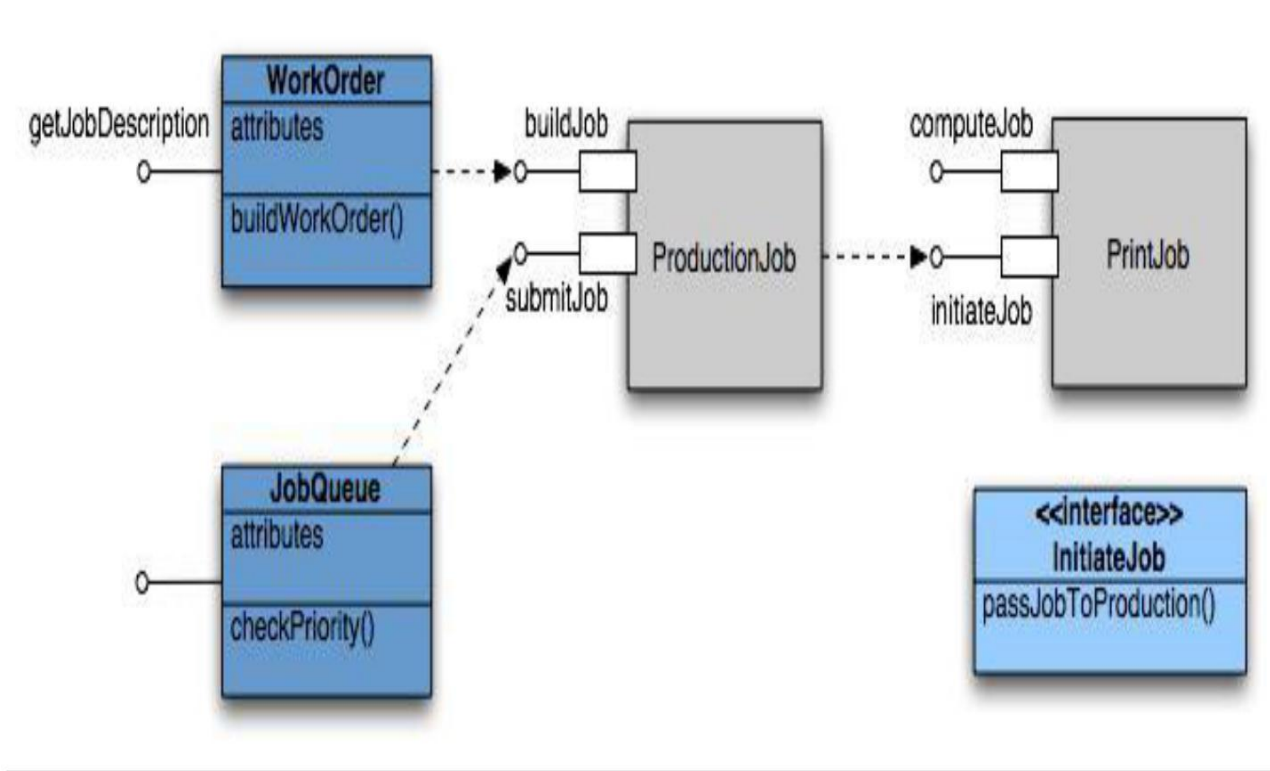
The interface design can be refactored as follows

- Define a new class *WorkOrder* that would take care of all activities associated with the assemble of a work order. The operation `buildWorkOrder()` becomes a part of that class
- Define a new class *JobQueue* that would incorporate the operation `checkPriority()`.
- A class *ProductionJob* would encompass all information associated with a production job to be passed to the production facility

The interface `InitiateJob` is now cohesive, focusing on one function

3b. Appropriate Interfaces

New interface InitiateJob



3c. Elaborate Attributes

Analysis classes will typically only list names of general attributes (ex. paperType).

List all attributes during component design.

UML syntax:

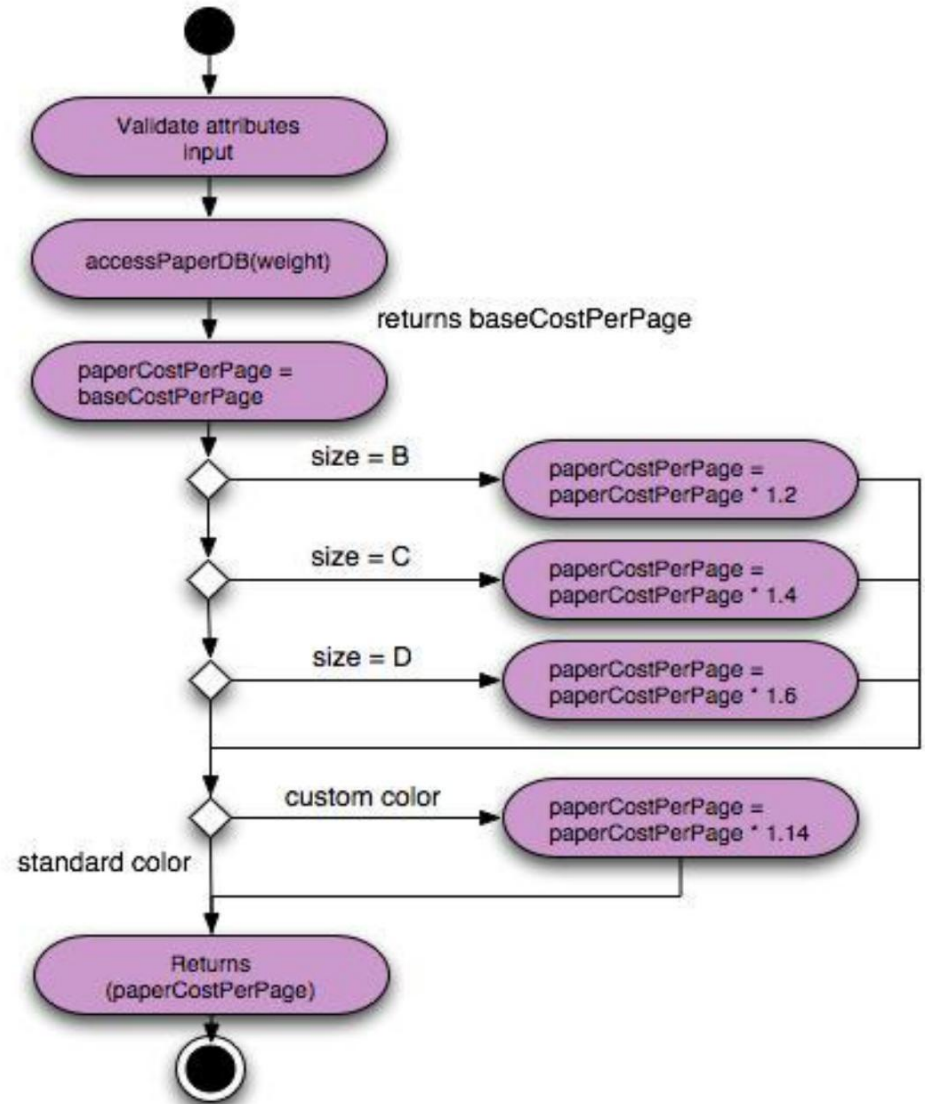
- name : type-expression = initial-value { property string }

For example, paperType can be broken into weight, size, and color. The weight attribute would be:

- paperType-weight: string =
 “A” { contains 1 of 4 values – A, B, C, or D }

3d. Describe Processing Flow

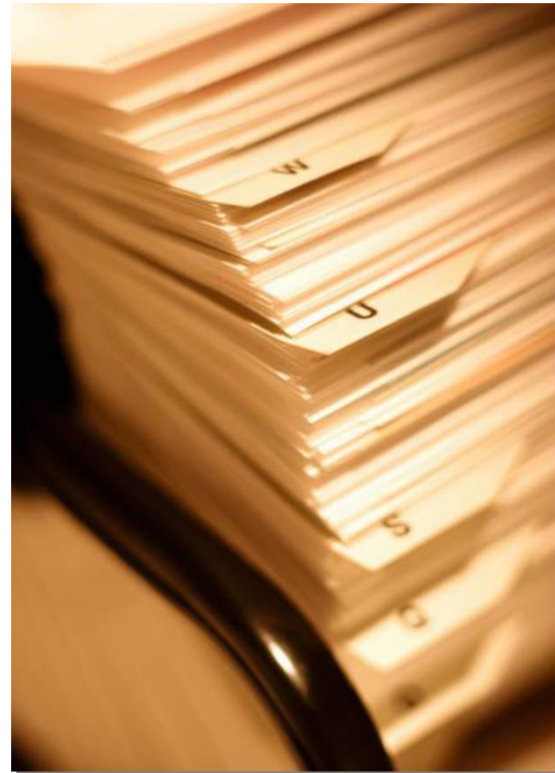
Activity diagram
for computePaperCost().



Step 4 – Persistent Data

Describe persistent

data sources (databases and files) and identify the classes required to manage them.



Step 5 – Elaborate Behavior

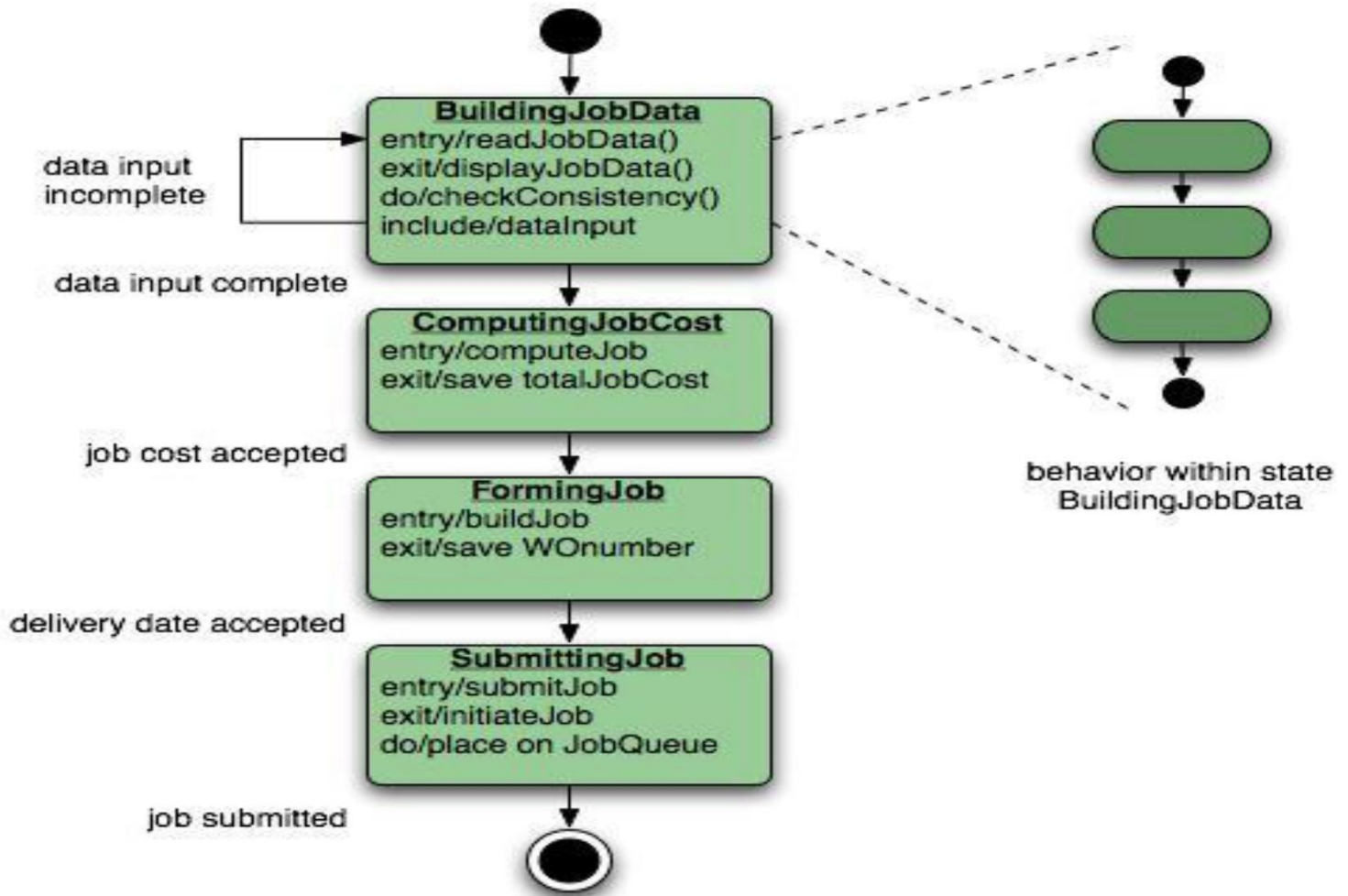
The dynamic behaviour of an object is affected by events that are external to it and the current state (mode of behaviour) of the object.

It is sometimes necessary to model the behavior of a design class.

The designer must examine all use- cases that are relevant to the design class throughout its life

The use-cases will help designer to delineate the events that affect the object and its states in which the object resides as time passes and events occur

Step 5 – Elaborate Behavior



Step 5 – Elaborate Behavior

Transitions from state to state (represented by rectangle with round corners) have the form:

- `event-name (parameter-list) [guard-condition] / action expression`
- Where `event-name` identifies the event
- `parameter-list` incorporates data that are associated with the event
- `guard-condition` specifies a condition that must be met before the event can occur
- `action-expression` defines an action that occurs as the transition takes place

Step 5 – Elaborate Behavior

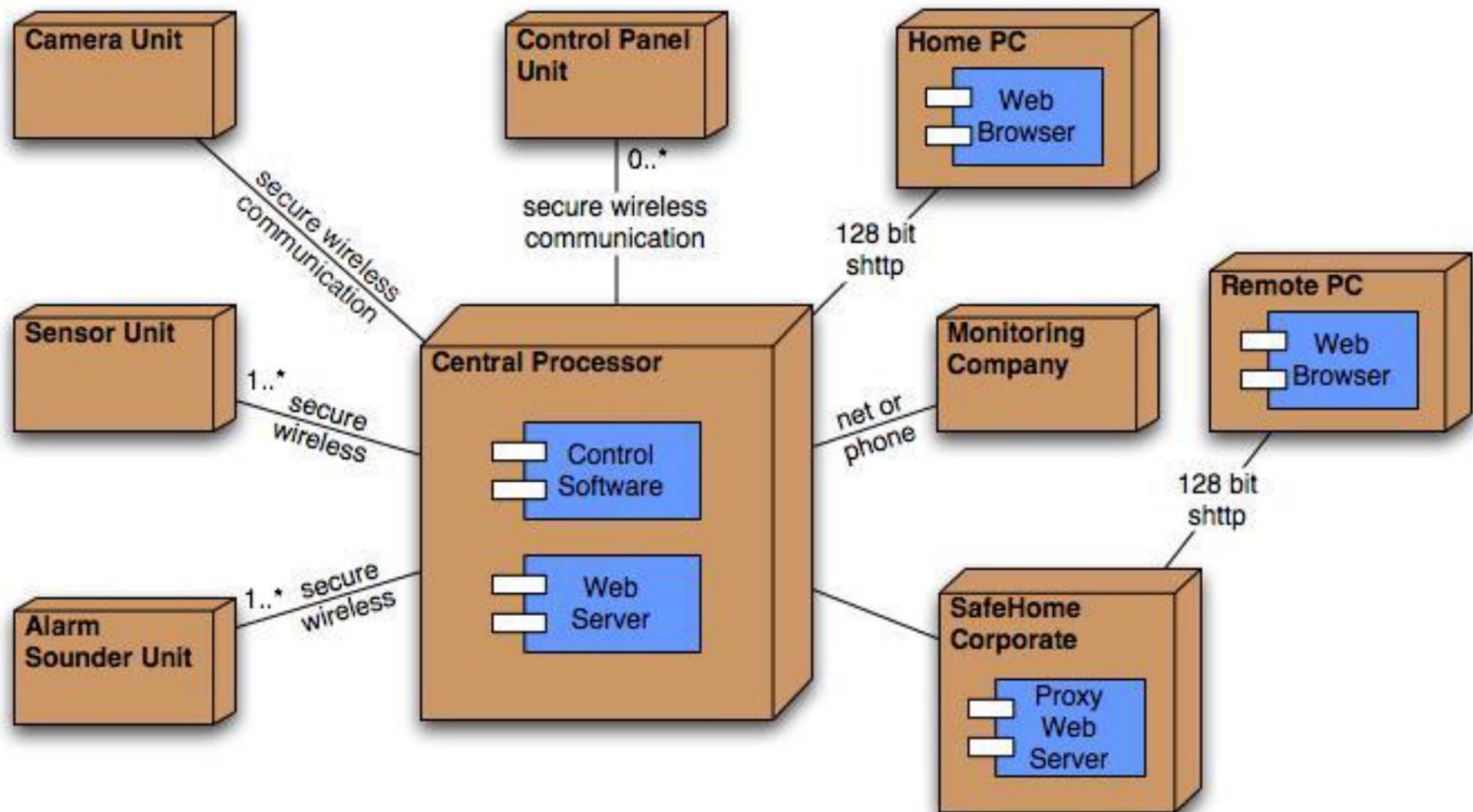
Each state may define entry/ and exit actions that occur as transitions into and out of the state occur.

The do indicator provides mechanism for indicating the activities that occur while in the state

The include indicator provides a means for elaborating the behaviour by embedding more state chart detail within the definition of the state

Step 6 – Elaborate Deployment

Deployment diagrams are elaborated to represent the location of key packages or components



Step 7 – Redesign/Reconsider

The first component-level model that was created will not be as complete, consistent, or accurate as the n^{th} iteration that is applied to the model.

The best designers will consider many alternative design solutions before settling on the final design model.

Object Constraint Language

The Object Constraint Language was designed to be used alongside UML to supplement the design specification

Usage

- To provide a formal specification of a system design.**
- To express constraints formally.**
- To express the semantics of methods formally through pre & post conditions.**

Object Constraint Language

OCL expressions

- OCL is used to define **invariants** (constraints) on the state of classes and to express the semantics of methods through **pre** and **post** conditions.
- Objects of the class containing the expression are referred to as *self*, and related objects in the class diagram are referenced
- When referring to a property in another package, the property name is prefixed with the package name :- *package::property*

Object Constraint Language

OCL expressions

- The simplest OCL statements are constructed in four parts

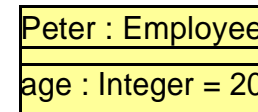
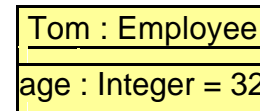
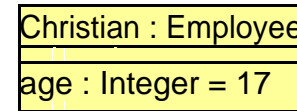
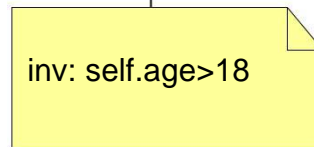
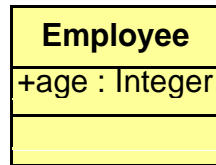
A context that defines the limited situation in which the statement is valid

A property that represents some characteristics of the context e.g., if the context is a class, a property might be an attribute

An operation (e.g., arithmetic, set oriented) that manipulates or qualifies the property

keywords that (e.g., if, then, and) that are used to specify conditional expressions

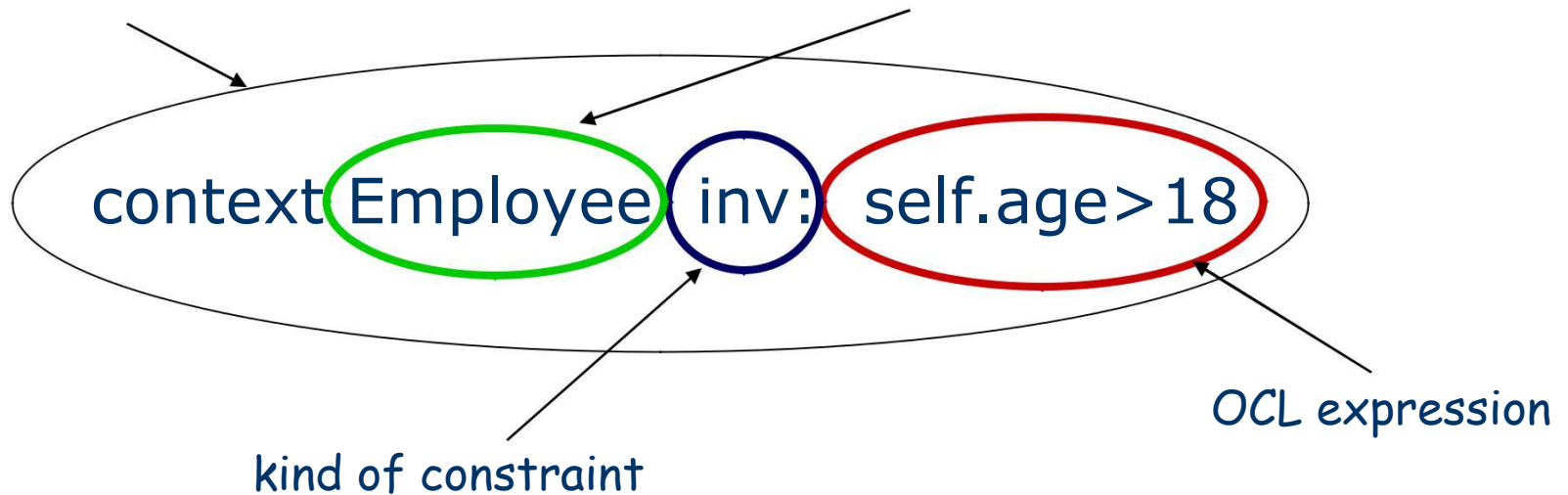
OCL Constraint



instances of the
type Employee

constraint

context for the expression



Constraints (Invariants)

Employee
age : Integer wage : Integer
raiseWage(newWage : Integer)

inv invariant: constraint must be true

for all instances of constrained type at any time

Constraint is always of the type Boolean

```
context Employee
```

```
inv: self.age > 18
```

Constraints (Pre- and Postconditions)

Employee
age : Integer wage : Integer
raiseWage(newWage : Integer)

pre precondition: constraint must be true, before execution of an Operation

post postcondition: constraint must be true, after execution of an Operation

self refers to the object on which the operation was called

return designates the result of the operation (if available)

The names of the parameters can also be used

```
context Employee::raiseWage (newWage : Integer)
```

```
pre: newWage > self.wage
```

```
post: wage = newWage
```

Designing Conventional Components

Conventional design constructs emphasize the maintainability of a functional/procedural program

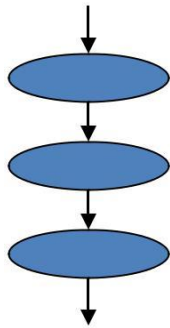
- Sequence, condition, and repetition

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

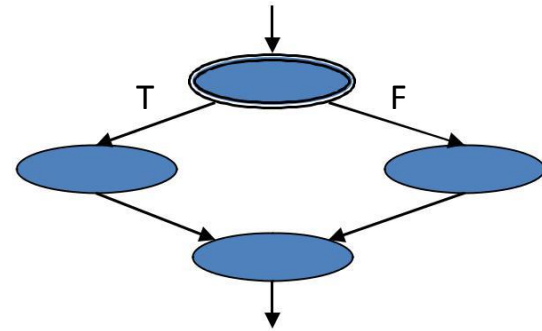
Various notations depict the use of these constructs

- Graphical design notation
 - Sequence, if-then-else, selection, repetition (see next slide)
- Tabular design notation
- Program design language

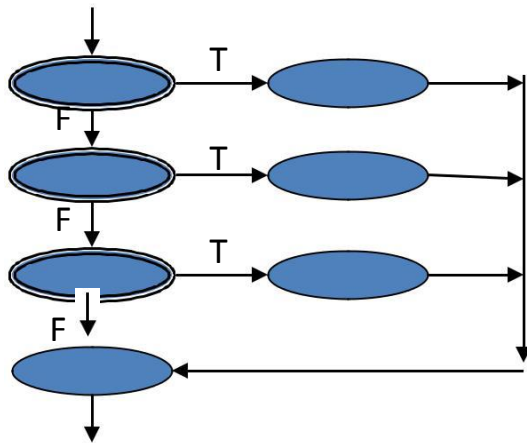
Graphical Design Notation



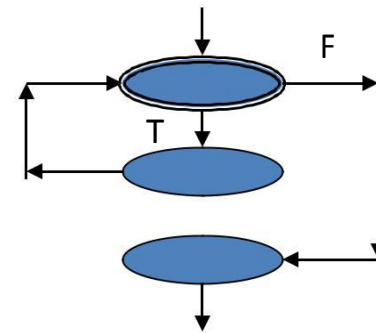
Sequence



If-then-else



Selection



Repetition

Tabular Design Notation

List all actions that can be associated with a specific procedure (or module)

List all conditions (or decisions made) during execution of the procedure

Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions

Define rules by indicating what action(s) occurs for a set of conditions

Tabular Design Notation (continued)

Rules

Conditions	1	2	3	4
Condition A	T	T		F
Condition B		F	T	
Condition C	T			T
Actions				
Action X	✓		✓	
Action Y				✓
Action Z	✓	✓		✓



Program Design Language

Program Design Language is also called structured English or psuedocode is a pidgin language in that it uses the vocabulary of one language (English) and the overall syntax of another (structured programming language)

Difference between PDL and real programming language lies in the use of narrative text embedded directly within the PDL statements

PDL cannot be compiled

Program Design Language

component alarmManagement;

The part of this component is to manage control panel switches and input from sensor by type and to act on any alarm condition that is encountered.

Program Design Language is also called structured English or pseudocode is a pidgin language in that it uses the vocabulary of one language (English) and the overall syntax of another (structured programming language)

Difference between PDL and real programming language lies in the use of narrative text embedded directly within the PDL statements

PDL cannot be compiled

```
make checkSensor procedure returning signalValue
if signalValue > bound [alarmType]
    then phone.message = message [alarmType]
    set alarmBell to "on" for alarmTimeSeconds
```


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

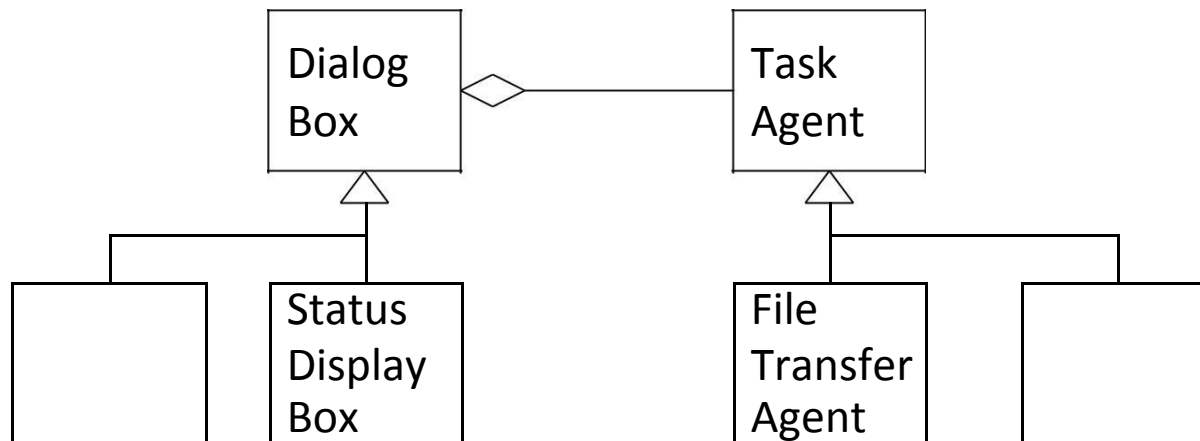
Derived from the analysis model of the requirements

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

Are the users trained professionals, technicians, clerical or manufacturing workers?

What level of formal education does the average user have?

Are the users capable of learning on their own from written materials or have they expressed a desire for classroom training?

Are the users expert typists or are they keyboard phobic?

What is the age range of the user community?

Will the users be represented predominately by one gender?

How are users compensated for the work they perform or are they volunteers?

User Interface Analysis

User Analysis Questions

Do users work normal office hours, or do they work whenever the job is required?

Is the software to be an integral part of the work users do, or will it be used only occasionally?

What is the primary spoken language among users?

What are the consequences if a user makes a mistake using the system?

Are users experts in the subject matter that is addressed by the system?

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

Are various types of data assigned to consistent locations on the screen (e.g., photos always in upper right corner)?

Are users able to customize the screen location for content?

Is proper on-screen identification assigned to all content?

Can large reports be partitioned for ease of understanding?

Are mechanisms available for moving directly to summary information for large collections of data?

Is graphical output scaled to fit within the bounds of the display device that is used?

How is color used to enhance understanding?

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)**

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

Using information developed during user interface analysis, define user interface objects and actions (operations)

Define events (user actions) that will cause the state of the user interface to change; model this behavior

Depict each interface state as it will actually look to the end user

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

Interface Objects and Actions

- 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

Design Issues to Consider

Four common design issues usually surface in any user interface

- **System response time (both length and variability)**

- **User help facilities**

 - 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 information handling**

 - How meaningful to the user, how descriptive of the problem**

- **Menu and command labeling**

 - Consistent, easy to learn, accessibility, internationalization**

Many software engineers do not address these issues until late in the design or construction process

- **This results in unnecessary iteration, project delays, and customer frustration**

User Interface Design

Guidelines for Error Messages

- 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

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

Questions for Menu Labeling and Typed Commands

- 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?

User Interface Design

Interface Design Patterns

- **Patterns are available for**
 - The complete UI**
 - Page layout**
 - Forms and input**
 - Tables**
 - Direct data manipulation**
 - Navigation**
 - Searching**
 - Page elements**
 - e-Commerce**

Design principles

User familiarity

- The interface should be based on user-oriented terms and concepts rather than computer concepts. For example, an office system should use concepts such as letters, documents, folders etc. rather than directories, file identifiers, etc.

Consistency

- The system should display an appropriate level of consistency. Commands and menus should have the same format, command punctuation should be similar, etc.

Minimal surprise

- If a command operates in a known way, the user should be able to predict the operation of comparable commands

Design principles

Recoverability

- **The system should provide some resilience to user errors and allow the user to recover from errors. This might include an undo facility, confirmation of destructive actions, 'soft' deletes, etc.**

User guidance

- **Some user guidance such as help systems, on-line manuals, etc. should be supplied**

User diversity

- **Interaction facilities for different types of user should be supported. For example, some users have seeing difficulties and so larger text should be available**

Interaction styles

Direct manipulation

Menu selection

Form fill-in

Command language

Natural language

Interaction styles

Interaction style	Main advantages	Main disadvantages	Application examples
Direct manipulation	Fast and intuitive interaction Easy to learn	May be hard to implement. Only suitable where there is a visual metaphor for tasks and objects.	Video games CAD systems
Menu selection	Avoids user error Little typing required	Slow for experienced users. Can become complex if many menu options.	Most general-purpose systems
Form fill-in	Simple data entry Easy to learn Checkable	Takes up a lot of screen space. Causes problems where user options do not match the form fields.	Stock control, Personal loan processing
Command language	Powerful and flexible	Hard to learn. Poor error management.	Operating systems, Command and control systems
Natural language	Accessible to casual users Easily extended	Requires more typing. Natural language understanding systems are unreliable.	Information retrieval systems

Colour displays

Colour adds an extra dimension to an interface and can help the user understand complex information structures.

Colour can be used to highlight exceptional events.

Common mistakes in the use of colour in interface design include:

- The use of colour to communicate meaning;**
- The over-use of colour in the display.**

Colour use guidelines

Limit the number of colours used and be conservative in their use.

Use colour change to show a change in system status.

Use colour coding to support the task that users are trying to perform.

Use colour coding in a thoughtful and consistent way.

Be careful about colour pairings.

Error messages

Error message design is critically important.

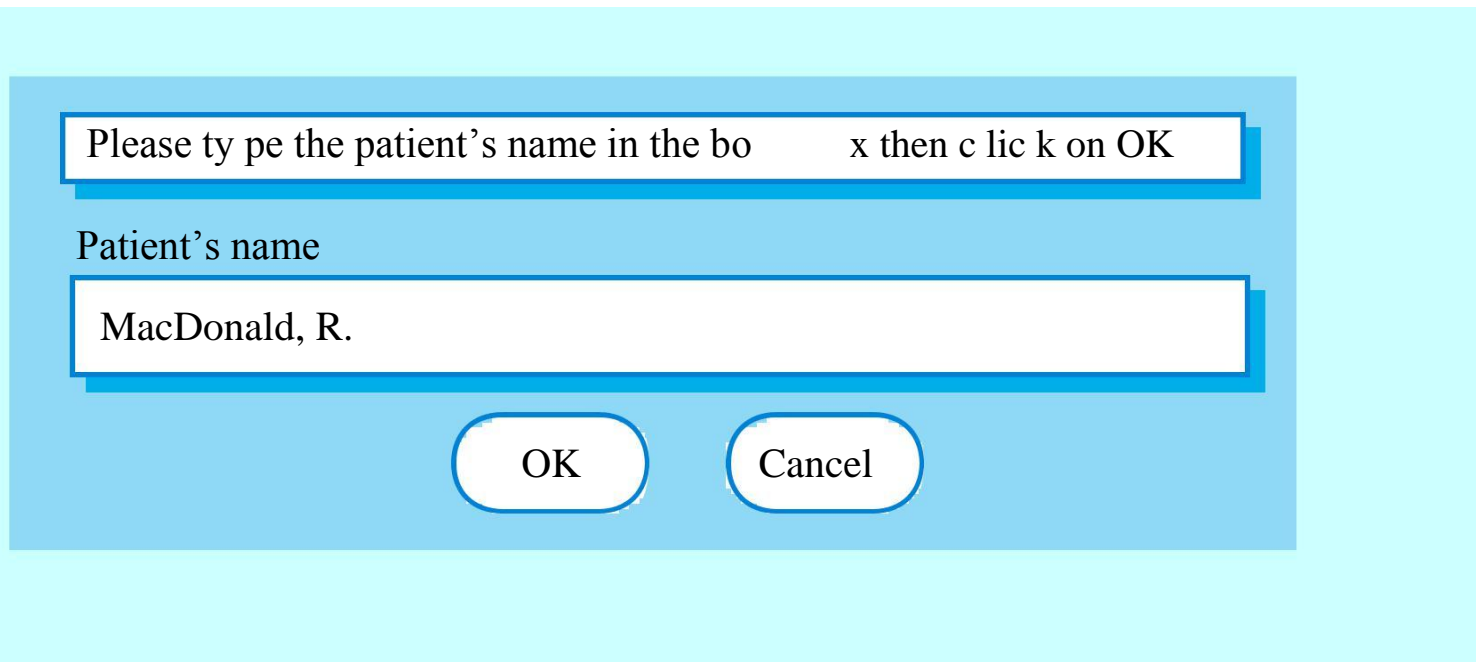
Poor error messages can mean that a user rejects rather than accepts a system.

Messages should be polite, concise, consistent and constructive.

The background and experience of users should be the determining factor in message design.

User error

Assume that a nurse misspells the name of a patient whose records she is trying to retrieve.



Please type the patient's name in the box then click on OK

Patient's name

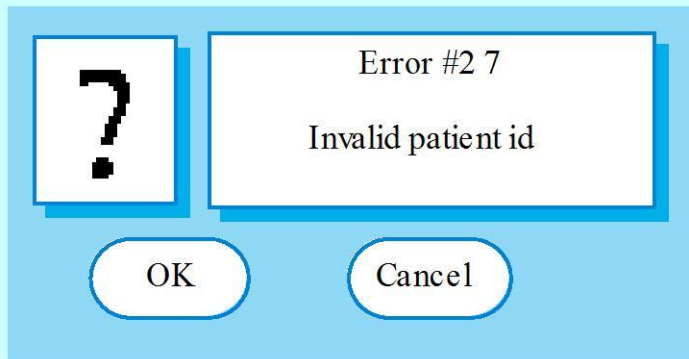
MacDonald, R.

OK Cancel

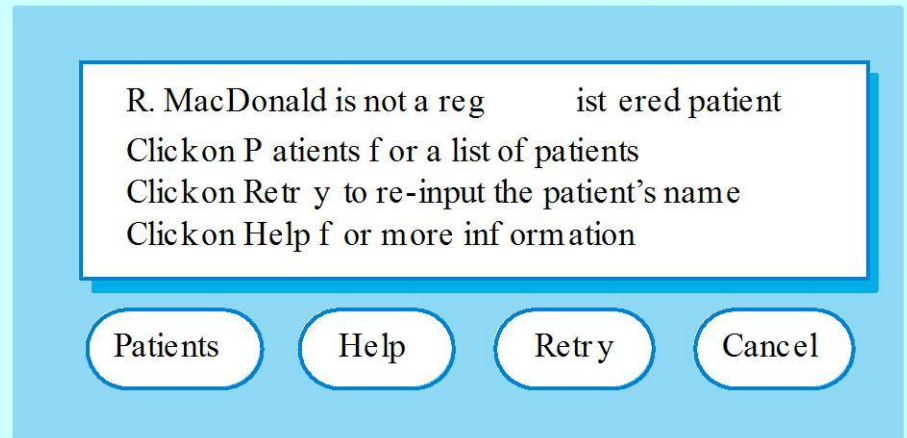
The image shows a light blue dialog box with a white text input field. The input field contains the text "MacDonald, R.". Below the input field are two buttons: "OK" and "Cancel". The text "Please type the patient's name in the box then click on OK" is displayed above the input field, and "Patient's name" is displayed to the left of the input field.

Good and bad message design

System-oriented error message



User-oriented error message



User Interface Evaluation

Design and Prototype Evaluation

- Before prototyping occurs, a number of evaluation criteria can be applied during design reviews to the design model itself

The amount of learning required by the users

- Derived from the length and complexity of the written specification and its interfaces

The interaction time and overall efficiency

- Derived from the number of user tasks specified and the average number of actions per task

The memory load on users

- Derived from the number of actions, tasks, and system states

The complexity of the interface and the degree to which it will be accepted by the user

- Derived from the interface style, help facilities, and error handling procedures

User Interface Evaluation

Prototype evaluation can range from an informal test drive to a formally designed study using statistical methods and questionnaires

The prototype evaluation cycle consists of prototype creation followed by user evaluation and back to prototype modification until all user issues are resolved

The prototype is evaluated for

- Satisfaction of user requirements**
- Conformance to the three golden rules of user interface design**
- Reconciliation of the four models of a user interface**

User Interface Evaluation -Usability attributes

Attribute	Description
Learnability	How long does it take a new user to become productive with the system?
Speed of operation	How well does the system response match the user's work practice?
Robustness	How tolerant is the system of user error?
Recoverability	How good is the system at recovering from user errors?
Adaptability	How closely is the system tied to a single model of work?

User Interface Evaluation

Simple evaluation techniques

- Questionnaires for user feedback.
- Video recording of system use and subsequent tape evaluation.
- Instrumentation of code to collect information about facility use and user errors.
- The provision of code in the software to collect on-line user feedback.

Syllabus

Testing Strategies : A strategic approach to software testing, test strategies for conventional software, Black-Box and White-Box testing, Validation testing, System testing, the art of Debugging.

Product metrics : Software Quality, Metrics for Analysis Model, Metrics for Design Model, Metrics for source code, Metrics for testing, Metrics for maintenance.

Strategic approach for software testing

Software testing is a formal process carried out by a specialized testing team in which a software unit, several integrated software units or an entire software package are examined by running the programs on a computer.

All the associated tests are performed according to approved test procedures on approved test cases.

Software is tested to uncover errors that were made inadvertently as it was designed and constructed

A strategy for software testing is developed by the project manager, software engineers and testing specialists

Strategic approach for software testing

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

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

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

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

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

Strategic approach for software testing

Testing often accounts for more project effort than any other software engineering activity. If it is conducted haphazardly, time is wasted, unnecessary effort is expended, and even worse, errors sneak through undetected. It would therefore necessary to establish a systematic strategy for testing software

Early testing focuses on a single component or a small group of related components and applies tests to uncover errors in the data and processing logic that have been encapsulated by the components.

Strategic approach for software testing

After components are tested they must be integrated until the complete system is constructed. At this point, a series of high-order tests are executed to uncover errors in the meeting customer requirements.

A test specification documents the software team's approach to testing by defining a plan that describes an overall strategy and a procedure that defines specific testing steps and the tests that will be conducted.

By reviewing the test specification prior to testing, the completeness of test cases and testing tasks.

An effective test plan and procedure will lead to the orderly construction of the software and the discovery of errors at each stage in the construction process.

Strategic approach for software testing

Software testing strategies provide a template for testing and have the following characteristics

- To perform effective testing, a software team should conduct effective formal technical reviews**
- Testing begins at the component level and works “outward” toward the integration of the entire computer-based system**
- Different testing techniques are appropriate at different points in time**
- Testing is conducted by the developer of the software and an independent test group**
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy**

Strategic approach for software testing

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

Verification and Validation

- Verification refers to the set of activities that ensure that software correctly implements a specific function or algorithm (are we building the product right) (Are the algorithms coded correctly?)**
- Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements (are we building the right product) (Does it meet user requirements?)**

Organizing for Software Testing

Testing should aim at "breaking" the software

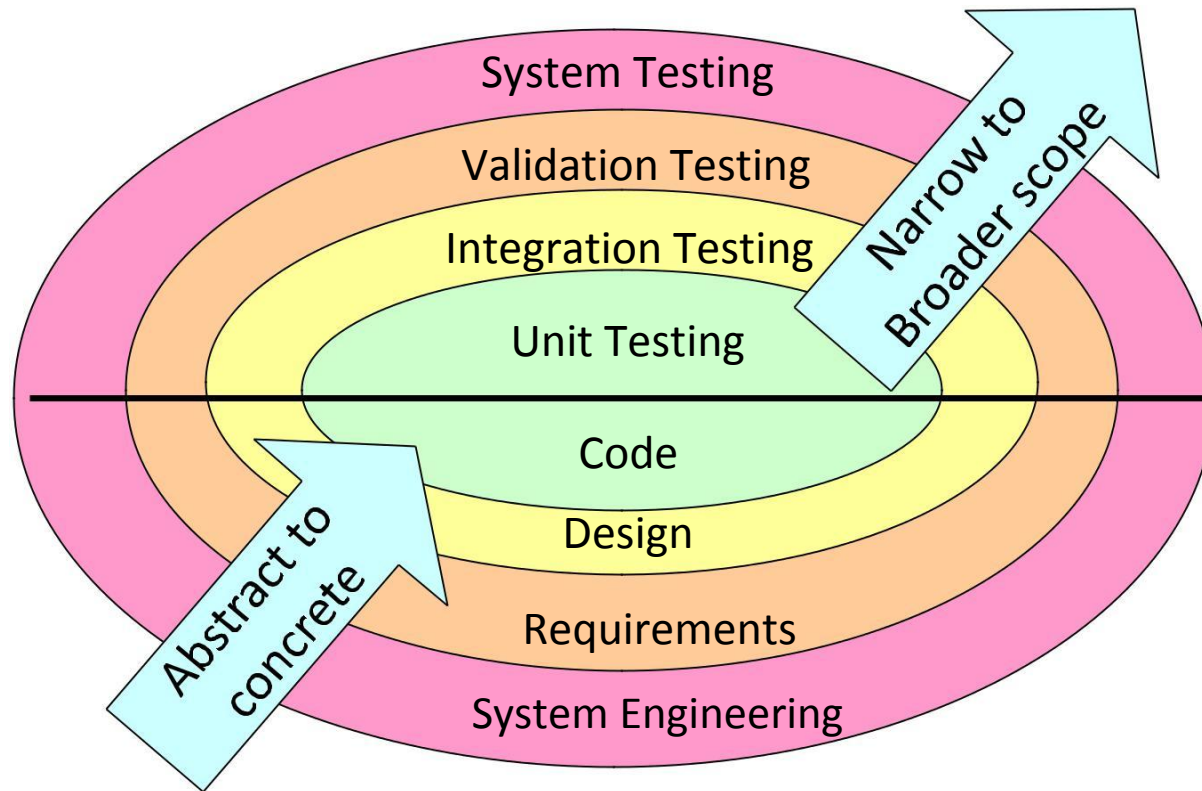
Common misconceptions

- The developer of software should do no testing at all
- The software should be given to a secret team of testers who will test it unmercifully
- The testers get involved with the project only when the testing steps are about to begin

Reality: Independent test group

- Removes the inherent problems associated with letting the builder test the software that has been built
- Removes the conflict of interest that may otherwise be present
- Works closely with the software developer during analysis and design to ensure that thorough testing occurs

Testing Strategy for Conventional Software



Testing Strategy for Conventional Software

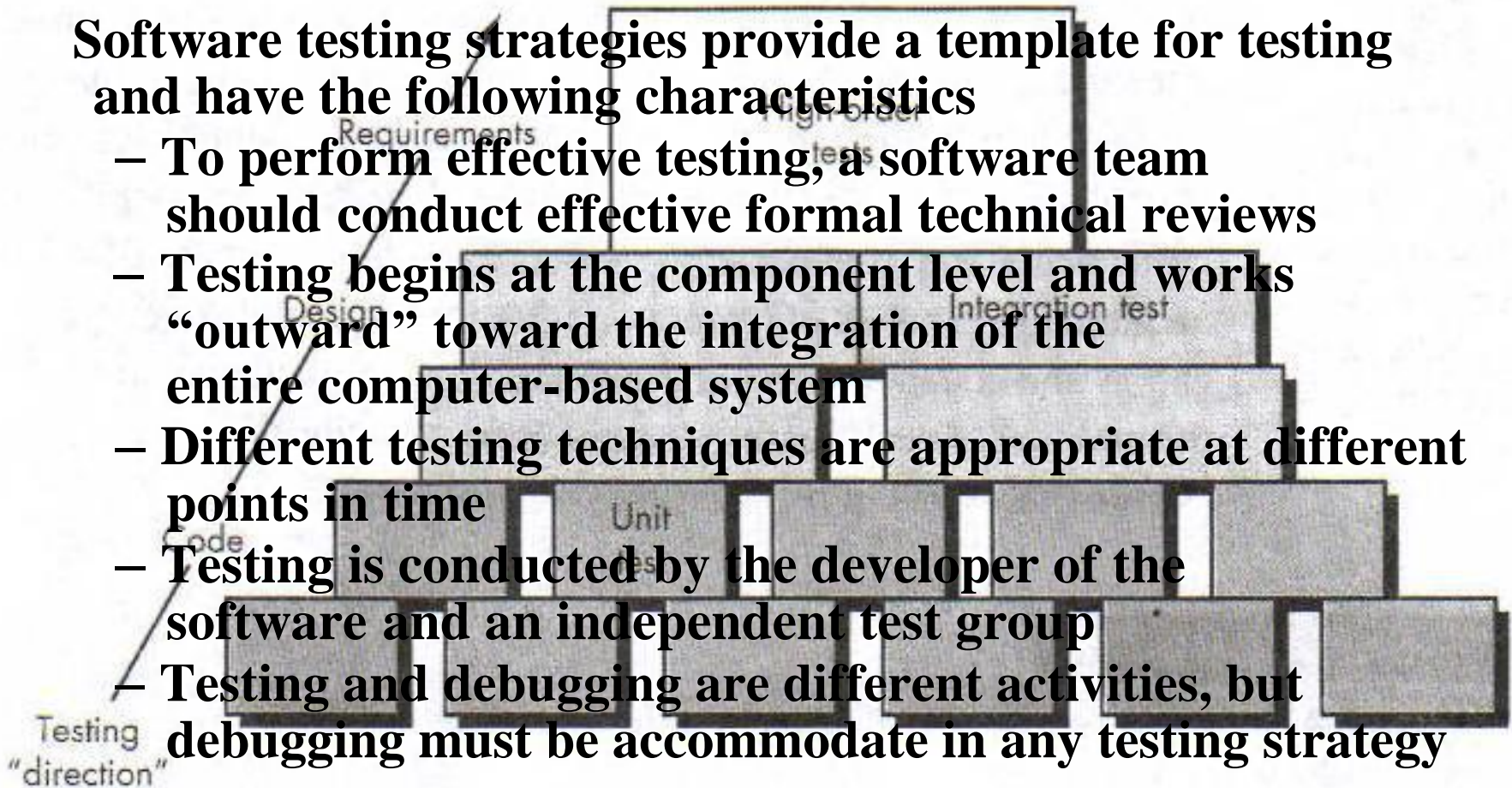
A strategy for software testing can be viewed in the context of the spiral.

- Unit testing begins at the vortex of the spiral and concentrates on each unit of the software as implemented in source code**
- Testing progresses by moving outward along the spiral to integration testing, where the focus is on design and the construction of the software architecture**
- Taking another turn outward on the spiral, we encounter validation testing, where requirements established as part of the software requirements analysis are validated against the software that has been constructed**
- Finally system testing where the software and other system elements are tested as a whole.**

Testing Strategy for Conventional Software

Software testing strategies provide a template for testing and have the following characteristics

- To perform effective testing, a software team should conduct effective formal technical reviews
- Testing begins at the component level and works “outward” toward the integration of the entire computer-based system
- Different testing techniques are appropriate at different points in time
- Testing is conducted by the developer of the software and an independent test group
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy



Testing Strategy for Conventional Software

Unit testing

- Concentrates on each component/function of the software as implemented in the source code
- Exercises specific paths in a component's control structure to ensure complete coverage and maximum error detection
- Components are then assembled and integrated

Integration testing

- Focuses on the design and construction of the software architecture
- Focuses on inputs and outputs, and how well the components fit together and work together

Validation testing

- Requirements are validated against the constructed software
- Provides final assurance that the software meets all functional, behavioral, and performance requirements

System testing

- The software and other system elements are tested as a whole
- Verifies that all system elements (software, hardware, people, databases) mesh properly and that overall system function and performance is achieved

Testing strategy for Object-Oriented Software

Must broaden testing to include detections of errors in analysis and design models

Unit testing loses some of its meaning and integration testing changes significantly

Use the same philosophy but different approach as in conventional software testing

Test "in the small" and then work out to testing "in the large"

- Testing in the small involves class attributes and operations; the main focus is on communication and collaboration within the class**
- Testing in the large involves a series of regression tests to uncover errors due to communication and collaboration among classes**

Finally, the system as a whole is tested to detect errors in fulfilling requirements

Criteria for Completion of Testing

When is Testing Complete?

- There is no definitive answer to this question
- Every time a user executes the software, the program is being tested
- Sadly, testing usually stops when a project is running out of time, money, or both
- One approach is to divide the test results into various severity levels

Then consider testing to be complete when certain levels of errors no longer occur or have been repaired or eliminated

Strategic Issues

- **Specify product requirements in a quantifiable manner long before testing commences. These should be specified in a way that is measurable so that testing results are unambiguous**
- **State testing objectives explicitly – The specific objective of testing should be stated in measurable terms. E.g. Test effectiveness, test coverage, mean time to failure, cost to find and fix defects etc**
- **Understand the users of the software and develop a profile for each user category.**
- **Develop a testing plan that emphasizes “ rapid cycle testing”**
- **Build “ robust” software that is designed to test itself**
- **Use effective formal technical reviews as a filter prior to testing**
- **Conduct formal technical reviews to assess the test strategy and test cases themselves**
- **Develop a continuous improvement approach for the testing process**

Test strategies for conventional software

- A testing strategy takes an incremental view of testing, beginning with the testing to individual program units, moving to tests designed to facilitate the integration of the units, and culminating with tests that exercise the constructed system**
- Classes of Tests that are generally conducted are**
 - Unit Testing**
 - Integration Testing**

Unit Testing

Focuses testing on the function or software module

Concentrates on the internal processing logic and data structures

Is simplified when a module is designed with high cohesion

- Reduces the number of test cases
- Allows errors to be more easily predicted and uncovered

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

Unit Testing

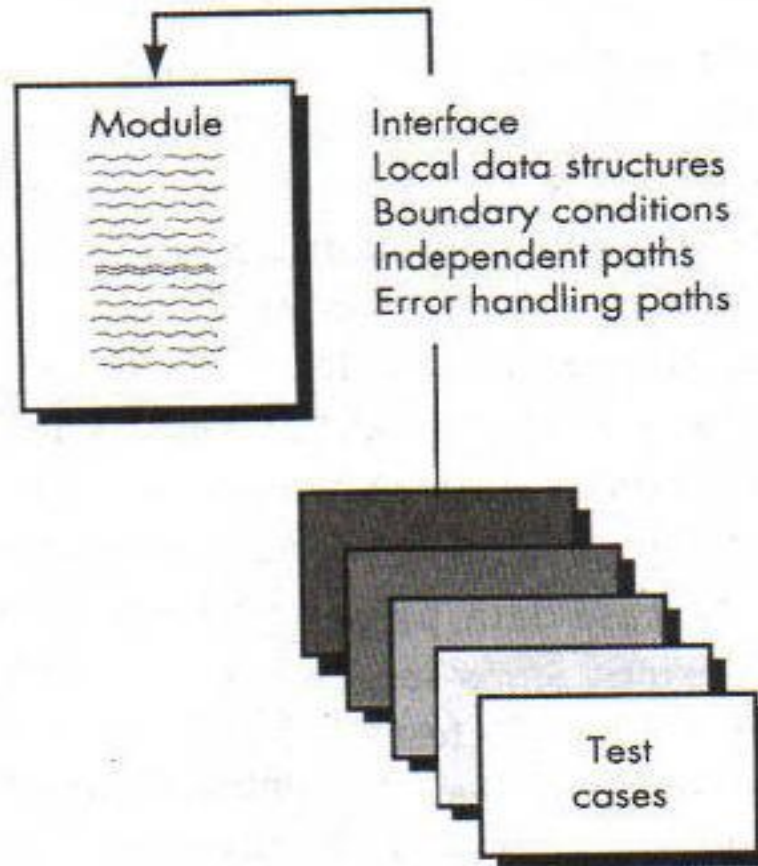
Targets for Unit Test Cases

- **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)

Unit Testing

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

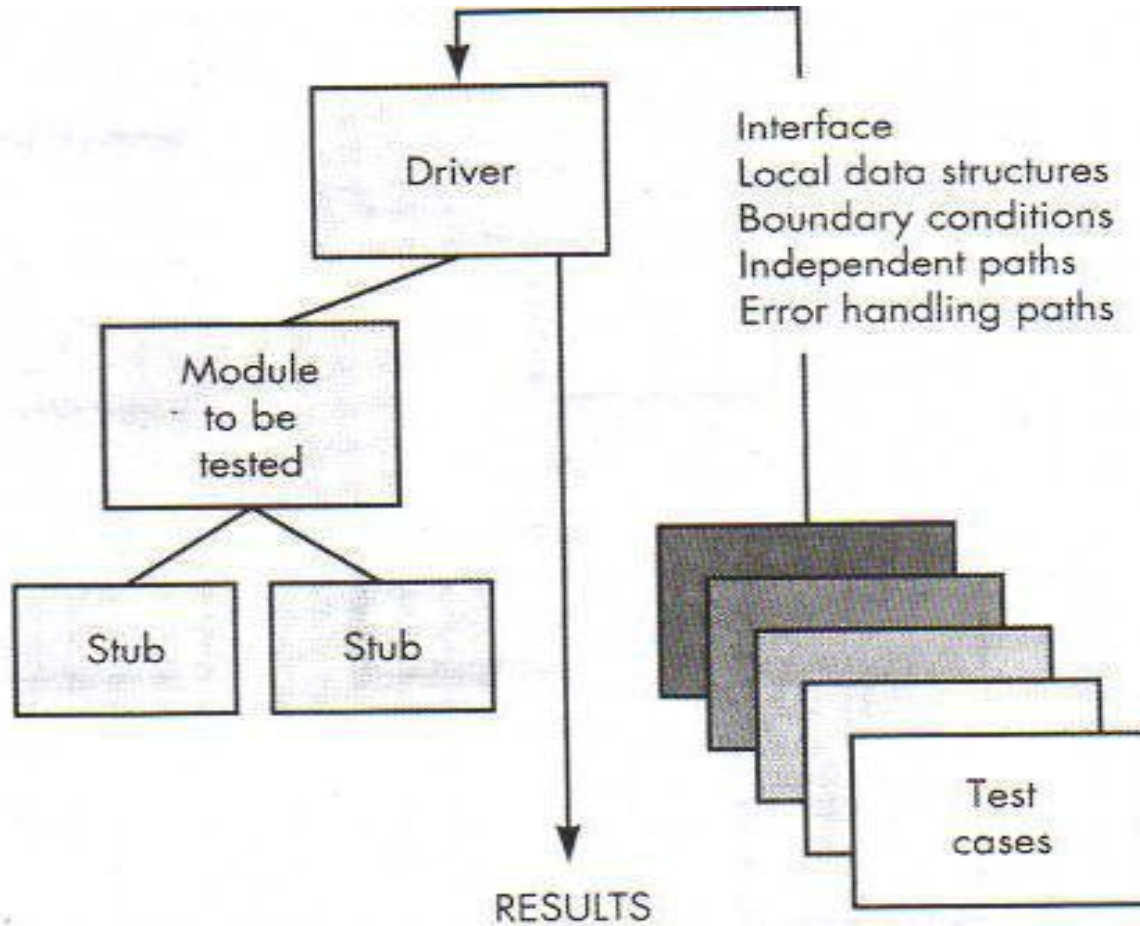
Stubs

- 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

Integration Testing

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

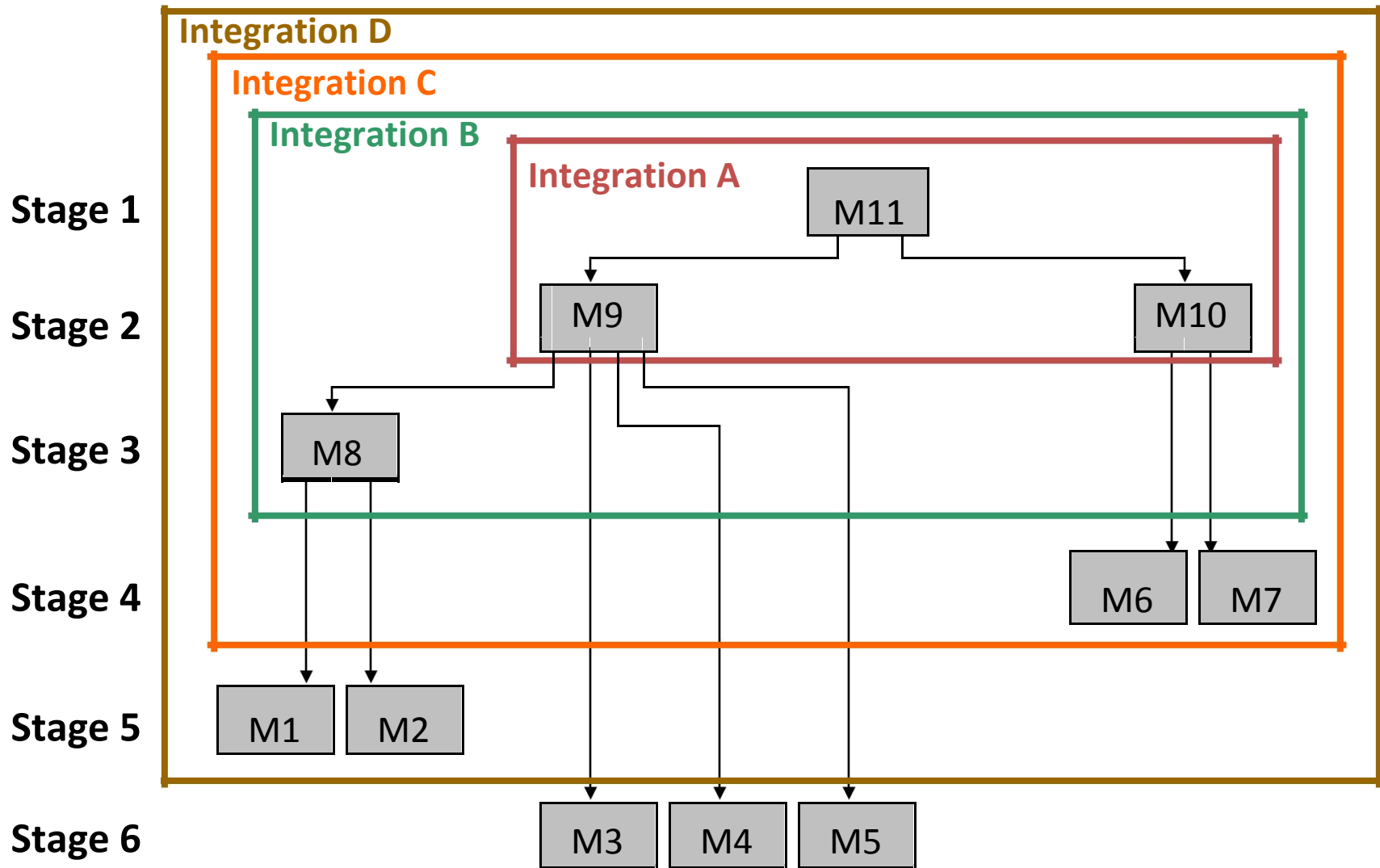
This approach verifies major control or decision points early in the test process

– 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 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

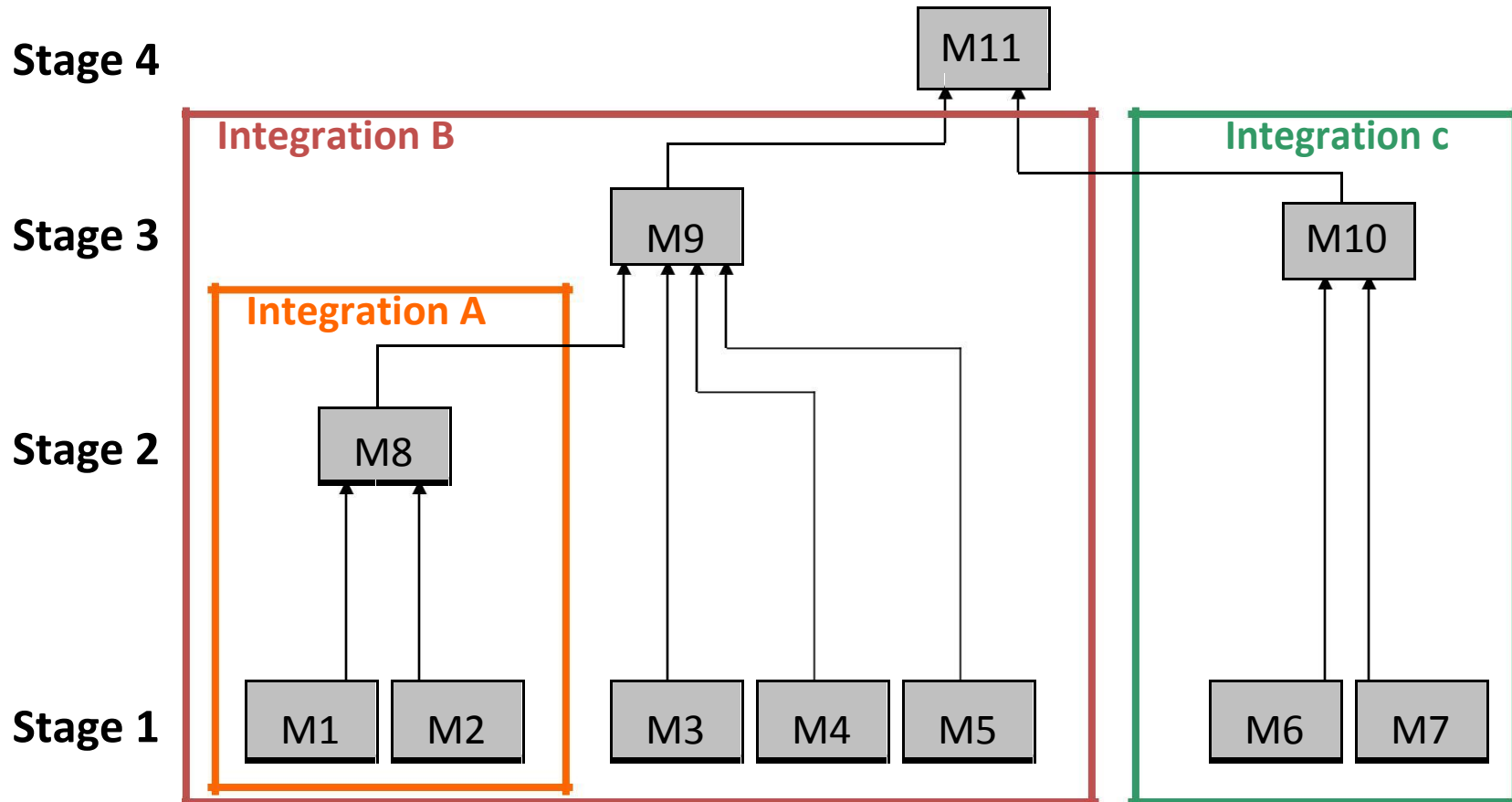
Need for stubs is eliminated

– 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

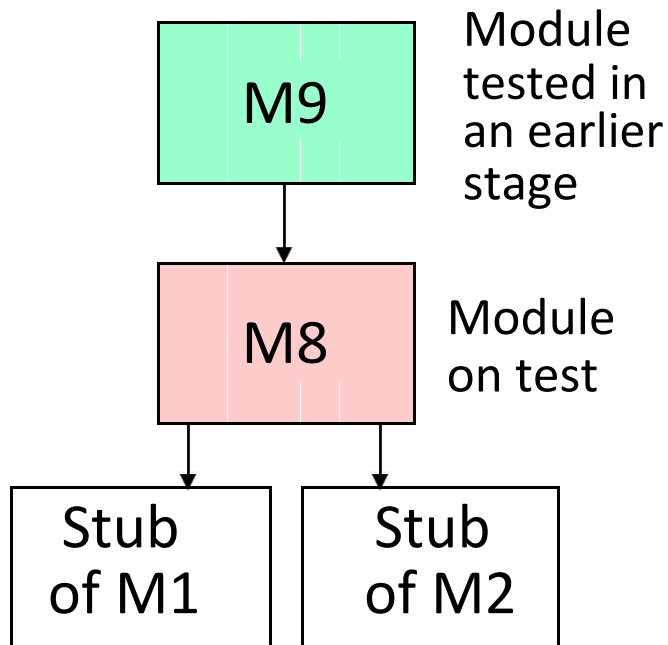
Drivers inherently do not contain the complete algorithms that will eventually use the services of the lower-level modules; consequently, testing may be incomplete or more testing may be needed later when the upper level modules are available

Bottom-up Integration

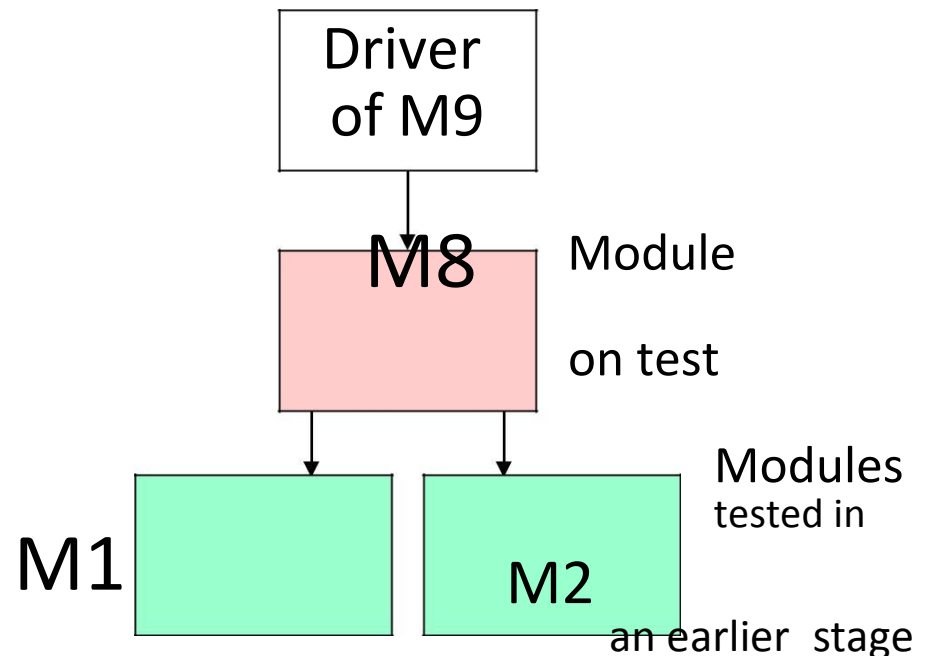


Use of stubs and drivers for incremental testing

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

Integration Testing

Strategic Options

- Selection of an integration strategy depends upon software characteristics and project schedule. Combined approach (sandwich testing) that uses top-down tests for upper levels of the program structure coupled with bottom-up tests for subordinate levels may be the best compromise

Integration test documentation

- Contains a test plan, a test procedure, is a work product of the software process, and becomes part of the software configuration
- The test plan describes the overall strategy for integration.
- A schedule for integration
- The detailed testing procedure that is required to accomplish the test plan
- A history of actual test results, problems, or peculiarities is recorded in a Test Report that can be appended to the Test Specification

Test Strategies for Object-Oriented Software

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

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

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

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

Test Strategies for Object-Oriented Software

Drivers can be used

- To test operations at the lowest level and for testing whole groups of classes
- To replace the user interface so that tests of system functionality can be conducted prior to implementation of the actual interface

Stubs can be used

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

Test Strategies for Object-Oriented Software

Two different object-oriented testing strategies

– Thread-based testing

Integrates the set of classes required to respond to one input or event for the system

Each thread is integrated and tested individually

Regression testing is applied to ensure that no side effects occur

– Use-based testing

First tests the independent classes that use very few, if any, server classes

Then the next layer of classes, called dependent classes, are integrated

This sequence of testing layer of dependent classes continues until the entire system is constructed

Test Strategies for Object-Oriented Software

– Integration Testing in the OO context

Drivers can be used to test operations at the lowest level and for the testing of whole group of classes.

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

Cluster of collaborating classes is exercised by designing test cases that attempt to uncover errors in the collaborations

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**

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

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

Validation Testing

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**

Validation Testing

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.

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

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.

Stress testing

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

Performance Testing

Performance testing is designed to test the run-time performance of software within the context of an integrated system.

Performance tests are often coupled with stress testing and usually requires both hardware and software instrumentation

Metrics for the Design Model

Architectural Design Metrics

- Structural complexity

$S(i) = f_{\text{out}}^2(i)$, where $f_{\text{out}}(i)$ is the “fan out” of module i

- Data complexity

$D(i) = v(i)/[f_{\text{out}}(i) + 1]$, where $v(i)$ is the number of input and output variables that are passed to and from module i

- System complexity

$$C(i) = S(i) + D(i)$$

- As each of these complexity values increases, the overall architectural complexity of the system also increases
- This leads to greater likelihood that the integration and testing effort will also increase

Metrics for the Design Model

Architectural Design Metrics

- **Shape complexity**

size = $n + a$, where n is the number of nodes and a is the number of arcs

Allows different program software architectures to be compared in a straightforward manner

- **Connectivity density (i.e., the arc-to-node ratio)**

$$r = a/n$$

May provide a simple indication of the coupling in the software architecture

Metrics for Object-Oriented Design

Size

- **Population:** a static count of all classes and methods
- **Volume:** a dynamic count of all instantiated objects at a given time
- **Length:** the depth of an inheritance tree

Coupling

- The number of collaborations between classes or the number of methods called between objects

Cohesion

- The cohesion of a class is the degree to which its set of properties is part of the problem or design domain

Primitiveness

- The degree to which a method in a class is atomic (i.e., the method cannot be constructed out of a sequence of other methods provided by the class)

Metrics for Object-Oriented Design

Specific Class-oriented Metrics

Weighted methods per class

- The normalized complexity of the methods in a class
- Indicates the amount of effort to implement and test a class

Depth of the inheritance tree

- The maximum length from the derived class (the node) to the base class (the root)
- Indicates the potential difficulties when attempting to predict the behavior of a class because of the number of inherited methods

Metrics for Object-Oriented Design

Specific Class-oriented Metrics

Number of children (i.e., subclasses)

- As the number of children of a class grows

Reuse increases

The abstraction represented by the parent class can be diluted by inappropriate children

The amount of testing required will increase

Coupling between object classes

- Measures the number of collaborations a class has with any other classes
- Higher coupling decreases the reusability of a class
- Higher coupling complicates modifications and testing
- Coupling should be kept as low as possible

Metrics for Object-Oriented Design

Specific Class-oriented Metrics

Response for a class

- This is the set of methods that can potentially be executed in a class in response to a public method call from outside the class
- As the response value increases, the effort required for testing also increases as does the overall design complexity of the class

Lack of cohesion in methods

- This measures the number of methods that access one or more of the same instance variables (i.e., attributes) of a class
- If no methods access the same attribute, then the measure is zero
- As the measure increases, methods become more coupled to one another via attributes, thereby increasing the complexity of the class design

Metrics for Maintenance

Software maturity index (SMI)

- Provides an indication of the stability of a software product based on changes that occur for each release

$$\text{SMI} = [M_T - (F_a + F_c + F_d)] / M_T$$

where

M_T = #modules in the current release

F_a = #modules in the current release that have been added

F_c = #modules in the current release that have been changed

F_d = #modules from the preceding release that were deleted in the current release

As the SMI (i.e., the fraction) approaches 1.0, the software product begins to stabilize

The average time to produce a release of a software product can be correlated with the SMI

Syllabus

Metrics for Process and Products : Software Measurement, Metrics for software quality.

Risk management : Reactive vs. Proactive Risk strategies, software risks, Risk identification, Risk projection, Risk refinement, RMMM, RMMM Plan.

Process and Project 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

Uses of Measurement

Can be applied to the software process with the intent of improving it on a continuous basis

Can be used throughout a software project to assist in estimation, quality control, productivity assessment, and project control

Can be used to help assess the quality of software work products and to assist in tactical decision making as a project proceeds

Software Metrics let you know when to laugh and when to cry

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 Domain

Process metrics are collected across all projects and over long periods of time

They are used for making strategic decisions

The intent is to provide a set of process indicators that lead to long-term software process improvement

The only way to know how/where to improve any process is to

- Measure specific attributes of the process
- Develop a set of meaningful metrics based on these attributes
- Use the metrics to provide indicators that will lead to a strategy for improvement

Metrics in the Process Domain

We measure the effectiveness of a process by deriving a set of metrics based on outcomes of the process such as

- Errors uncovered before release of the software**
- Defects delivered to and reported by the end users**
- Work products delivered**
- Human effort expended**
- Calendar time expended**
- Conformance to the schedule**
- Time and effort to complete each generic activity**

Etiquette of Process Metrics

Use common sense and organizational sensitivity when interpreting metrics data

Provide regular feedback to the individuals and teams who collect measures and metrics

Don't use metrics to evaluate individuals

Work with practitioners and teams to set clear goals and metrics that will be used to achieve them

Never use metrics to threaten individuals or teams

Metrics data that indicate a problem should not be considered “negative”

- Such data are merely an indicator for process improvement

Don't obsess on a single metric to the exclusion of other important metrics

Metrics in the Project Domain

Project metrics enable a software project manager to

- **Assess the status of an ongoing project**
- **Track potential risks**
- **Uncover problem areas before their status becomes critical**
- **Adjust work flow or tasks**
- **Evaluate the project team's ability to control quality of software work products**

Many of the same metrics are used in both the process and project domain

Project metrics are used for making tactical decisions

- **They are used to adapt project workflow and technical activities**

Use of Project Metrics

The first application of project metrics occurs during estimation

- **Metrics from past projects are used as a basis for estimating time and effort**

As a project proceeds, the amount of time and effort expended are compared to original estimates

As technical work commences, other project metrics become important

- **Production rates are measured (represented in terms of models created, review hours, function points, and delivered source lines of code)**
- **Error uncovered during each generic framework activity (i.e, communication, planning, modeling, construction, deployment) are measured**

Use of Project Metrics

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

Two categories of software measurement

- Direct measures of the

 - Software process (cost, effort, etc.)

 - Software product (lines of code produced, execution speed, defects reported over time, etc.)

- Indirect measures of the

 - Software product (functionality, quality, complexity, efficiency, reliability, maintainability, etc.)

Project metrics can be consolidated to create process metrics for an organization

Size-oriented Metrics

Derived by normalizing quality and/or productivity measures by considering the size of the software produced

Thousand lines of code (KLOC) are often chosen as the normalization value

Metrics include

- Errors per KLOC** - **Errors per person-month**
- Defects per KLOC** - **KLOC per person-month**
- Dollars per KLOC** - **Dollars per page of documentation**
- Pages of documentation per KLOC**

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**

Getting Started with Metrics

Establish a measurement collection process

a) What is the source of the data? b) Can tools be used to collect the data?

c) Who is responsible for collecting the data?

d) When are the data collected and recorded?

e) How are the data stored?

What validation mechanisms are used to ensure the data are correct?

Acquire appropriate tools to assist in collection and assessment

Establish a metrics database

Define appropriate feedback mechanisms on what the metrics indicate about your process so that the process and the metrics program can be improved

Establishing a Software Metrics Program

1. Identify your business goals.
2. Identify what you want to know or learn.
3. Identify your subgoals.
4. Identify the entities and attributes related to your subgoals.
5. Formalize your measurement goals.
6. Identify quantifiable questions and the related indicators that you will use to help you achieve your measurement goals.
7. Identify the data elements that you will collect to construct the indicators that help answer your questions.
8. Define the measures to be used, and make these definitions operational.
9. Identify the actions that you will take to implement the measures.
10. Prepare a plan for implementing the measures.

Definition of Risk

A risk is a potential problem – it might happen and it might not

Conceptual definition of risk

- Risk concerns future happenings
- Risk involves change in mind, opinion, actions, places, etc.
- Risk involves choice and the uncertainty that choice entails

Two characteristics of risk

- **Uncertainty** – the risk may or may not happen, that is, there are no 100% risks (those, instead, are called constraints)
- **Loss** – the risk becomes a reality and unwanted consequences or losses occur

Risk Categorization – Approach #1

Project risks

- They threaten the project plan
- If they become real, it is likely that the project schedule will slip and that costs will increase

Technical risks

- They threaten the quality and timeliness of the software to be produced
- If they become real, implementation may become difficult or impossible

Business risks

- They threaten the viability of the software to be built
- If they become real, they jeopardize the project or the product

Risk Categorization – Approach #1

Sub-categories of Business risks

- **Market risk** – building an excellent product or system that no one really wants
- **Strategic risk** – building a product that no longer fits into the overall business strategy for the company
- **Sales risk** – building a product that the sales force doesn't understand how to sell
- **Management risk** – losing the support of senior management due to a change in focus or a change in people
- **Budget risk** – losing budgetary or personnel commitment

Risk Categorization – Approach #2

Known risks

- Those risks that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date)

Predictable risks

- Those risks that are extrapolated from past project experience (e.g., past turnover)

Unpredictable risks

- Those risks that can and do occur, but are extremely difficult to identify in advance

Reactive vs. Proactive Risk Strategies

Reactive risk strategies

- "Don't worry, I'll think of something"
- The majority of software teams and managers rely on this approach
- Nothing is done about risks until something goes wrong
 - The team then flies into action in an attempt to correct the problem rapidly (fire fighting)
- Crisis management is the choice of management techniques

Proactive risk strategies

- Steps for risk management are followed
- Primary objective is to avoid risk and to have a contingency plan in place to handle unavoidable risks in a controlled and effective manner

Steps for Risk Management

Identify possible risks; recognize what can go wrong

Analyze each risk to estimate the probability that it will occur and the impact (i.e., damage) that it will do if it does occur

Rank the risks by probability and impact

Impact may be negligible, marginal, critical, and catastrophic

Develop a contingency plan to manage those risks having high probability and high impact

Risk Identification

Background

- Risk identification is a systematic attempt to specify threats to the project plan
- By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary
- Generic risks
 - Risks that are a potential threat to every software project

Risk Identification

Product-specific risks

- Risks that can be identified only by those a with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built
- This requires examination of the project plan and the statement of scope
- "What special characteristics of this product may threaten our project plan

Risk Item Checklist

Used as one way to identify risks

Focuses on known and predictable risks in specific subcategories

Can be organized in several ways

- A list of characteristics relevant to each risk subcategory
- Questionnaire that leads to an estimate on the impact of each risk
- A list containing a set of risk component and drivers and their probability of occurrence

Known and Predictable Risk Categories

Product size – Risks associated with overall size of the software to be built

Business impact – Risks associated with constraints imposed by management or the marketplace

Customer characteristics – Risks associated with sophistication of the customer and the developer's ability to communicate with the customer in a timely manner

Process definition – Risks associated with the degree to which the software process has been defined and is followed

Known and Predictable Risk Categories

Development environment – Risks associated with availability and quality of the tools to be used to build the project

Technology to be built – Risks associated with complexity of the system to be built and the "newness" of the technology in the system

Staff size and experience – Risks associated with overall technical and project experience of the software engineers who will do the work

Risk Control Strategies

An organization must choose one of four basic strategies to control risks:

Avoidance: applying safeguards that eliminate or reduce the remaining uncontrolled risks for the vulnerability

Transference: shifting the risk to other areas or to outside entities

Mitigation: reducing the impact should the vulnerability be exploited

Acceptance: understanding the consequences and accept the risk without control or mitigation

Risk Control Strategies

Avoidance

Avoidance is the risk control strategy that attempts to prevent the exploitation of the vulnerability.

Avoidance is accomplished through:

Application of policy

Application of training and education

Countering threats

Implementation of technical security controls and safeguards

Risk Control Strategies

Transference

Transference is the control approach that attempts to shift the risk to other assets, other processes, or other organizations.

- This may be accomplished by rethinking how services are offered, revising deployment models, outsourcing to other organizations, purchasing insurance, or by implementing service contracts with providers.**

Risk Control Strategies

Mitigation

Mitigation is the control approach that attempts to reduce, by means of planning and preparation, the damage caused by the exploitation of vulnerability.

This approach includes three types of plans:

The disaster recovery plan (DRP),

Incident response plan (IRP)

Business continuity plan (BCP).

Mitigation depends upon the ability to detect and respond to an attack as quickly as possible.

Assessing Risk Impact

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

- Its nature – This indicates the problems that are likely if the risk occurs
- Its scope – This combines the severity of the risk (how serious was it) with its overall distribution (how much was affected)
- Its timing – This considers when and for how long the impact will be felt

Assessing Risk Impact

The overall risk exposure formula is $RE = P \times C$

- P = the probability of occurrence for a risk
- C = the cost to the project should the risk actually occur

Example

- P = 80% probability that 18 of 60 software components will have to be developed
- C = Total cost of developing 18 components is \$25,000
- $RE = .80 \times \$25,000 = \$20,000$

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**

Risk Mitigation, Monitoring, and Management

Strategy for Reducing Staff Turnover

Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market)

Mitigate those causes that are under our control before the project starts

Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave

Risk Mitigation, Monitoring, and Management

Strategy for Reducing Staff Turnover

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

Risk Mitigation, Monitoring, and Management

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

Risk Mitigation, Monitoring, and Management

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

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**

The RMMM Plan

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

Seven Principles of Risk Management

Maintain a global perspective

- View software risks within the context of a system and the business problem that is intended to solve

Take a forward-looking view

- Think about risks that may arise in the future; establish contingency plans

Encourage open communication

- Encourage all stakeholders and users to point out risks at any time

Seven Principles of Risk Management

Integrate risk management

- Integrate the consideration of risk into the software process**

Emphasize a continuous process of risk management

- Modify identified risks as more becomes known and add new risks as better insight is achieved**

Develop a shared product vision

- A shared vision by all stakeholders facilitates better risk identification and assessment**

Encourage teamwork when managing risk

- Pool the skills and experience of all stakeholders when conducting risk management activities**

Summary of Risk Management

Whenever much is riding on a software project, common sense dictates risk analysis

- Yet, most project managers do it informally and superficially, if at all**

However, the time spent in risk management results in

- Less upheaval during the project**
- A greater ability to track and control a project**
- The confidence that comes with planning for problems before they occur**

Risk management can absorb a significant amount of the project planning effort...but the effort is worth it

Syllabus

Quality Management : Quality concepts, Software quality assurance, Software Reviews, Formal technical reviews, Statistical Software quality Assurance, Software reliability, The ISO 9000 quality standards.

Quality Concepts

Quality Management

- Also called software quality assurance (SQA)
- Serves as an umbrella activity that is applied throughout the software process
- Involves doing the software development correctly versus doing it over again
- Reduces the amount of rework, which results in lower costs and improved time to market

Quality Concepts

Quality Management

– Encompasses

A software quality assurance process

Specific quality assurance and quality control tasks (including formal technical reviews and a multi-tiered testing strategy)

Effective software engineering practices (methods and tools)

Control of all software work products and the changes made to them

A procedure to ensure compliance with software development standards

Measurement and reporting mechanisms

Quality

Two kinds of quality are sought out

- Quality of design

 - The characteristic that designers specify for an item

 - This encompasses requirements, specifications, and the design of the system

- Quality of conformance (i.e., implementation)

 - The degree to which the design specifications are followed during manufacturing

 - This focuses on how well the implementation follows the design and how well the resulting system meets its requirements

Quality

Quality also can be looked at in terms of user satisfaction

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

Quality Control

Involves a series of inspections, reviews, and tests used throughout the software process

Ensures that each work product meets the requirements placed on it

Includes a feedback loop to the process that created the work product

- This is essential in minimizing the errors produced

Combines measurement and feedback in order to adjust the process when product specifications are not met

Requires all work products to have defined, measurable specifications to which practitioners may compare to the output of each process

Quality Assurance Functions

Consists of a set of auditing and reporting functions that assess the effectiveness and completeness of quality control activities

Provides management personnel with data that provides insight into the quality of the products

Alerts management personnel to quality problems so that they can apply the necessary resources to resolve quality issues

Quality Assurance Functions

Consists of a set of auditing and reporting functions that assess the effectiveness and completeness of quality control activities

Provides management personnel with data that provides insight into the quality of the products

Alerts management personnel to quality problems so that they can apply the necessary resources to resolve quality issues

The Cost of Quality

Includes all costs incurred in the pursuit of quality or in performing quality-related activities

Is studied to

- **Provide a baseline for the current cost of quality**
- **Identify opportunities for reducing the cost of quality**
- **Provide a normalized basis of comparison (which is usually dollars)**

Involves various kinds of quality costs Increases dramatically as the activities progress from

- Prevention → Detection → Internal failure → External failure

"It takes less time to do a thing right than to explain why you did it wrong." Longfellow

Kinds of Quality Costs

Prevention costs

- Quality planning, formal technical reviews, test equipment, training

Appraisal costs

- Inspections, equipment calibration and maintenance, testing

Failure costs – subdivided into internal failure costs and external failure costs

- Internal failure costs

Incurred when an error is detected in a product prior to shipment
Include rework, repair, and failure mode analysis

- External failure costs

Involves defects found after the product has been shipped
Include complaint resolution, product return and replacement, help line support, and warranty work

Software Quality Assurance

Software Quality

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

This definition emphasizes three points

- Software requirements are the foundation from which quality is measured; lack of conformance to requirements is lack of quality
- Specified standards define a set of development criteria that guide the manner in which software is engineered; if the criteria are not followed, lack of quality will almost surely result
- A set of implicit requirements often goes unmentioned; if software fails to meet implicit requirements, software quality is suspect

Software Quality Assurance

Software quality is no longer the sole responsibility of the programmer

- It extends to software engineers, project managers, customers, salespeople, and the SQA group
- Software engineers apply solid technical methods and measures, conduct formal technical reviews, and perform well-planned software testing

Software Quality Assurance

The SQA Group

- Serves as the customer's in-house representative
- Assists the software team in achieving a high-quality product
- Views the software from the customer's point of view
 - Does the software adequately meet quality factors?
 - Has software development been conducted according to pre-established standards?
 - Have technical disciplines properly performed their roles as part of the SQA activity?
- Performs a set of activities that address quality assurance planning, oversight, record keeping, analysis, and reporting

SQA Activities

Prepares an SQA plan for a project

Participates in the development of the project's software process description

Reviews software engineering activities to verify compliance with the defined software process

Audits designated software work products to verify compliance with those defined as part of the software process

Ensures that deviations in software work and work products are documented and handled according to a documented procedure

Records any noncompliance and reports to senior management

Coordinates the control and management of change

Helps to collect and analyze software metrics

Software Reviews

Purpose of Reviews

- Serve as a filter for the software process
- Are applied at various points during the software process
- Uncover errors that can then be removed
- Purify the software analysis, design, coding, and testing activities
- Catch large classes of errors that escape the originator more than other practitioners
- Include the formal technical review (also called a walkthrough or inspection)
 - Acts as the most effective SQA filter
 - Conducted by software engineers for software engineers
 - Effectively uncovers errors and improves software quality
 - Has been shown to be up to 75% effective in uncovering design flaws (which constitute 50-65% of all errors in software)
- Require the software engineers to expend time and effort, and the organization to cover the costs

Formal Technical Review (FTR)

Objectives

- To uncover errors in function, logic, or implementation for any representation of the software
- To verify that the software under review meets its requirements
- To ensure that the software has been represented according to predefined standards
- To achieve software that is developed in a uniform manner
- To make projects more manageable

Serves as a training ground for junior software engineers to observe different approaches to software analysis, design, and construction

Promotes backup and continuity because a number of people become familiar with other parts of the software

May sometimes be a sample-driven review

- Project managers must quantify those work products that are the primary targets for formal technical reviews
- The sample of products that are reviewed must be representative of the products as a whole

Formal Technical Review (FTR)

The FTR Meeting

- Has the following constraints

From 3-5 people should be involved

Advance preparation (i.e., reading) should occur for each participant but should require no more than two hours a piece and involve only a small subset of components

The duration of the meeting should be less than two hours

- Focuses on a specific work product (a software requirements specification, a detailed design, a source code listing)

Formal Technical Review (FTR)

The FTR Meeting

- Activities before the meeting

The producer informs the project manager that a work product is complete and ready for review

The project manager contacts a review leader, who evaluates the product for readiness, generates copies of product materials, and distributes them to the reviewers for advance preparation

Each reviewer spends one to two hours reviewing the product and making notes before the actual review meeting

The review leader establishes an agenda for the review meeting and schedules the time and location

Formal Technical Review (FTR)

The FTR Meeting

- Activities during the meeting

The meeting is attended by the review leader, all reviewers, and the producer

One of the reviewers also serves as the recorder for all issues and decisions concerning the product

After a brief introduction by the review leader, the producer proceeds to "walk through" the work product while reviewers ask questions and raise issues

The recorder notes any valid problems or errors that are discovered; no time or effort is spent in this meeting to solve any of these problems or errors

Formal Technical Review (FTR)

The FTR Meeting

- **Activities at the conclusion of the meeting**

All attendees must decide whether to

- **Accept the product without further modification**
- **Reject the product due to severe errors (After these errors are corrected, another review will then occur)**
- **Accept the product provisionally (Minor errors need to be corrected but no additional review is required)**

All attendees then complete a sign-off in which they indicate that they took part in the review and that they concur with the findings

Formal Technical Review (FTR)

The FTR Meeting

- Activities following the meeting

The recorder produces a list of review issues that

- Identifies problem areas within the product
- Serves as an action item checklist to guide the producer in making corrections

The recorder includes the list in an FTR summary report

- This one to two-page report describes what was reviewed, who reviewed it, and what were the findings and conclusions

The review leader follows up on the findings to ensure that the producer makes the requested corrections