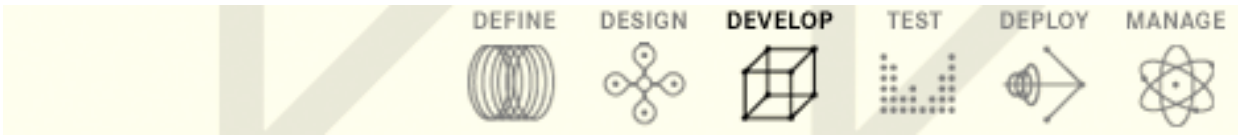


Software Engineering

Pramod Parajuli

Software Engineering

Objectives

This course aims to introduce students to problems in large-scale software production. This course should be associated with laboratory experiments to augment the concepts taught in the class.

Contents

1. **Introduction**
   - 1.1. Introduction of Software and Software Engineering
   - 1.2. History of Software Engineering
   - 1.3. Software Characteristics
2. **System Engineering**
   - 2.1. Introduction of system
   - 2.2. Properties of system
   - 2.3. System and its environment
   - 2.4. System Modeling
3. **Software Process**
   - 3.1. Software Process Model
   - 3.2. The Linear Sequential Model
   - 3.3. The Prototyping Model
   - 3.4. Rapid Application Model (RAD)
   - 3.5. Evolutionary Software Process Model
4. **Project Management**
   - 4.1. Management Activities
   - 4.2. Project Planning
   - 4.3. Project Scheduling
   - 4.4. Risk Management
5. **Requirement & Specification**
   - 5.1. Software Requirements
   - 5.2. Functional Requirements & Non Functional Requirements
6. **Requirement Engineering**
   - 6.1. Requirement Engineering Process
   - 6.2. Software Engineering Document
   - 6.3. Requirements Validation
7. **View Point Oriented Analysis**
8. **Architectural Design**
9. **Software Testing**
   - 9.1. Types of Software Testing
10. **Introduction of Software Reuse, Software Re-engineering and OOSE.**

Reference Book:

1. Software Engineering, Ian Sommerville, 6th Edition, Pearson.
2. Software Engineering, A Practitioner's Approach, Roger S. Pressman, 5th Edition, McGraw Hill.

Functions of a 'media player'

Example: 'chaos' and Software Engineering.

Managing chaos.

Design an ad-hoc 'media player'

Why Software Engineering required? Example: NASA & Russia meeting

Nature of programs

Programs are like Gods. They do everything you ask them to do.
- Brooks, Example. (Explain)

What is SE?
- use of existing approaches
- S/W Engineering --> focus on architectural design
- Engineering doesn't mean use available tools, rather explore the tools and find the best one

Why S/W crashes?
- mechanical, civil, electrical, -> matured
- SE just got the market in late 80s'

It is better to solve the right problems the wrong way than wrong problems the right way.

Expectations

Google.com

Why people use Google?
- For advanced research, information search

Guess what is the weirdest search in Google?

People look into refrigerator, search for food items and search on the Google to make up a recipe!

The Google professionals never thought about such search when it was established. The expectations of customers before S/W project starts and at the end of the S/W project also change.

☞ Requires - Expectation Management

Some of the topics that might not be covered in the class;
[Search at *http://www.wikipedia.org/*]

- Refactoring - in chaos also, some kind of organization do exist.
- Agile programming
- eXtreme programming

Recommended References

The Mythical Man-Month - Frederick P. Brooks, Jr.
Cathedral and Bazaar – Eric Raymond
Unix Philosophy – The Book

Freelancing
Brook's Law
Open Source and General Public License
GNU – GNU is Not Unix
Unified Modeling Language
Literate Programming
Knowledge Management
Business Intelligence

Agilealliance.org
Agilemanifesto.org
Wikipedia.org
Journal of object technology – http://www.jot.com
Roger S. Pressman and Associates - http://www.rspa.com
GNU Free Manuals - http://www.gnu.org/philosophy/free-doc.html
Eclipse IDE – http://www.eclipse.org
Rational Rose – http://www.rational.com
OpenSource Foundation – http://www.osef.org
IBM OpenSource - http://www-128.ibm.com/developerworks/opensource/
IBM DB2 - Information Management Software
Free UML Eclipse Tool - SDE for Eclipse Community Edition
- http://www.visual-paradigm.com/product/sde/ec/productinfosdeceec.jsp

Project

- Projects from DBMS, OOP or
- new project
  - Customer choice tracking for marketing using association rule
  - Decision tree for decision support system using Gini Index
  - Intelligent library information management system
  - Market prediction system using Markov chain for different brands of products
  - Document management system

- Study of at least two existing automation systems relative to the project
- Two Information Management functions as part of the project
- Refactoring will be applied on the projects

- Group division
  - Separate module for individuals
  - Detailed requirement analysis, design, implementation, testing, documentation, and presentation for the modules separately
  - Select one **team leader** – responsible for leadership, project breakdown, integration and module compliance testing, alpha testing, project monitoring and tracking, project progress report generation etc.

Schedule

Phase	Date
Project Title and Group Finalization, System study	June 23, 2006
Ad-hoc system design	June 30, 2006
Refactoring	July 14, 2006
Requirement Analysis	August 04, 2006
Design	September 08, 2006
Implementation, Testing, Presentation	October 02, 2006

Note:

- After every phase a complete documentation of the phase should be submitted.
- The final project report will be composed of letter of recommendation, abstract, all of the reports submitted in order, context diagram, acknowledgement, references and bibliography, and other formal structures.

The Product

The deliverables

- Programs
- Associated documents
  - ☑ System documentation
  - ☑ User documentation/guide
  - ☑ Manuals, websites
  - ☑ Advertisements
- Configuration data, data structures

Software

Two types

- Generic – developers get full control
- Bespoke (customized) – users get full control

System S/W	Application S/W	Eng./Scientific S/W
Embedded S/W	Product Line S/W	Web Application
Artificial Intelligence S/W	Ubiquitous Computing	Netsourcing
OpenSource S/W	Legacy S/W	

S/W Characteristics

- Software is developed, but not manufactured.
- Software doesn't wear out.
- Most of software are custom built.

S/W Evolution

The early years	The second era	The third era	The fourth era
Batch orientation Limited distribution Custom software	Multiuser Real-time Database Product software	Distributed systems Embedded intelligence Low cost h/w Consumer impact	Powerful desktops OO technologies Expert systems ANN Parallel computing Network computers
1950 – 1960	- 1970	- 1990	- 2000

S/W Engineering

Use of engineering discipline in S/W development

SE methods

- Structured analysis, function oriented method (1978)
- Object oriented (1980's)
- UML (1997)

Software Myths

Motivation - unlearn best practices (Modern management)

- Management myths (e.g. add more people behind schedule, why should we use new tools?)
- Customer myths (e.g. statement of objectives is sufficient for S/W development, change can be accommodated easily as S/W itself is flexible)
- Practitioner's myths (e.g. Once S/W dev. is finished, it's all done, S/W quality can't be assessed before S/W is running)

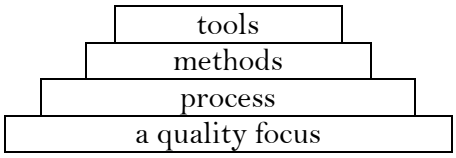
References: *Pressman – Chap 1, Sommerville – Chap 1* …

Software Engineering **Day 2**

The Process

- Engineering discipline
- All aspects of S/W production
  "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is the application of engineering to software. The study of approaches as in previously defined activities of software development."

SE as a layered technology



S/W Process

1. Software specification – definition phase (focus on **what**)
2. Software development – development phase (focus on **how**)
3. Software validation – customer validation
4. Software evolution – maintenance phase
   a. Corrective maintenance
   b. Adaptive maintenance
   c. Enhancement maintenance
   d. Preventive maintenance

Umbrella activities

All the phases are complemented by *umbrella activities*

- Software project tracking and control
- Software quality assurance
- Document preparation and production
- Measurement
- Formal technical reviews
- Software configuration management
- Reusability management
- Risk management

S/W process model

- Workflow model
- Dataflow or activity model
- Role/action model

S/W approaches

- Waterfall
- Iterative
- Component based software engineering (CBSE)

S/W cost

S/W development cost

Waterfall model	Spec	Design	Dev.	Integration and testing
Iterative dev.	Spec	Iterative Development		System testing
CBSE	Spec	Development	Integration and testing	
Long life time	System dev.	System Evolution		

CASE (Computer Aided Software Engineering)

- High level CASE tools
  - Analysis & design
  - Code generator
  - Report renerator
- Low level CASE tools
  - Debuggers
  - Program analyzers
  - Test case generators
  - Program editors

Attributes of good software

- Maintainability
- Dependability (reliable, secure, safety) – should not cause physical/economic damage to existing system
- Efficiency – resource optimization
- Usability – useful without undue effort to learn

Software crisis

- Rapid increase in computing power, complexity of problems, ..
- Correct, understandable, verifiable computer programs

"[The major cause of the software crisis is] that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."
Edsger Dijkstra: The Humble Programmer (PDF, 473Kb)

Definite measures for software crisis;

1. Projects running over budget
2. Projects running over time
3. Software was of low quality
4. Software often did not meet requirements (functional, non-functional)
5. Projects unmanageable, code difficult to maintain

Key challenges of SE

1. The legacy challenge
2. Heterogeneity challenge
3. Delivery challenge

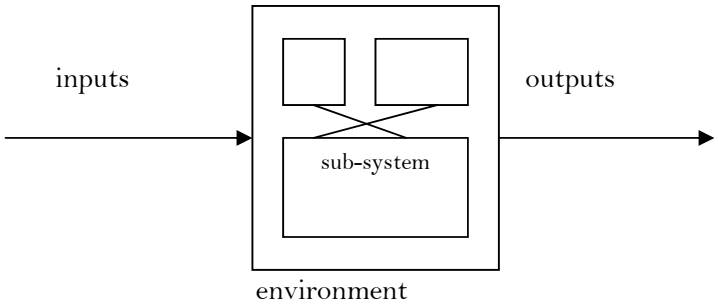
Professional & Ethical responsibilities

1. Confidentiality
2. Competence
3. Intellectual property rights
4. Computer Misuse

Refer to – Code of Ethics & Professional Practice, ACM/IEEE (Sommerville)

System Engineering

System
A system is combination of components that work together to achieve some objective.



Open System and Closed System

System Engineering
- Activity of specifying, designing, implementing, validating, deploying, & maintaining system
- Concerned with proper selection of product, service, and technology for system development

System properties
- Functional
- Non-functional (behavioral)
    - Reliability (H/W, S/W, Operator – how likely the operator of the system will make an error?)
    - Performance
    - Safety
    - Security

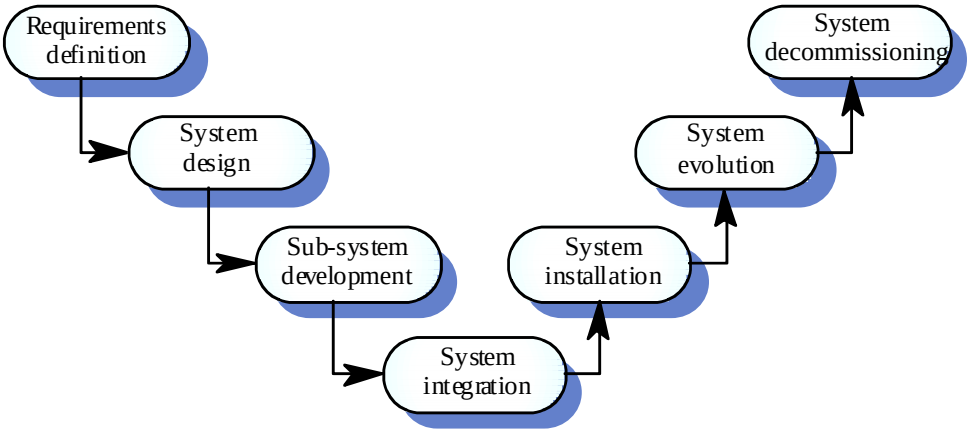
Example - Airbus
- Establishment
- Innovation – A380
- Leading
- Crisis

Changes affecting the system design
- Process changes
- Job changes
- Organizational changes

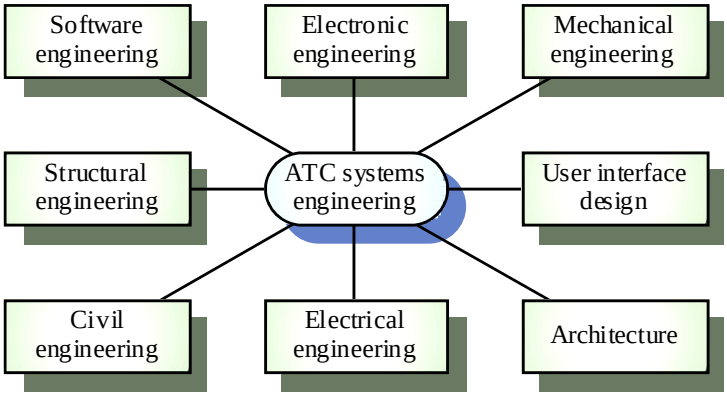
System Modeling

Generally used to represent the existing system to indicate that the developers understand the system correctly. [Refer to System Modeling – Sommerville, p. 26]

System Engineering Process



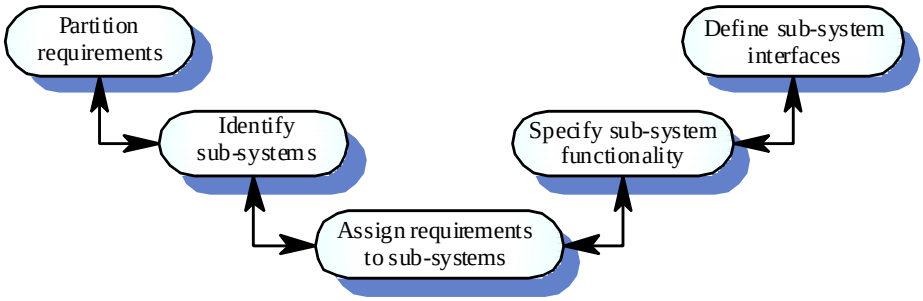
Inter-disciplinary involvement



System Requirements

1. Abstract functional requirements
2. System properties
3. Characteristics which the system must not exhibit

System Design



Sub-system development

- Modularization
- Module testing

System integration

- Synchronize
- Incremental integration

System installation

Problems

- Environmental assumptions may be incorrect
- May be human resistance to the introduction of a new system
- System may have to coexist with alternative systems for some time
- May be physical installation problems (e.g. cabling problems)
- Operator training has to be identified

System operation

- Will bring unforeseen requirements to light
- Users may use the system in a way which is not anticipated by system designers
- May reveal problems in the interaction with other systems
    - Physical problems of incompatibility
    - Data conversion problems
    - Increased operator error rate because of inconsistent interfaces

System evolution

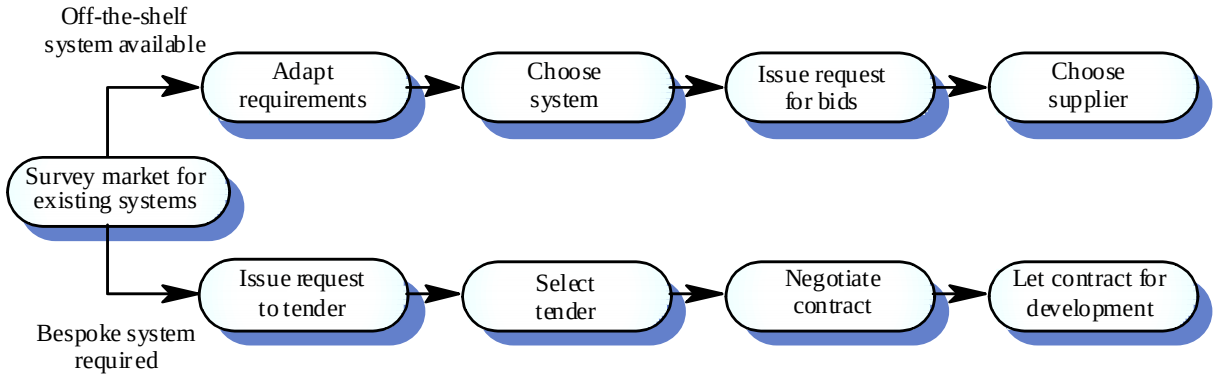
- Large systems have a long lifetime. They must evolve to meet changing requirements
- Evolution is inherently costly
    - Changes must be analysed from a technical and business perspective
    - Sub-systems interact so unanticipated problems can arise
    - There is rarely a rationale for original design decisions
    - System structure is corrupted as changes are made to it
- Existing systems which must be maintained are sometimes called legacy systems

System decommissioning

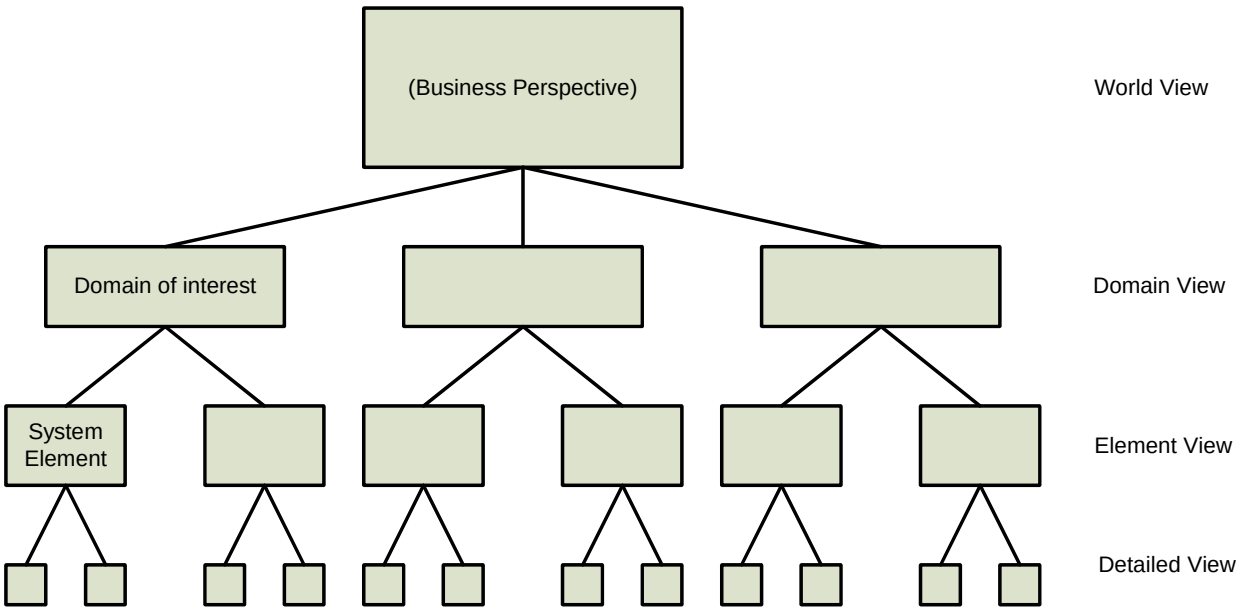
- Taking the system out of service after its useful lifetime
- May require removal of materials (e.g. dangerous chemicals) which pollute the environment
    - Should be planned for in the system design by encapsulation
- May require data to be restructured and converted to be used in some other system

System procurement

- Acquiring a system for an organization to meet some need
- Some system specification and architectural design is usually necessary before procurement
    - You need a specification to let a contract for system development
    - The specification may allow you to buy a commercial off-the-shelf (COTS) system. Almost always cheaper than developing a system from scratch



System Engineering Hierarchy



Factors to be considered for system modeling

- Assumption – possible permutations & variations
- Simplifications
- Limitations
- Constraints
- Preferences

System simulation

(S/W tools for System Simulation – Pressman (6) – p. 160)

Modeling methods

- Hatley-Pirbhai modeling (input → processing → output model, user interface, maintenance, self test)
- UML (http://www.rational.com/uml/index.jsp)

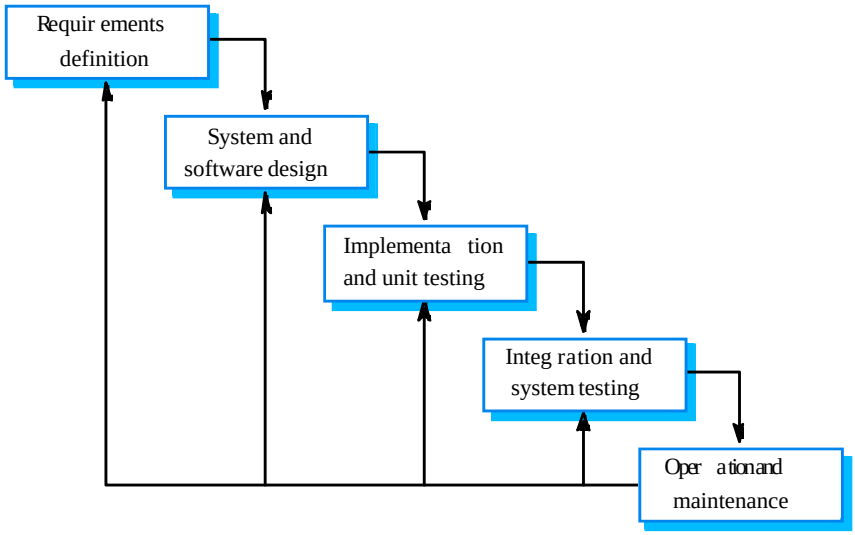
References: *Pressman (6) – Chap 6, Sommerville – Chap 2*

Software Process

Software Process Models

- Waterfall
- Evolutionary
- Formal systems
- Reuse based

Waterfall model

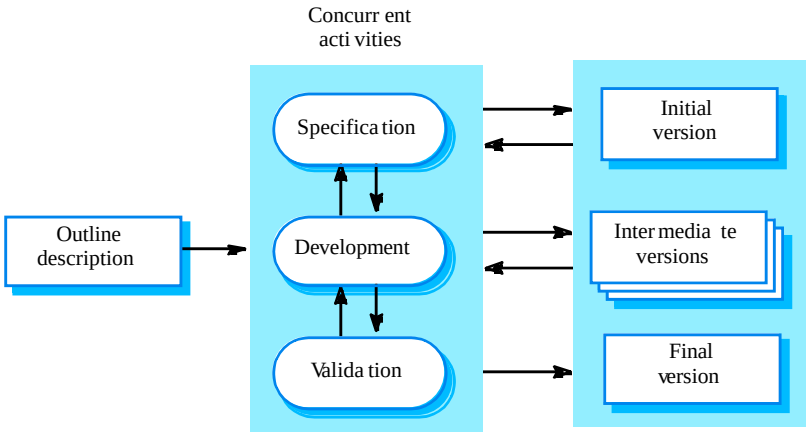


- The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway. One phase has to be complete before moving onto the next phase.
- Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.
- Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.

Evolutionary development model

Expose initial development to customer them refine through many versions.

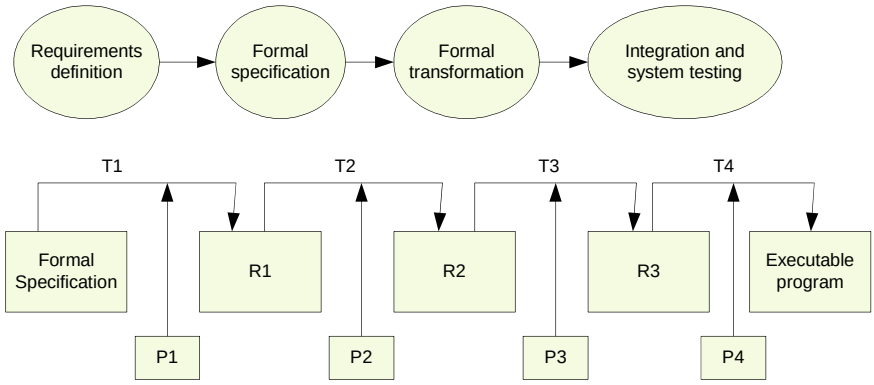
- Exploratory development
  Objective is to work with customers and to evolve a final system from an initial outline specification. Should start with well-understood requirements and add new features as proposed by the customer.
- Throw-away prototyping
  Objective is to understand the system requirements. Should start with poorly understood requirements to clarify what is really needed.



- 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.

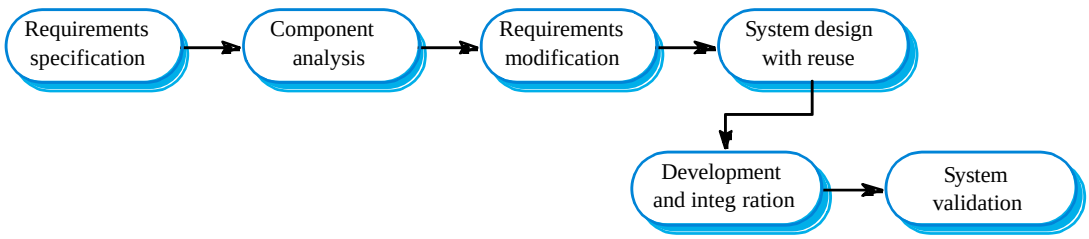
Formal Systems development

- Detail the requirements formally and express using mathematical notation
- Transform formal specifications into programs
- Used for development of critical systems



Reuse oriented development

Requires large base of reusable software components



References:, *Sommerville – Chap 3* [**Next**]: *Pressman (6) – Chap 3 Process Models*

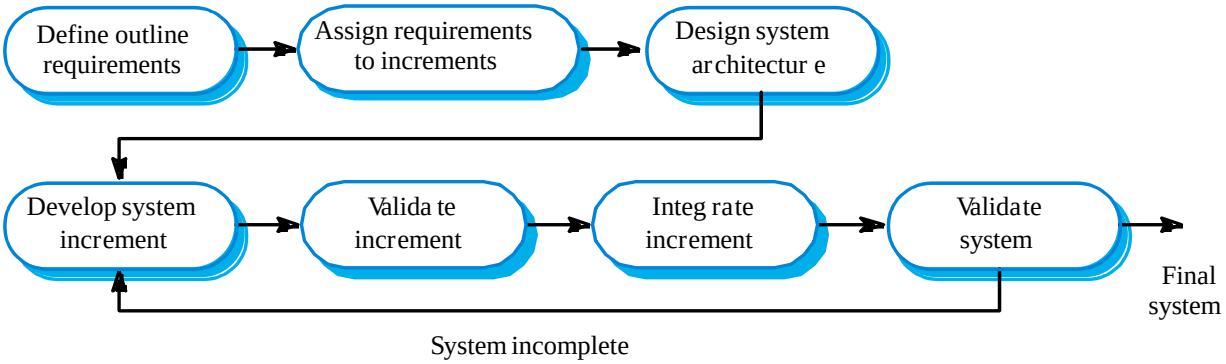
Prescriptive models
Prescribe a set of process elements – framework activities, S/W engineering actions, tasks, work products, quality assurance, and change control mechanisms.

Linear sequential models
Waterfall model

- Changes might cause confusion
- It is difficult for the customer to state all requirements explicitly
- Customer must have patience for delivery of product

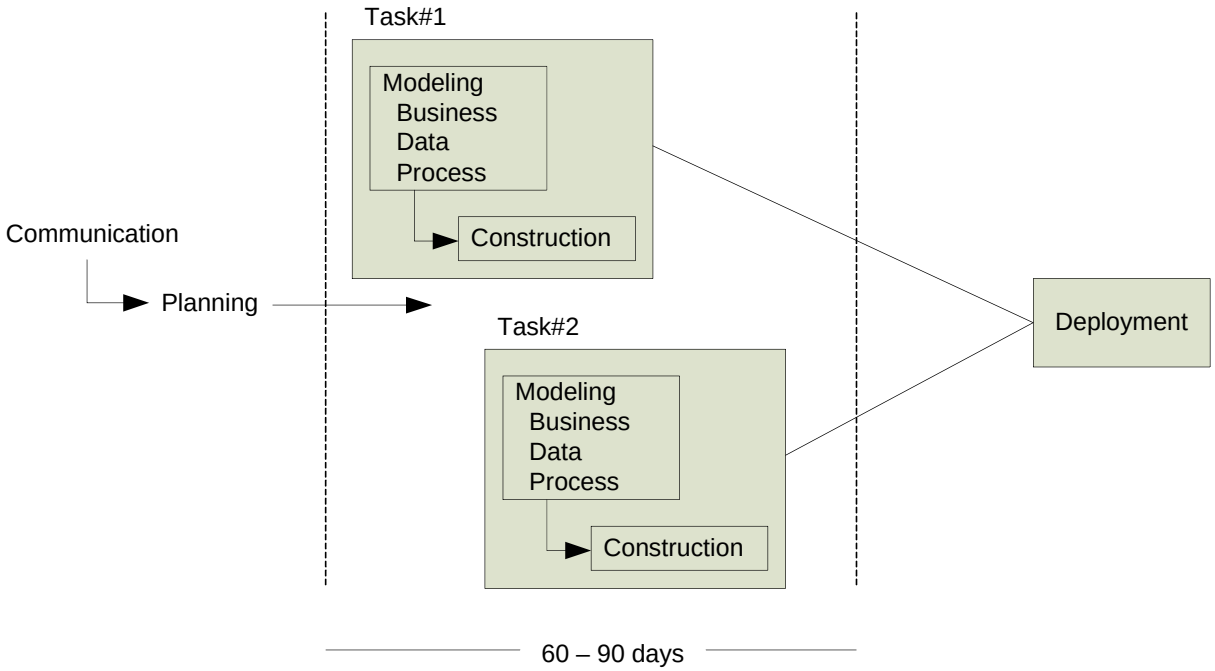
Incremental process models
Incremental model

- First increment – core product, addresses basic requirements
- Focuses on 'operational product' after each increment unlike prototyping
- Used for all 'agile' process



The RAD model

- Incremental S/W process emphasizing on short development cycle

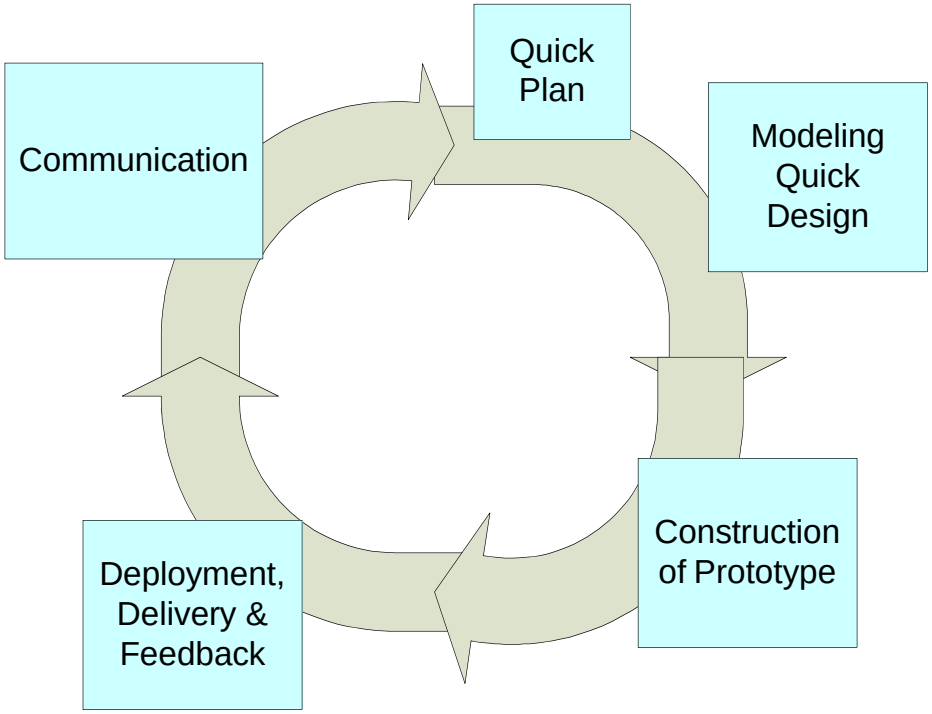


Disadvantages

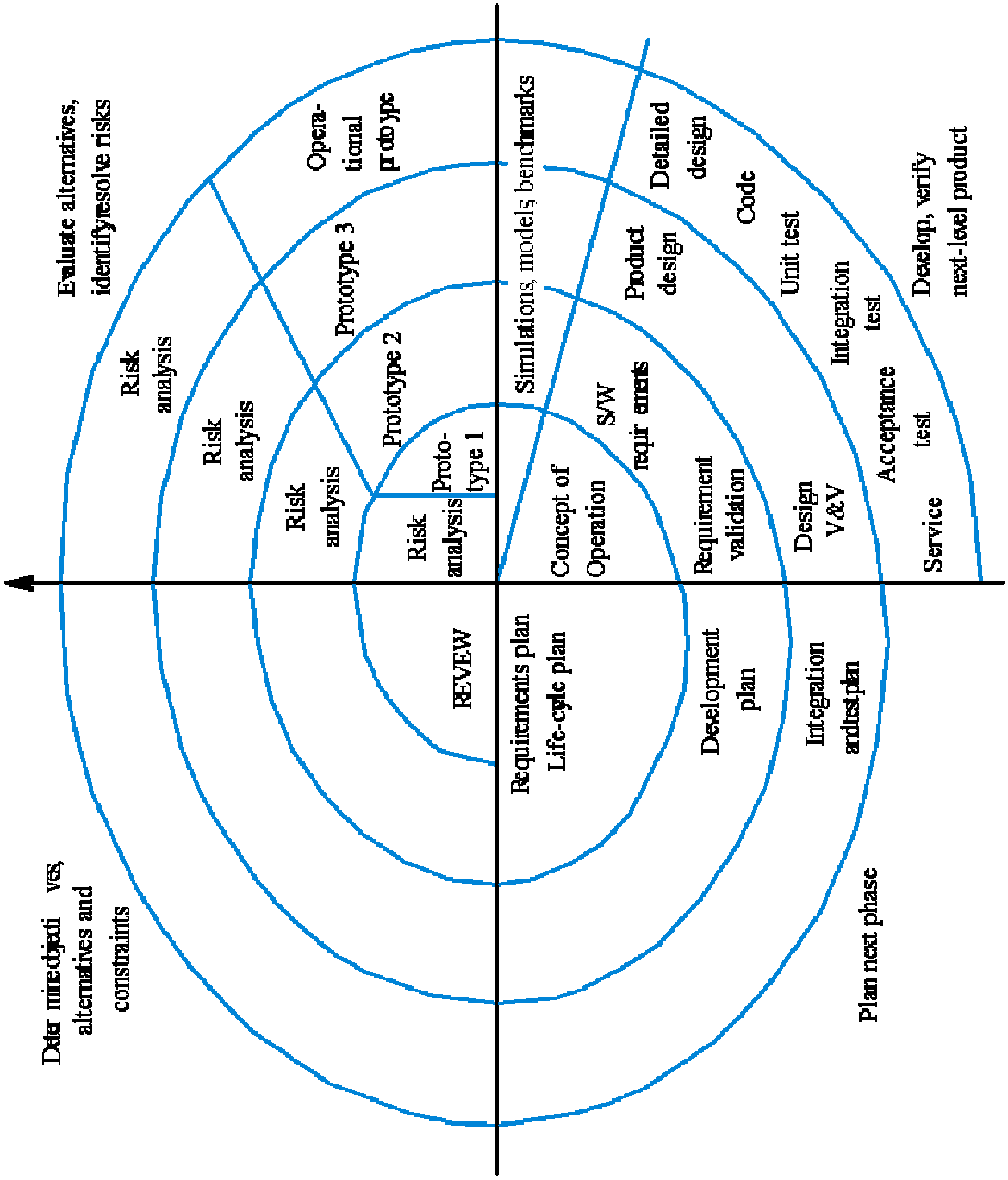
1. For large projects, requires large human resources
2. Developers & customers must be ready for rapid-fire activities
3. System must be modular
4. RAD may not be appropriate if technical risks are high

Evolutionary press model

Prototype model



Spiral Model



- Customer communication
- Planning (estimation, scheduling, risk analysis)
- Modeling (analysis, design)
- Construction (code, test)
- Deployment (delivery, feedback)

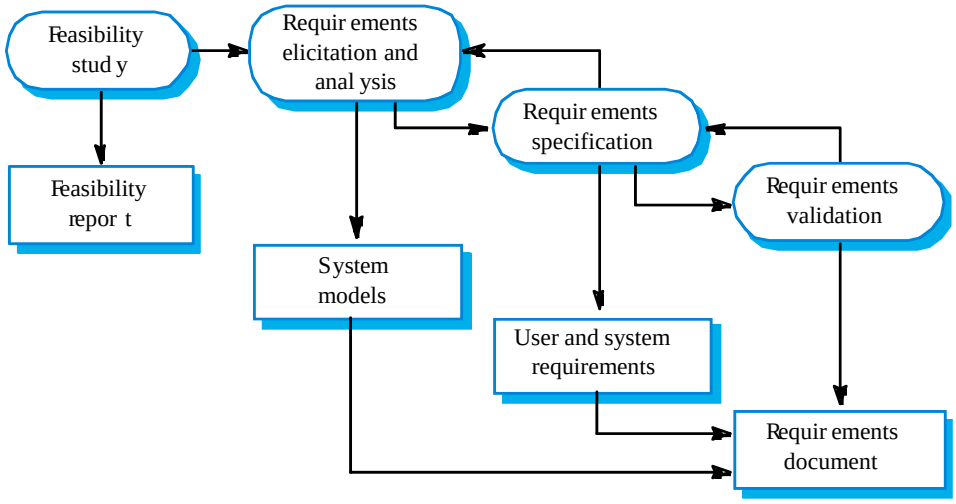
(Ref. http://www.sei.cmu.edu/cbs/spiral2000/)

Process activities

- Software specification
- Software design and implementation
- Software validation
- Software evolution

Software specification

- The process of establishing what services are required and the constraints on the system's operation and development.
- Requirements engineering process
  - Feasibility study;
  - Requirements elicitation and analysis;
  - Requirements specification;
  - Requirements validation.

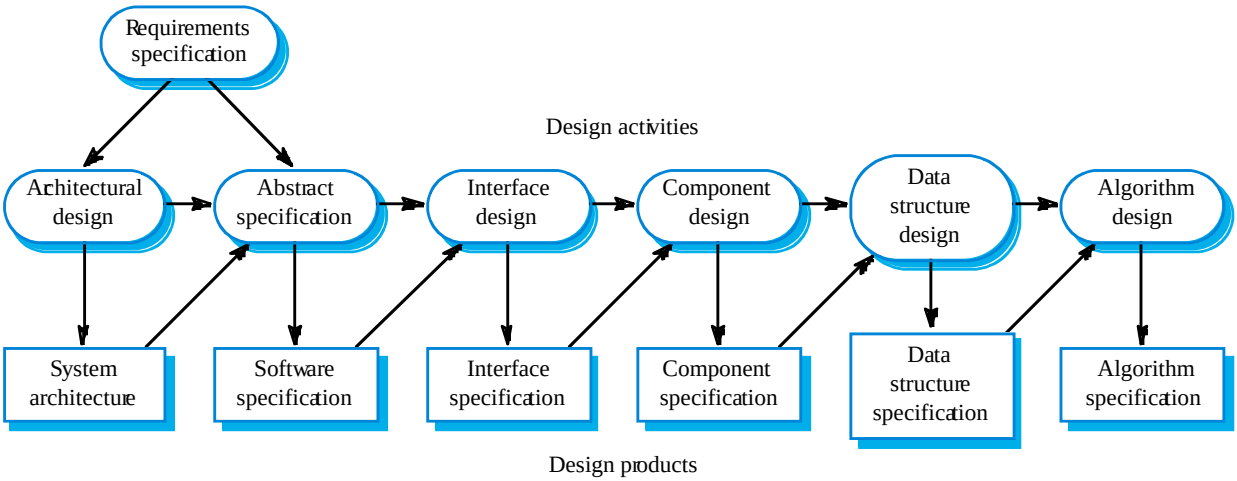


Software design and implementation

- The process of converting the system specification into an executable system.
- Software design
  - Design a software structure that realises the specification;
- Implementation
  - Translate this structure into an executable program;
- The activities of design and implementation are closely related and may be inter-leaved.

Design process activities

- Architectural design
- Abstract specification
- Interface design
- Component design
- Data structure design
- Algorithm design



Interface design

- User interface
- H/W interface
- Other system components interface

Structured methods

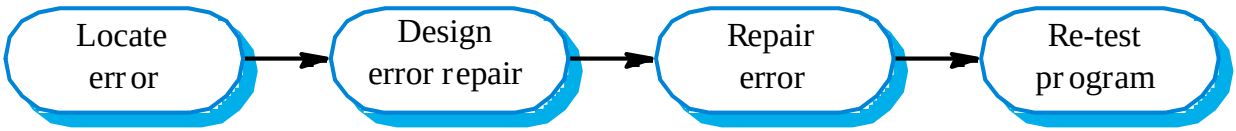
- Systematic approaches to developing a software design.
- The design is usually documented as a set of graphical models.
- Possible models
    1. Object model;
    2. Sequence model;
    3. State transition model;
    4. Structural model;
    5. Data-flow model.

Selection of design models

- A data flow model
- An E-R model to represent relationships
- Structural model for system components and their interactions
- OO methods for inherited model, object oriented partitioning

Programming and debugging

- Translating a design into a program and removing errors from that program.
- Programming is a personal activity - there is no generic programming process.
- Programmers carry out some program testing to discover faults in the program and remove these faults in the debugging process.

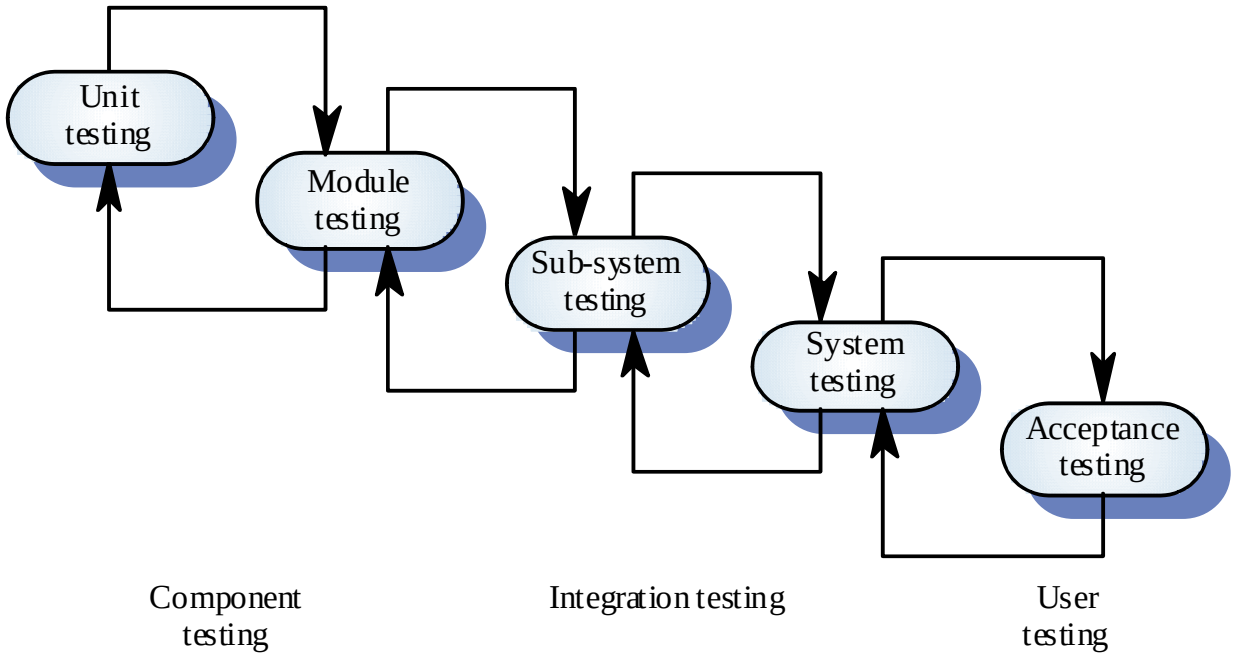


Software validation

- Verification and validation (V & V) is intended to show that a system conforms to its specification and meets the requirements of the system customer.
- Involves checking and review processes and system testing.

- System testing involves executing the system with test cases that are derived from the specification of the real data to be processed by the system.

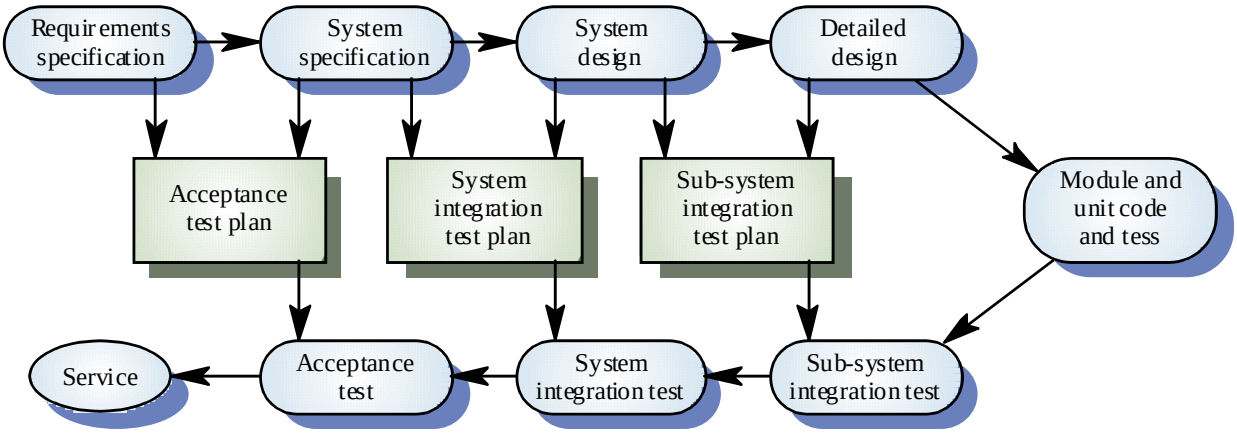
The testing process



Testing stages

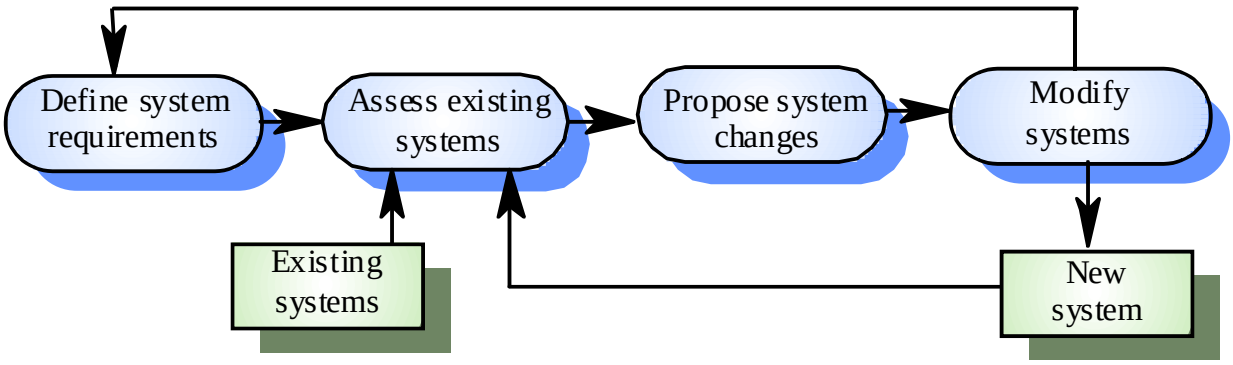
- Component or unit testing
  - Individual components are tested independently;
  - Components may be functions or objects or coherent groupings of these entities.
- System testing
  - Testing of the system as a whole. Testing of emergent properties is particularly important.
  - Alpha testing – system testing carried out by the test team within the organization
  - Beta testing – system testing carried out by a selected group of friendly customers
- Acceptance testing
  - Testing with customer data to check that the system meets the customer's needs.
  - Customer conducts the test and determines whether or not to accept the delivery of the system

Testing phases



Software evolution

- Software is inherently flexible and can change
- Requirements change through changing business circumstances



Automated process support (CASE)

- Computer-aided software engineering (CASE) is software to support software development and evolution processes
- Activity automation
  - Graphical editors for system model development
  - Data dictionary to manage design entities
  - Graphical UI builder for user interface construction
  - Debuggers to support program fault finding
  - Automated translators to generate new versions of a program

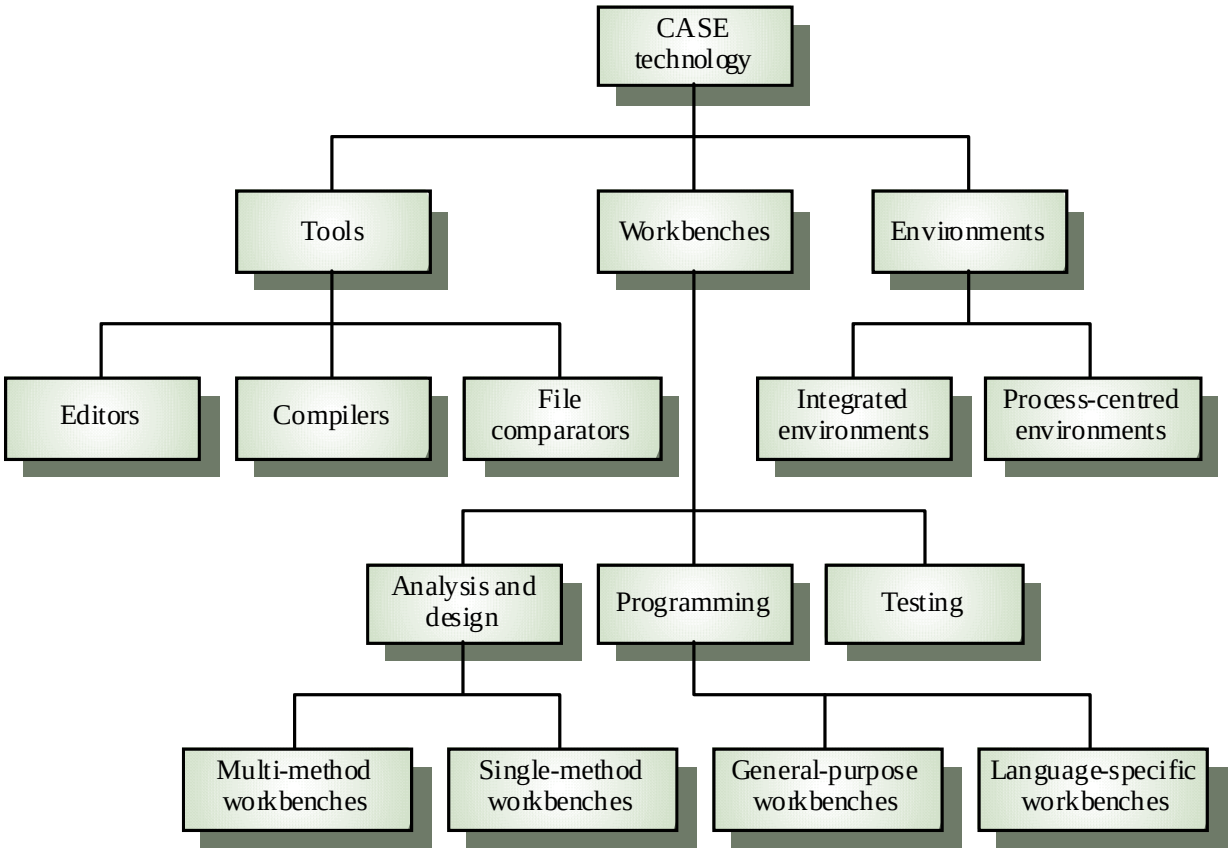
(Refer to 'CASE activities', Sommerville p. 64)

CASE classification

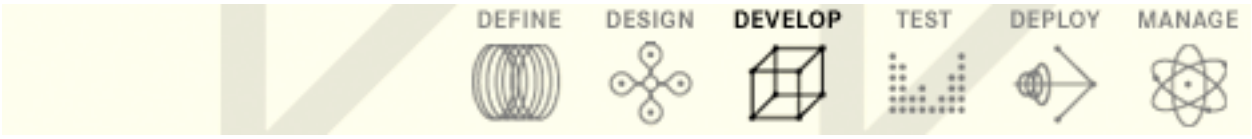
- Classification helps us understand the different types of CASE tools and their support for process activities
- Functional perspective
  - Tools are classified according to their specific function
- Process perspective
  - Tools are classified according to process activities that are supported
- Integration perspective
  - Tools are classified according to their organisation into integrated units

Functional tools

Tool type	Examples
Planning tools	PERT tools, estimation tools, spreadsheets
Editing tools	Text editors, diagram editors, word processors
Change management tools	Requirements traceability tools, change control systems
Configuration management tools	Version management systems, system building tools
Prototyping tools	Very high-level languages, user interface generators
Method-support tools	Design editors, data dictionaries, code generators
Language-processing tools	Compilers, interpreters
Program analysis tools	Cross reference generators, static analysers, dynamic analysers
Testing tools	Test data generators, file comparators
Debugging tools	Interactive debugging systems
Documentation tools	Page layout programs, image editors
Re-engineering tools	Cross-reference systems, program restructuring systems



References: *Pressman (6) – Chap 6, Sommerville – Chap 3*



Software Engineering

Project Management

Pramod Parajuli

Software Engineering **Day**

Project Management

Overview

- Project management involves the planning, monitoring, and control of people, process, and events that occur during software development.
- Everyone manages, but the scope of each person's management activities varies according his or her role in the project.
- Software needs to be managed because it is a complex undertaking with a long duration time.
- Managers must focus on the fours P's to be successful (people, product, process, and project).
- A project plan is a document that defines the four P's in such a way as to ensure a cost effective, high quality software product.
- The only way to be sure that a project plan worked correctly is by observing that a high quality product was delivered on time and under budget.

Management Spectrum

- People (recruiting, selection, performance management, training, compensation, career development, organization, work design, team/culture development)
- Product (product objectives, scope, alternative solutions, constraint tradeoffs)
- Process (framework activities populated with tasks, milestones, work products, and QA points)
- Project (planning, monitoring, controlling)

People

- Players (senior managers, technical managers, practitioners, customers, end-users)
- Team leadership model (motivation, organization, skills)
- Characteristics of effective project managers (problem solving, managerial identity, achievement, influence and team building)

Software Team Organization

- Democratic decentralized (rotating task coordinators and group consensus)
- Controlled decentralized (permanent leader, group problem solving, subgroup implementation of solutions)
- Controlled centralized (top level problem solving and internal coordination managed by team leader)

Factors Affecting Team Organization

- Difficulty of problem to be solved
- Size of resulting program
- Team lifetime
- Degree to which problem can be modularized
- Required quality and reliability of the system to be built
- Rigidity of the delivery date
- Degree of communication required for the project

Coordination and Communication Issues

- Formal, impersonal approaches (e.g. documents, milestones, memos)
- Formal interpersonal approaches (e.g. review meetings, inspections)
- Informal interpersonal approaches (e.g. information meetings, problem solving)
- Electronic communication (e.g. e-mail, bulletin boards, video conferencing)
- Interpersonal network (e.g. informal discussion with people other than project team members)

The Product

- Software scope (context, information objectives, function, performance)
- Problem decomposition (partitioning or problem elaboration - focus is on functionality to be delivered and the process used to deliver it)

The Process

- Process model chosen must be appropriate for the: customers and developers, characteristics of the product, and project development environment
- Project planning begins with melding the product and the process
- Each function to be engineered must pass though the set of framework activities defined for a software organization
- Work tasks may vary but the common process framework (CPF) is invariant (project size does not change the CPF)
- The job of the software engineer is to estimate the resources required to move each function though the framework activities to produce each work product
- Project decomposition begins when the project manager tries to determine how to accomplish each CPF activity

The Project

- Start on the right foot
- Maintain momentum
- Track progress
- Make smart decisions
- Conduct a postmortem analysis

W5HH Principle

- Why is the system being developed?
- What will be done by When?
- Who is responsible for a function?
- Where are they organizationally located?
- How will the job be done technically and managerially?
- How much of each resource is needed?

Critical Practices

- Formal risk management
- Empirical cost and schedule estimation
- Metric-based project management
- Earned value tracking
- Defect tracking against quality targets
- People-aware program management

Software Process and Project Metrics

Overview

- Software process and project metrics are quantitative measures that enable software engineers to gain insight into the efficiency of the software process and the projects conducted using the process framework. In software project management, we are primarily concerned with productivity and quality metrics. The four reasons for measuring software processes, products, and resources (to characterize, to evaluate, to predict, and to improve).

Measures, Metrics, and Indicators

- Measure - provides a quantitative indication of the size of some product or process attribute
- Measurement - is the act of obtaining a measure
- Metric - is a quantitative measure of the degree to which a system, component, or process possesses a given attribute

Process and Project Indicators

- Metrics should be collected so that process and product indicators can be ascertained
- Process indicators enable software project managers to: assess project status, track potential risks, detect problem area early, adjust workflow or tasks, and evaluate team ability to control product quality

Process Metrics

- Private process metrics (e.g. defect rates by individual or module) are known only to the individual or team concerned.
- Public process metrics enable organizations to make strategic changes to improve the software process.
- Metrics should not be used to evaluate the performance of individuals.
- Statistical software process improvement helps and organization to discover where they are strong and where are week.

Project Metrics

- Software project metrics are used by the software team to adapt project workflow and technical activities.
- Project metrics are used to avoid development schedule delays, to mitigate potential risks, and to assess product quality on an on-going basis.
- Every project should measure its inputs (resources), outputs (deliverables), and results (effectiveness of deliverables).

Software Measurement

- Direct measures of software engineering process include cost and effort.
- Direct measures of the product include lines of code (LOC), execution speed, memory size, defects per reporting time period.
- Indirect measures examine the quality of the software product itself (e.g. functionality, complexity, efficiency, reliability, maintainability).

Size-Oriented Metrics

- Derived by normalizing (dividing) any direct measure (e.g. defects or human effort) associated with the product or project by LOC.
- Size oriented metrics are widely used but their validity and applicability is widely debated.

Function-Oriented Metrics

- Function points are computed from direct measures of the information domain of a business software application and assessment of its complexity.
- Once computed function points are used like LOC to normalize measures for software productivity, quality, and other attributes.
- Feature points and 3D function points provide a means of extending the function point concept to allow its use with real-time and other engineering applications.
- The relationship of LOC and function points depends on the language used to implement the software.

Software Quality Metrics

- Factors assessing software quality come from three distinct points of view (product operation, product revision, product modification).
- Software quality factors requiring measures include correctness (defects per KLOC), maintainability (mean time to change), integrity (threat and security), and usability (easy to learn, easy to use, productivity increase, user attitude).
- Defect removal efficiency (DRE) is a measure of the filtering ability of the quality assurance and control activities as they are applied through out the process framework.

Integrating Metrics with Software Process

- Many software developers do not collect measures.
- Without measurement it is impossible to determine whether a process is improving or not.
- Baseline metrics data should be collected from a large, representative sampling of past software projects.
- Getting this historic project data is very difficult, if the previous developers did not collect data in an on-going manner.

Statistical Process Control

- It is important to determine whether the metrics collected are statistically valid and not the result of noise in the data.
- Control charts provide a means for determining whether changes in the metrics data are meaningful or not.
- Zone rules identify conditions that indicate out of control processes (expressed as distance from mean in standard deviation units).

Metrics for Small Organizations

- Most software organizations have fewer than 20 software engineers.
- Best advice is to choose simple metrics that provide value to the organization and don't require a lot of effort to collect.
- Even small groups can expect a significant return on the investment required to collect metrics, if this activity leads to process improvement.

Establishing a Software Metrics Program

- Identify business goal
- Identify what you want to know
- Identify subgoals
- Identify subgoal entities and attributes
- Formalize measurement goals
- Identify quantifiable questions and indicators related to subgoals
- Identify data elements needed to be collected to construct the indicators
- Define measures to be used and create operational definitions for them
- Identify actions needed to implement the measures
- Prepare a plan to implement the measures

References: *Pressman - Chap. 22, Sommerville – Chap. 4* [**Next**] …

Software Project Planning

Overview

- Software planning involves estimating how much time, effort, money, and resources will be required to build a specific software system. After the project scope is determined and the problem is decomposed into smaller problems, software managers use historical project data (as well as personal experience and intuition) to determine estimates for each. The final estimates are typically adjusted by taking project complexity and risk into account. The resulting work product is called a project management plan.

Estimation Reliability Factors

- Project complexity
- Project size
- Degree of structural uncertainty (degree to which requirements have solidified, the ease with which functions can be compartmentalized, and the hierarchical nature of the information processed)
- Availability of historical information

Project Planning Objectives

- To provide a framework that enables software manager to make a reasonable estimate of resources, cost, and schedule.
- Project outcomes should be bounded by 'best case' and 'worst case' scenarios.
- Estimates should be updated as the project progresses.

Software Scope

- Describes the data to be processed and produced, control parameters, function, performance, constraints, external interfaces, and reliability.
- Often functions described in the software scope statement are refined to allow for better estimates of cost and schedule.

Customer Communication and Scope

- Determine the customer's overall goals for the proposed system and any expected benefits.
- Determine the customer's perceptions concerning the nature if a good solution to the problem.
- Evaluate the effectiveness of the customer meeting.

Feasibility

- Technical feasibility is not a good enough reason to build a product.
- The product must meet the customer's needs and not be available as an off-the-shelf purchase.

Estimation of Resources

- Human Resources (number of people required and skills needed to complete the development project)

- Reusable Software Resources (off-the-shelf components, full-experience components, partial-experience components, new components)
- Development Environment (hardware and software required to be accessible by software team during the development process)

Software Project Estimation Options

- Delay estimation until late in the project.
- Base estimates on similar projects already completed.
- Use simple decomposition techniques to estimate project cost and effort.
- Use empirical models for software cost and effort estimation.
- Automated tools may assist with project decomposition and estimation.

Decomposition Techniques

- Software sizing (fuzzy logic, function point, standard component, change)
- Problem-based estimation (using LOC decomposition focuses on software functions, using FP decomposition focuses on information domain characteristics)
- Process-based estimation (decomposition based on tasks required to complete the software process framework)

Empirical Estimation Models

- Typically derived from regression analysis of historical software project data with estimated person-months as the dependent variable and KLOC or FP as independent variables.
- Constructive Cost Model (COCOMO) is an example of a static estimation model.
- The Software Equation is an example of a dynamic estimation model.

Make-Buy Decision

- It may be more cost effective to acquire a piece of software rather than develop it.
- Decision tree analysis provides a systematic way to sort through the make-buy decision.
- As a rule outsourcing software development requires more skillful management than does in-house development of the same product.

Automated Estimation Tool Capabilities

- Sizing of project deliverables
- Selecting project activities
- Predicting staffing levels
- Predicting software effort
- Predicting software cost
- Predicting software schedule

References: *Pressman - Chap. 23, Sommerville – Chap. 4* [**Next**] …

Software Metric

Software metric let you know when to laugh and when to cry.
- Tom Gilb

S/W Measurement

- Size oriented metric
  - LOC
  - Effort
  - $
  - Documentation
  - Errors
  - People

- Function oriented metric
  - Define function categories and assign weights
  - For particular function/module find total function points
  - Ref. to 'Function Point' document
  - Ref. to LOC/FP (Pressman 6, p. 657)

- Object oriented metric
  - No. of scenario
  - No. of key classes
  - No. of support classes
  - No. of subsystems

Measuring Quality

- Correctness
- Maintainability
- Integrity
- Usability

Estimation

- S/W scope and feasibility
- Resources
- Human Resources
- Reusable s/w resource
- Environmental resources

Sample – LOC based estimation example

Function	**Estimated LOC**
User Interface and control facilities (UICF)	2,300
Two-dimensional geometric analysis (2DGA)	5,300
Three-dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (CGDF)	4,950
Peripheral control function (PCF)	2,100
Design analysis modules (DAM)	8,400
Estimated lines of code	**33,200**

Sample – Function point estimation example

Information domain	Simple	Avg.	Complex	Count	FP count	
No. of external inputs	20	**24**	30	4	96	EI
No. of external outputs	**12**	15	22	5	60	EO
No. of external inquiries	**16**	22	28	10	160	EQ
No. of internal logical files	4	**4**	5	3	12	ILF
No. of external interface files	2	2	**3**	2	6	ELF
Count total					334	

Use of value using general system characteristics

Factor	Value
Backup and recovery	4
Data communication	2
Distributed processing	0
Performance critical	4
Existing operating environment	3
Online data entry	4
Input transaction over multiple screens	5
ILFs updated online	3
Information domain values complex	5
Internal processing complex	5
code designed for reuse	4
Conversion/installation in design	3
Multiple installations	5
Application designed for change	5

Estimated no. of FP is derived as;

$$FP_{estimated} = count\_total \times [0.65 + 0.01 \times \sum (Fi)]$$

Estimation techniques

- Empirical
  - Expert judgment
  - Delphi cost
- Heuristic
- Analytical

Good S/W Engineer

- Exposure to systematic techniques
- Good technical knowledge of the project areas (domain knowledge)
- Good programming abilities
- Good communication skills (oral, written, interpersonal)
- High motivation
- Sound knowledge of fundamentals of computer science
- Intelligence
- Ability to work in team
- Discipline

References: *Pressman - Chap. 23, Sommerville – Chap. 4* [**Next**] …

Risk Management

'If you don't actively attack the risks, they will actively attack you.'
- Tom Glib

- Risk management is concerned with identifying risks and drawing up plans to minimise their effect on a project.
- A risk is a probability that some adverse circumstance will occur.
  - Project risks affect schedule or resources
  - Product risks affect the quality or performance of the software being developed
  - Business risks affect the organisation developing or procuring the software

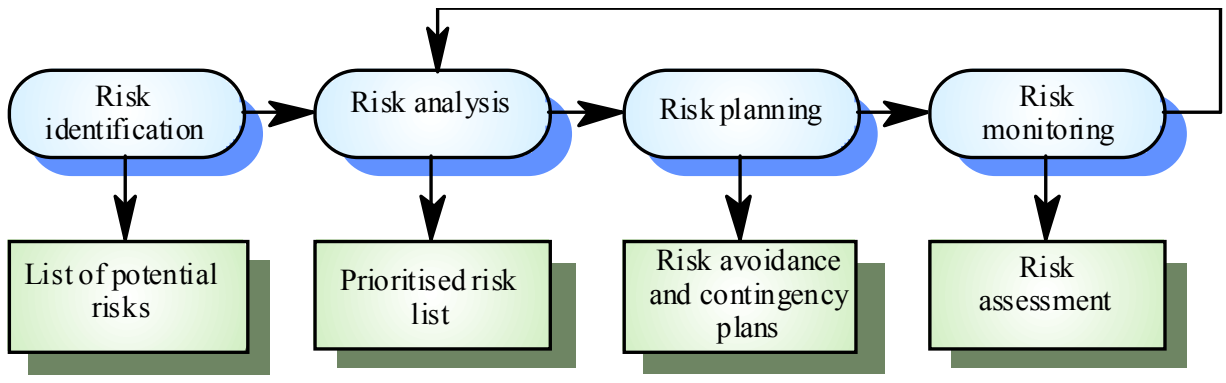
Software Risks

- Project risks – project schedule and resources
- Product risks – quality or performance of the software product
- Business risks – organization

Risk	Risk type	Description
Staff turnover	Project	Experienced staff will leave the project
Management change	Project	
H/W unavailability	Project	
Requirements change	Project, product	
Specs. Delay	Project, product	
Size underestimate	Project, product	
CASE tool under-performance	Product	
Technology change	Business	
Product competition	Business	

The risk management process

- Risk identification
  - Identify project, product and business risks
- Risk analysis
  - Assess the likelihood and consequences of these risks
- Risk planning
  - Draw up plans to avoid or minimise the effects of the risk
- Risk monitoring
  - Monitor the risks throughout the project



Risk identification

Risk type	Possible risks
Technology	The database used in the system cannot process as many transactions per second as expected. Software components which should be reused contain defects which limit their functionality.
People	It is impossible to recruit staff with the skills required. Key staff are ill and unavailable at critical times. Required training for staff is not available.
Organisational	The organisation is restructured so that different management are responsible for the project. Organisational financial problems force reductions in the project budget.
Tools	The code generated by CASE tools is inefficient. CASE tools cannot be integrated.
Requirements	Changes to requirements which require major design rework are proposed. Customers fail to understand the impact of requirements changes.
Estimation	The time required to develop the software is underestimated. The rate of defect repair is underestimated. The size of the software is underestimated.

Risk analysis

- Assess probability and seriousness of each risk
- Probability may be very low, low, moderate, high or very high
- Risk effects might be catastrophic, serious, tolerable or insignificant

Risk	Probability	Effects
Organisational financial problems force reductions in the project budget.	Low	Catastrophic
It is impossible to recruit staff with the skills required for the project.	High	Catastrophic
Key staff are ill at critical times in the project.	Moderate	Serious
Software components which should be reused contain defects which limit their functionality.	Moderate	Serious
Changes to requirements which require major design rework are proposed.	Moderate	Serious
The organisation is restructured so that different management are responsible for the project.	High	Serious
The database used in the system cannot process as many transactions per second as expected.	Moderate	Serious
The time required to develop the software is underestimated.	High	Serious
CASE tools cannot be integrated.	High	Tolerable
Customers fail to understand the impact of requirements changes.	Moderate	Tolerable
Required training for staff is not available.	Moderate	Tolerable
The rate of defect repair is underestimated.	Moderate	Tolerable
The size of the software is underestimated.	High	Tolerable
The code generated by CASE tools is inefficient.	Moderate	Insignificant

Risk planning

- Consider each risk and develop a strategy to manage that risk
- Avoidance strategies
  - The probability that the risk will arise is reduced
- Minimisation strategies
  - The impact of the risk on the project or product will be reduced
- Contingency plans
  - If the risk arises, contingency plans are plans to deal with that risk

Risk management strategies

Risk	Strategy
Organisational financial problems	Prepare a briefing document for senior management show how the project is making a very important contribution to goals of the business.
Recruitment problems	Alert customer of potential difficulties and the possibility delays, investigate buying-in components.
Staff illness	Reorganise team so that there is more overlap of work a people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bough components of known reliability.
Requirements changes	Derive traceability information to assess requireme change impact, maximise information hiding in the design.
Organisational restructuring	Prepare a briefing document for senior management show how the project is making a very important contribution to goals of the business.
Database performance	Investigate the possibility of buying a higher-performan database.
Underestimated development time	Investigate buying in components, investigate use of program generator.

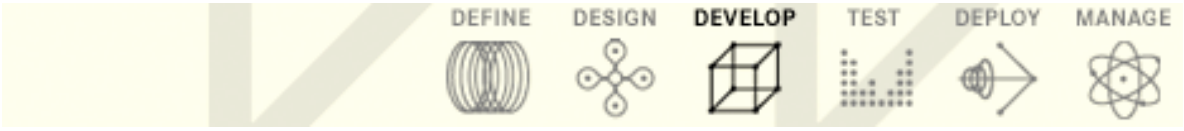
Risk monitoring

- Assess each identified risks regularly to decide whether or not it is becoming less or more probable
- Also assess whether the effects of the risk have changed
- Each key risk should be discussed at management progress meetings

Risk factors

Risk type	Potential indicators
Technology	Late delivery of hardware or support software, many reported technology problems
People	Poor staff morale, poor relationships amongst team member, job availability
Organisational	organisational gossip, lack of action by senior management
Tools	reluctance by team members to use tools, complaints about CASE tools, demands for higher-powered workstations
Requirements	many requirements change requests, customer complaints
Estimation	failure to meet agreed schedule, failure to clear reported defects

References: *Pressman - Chap.25 , Sommerville – Chap. 4* [**Next**] *Requirements & Specifications*



Software Engineering

Requirement & Specification
Requirement Engineering
View Point Oriented Analysis (VORD)

Pramod Parajuli

Software Engineering **Day**

Software Requirements

Descriptions and specifications of a system

- Functional and non-functional requirements
- User requirements
- System requirements
- Interface specification
- SRS

Requirements engineering

- The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed
- The requirements themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process

What is a requirement?

- May range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification
- This is inevitable as requirements may serve a dual function
    - May be the basis for a bid for a contract - must be open to interpretation
    - May be the basis for the contract itself - therefore must be defined in detail
    - Both these statements may be called requirements

Types of requirement

- User requirements
    - Statements in natural language, diagrams of the services. Written for customers.
- System requirements
    - Structured document, detailed descriptions of the system services. Written as a contract between client and contractor
- Software specification
    - A detailed software description which can serve as a basis for a design or implementation. Written for developers

Example

User requirement definition

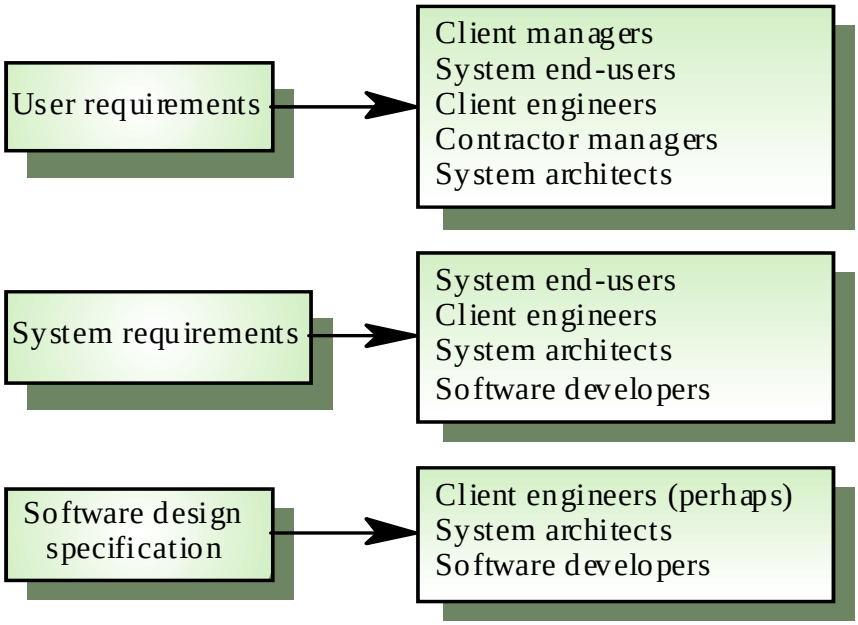
1. LIBSS shall keep track of all data required by copyright licensing agencies in the UK and elsewhere.

System requirements specification

1.1 On making request for a document from LIBSYS, the requestor shall be presented with a form that records details of the user and the request made.

1.2 LIBSYS request forms shall be stored on the system for fie years from the date of request.

1.3 …



Functional and non-functional requirements

- Functional requirements
  - Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
- Non-functional requirements
  - Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
- Domain requirements
  - Requirements that come from the application domain of the system and that reflect characteristics of that domain

Functional requirements

- Describe functionality or system services
- Functional user requirements may be high-level statements of what the system should do but functional system requirements should describe the system services in detail

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 completeness and consistency

- In principle requirements should be both complete and consistent
- Complete - they should include descriptions of all facilities required
- Consistent - there should be no conflicts or contradictions in the descriptions of the system facilities
- In practice, it is impossible to produce a complete and consistent requirements document

Non-functional requirements

- 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

Goals and requirements

- Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify.
- Goal - A general intention of the user such as ease of use
- Verifiable non-functional requirement - A statement using some measure that can be objectively tested

"The system should be easy to use by experienced controllers and should be organised in such a way that user errors are minimised."

Requirements measures

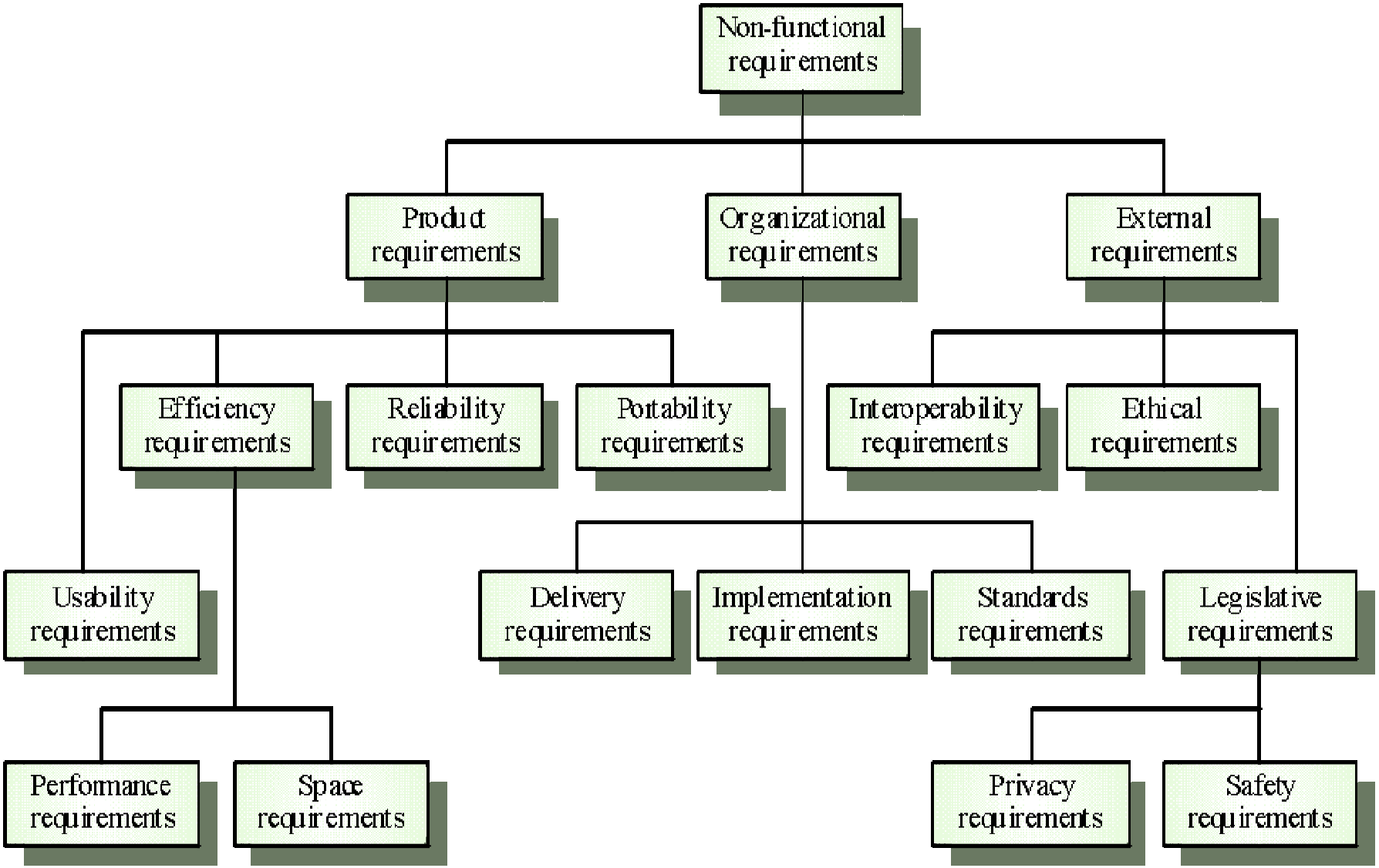
Property	Measure
Speed	Processed transactions/second User/Event response time Screen refresh time
Size	K bytes No. of RAM chips
Ease of use	Training time No. of help frames
Reliability	Mean time of failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target-dependent statements No. of target systems

Requirements interaction - Conflicts between different non-functional requirements are common in complex systems.

Domain requirements

- Derived from the application domain and describe system characteristics and features that reflect the domain
- May be new functional requirements, constraints on existing requirements or define specific computations
- If domain requirements are not satisfied, the system may be unworkable

e.g. **Library system domain requirements**
There shall be a standard user interface to all databases which shall be based on the Z39.50 standard.


Non-functional requirements
Product requirements
Organizational requirements
External requirements
Efficiency requirements
Reliability requirements
Portability requirements
Interoperability requirements
Ethical requirements
Usability requirements
Delivery requirements
Implementation requirements
Standards requirements
Legislative requirements
Performance requirements
Space requirements
Privacy requirements
Safety requirements

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

User requirements

- Should describe functional and non-functional requirements so that they are understandable by system users who don't have detailed technical knowledge
- User requirements are defined using natural language, tables and diagrams

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

Detailed user requirement

3.5.1 Adding nodes to a design

3.5.1.1 **The editor shall provide a facility for users to add nodes of a specified type to their design.**

3.5.1.2 The sequence of actions to add a node should be as follows:

1. The user should select the type of node to be added.
2. The user should move the cursor to the approximate node position in the diagram and indicate that the node symbol should be added at that point.
3. The user should then drag the node symbol to its final position.

Rationale: The user is the best person to decide where to position a node on the diagram. This approach gives the user direct control over node type selection and positioning.

Specification: ECLIPSE/WS/Tools/DE/FS. Section 3.5.1

Guidelines for writing requirements

- Invent a standard format and use it for all requirements
- Use language in a consistent way. Use shall for mandatory requirements, should for desirable requirements
- Use text highlighting to identify key parts of the requirement
- Avoid the use of computer jargon

System requirements

- More detailed specifications of user requirements
- Serve as a basis for designing the system
- May be used as part of the system contract
- System requirements may be expressed using system models discussed in Chapter 7

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

Problems with NL specification

- Ambiguity
- Over-flexibility
- Lack of modularisation

Alternatives to NL specification

- Structured natural language – use standard forms and templates
- Design description language – like programming language, demonstrates operational model
- Graphical notations
- Mathematical specifications

Structured language specifications

- A limited form of natural language may be used to express requirements
- This removes some of the problems resulting from ambiguity and flexibility and imposes a degree of uniformity on a specification
- Often supported using a forms-based approach

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
- Pre and post conditions (if appropriate)
- The side effects (if any)

PDL-based requirements definition

- Requirements may be defined operationally using a language like a programming language but with more flexibility of expression
- Most appropriate in two situations
  - Where an operation is specified as a sequence of actions and the order is important
  - When hardware and software interfaces have to be specified
- Disadvantages are
  - The PDL may not be sufficiently expressive to define domain concepts
  - The specification will be taken as a design rather than a specification

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

Form based specification

ECLIPSE/Workstation/Tools/DE/FS/3.5.1	
Function	Add node
Description	Adds a node to an existing design. The user selects the type of nod When added to the design, the node becomes the current selection. The user chooses the nc moving the cursor to the area where the node is added.
Inputs	Node type, Node position, Design identifier.
Source	Node type and Node position are input by the user, Design identi
Outputs	Design identifier.
Destination	The design database. The design is committed to the database on operation.
Requires	Design graph rooted at input design identifier.
Pre-condition	The design is open and displayed on the user's screen.
Post-condition	The design is unchanged apart from the addition of a node of the spec at the given position.
Side-effects	None
Definition: ECLIPSE/Workstation/Tools/DE/RD/3.5.1	

Interface specification

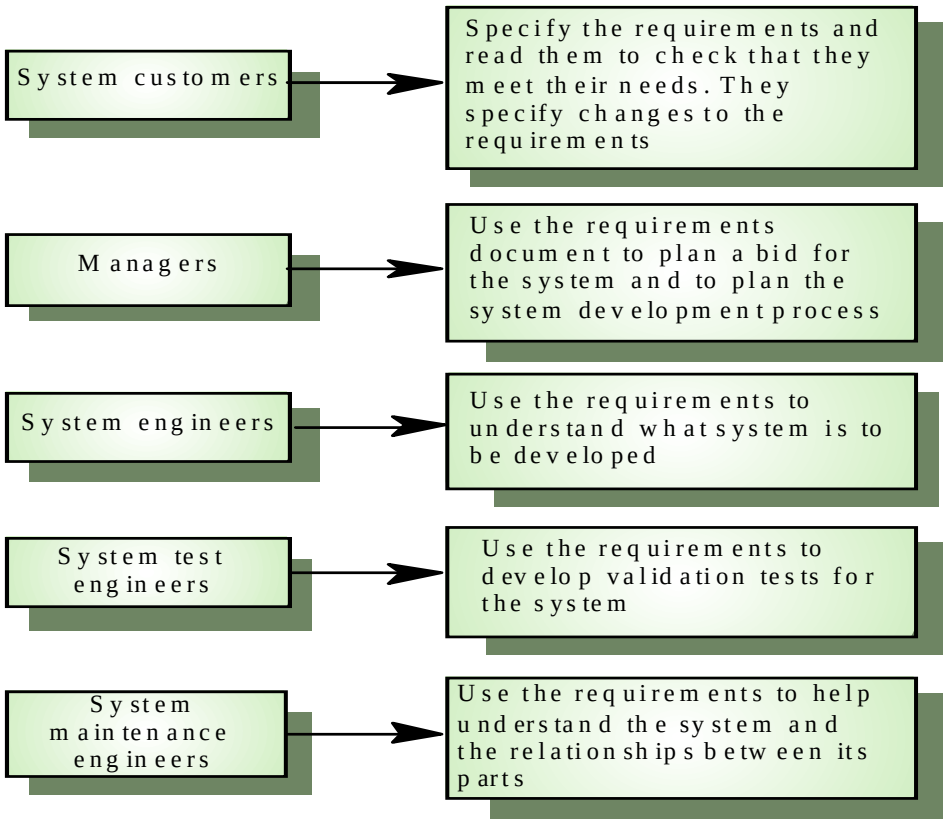
```
interface PrintServer {

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

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

The requirements document

- The requirements document is the official statement of what is required of the system developers
- Should include both a definition and a specification of 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



Requirements document requirements

- Specify external system behaviour
- Specify implementation constraints
- Easy to change
- Serve as reference tool for maintenance
- Record forethought about the life cycle of the system i.e. predict changes
- Characterise responses to unexpected events

IEEE requirements standard

- Introduction
- General description
- Specific requirements
- Appendices
- Index
- This is a generic structure that must be instantiated for specific systems

Requirements document structure

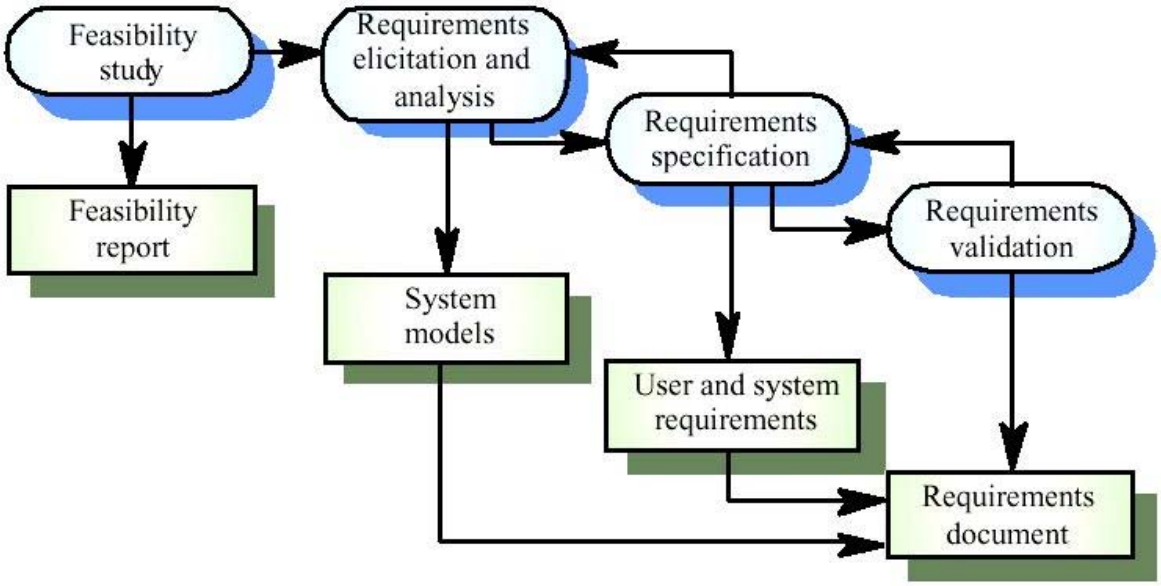
- Preface
- Introduction
- Glossary
- User requirements definition
- System architecture
- System requirements specification
- System models
- System evolution
- Appendices
- Index

References: *Pressman - Chap. , Sommerville – Chap. 5* [**Next**] *Requirement Engineering*

Software Engineering **Day**

Requirement Engineering

Requirement engineering process



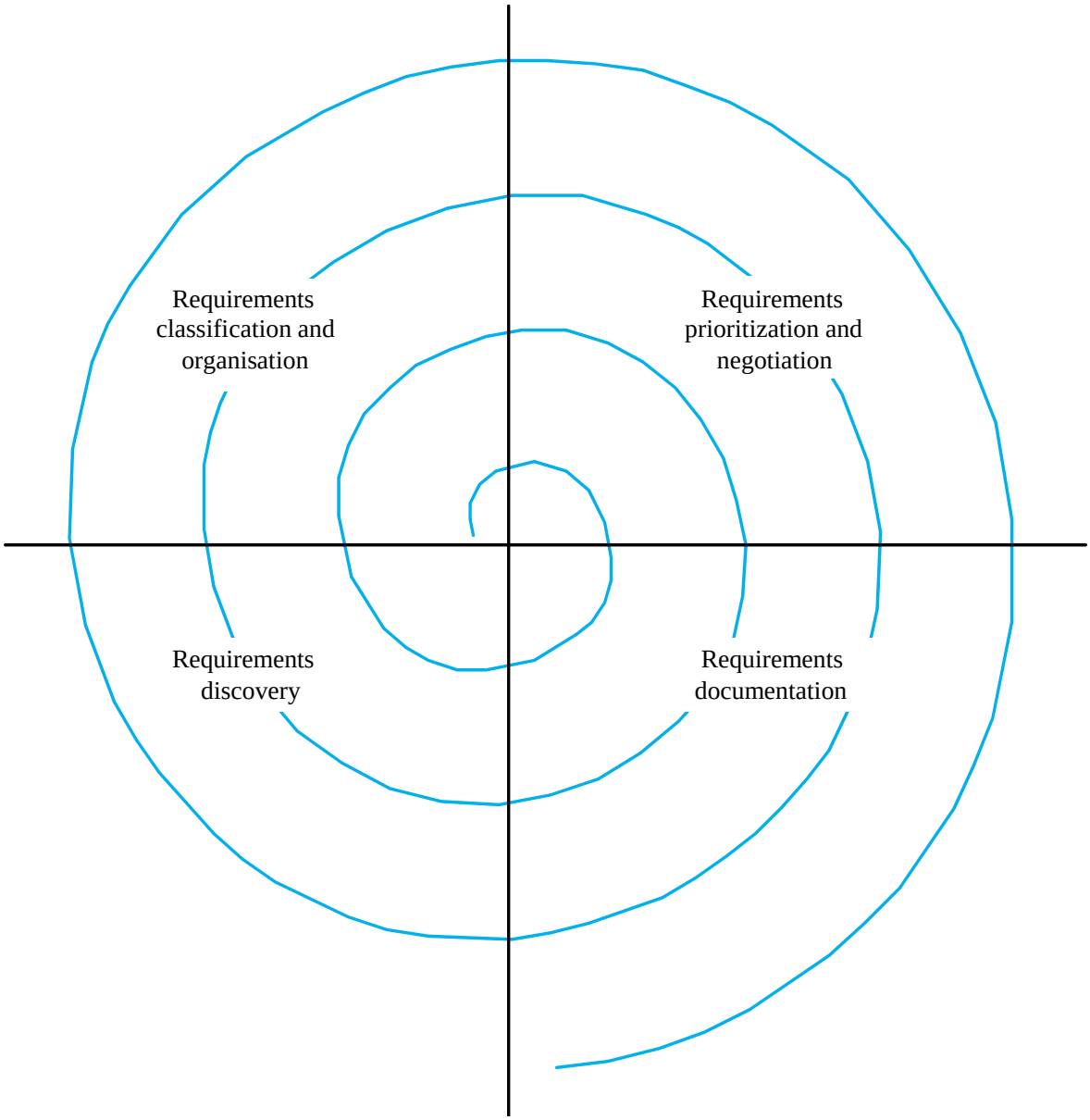
Feasibility studies

- A feasibility study decides whether or not the proposed system is worthwhile
- A short focused study that checks
    - If the system contributes to organisational objectives
    - If the system can be engineered using current technology and within budget
    - If the system can be integrated with other systems that are used

- Questions for people in the organisation
    - What if the system wasn't implemented?
    - What are current process problems?
    - How will the proposed system help?
    - What will be the integration problems?
    - Is new technology needed? What skills?
    - What facilities must be supported by the proposed system?

Requirements Engineering Tasks

- Inception
- Elicitation
    - Scope
    - Understanding
    - Volatility (requirements)
- Elaboration
- Negotiation
- Specification
- Validation (Refer to 'Requirements Validation Checklist', Pressman 6, p. 179)
- Requirements management.



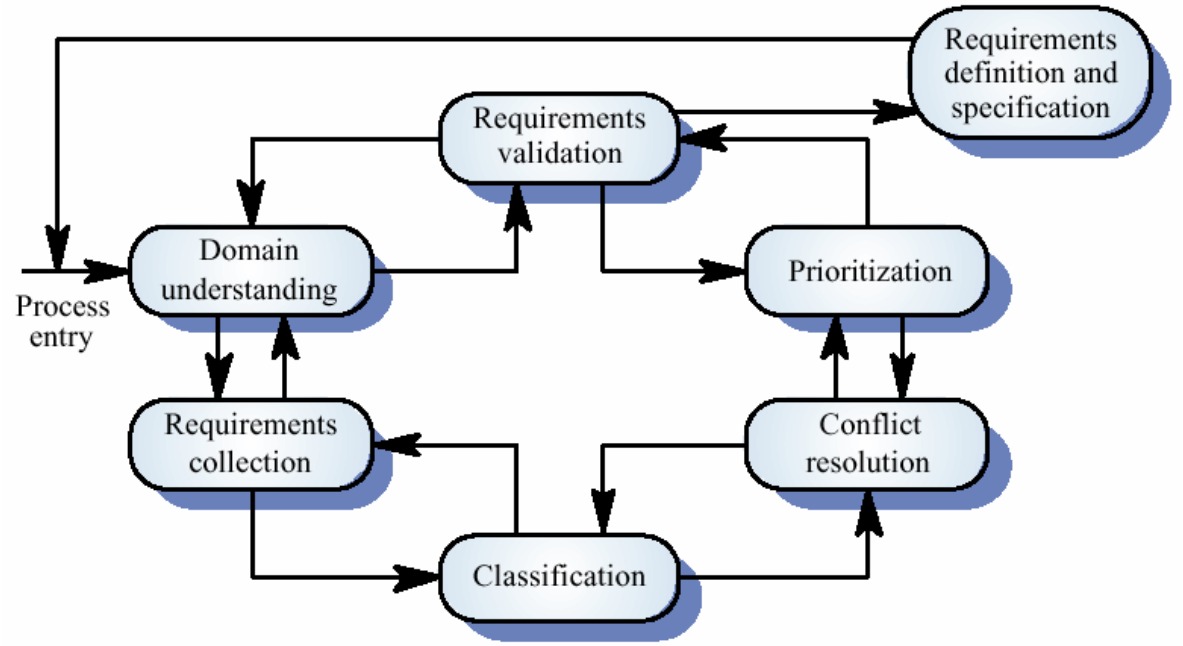
Elicitation and analysis

- Sometimes called requirements elicitation or requirements discovery
- Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints
- May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders*

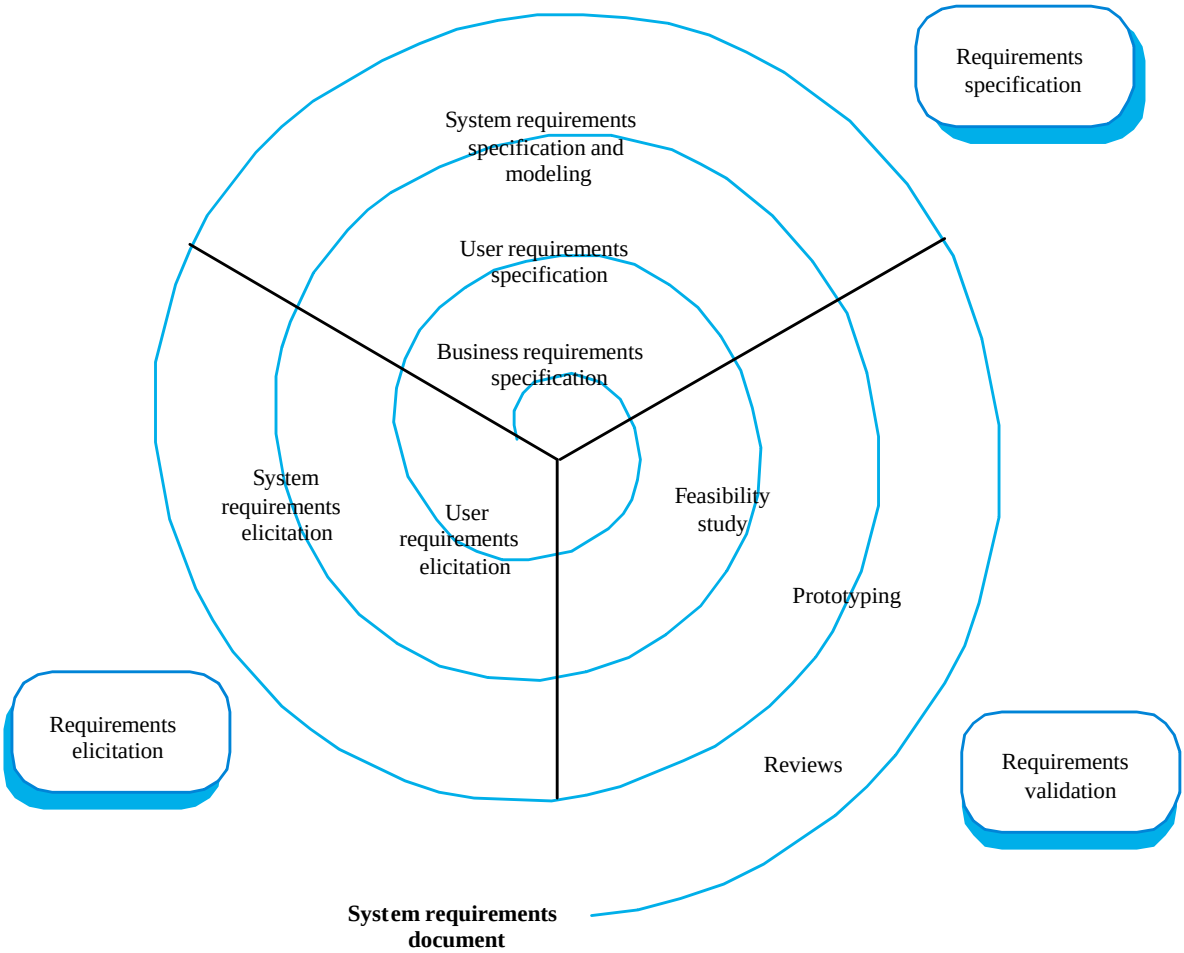
Problems of requirements analysis

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

The requirements analysis process



Requirements elicitation and analysis process



Process activities

- Domain understanding
- Requirements discovery and collection
- Classification and organization
- Conflict resolution
- Prioritisation and negotiation
- Requirements checking
- Requirements documentation

System models

- Different models may be produced during the requirements analysis activity
- Requirements analysis may involve three structuring activities which result in these different models
    - Partitioning: Identifies the structural (part-of) relationships between entities
    - Abstraction: Identifies generalities among entities
    - Projection: Identifies different ways of looking at a problem

Viewpoint-oriented elicitation

- Stakeholders represent different ways of looking at a problem or problem viewpoints
- This multi-perspective analysis is important as there is no single correct way to analyse system requirements

Banking ATM system

Services - cash withdrawal, message passing to request a service, ordering a statement and transferring funds.

Autoteller viewpoints

- Bank customers
- Representatives of other banks
- Hardware and software maintenance engineers
- Marketing department
- Bank managers and counter staff
- Database administrators and security staff
- Communications engineers
- Personnel department

Types of viewpoint

- Data sources or sinks
  Viewpoints are responsible for producing or consuming data. Analysis involves checking that data is produced and consumed and that assumptions about the source and sink of data are valid
- Representation frameworks
  Viewpoints represent particular types of system model. These may be compared to discover requirements that would be missed using a single representation. Particularly suitable for real-time systems
- Receivers of services
  Viewpoints are external to the system and receive services from it. Most suited to interactive systems

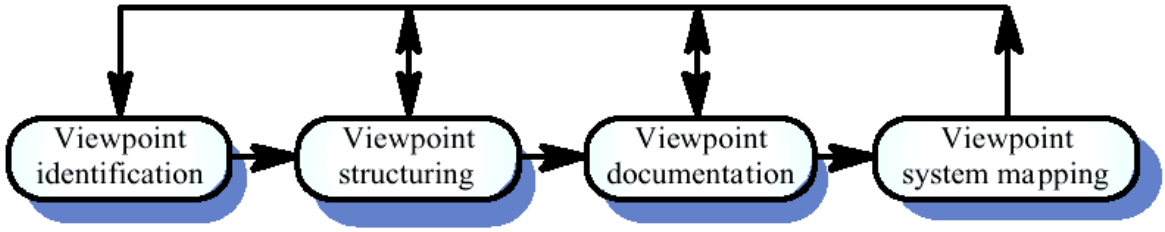
External viewpoints

- Natural to think of end-users as receivers of system services
- Viewpoints are a natural way to structure requirements elicitation
- It is relatively easy to decide if a viewpoint is valid
- Viewpoints and services may be sued to structure non-functional requirements

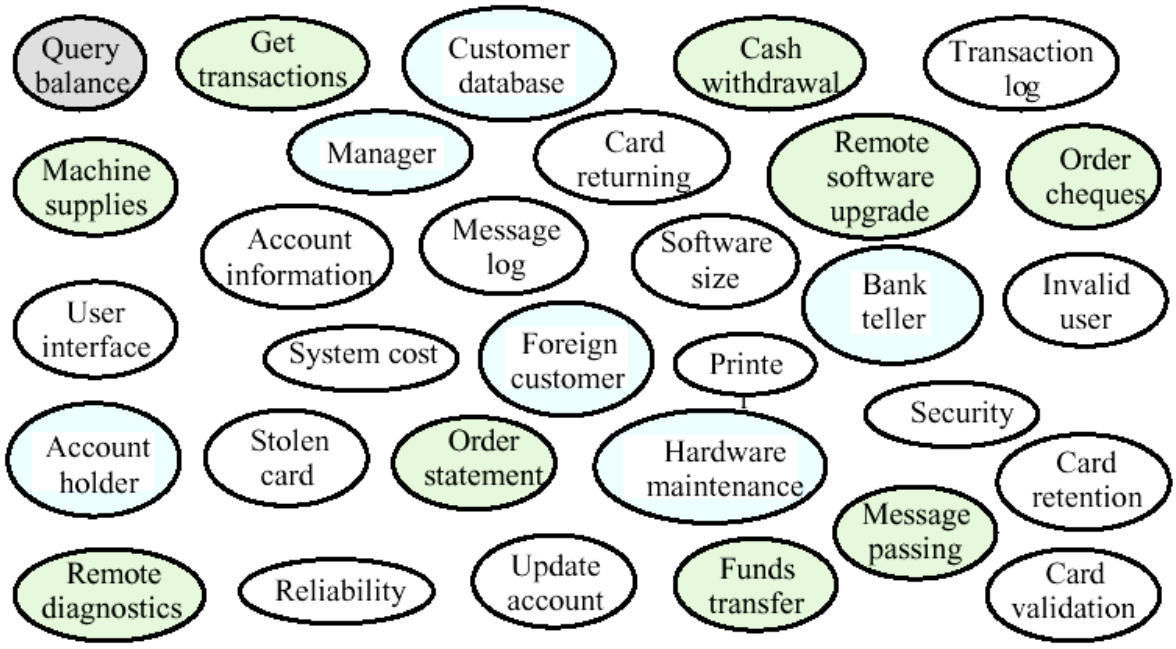
Method-based analysis

- Widely used approach to requirements analysis. Depends on the application of a structured method to understand the system
- Methods have different emphases. Some are designed for requirements elicitation, others are close to design methods
- A viewpoint-oriented method (VORD) is used as an example here. It also illustrates the use of viewpoints

The VORD process model



Viewpoint identification



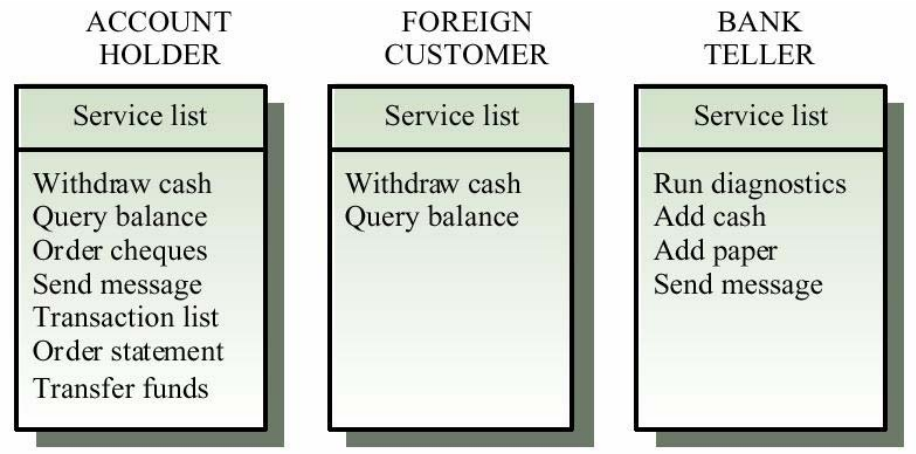
Viewpoint template

Reference:	The viewpoint name.
Attributes:	Attributes providing viewpoint information.
Events:	A reference to a set of event scenarios describing how the system reacts to viewpoint events.
Services	A reference to a set of service descriptions.
Sub-VPs:	The names of sub-viewpoints.

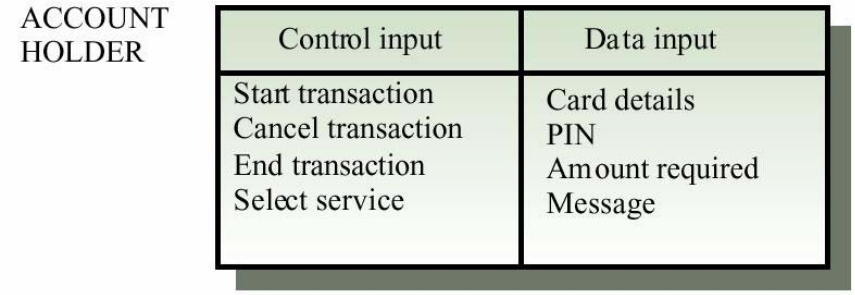
Service template

Reference:	The service name.
Rationale:	Reason why the service is provided.
Specification:	Reference to a list of service specifications. These may be expressed in different notations.
Viewpoints:	List of viewpoint names receiving the service.
Non-functional requirements:	Reference to a set of non-functional requirements which constrain the service.
Provider:	Reference to a list of system objects which provide the service.

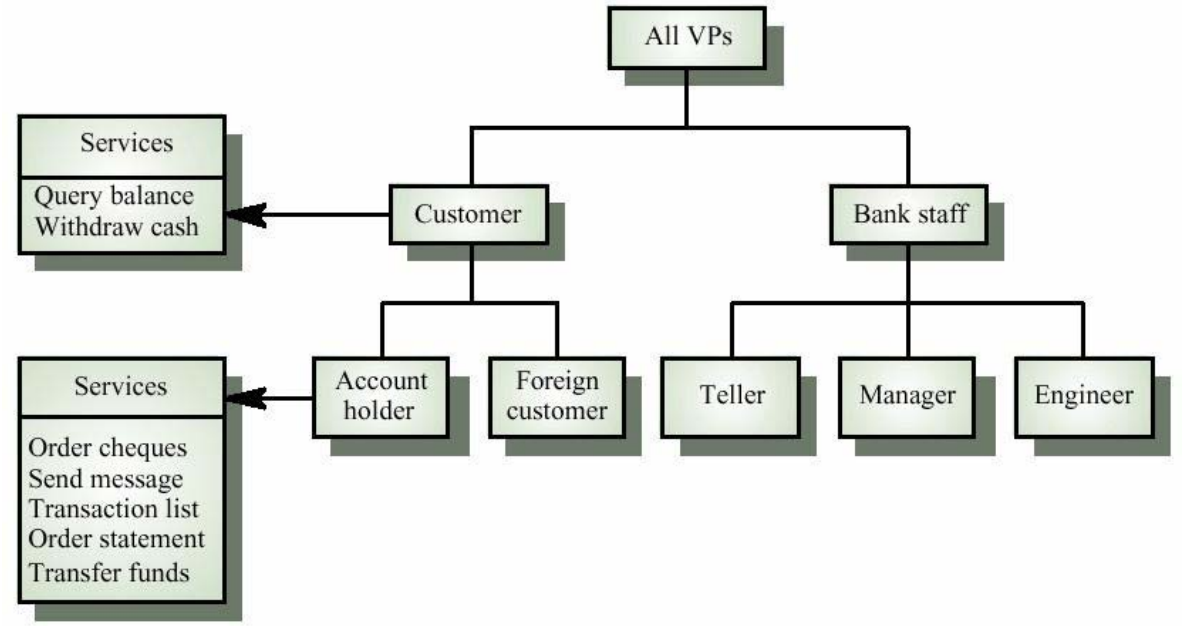
Viewpoint service information



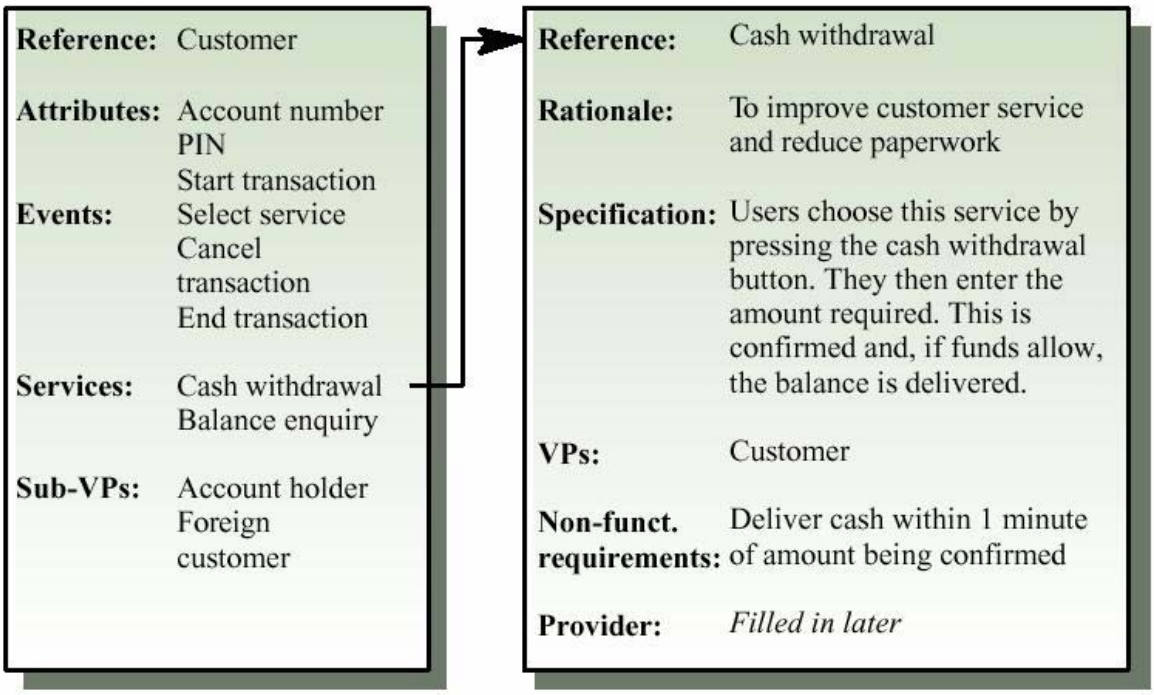
Viewpoint data/control



Viewpoint hierarchy



Customer/cash withdrawal templates



Scenarios

- Scenarios are descriptions of how a system is used in practice
- They are helpful in requirements elicitation as people can relate to these more readily than abstract statement of what they require from a system
- Scenarios are particularly useful for adding detail to an outline requirements description

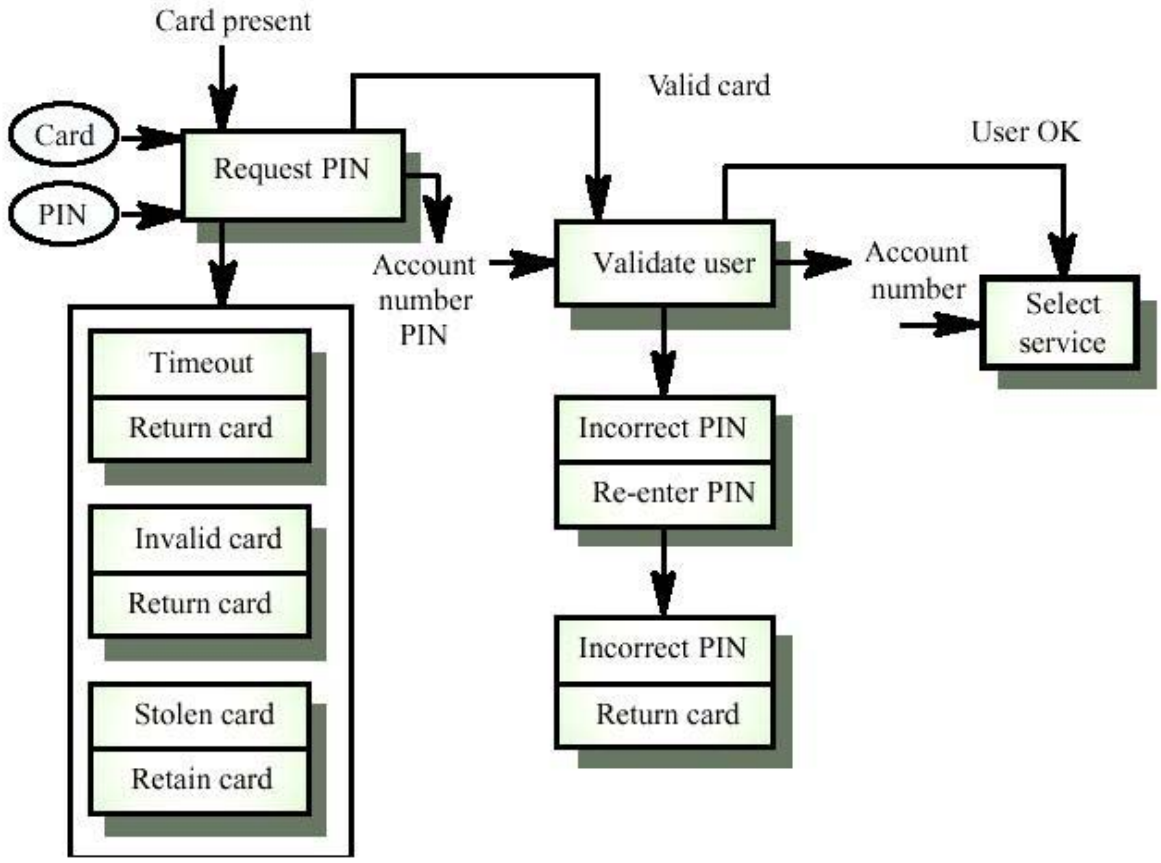
Scenario descriptions

- System state at the beginning of the scenario
- Normal flow of events in the scenario
- What can go wrong and how this is handled
- Other concurrent activities
- System state on completion of the scenario

Event scenarios

- Event scenarios may be used to describe how a system responds to the occurrence of some particular event such as 'start transaction'
- VORD includes a diagrammatic convention for event scenarios.
    - Data provided and delivered
    - Control information
    - Exception processing
    - The next expected event

Event scenario - start transaction



Notation for data and control analysis

- Ellipses. data provided from or delivered to a viewpoint
- Control information enters and leaves at the top of each box
- Data leaves from the right of each box
- Exceptions are shown at the bottom of each box
- Name of next event is in box with thick edges

Exception description

- Most methods do not include facilities for describing exceptions
- In this example, exceptions are
    - Timeout. Customer fails to enter a PIN within the allowed time limit
    - Invalid card. The card is not recognised and is returned
    - Stolen card. The card has been registered as stolen and is retained by the machine

FAST (Facilitated Application Specification Technique)

- Meeting between customers and developers at a neutral site (no home advantage).
- Goals
    - identify the problem
    - propose elements of solution
    - negotiate different approaches
    - specify preliminary set of requirements

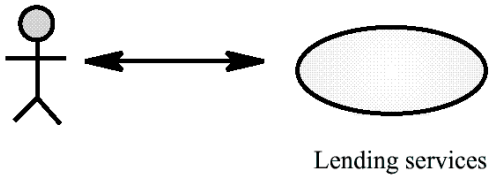
- Rules for participation and preparation established ahead of time.

- Agenda suggested
  - brainstorming encouraged
- Facilitator appointed.
- Definition mechanism
  - Sheets, flipcharts, wallboards, stickers, etc.

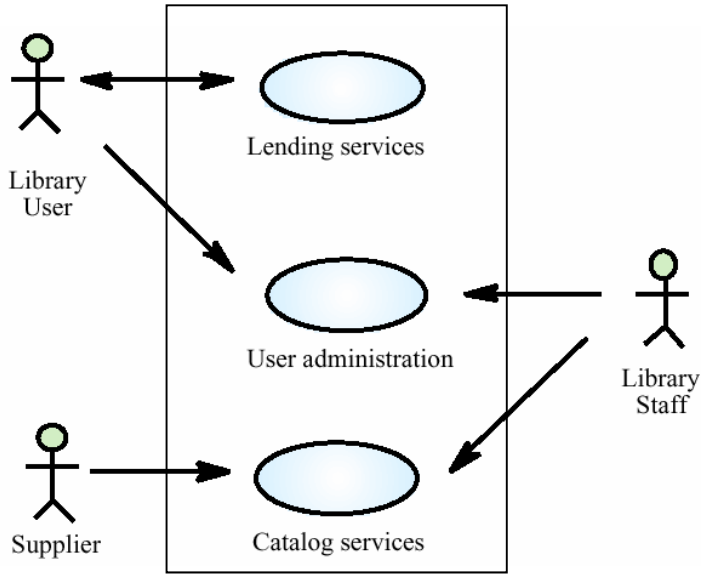
Use cases (requirements)

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

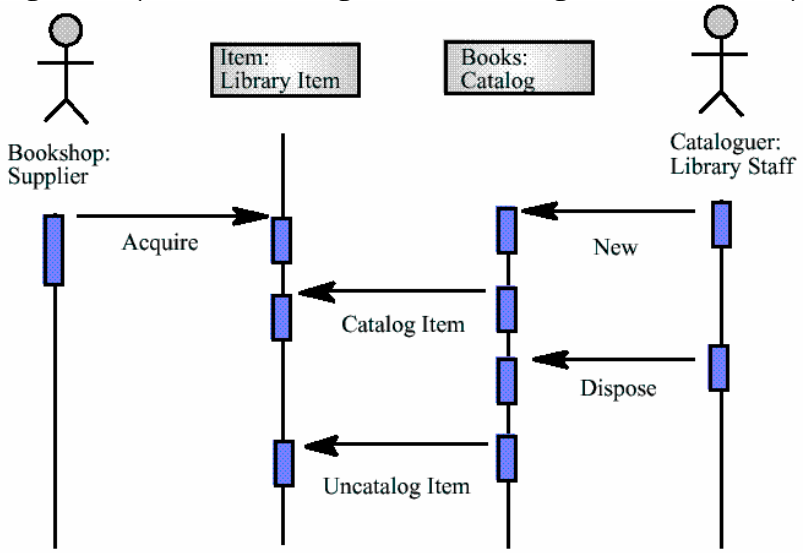
Lending use-case



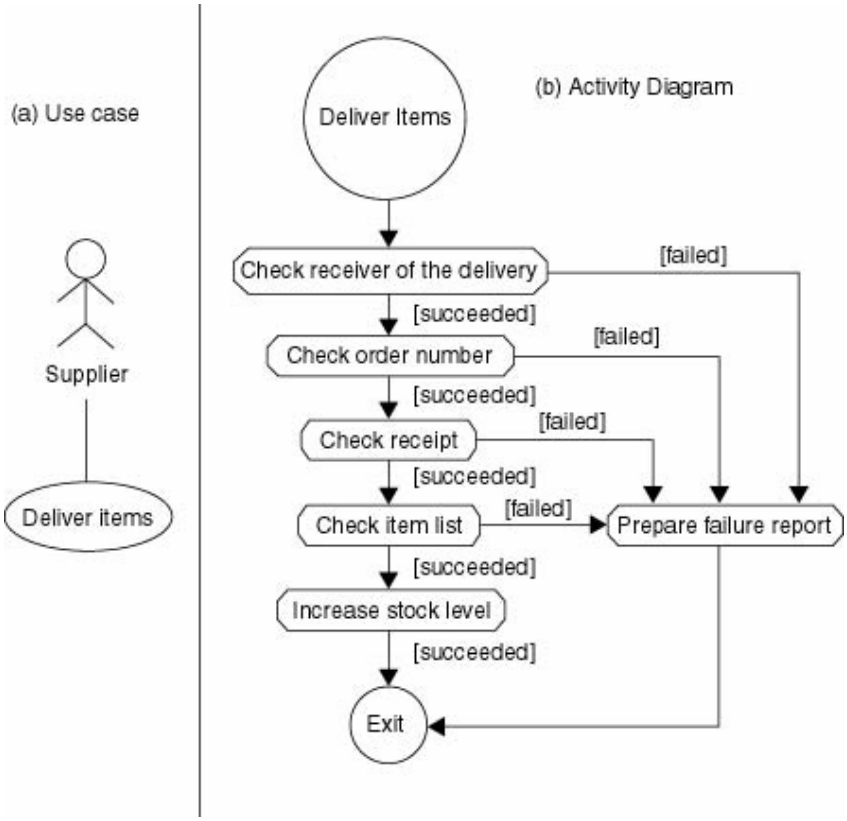
Library use-cases



Catalogue management (interaction diagram – modelling the control flow)



Activity Diagrams

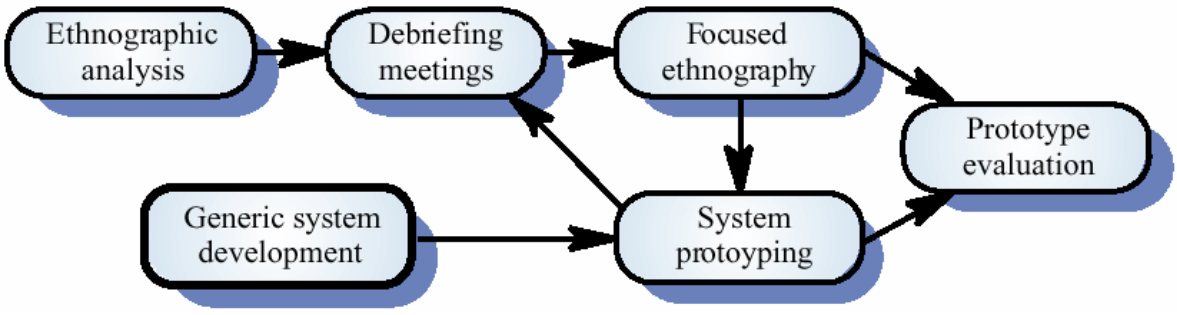


Social and organisational factors

- 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 are influences on all viewpoints

Ethnography

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



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.
- Consistency.
- Completeness.
- Realism.
- Verifiability.

Requirements validation techniques

- Requirements reviews
- Prototyping
- Test-case generation
- Automated consistency analysis (structured)

Requirements reviews

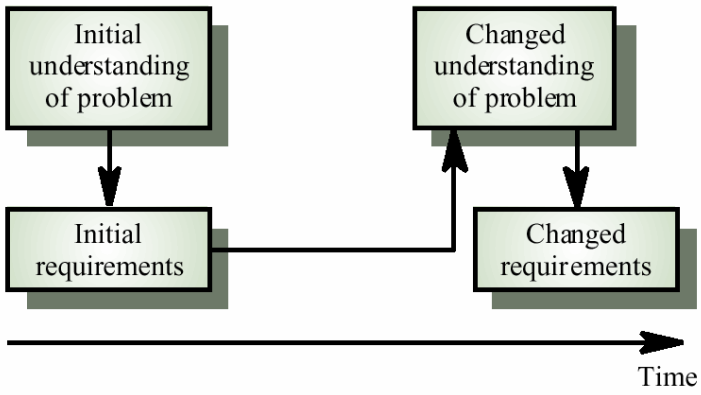
Review checks

- 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
- 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. Stable requirements derived from the core activity of the customer organisation

- Volatile requirements. Requirements which change during development or when the system is in use

Classification of requirements

- Mutable requirements
- Emergent requirements
- Consequential requirements
- Compatibility requirements

Requirements management planning

- 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

Various traceability tables are created to manage requirements.

1. Features traceability table
2. Source traceability table
3. Dependency traceability table
4. Subsystem traceability table
5. Interface traceability table

	Specific aspect of the system or its environment					
Requirements	A1	A2	A3	…	…	Ai
R1		√		√		
R2	√		√		√	√
R3		√	√		√	
…				√	√	
Rn	√	√	√			

Requirement change management

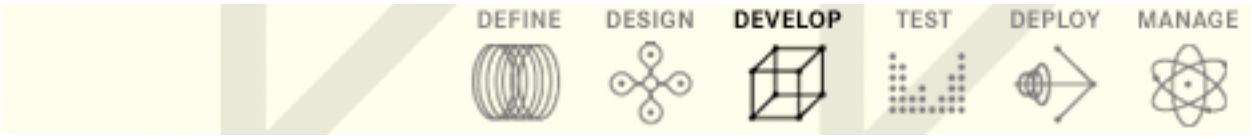
CASE tool support

- 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

http://www.livespecs.com/modules.php?op=modload&name=News&file=index&catid=14&topic=&allstories=1

http://faculty.fuqua.duke.edu/~charvey/Video/Digital_video.htm **[Finance]**

References: *Pressman - Chap. 7 , Sommerville – Chap. 6* **[Next]** *Architectural Design*



Software Engineering

Software Design
Architectural Design

Pramod Parajuli

Software Engineering **Day**

Software Design

- High level design
- Detailed design

Fundamental Software Design Concepts

- Abstraction - hide irrelevant lower level details
  - procedural abstraction - named sequence of events
  - data abstraction - named collection of data objects
- Refinement - process of elaboration for each design component
- Modularity - the degree to examine components independently of one another
- Software architecture - overall structure of the software components
- Control hierarchy or program structure - represents the module organization and implies a control hierarchy, but does not represent the procedural aspects of the software (e.g. event sequences)
- Structural partitioning –
  - horizontal partitioning - input, data transformations, and output
  - vertical partitioning (factoring) distributes control in a top-down manner (control decisions in top level modules and processing work in the lower level modules)
- Data structure - representation of the logical relationship among individual data elements
- Software procedure - precise specification of processing (event sequences, decision points, repetitive operations, data organization/structure)
- Information

Good S/W design

- Capture all functionalities of system correctly
- Easily understandable
- Easily amenable to change
- exhibit good architectural structure
- be modular
- appropriate data structures
- 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

How to?

- Use of consistent and meaningful names
- Cleanly decomposed set of modules
- Need arrangement of modules in hierarchy
  - Low fan out
  - Abstraction

Cohesion & coupling

- Increased functionally independence - high cohesion and low coupling
- Cohesion – measure of functional strength
- Coupling – measure of the degree of functional interdependence

+ reduces error propagation
+ reuse of a module is possible
+ complexity of design reduced

Classification of cohesion

coincidental	logical	temporal	procedural	communicational	sequential	functional

Low ⟶ High

Classification of coupling

data	stamp	control	common	content

Low ⟶ High

Design

- Data design
- Architectural design
- Interface design
- Procedural design
- Component level design

Data design

- Data structures and the operations to be performed on each should be identified
- Data dictionary should be developed
- Low-level data design must be left for detailed design
- Information hiding and modularization must be emphasized
- Library of useful data structures and operations on them should be developed
- Abstract data types defined must be supported by programming language

Architectural design

- Used to generate formal specification for detailed design.
- Establishing the overall structure of a software system
- The design process for identifying the sub-systems making up a system and the framework for sub-system control and communication is ***architectural design***
- The output of this design process is a description of the ***software architecture***
- Represents the link between specification and design processes
- Often carried out in parallel with some specification activities

Advantages of explicit architecture

- Stakeholder communication
  - Architecture may be used as a focus of discussion by system stakeholders
- System analysis
  - Means that analysis of whether the system can meet its non-functional requirements is possible
- Large-scale reuse
  - The architecture may be reusable across a range of systems

Architectural design process

- System structuring
  - Decomposing into principal sub-systems and identifying the communications
- Control modelling
  - A model of the control relationships between the different parts of the system
- Modular decomposition
  - The identified sub-systems are decomposed into modules

Sub-systems and modules

- A sub-system is a system in its own right whose operation is independent of the services provided by other sub-systems.
- A module is a system component that provides services to other components but would not normally be considered as a separate system

Architectural models

- Different architectural models may be produced during the design process
- Each model presents different perspectives on the architecture
- Static structural model that shows the major system components
- Dynamic process model that shows the **process structure** of the system
- Interface model that defines sub-system interfaces
- Relationships model such as a data-flow model

Architectural styles

- The architectural model of a system may conform to a generic architectural model or style
- An awareness of these styles can simplify the problem of defining system architectures
- However, most large systems are heterogeneous and do not follow a single architectural style

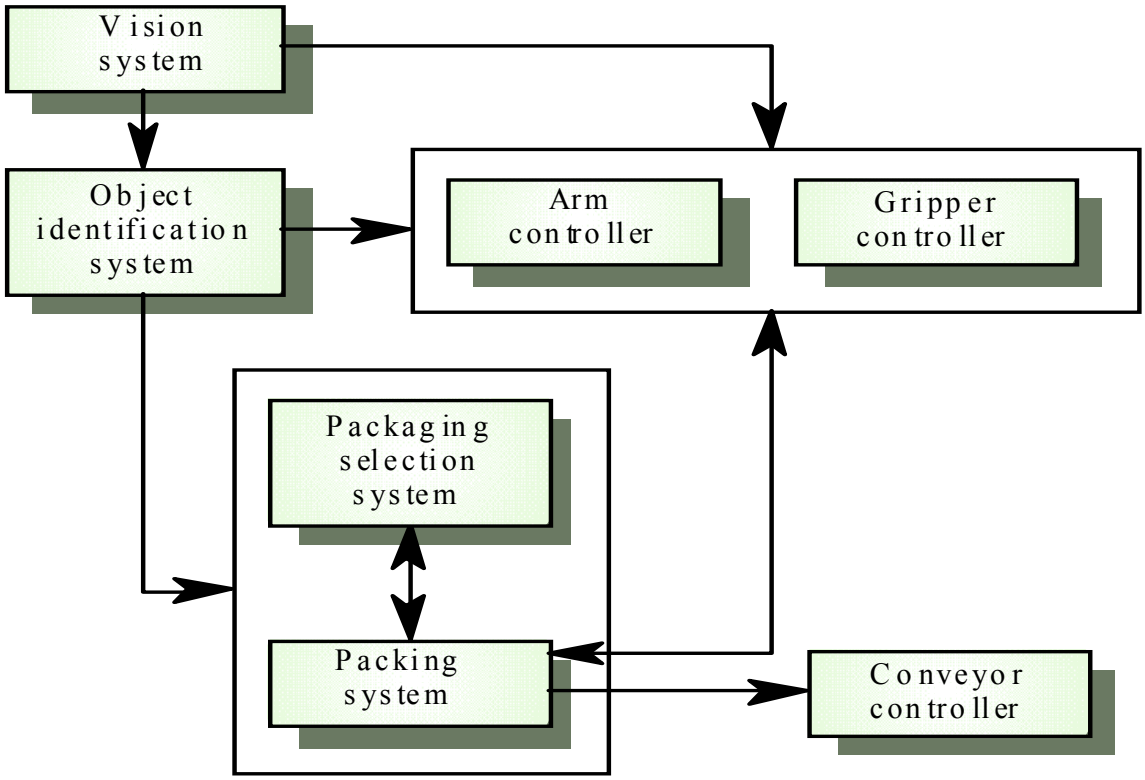
Architecture attributes

- Performance -
    - Localise operations to minimise sub-system communication
- Security -
    - Use a layered architecture with critical assets in inner layers
- Safety -
    - Isolate safety-critical components
- Availability -
    - Include redundant components in the architecture
- Maintainability -
    - Use fine-grain, self-contained components

System structuring

- Concerned with decomposing the system into interacting sub-systems
- The architectural design is normally expressed as a block diagram presenting an overview of the system structure
- More specific models showing how sub-systems share data, are distributed and interface with each other may also be developed

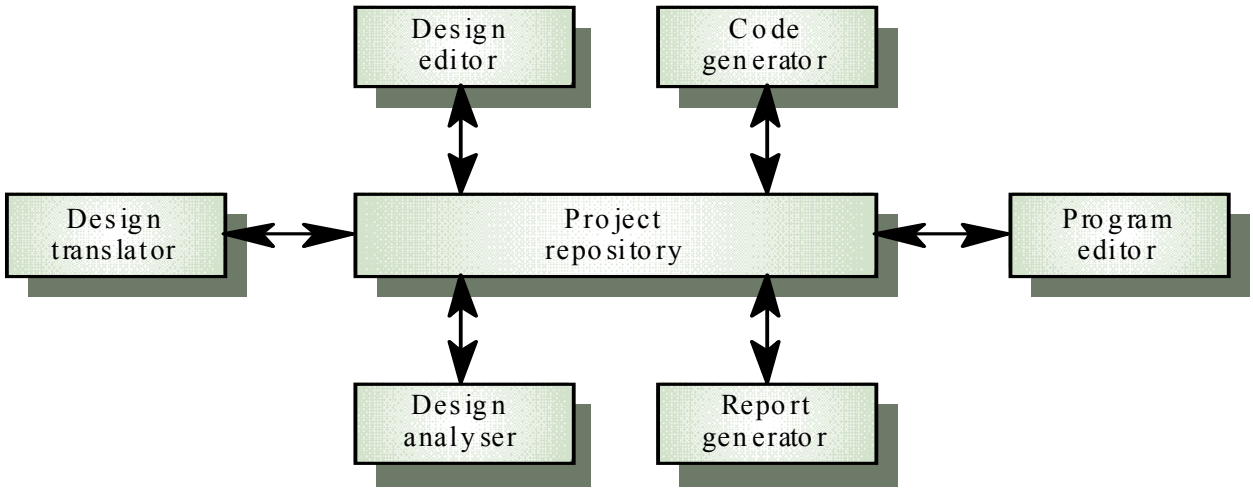
e.g. packing robot control system



The repository model

- Sub-systems must exchange data. This may be done in two ways:
  - Through central database (blackboard)
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems
- When large amounts of data are to be shared, the repository model of sharing is most commonly used

CASE toolset architecture



Repository model characteristics

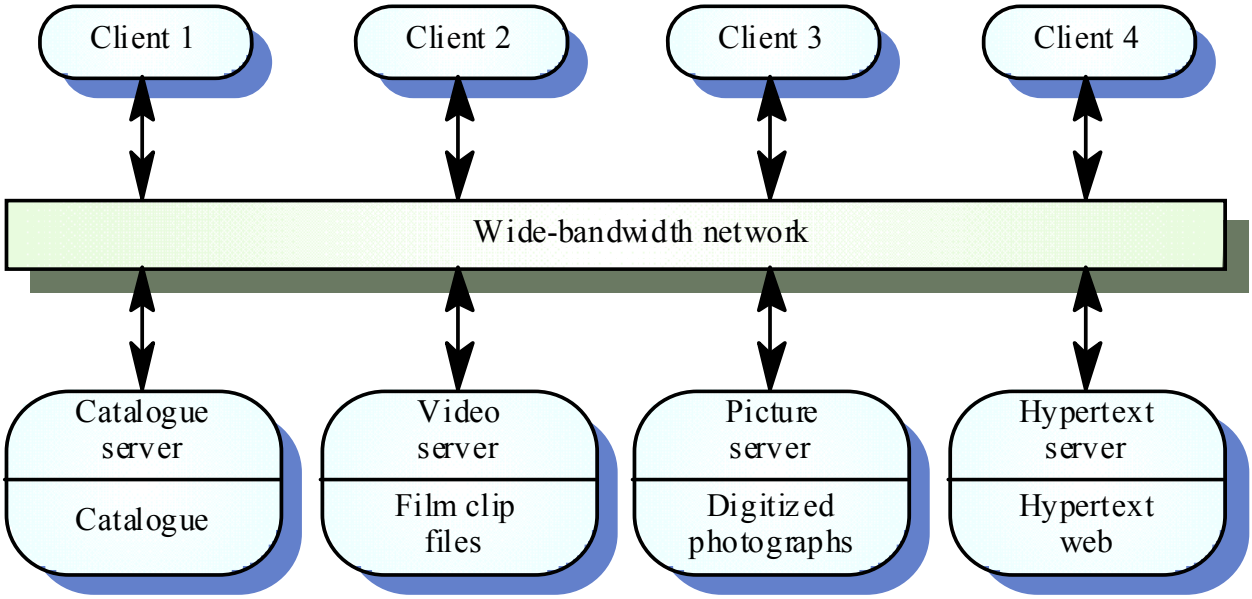
- Advantages
  - Efficient way to share large amounts of data
  - Sub-systems need not be concerned with how data is produced Centralised management e.g. backup, security, etc.
  - Sharing model is published as the repository schema

- Disadvantages
    - Sub-systems must agree on a repository data model. Inevitably a compromise
    - Data evolution is difficult and expensive
    - No scope for specific management policies
    - Difficult to distribute efficiently

Client-server architecture

- How data and processing is distributed across a range of components?
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services
- Network which allows clients to access servers

e.g. Film and picture library



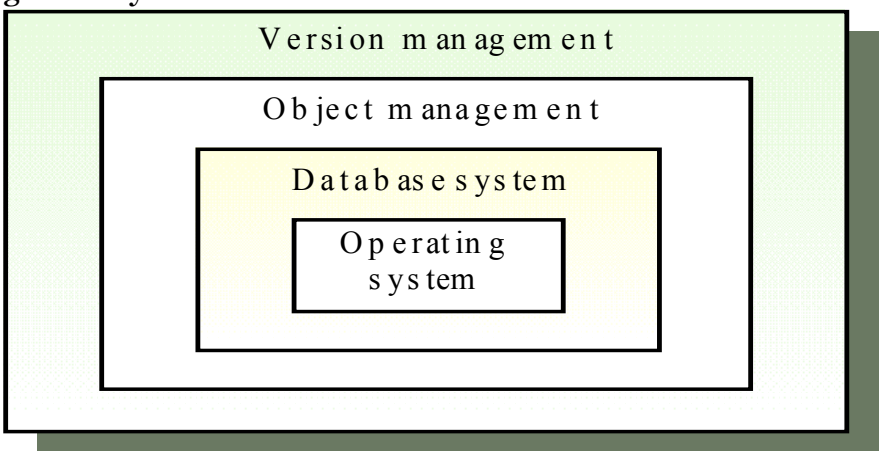
Client-server characteristics

- Advantages
    - Distribution of data is straightforward
    - Makes effective use of networked systems. May require cheaper hardware
    - Easy to add new servers or upgrade existing servers
- Disadvantages
    - No shared data model so sub-systems use different data organisation. data interchange may be inefficient
    - Redundant management in each server
    - No central register of names and services - it may be hard to find out what servers and services are available

Abstract machine model

- Used to model the interfacing of sub-systems
- Organises the system into a set of layers (or abstract machines) each of which provide a set of services
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected
- However, often difficult to structure systems in this way

Version management system



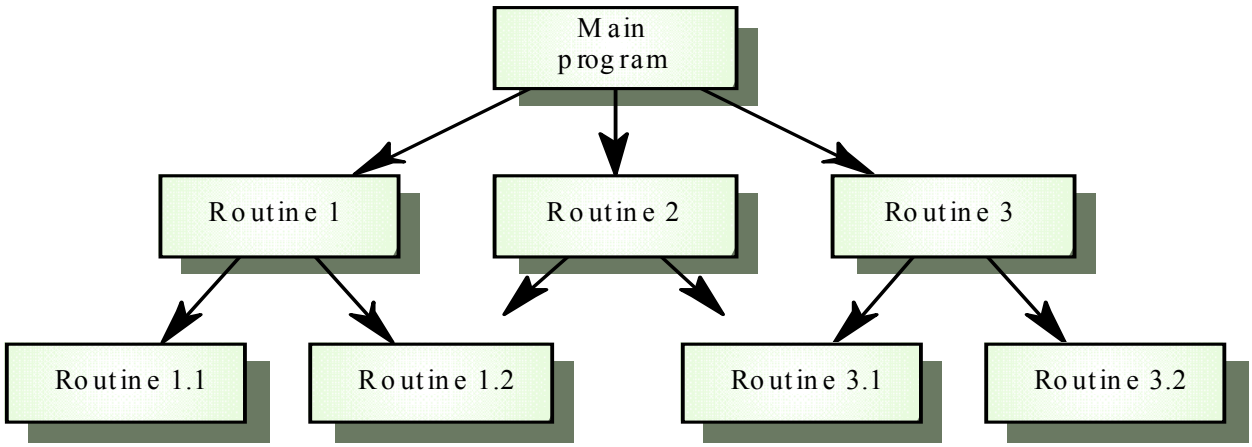
Control models

- Concerned with the control flow between sub-systems. Distinct from the system decomposition model
- Centralised control
  - One sub-system has overall responsibility for control and starts and stops other sub-systems
- Event-based control
  - Each sub-system can respond to externally generated events from other sub-systems or the system's environment

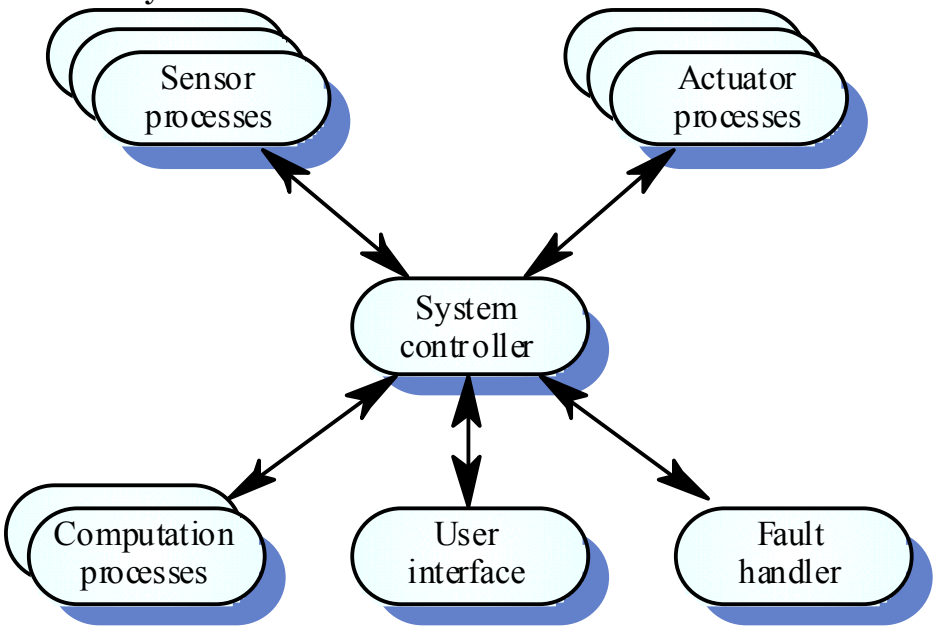
Centralised control

- A control sub-system takes responsibility for managing the execution of other sub-systems
- Call-return model
  - Top-down subroutine model where control starts at the top of a subroutine hierarchy and moves downwards. Applicable to sequential systems
- Manager model
  - Applicable to concurrent systems. One system component controls the stopping, starting and coordination of other system processes. Can be implemented in sequential systems as a case statement

e.g. call-return model



e.g. real-time system control



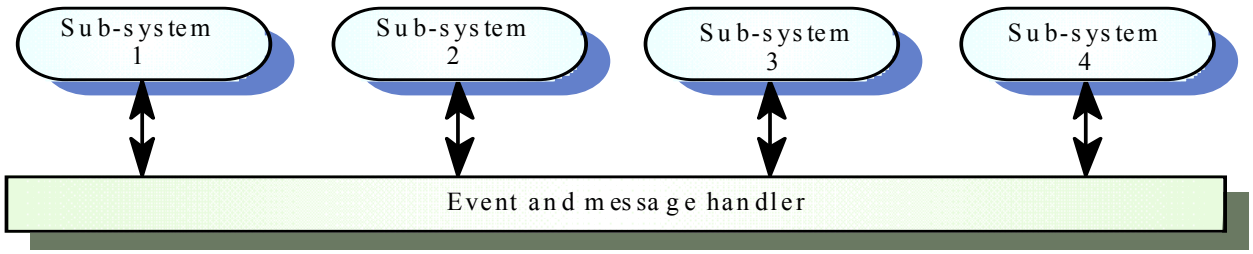
Event-driven systems

- Driven by externally generated events where the timing of the event is out with the control of the sub-systems which process the event
- Two principal event-driven models
  - **Broadcast models**. An event is broadcast to all sub-systems. Any sub-system which can handle the event may do so
  - **Interrupt-driven models**. Used in real-time systems where interrupts are detected by an interrupt handler and passed to some other component for processing
- Other event driven models include spreadsheets and production systems

Broadcast model

- Effective in integrating sub-systems on different computers in a network
- Sub-systems register an interest in specific events. When these occur, control is transferred to the sub-system which can handle the event
- Control policy is not embedded in the event and message handler. Sub-systems decide on events of interest to them
- However, sub-systems don't know if or when an event will be handled

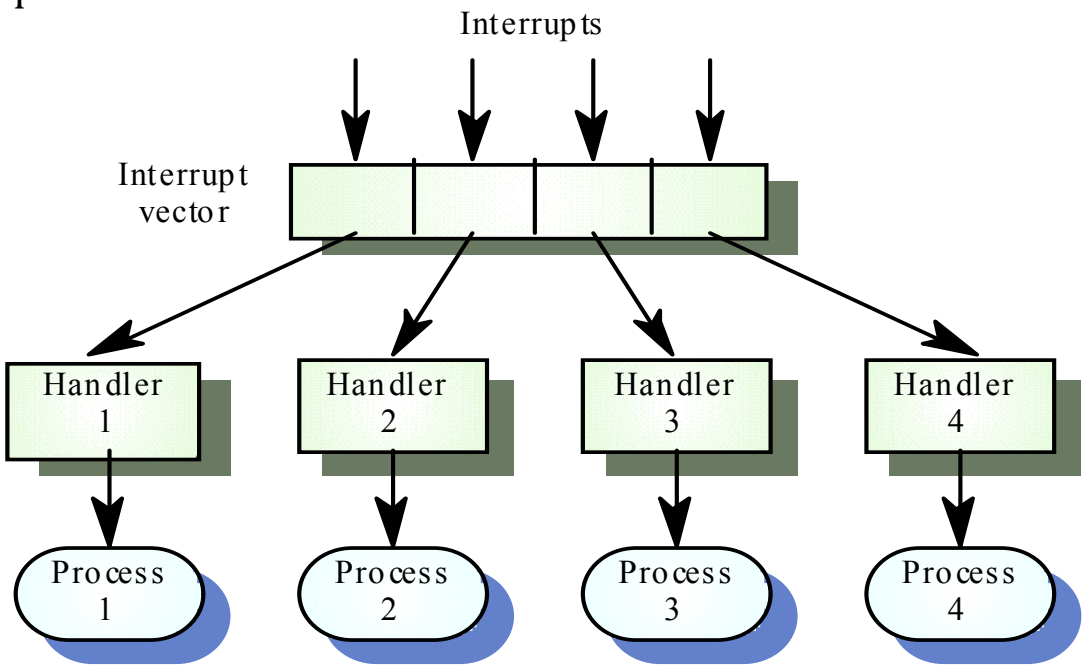
Selective broadcasting



Interrupt-driven systems

- Used in real-time systems where fast response to an event is essential
- There are known interrupt types with a handler defined for each type
- Each type is associated with a memory location and a hardware switch causes transfer to its handler
- Allows fast response but complex to program and difficult to validate

Interrupt-driven control



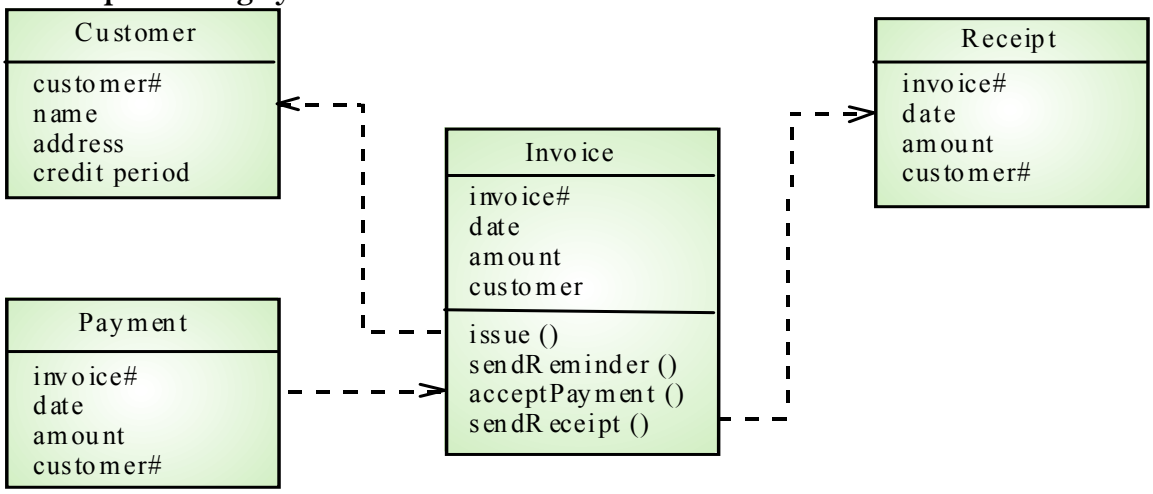
Modular decomposition

- Another structural level where sub-systems are decomposed into modules
- Two modular decomposition models covered
  - An object model where the system is decomposed into interacting objects
  - A data-flow model where the system is decomposed into functional modules which transform inputs to outputs. Also known as the pipeline model
- If possible, decisions about concurrency should be delayed until modules are implemented

Object models

- Structure the system into a set of loosely coupled objects with well-defined interfaces
- Object-oriented decomposition is concerned with identifying object classes, their attributes and operations
- When implemented, objects are created from these classes and some control model used to coordinate object operations

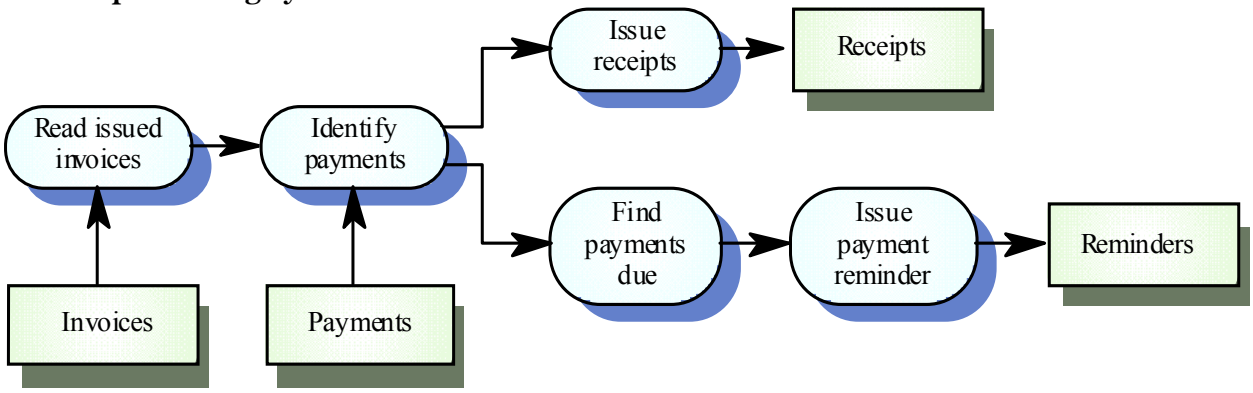
Invoice processing system



Data-flow models

- Functional transformations process their inputs to produce outputs
- May be referred to as a pipe and filter model (as in UNIX shell)
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems
- Not really suitable for interactive systems

Invoice processing system



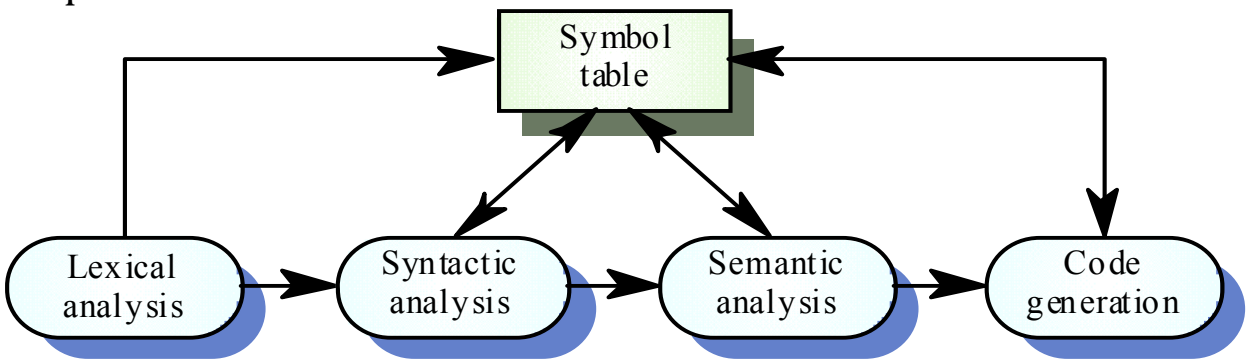
Domain-specific architectures

- Architectural models which are specific to some application domain
- Two types of domain-specific model
  - Generic models - abstractions from a number of real systems, encapsulate the principal characteristics of these systems
  - Reference models - more abstract, idealised model, provide a means of information about that class of system and of comparing different architectures
- Generic models are usually bottom-up models; Reference models are top-down models

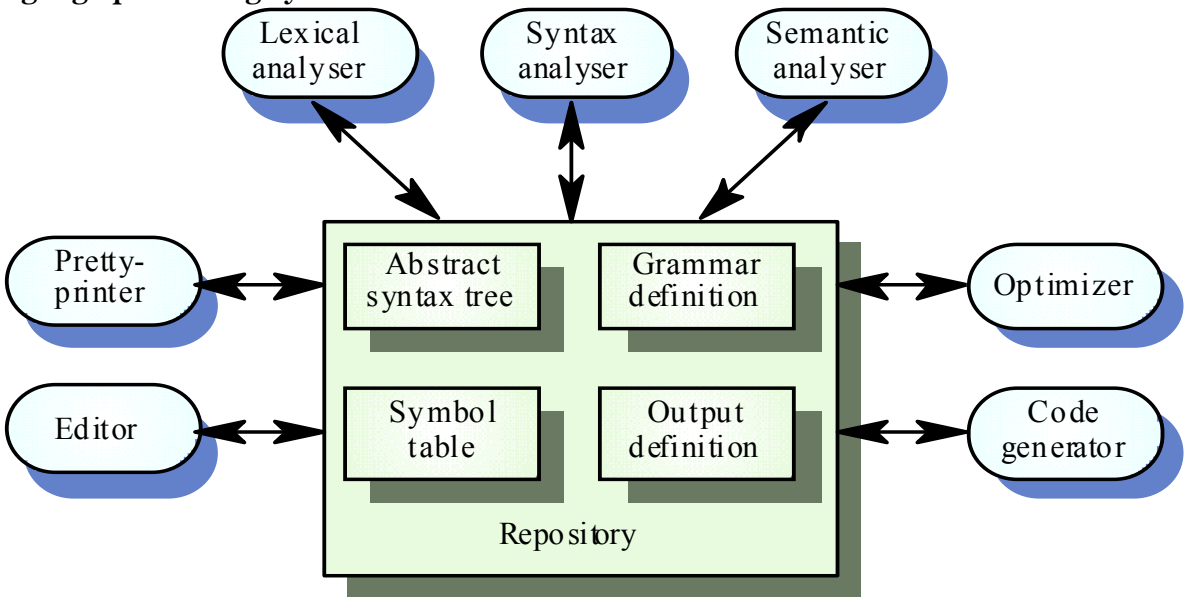
Generic models

- Compiler model is a well-known example although other models exist in more specialised application domains
  - Lexical analyser
  - Symbol table
  - Syntax analyser
  - Syntax tree
  - Semantic analyser
  - Code generator
- Generic compiler model may be organised according to different architectural models

Compiler model



Language processing system



Reference architectures

- Reference models are derived from a study of the application domain rather than from existing systems
- May be used as a basis for system implementation or to compare different systems. It acts as a standard against which systems can be evaluated
- OSI model is a layered model for communication systems

Interface design

- Internal and external interface design
- User interface design (novice, knowledgeable, expert)
- Implementation tools
- Interface design guideline
    - General interaction
    - Information display
    - Data input

Procedural design

- Structured programming
- Graphical design notation
- Tabular design notation
- Program design language

Refer to;

Distributed System Architectures
Application Architectures

References: *Pressman - Chap. 13, 14 , Sommerville – Chap. 10*

[**Next**] *Software Testing*

Design documentation

1. Scope
    a. System objectives
    b. Major software requirements
    c. Design constraints, limitations

2. Data Design
    a. Data objects and resultant data structures
    b. File and database structures
        i. external file structure
            1. logical structure
            2. logical record description
            3. access method
        ii. global data
        iii. file and data cross reference

3. Architectural Design
    a. Review of data and control flow
    b. Derived program structure (use any/combination of proper architectural styles and representation)

4. Interface Design
    a. Human-machine interface specification
    b. Human-machine interface design rules
    c. External interface design
        i. Interface to external data
        ii. Interface to external systems or devices
    d. Internal interface design rules

5. Procedural Design (if functional, use DFDs and flowcharts, and if object oriented concept, then use OOD representation)
    a. Processing narrative
    b. Interface description
    c. Design language description
    d. Modules used
    e. Internal data structures
    f. Comments/restrictions/limitations

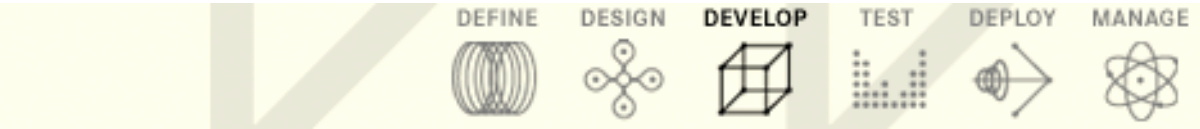
6. Requirements cross-reference

7. Test Provisions
    a. Test guidelines
    b. Integration strategy
    c. Special considerations

8. Special Notes

9. Appendices

☞ **Submit the design document one week before the final examination schedule.**



Software Engineering

Coding
Software Testing

Pramod Parajuli

Software Engineering **Day**

Coding

- Select proper language with enough library routines
- Use of code library

Coding standards

- Rules for limiting use of globals
- Contents of headers (name of module, data, author's name, modification history, synopsis, different functions supported, global variables accessed and modified, return types etc.)
- Naming convention for global variables, local variables, and constants
- strong or weak typing?
- references or pointers in interfaces?
- stream I/O or stdio?
- should C++ code call C code? vice versa?
- interfaces uniformly have a get() and/or set() member function for each data member?
- should interfaces be designed from the outside-in or the inside-out?
- should errors be handled by try/catch/throw or by return codes?

Coding guidelines

- do not use too clever and difficult to understand coding style
- avoid obscure side effects
- do not use on identifier for multiple purpose

Code walk-throughs

- tracing the code by development team
- focus on discovery of errors

Code inspections

- discovery of commonly made errors
- code is examined for certain kinds of errors

Use of uninitialized variables, jumps into loops, nonterminating loops, incompatible assignments, array indices out of bounds, improper storage allocation and deallocation, mismatches between actual and formal parameters, incorrect logical operators and precedence, modification of loop variables, comparison of equality of floating point values etc.

Documentation

- internal documentation (comments – 4X the lines of codes)
- external documentation

CASE tools

- use of case tools for documentation
- code generation, code reverse engineering (http://www.crystalflowcharts.com)

STL (Standard Template Libraries) - standard templates developed by standard organizations and authorities

CVS (Concurrent Versioning System) - automate the changes in code

Software Testing

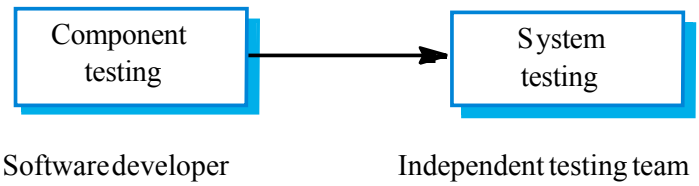
- process of executing a program with the intent of finding an error
- good test case has high probability of finding an as-yet undiscovered error
- a successful test is one that uncovers an as-yet undiscovered error

Testing principle

- all test should be traceable to customer requirements
- test should be planned long before testing begins
- Pareto principle applies to S/W testing
- Testing should begin 'in the small' and progress towards testing 'in large'
- Exhaustive testing is not possible

Types

- Unit/component/defect testing
- Mutation testing
- Integration testing
- System testing
  - Alpha testing
  - Beta testing
  - Acceptance testing



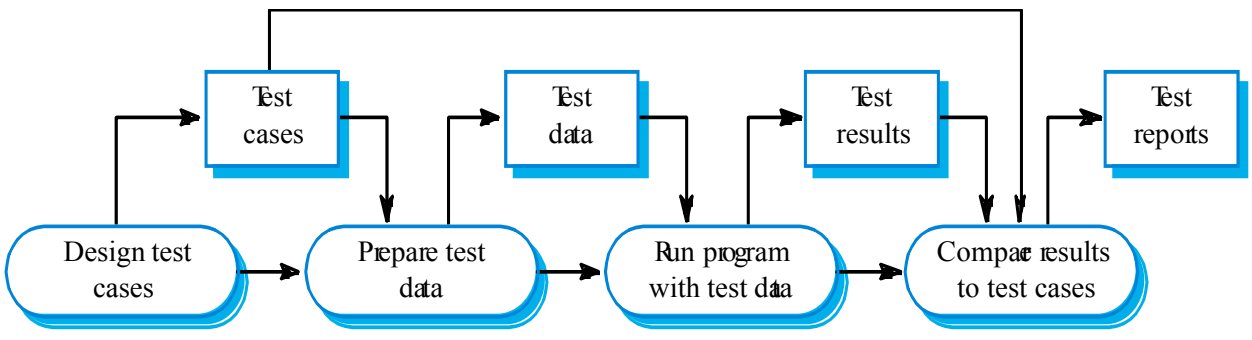
Testing priorities

- Only exhaustive testing can show a program is free from defects. However, exhaustive testing is impossible
- Tests should exercise a system's capabilities rather than its components
- Testing old capabilities is more important than testing new capabilities
- Testing typical situations is more important than boundary value cases

Test data and test cases

- *Test data* Inputs which have been devised to test the system
- *Test cases* Inputs to test the system and the predicted outputs from these inputs if the system operates according to its specification

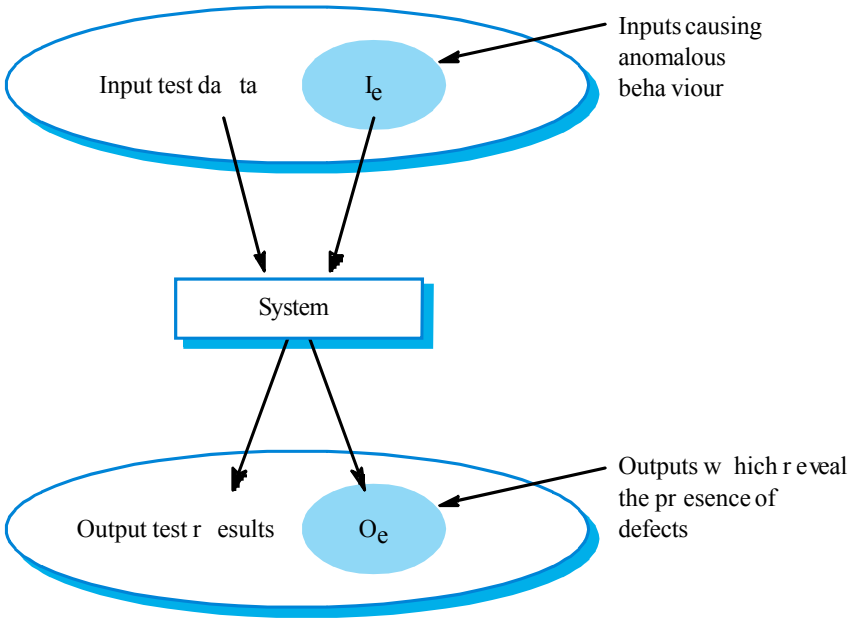
The defect (unit) testing process



Black-box testing

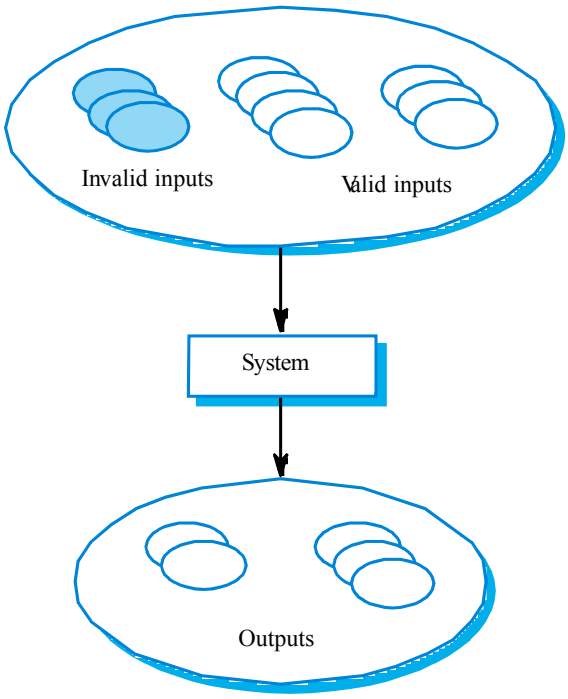
- An approach to testing where the program is considered as a 'black-box'
- The program test cases are based on the system specification
- Test planning can begin early in the software process

1. Equivalence class partitioning
2. Boundary value analysis



Equivalence partitioning

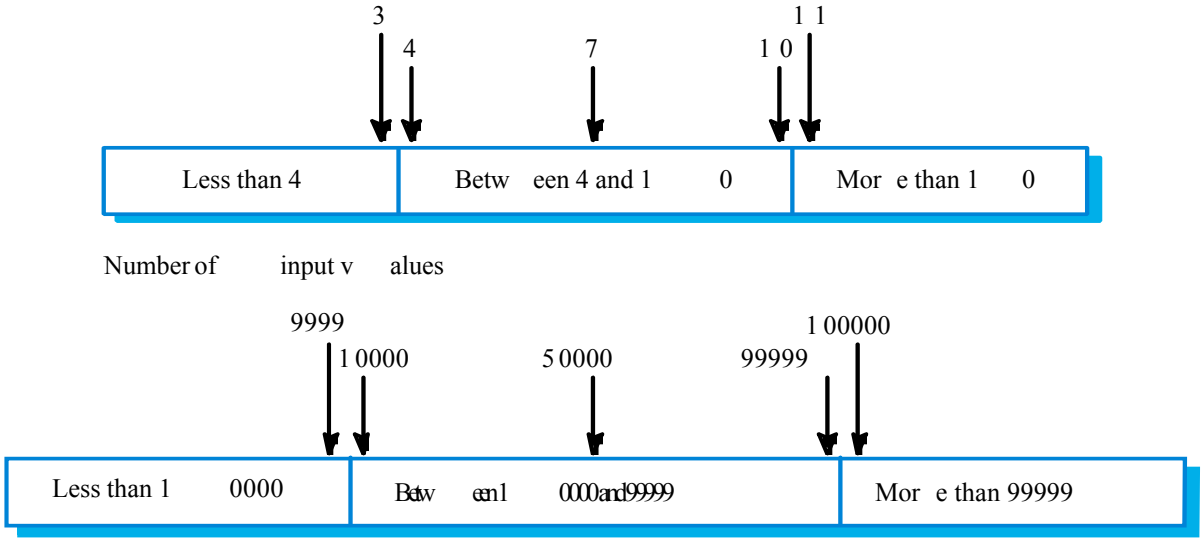
- Input data and output results often fall into different classes where all members of a class are related
- Each of these classes is an equivalence partition where the program behaves in an equivalent way for each class member
- Test cases should be chosen from each partition



- Partition system inputs and outputs into 'equivalence sets'
  - If input is a 5-digit integer between 10,000 and 99,999, equivalence partitions are <10,000, 10,000-99, 999 and > 10, 000

Choose test cases at the boundary of these sets

- 00000, 09999, 10000, 99999, 10001



Search routine specification

procedure Search (Key : ELEM ; T: ELEM_ARRAY;
Found : in out BOOLEAN; L: in out ELEM_INDEX) ;

Pre-condition
-- the array has at least one element
T'FIRST <= T'LAST
Post-condition
-- the element is found and is referenced by L
(Found and T (L) = Key)
or
-- the element is not in the array
(not Found and
not (exists i, T'FIRST >= i <= T'LAST, T (i) = Key))

Search routine - input partitions

- Inputs which conform to the pre-conditions
- Inputs where a pre-condition does not hold
- Inputs where the key element is a member of the array
- Inputs where the key element is not a member of the array

Testing guidelines (sequences)

- Test software with sequences which have only a single value
- Use sequences of different sizes in different tests
- Derive tests so that the first, middle and last elements of the sequence are accessed
- Test with sequences of zero length

Search routine - input partitions

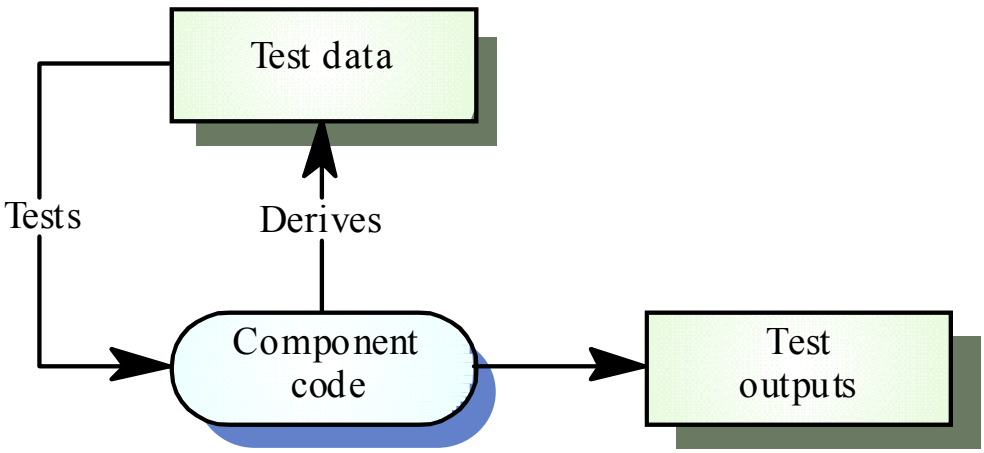
Array	Element
Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

Input sequence (T)	**Key** (Key)	**Output** (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

Structural testing

- Sometime called white-box testing (glass-box testing)
- Derivation of test cases according to program structure. Knowledge of the program is used to identify additional test cases
- Objective is to exercise all program statements (not all path combinations)

1. statement coverage
2. branch coverage
3. condition coverage
4. path coverage



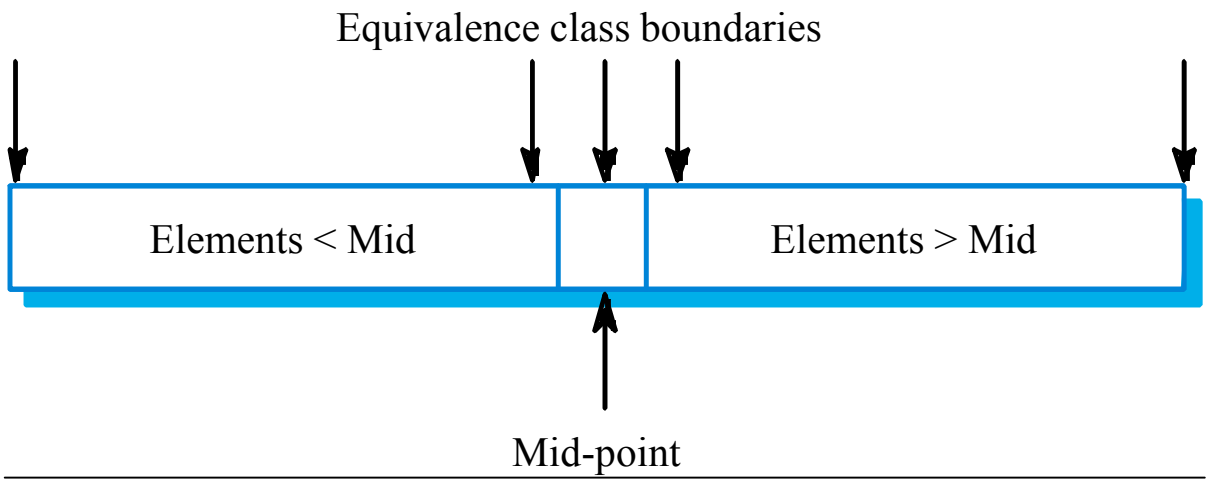
```
class BinSearch {

// This is an encapsulation of a binary search function that takes an array of
// ordered objects and a key and returns an object with 2 attributes namely
// index - the value of the array index
// found - a boolean indicating whether or not the key is in the array
// An object is returned because it is not possible in Java to pass basic types by
// reference to a function and so return two values
// the key is -1 if the element is not found

    public static void search ( int key, int [] elemArray, Result r )
    {
        int bottom = 0 ;
        int top = elemArray.length - 1 ;
        int mid ;
        r.found = false ; r.index = -1 ;
        while ( bottom <= top )
        {
            mid = (top + bottom) / 2 ;
            if (elemArray [mid] == key)
            {
                r.index = mid ;
                r.found = true ;
                return ;
            } // if part
            else
            {
                if (elemArray [mid] < key)
                    bottom = mid + 1 ;
                else
                    top = mid - 1 ;
            }
        } //while loop
    } // search
} //BinSearch
```

Binary search - equiv. partitions

- Pre-conditions satisfied, key element in array
- Pre-conditions satisfied, key element not in array
- Pre-conditions unsatisfied, key element in array
- Pre-conditions unsatisfied, key element not in array
- Input array has a single value
- Input array has an even number of values
- Input array has an odd number of values



Input array (T)	**Key** (Key)	**Output** (Found, L)
17	17	true, 1
17	0	false, ??
17, 21, 23, 29	17	true, 1
9, 16, 18, 30, 31, 41, 45	45	true, 7
17, 18, 21, 23, 29, 38, 41	23	true, 4
17, 18, 21, 23, 29, 33, 38	21	true, 3
12, 18, 21, 23, 32	23	true, 4
21, 23, 29, 33, 38	25	false, ??

Path testing

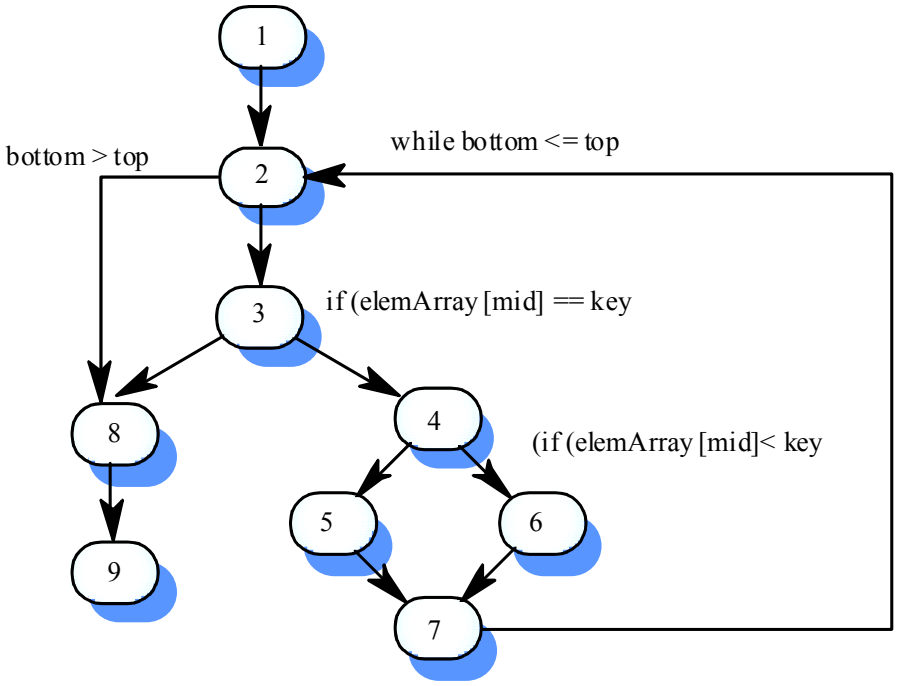
- The objective of path testing is to ensure that the set of test cases is such that each path through the program is executed at least once
- The starting point for path testing is a program flow graph that shows nodes representing program decisions and arcs representing the flow of control
- Statements with conditions are therefore nodes in the flow graph

Program flow graphs

- Describes the program control flow. Each branch is shown as a separate path and loops are shown by arrows looping back to the loop condition node
- Used as a basis for computing the cyclomatic complexity
- Cyclomatic complexity = Number of edges - Number of nodes +2

Cyclomatic complexity

- The number of tests to test all control statements equals the cyclomatic complexity
- Cyclomatic complexity equals number of conditions in a program
- Useful if used with care. Does not imply adequacy of testing.
- Although all paths are executed, all combinations of paths are not executed



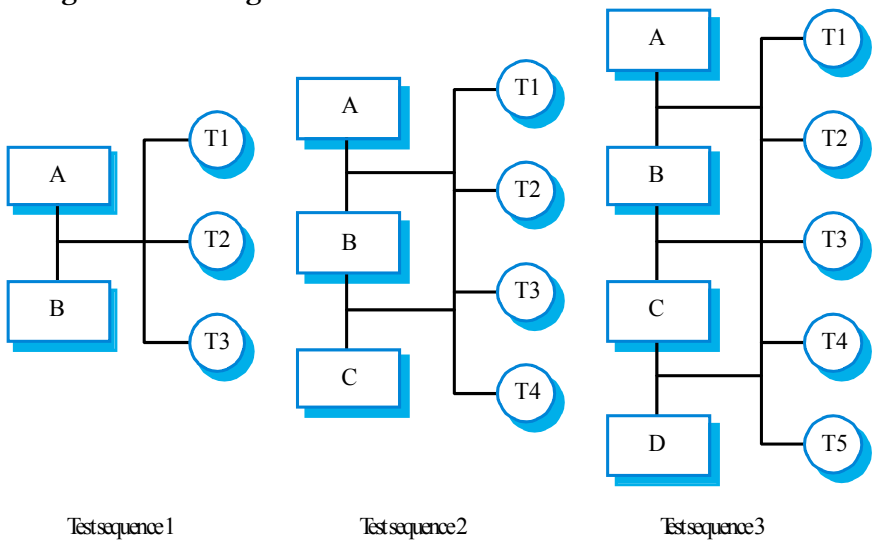
Independent paths

- 1, 2, 3, 8, 9
- 1, 2, 3, 4, 6, 7, 2
- 1, 2, 3, 4, 5, 7, 2
- 1, 2, 3, 4, 6, 7, 2, 8, 9
- Test cases should be derived so that all of these paths are executed
- A dynamic program analyser may be used to check that paths have been executed

Integration testing

- Tests complete systems or subsystems composed of integrated components
- Integration testing should be black-box testing with tests derived from the specification
- Main difficulty is localising errors
- Incremental integration testing reduces this problem

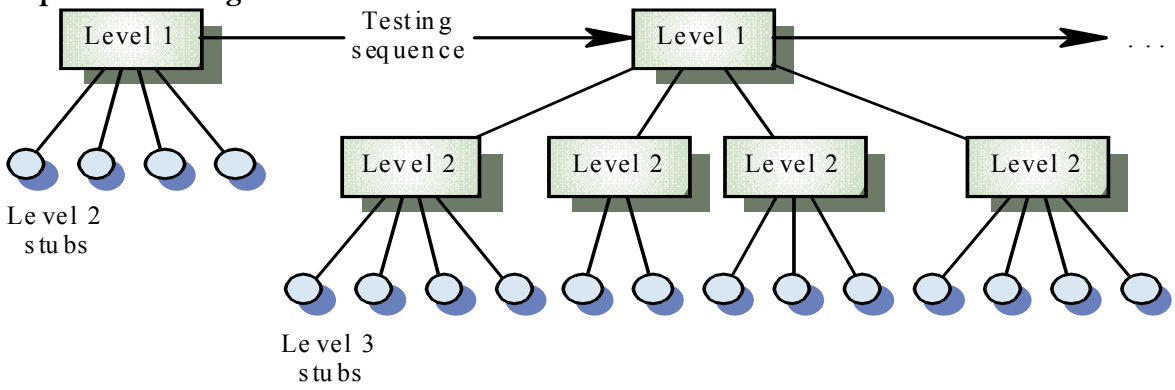
Incremental integration testing



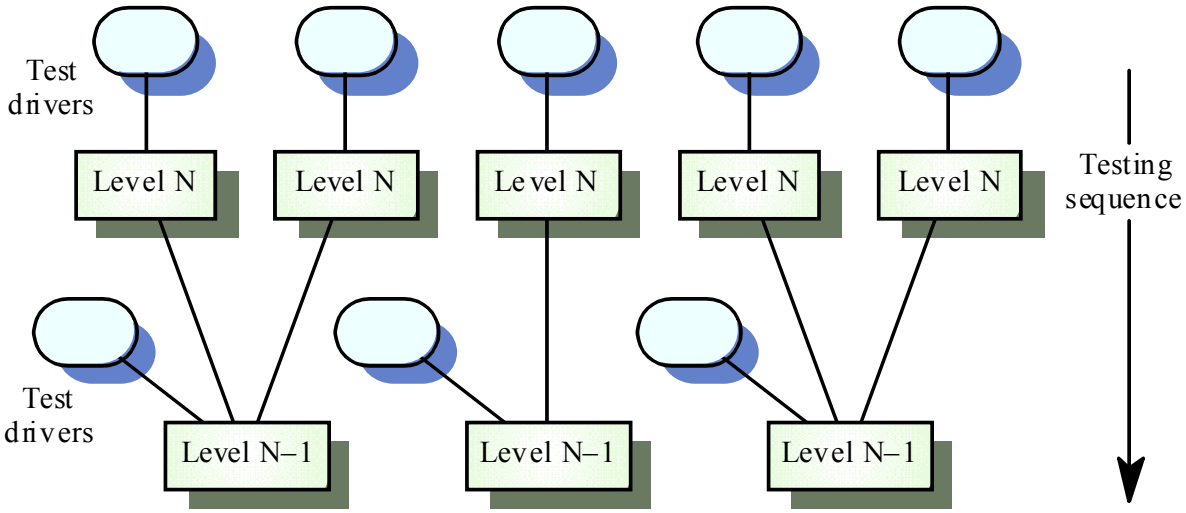
Approaches to integration testing

- Big-bang integration testing
- Top-down testing
  - Start with high-level system and integrate from the top-down replacing individual components by stubs where appropriate
- Bottom-up testing
  - Integrate individual components in levels until the complete system is created
- In practice, most integration involves a combination of these strategies

Top-down testing



Bottom-up testing

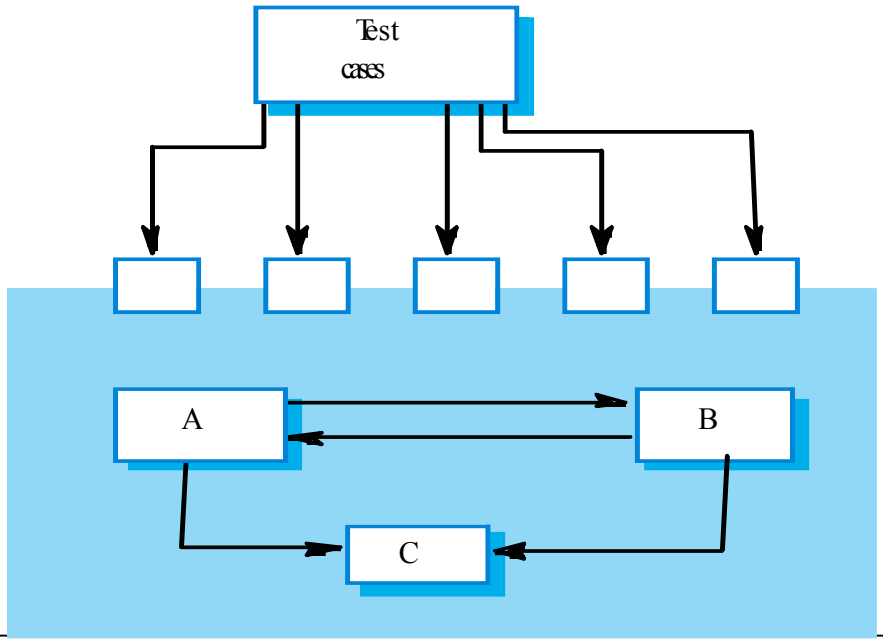


Testing approaches

- Architectural validation
  - Top-down integration testing is better at discovering errors in the system architecture
- System demonstration
  - Top-down integration testing allows a limited demonstration at an early stage in the development
- Test implementation
  - Often easier with bottom-up integration testing
- Test observation
  - Problems with both approaches. Extra code may be required to observe tests

Interface testing

- Takes place when modules or sub-systems are integrated to create larger systems
- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces
- Particularly important for object-oriented development as objects are defined by their interfaces



Interfaces types

- Parameter interfaces
  - Data passed from one procedure to another
- Shared memory interfaces
  - Block of memory is shared between procedures
- Procedural interfaces
  - Sub-system encapsulates a set of procedures to be called by other sub-systems
- Message passing interfaces
  - Sub-systems request services from other sub-systems

Interface errors

- Interface misuse
  - A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order
- Interface misunderstanding
  - A calling component embeds assumptions about the behaviour of the called component which are incorrect
- Timing errors
  - The called and the calling component operate at different speeds and out-of-date information is accessed

Interface testing guidelines

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges
- Always test pointer parameters with null pointers
- Design tests which cause the component to fail
- Use stress testing in message passing systems
- In shared memory systems, vary the order in which components are activated

System Testing

- Alpha testing – test carried by test team within the organization
- Beta testing – testing performed by a selected group of friendly customers
- Acceptance testing – performed by customer to determine whether or not to accept the delivery of the system

Stress testing

- Exercises the system beyond its maximum design load. Stressing the system often causes defects to come to light
- Stressing the system test failure behaviour. Systems should not fail catastrophically. Stress testing checks for unacceptable loss of service or data
- Particularly relevant to distributed systems which can exhibit severe degradation as a network becomes overloaded

Error seeding

- Seed the code with some known errors
- Expect more errors to be uncovered

Object-oriented testing

- The components to be tested are object classes that are instantiated as objects
- Larger grain than individual functions so approaches to white-box testing have to be extended
- No obvious 'top' to the system for top-down integration and testing

Testing levels

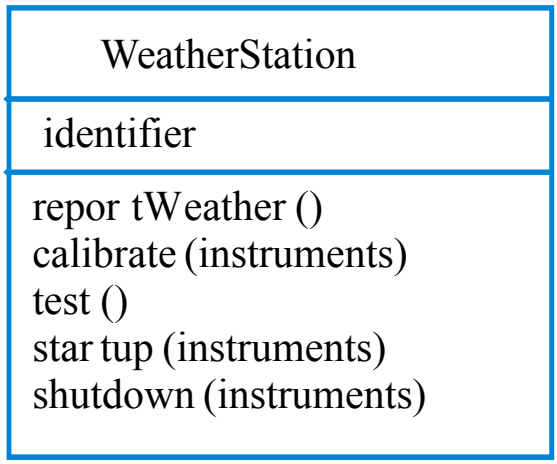
- Testing operations associated with objects
- Testing object classes
- Testing clusters of cooperating objects
- Testing the complete OO system

Object class testing

- Complete test coverage of a class involves
  - Testing all operations associated with an object
  - Setting and interrogating all object attributes
  - Exercising the object in all possible states
- Inheritance makes it more difficult to design object class tests as the information to be tested is not localised

Weather station object interface

- Test cases are needed for all operations
- Use a state model to identify state transitions for testing
- Examples of testing sequences
  - Shutdown -> Waiting -> Shutdown
  - Waiting -> Calibrating -> Testing -> Transmitting -> Waiting
  - Waiting -> Collecting -> Waiting -> Summarising -> Transmitting -> Waiting



Object integration

- Levels of integration are less distinct in object-oriented systems
- Cluster testing is concerned with integrating and testing clusters of cooperating objects
- Identify clusters using knowledge of the operation of objects and the system features that are implemented by these clusters

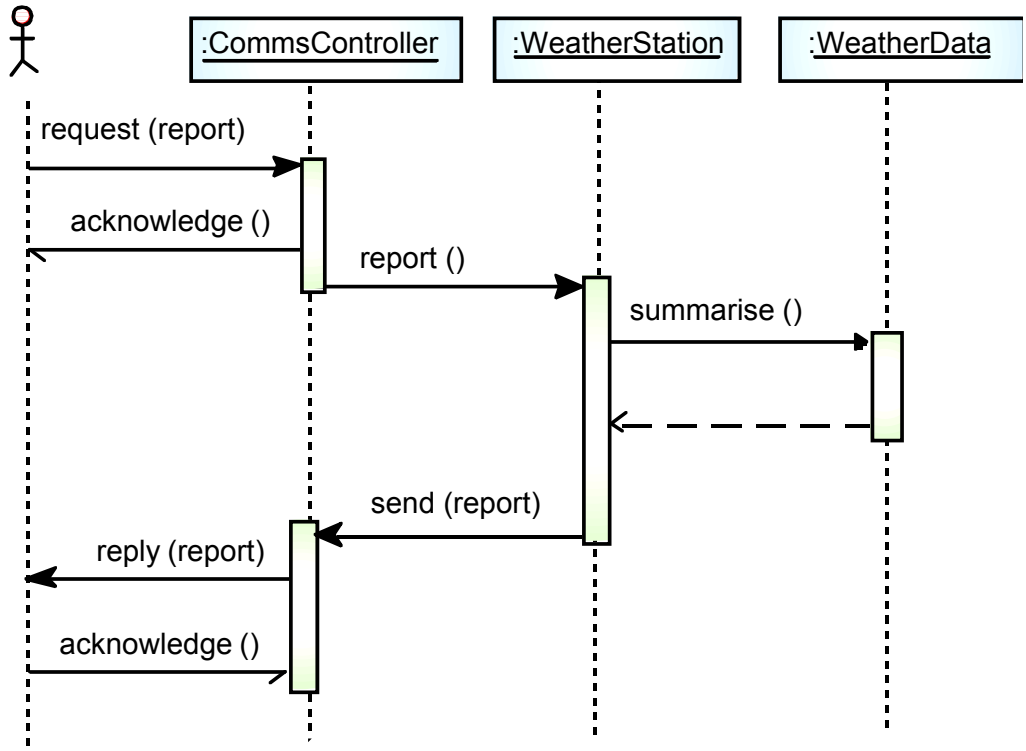
Approaches to cluster testing

- Use-case or scenario testing
  - Testing is based on a user interactions with the system
  - Has the advantage that it tests system features as experienced by users
- Thread testing
  - Tests the systems response to events as processing threads through the system
- Object interaction testing
  - Tests sequences of object interactions that stop when an object operation does not call on services from another object

Scenario-based testing

- Identify scenarios from use-cases and supplement these with interaction diagrams that show the objects involved in the scenario
- Consider the scenario in the weather station system where a report is generated

Collect weather data



Weather station testing

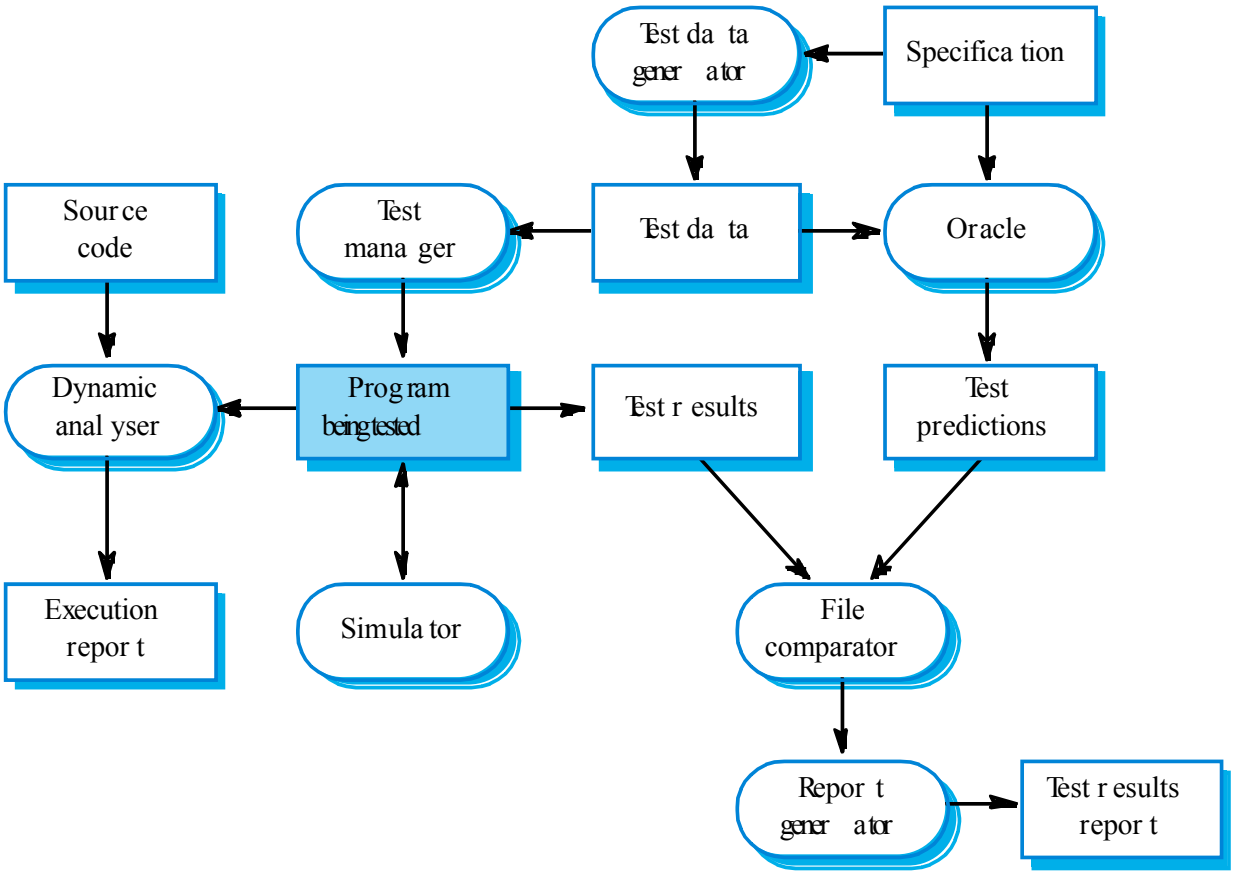
- Thread of methods executed
  - CommsController:request -> WeatherStation:report -> WeatherData:summarise
- Inputs and outputs
  - Input of report request with associated acknowledge and a final output of a report
  - Can be tested by creating raw data and ensuring that it is summarised properly
  - Use the same raw data to test the WeatherData object

Testing documentation and help facilities

- Does the documentation accurately describe how to accomplish each mode of use?
- Is the description of each interaction sequence accurate?
- Are examples accurate?
- Are terminology, menu descriptions, and system responses consistent with the actual program?
- Is it relatively easy to locate guidance within the documentation?
- Can troubleshooting be accomplished easily with the documentation?
- Are all error messages displayed for the use described in more detail in the document?
- etc.

Testing workbenches

- Testing is an expensive process phase. Testing workbenches provide a range of tools to reduce the time required and total testing costs
- Most testing workbenches are open systems because testing needs are organisation-specific
- Difficult to integrate with closed design and analysis workbenches

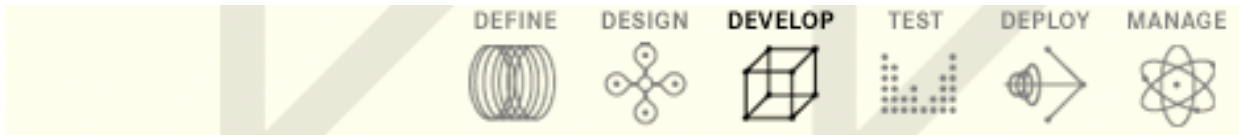


Testing workbench adaptation

- Scripts may be developed for user interface simulators and patterns for test data generators
- Test outputs may have to be prepared manually for comparison
- Special-purpose file comparators may be developed

References: *Pressman – Chap. 17, 18, Sommerville – Chap. 20*

[**Next**] *S/W Reuse, S/W Re-engineering, OOSE*



Software Engineering

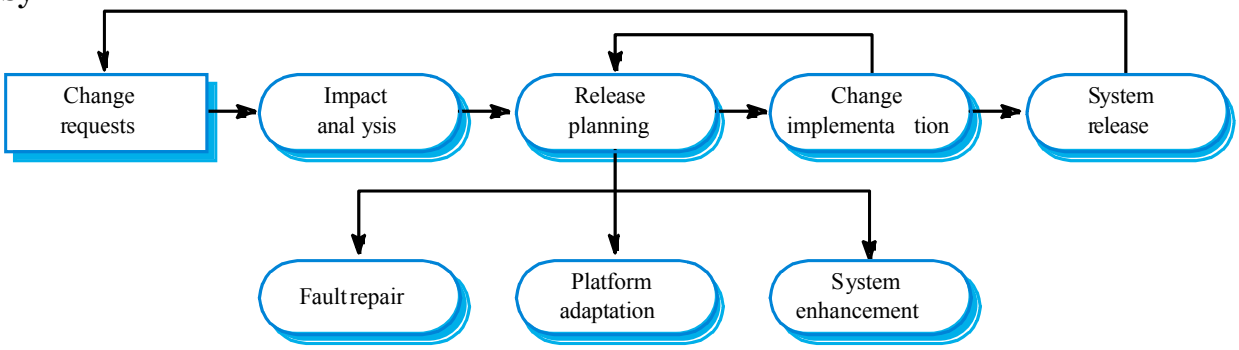
Software Re-engineering

Pramod Parajuli

Software Re-engineering

- Software change is inevitable
- A key problem for organisations is implementing and managing change to their existing software systems
- **Program evolution dynamics** is the study of the processes of system change

System Evolution Process



Types of maintenance

- Maintenance to repair software faults
- Maintenance to adapt software to a different operating environment
- Maintenance to add to or modify the system's functionality

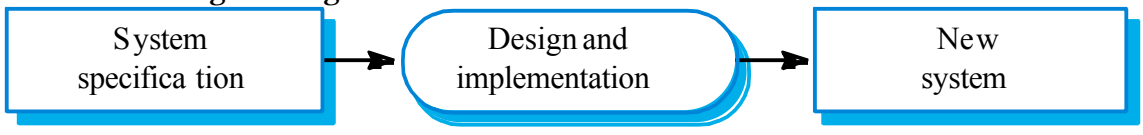
System re-engineering

- Re-structuring or re-writing part or all of a legacy system without changing its functionality.
- Applicable where some but not all sub-systems of a larger system require frequent maintenance.
- Re-engineering involves adding effort to make them easier to maintain. The system may be re-structured and re-documented.

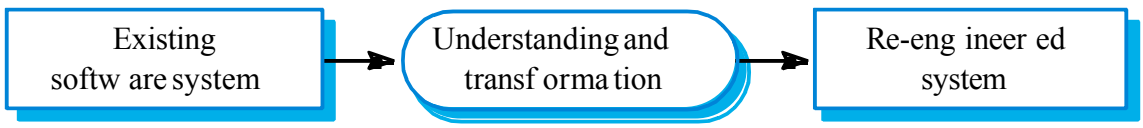
Advantages of reengineering

- Reduced risk
  - There is a high risk in new software development. There may be development problems, staffing problems and specification problems.
- Reduced cost
  - The cost of re-engineering is often significantly less than the costs of developing new software.

Forward and re-engineering

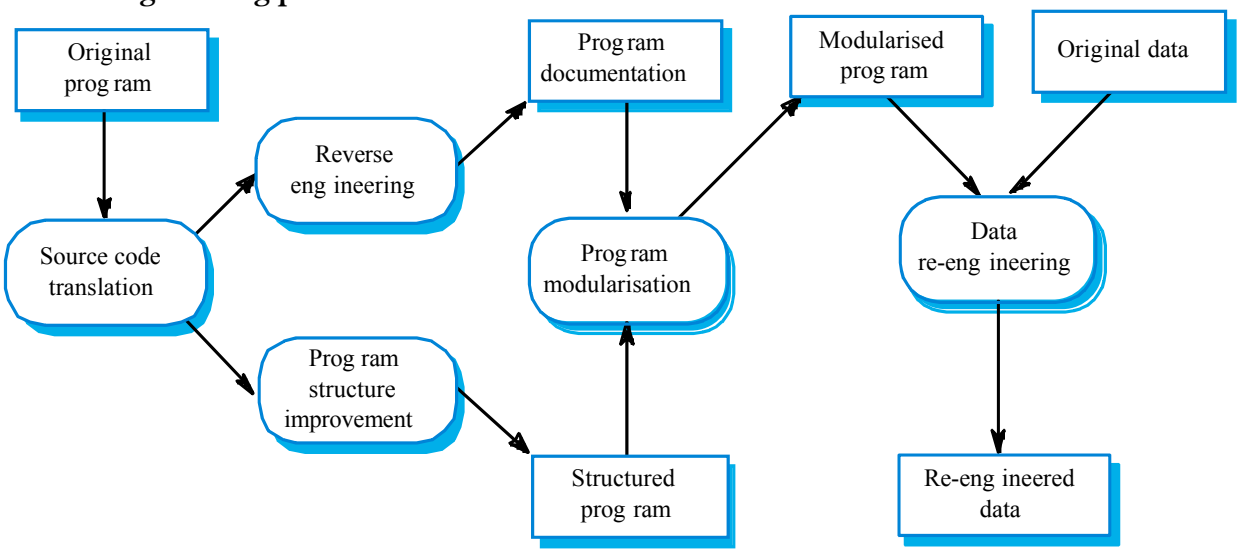


Forward eng ineering



Softw are re-eng ineering

The re-engineering process



Reengineering process activities

- Source code translation
  - Convert code to a new language.
- Reverse engineering
  - Analyze the program to understand it;
- Program structure improvement
  - Restructure automatically for understandability;
- Program modularization
  - Re-organize the program structure;
- Data reengineering
  - Clean-up and restructure system data.

Re-engineering approaches

Automated program restructuring

Program and data restructuring

Automated source code conversion

Automated restructuring with manual changes

Restructuring plus architectural changes

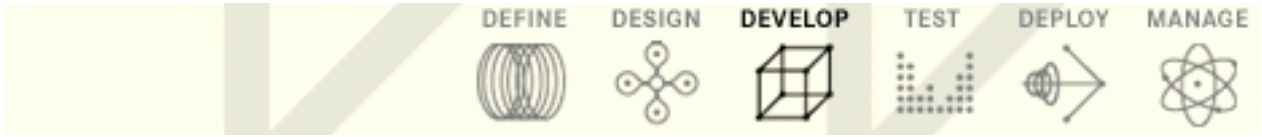
Increased cost

Reengineering cost factors

- The quality of the software to be reengineered.
- The tool support available for reengineering.
- The extent of the data conversion which is required.
- The availability of expert staff for reengineering.
  - This can be a problem with old systems based on technology that is no longer widely used.

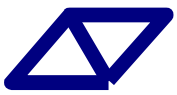
References: *Sommerville (7) – Chap. 21*

[**Next**] *S/W Reuse*



Software Engineering

Software Reuse



Pramod Parajuli

Software Reuse

- In most engineering disciplines, systems are designed by composing existing components that have been used in other systems.
- Software engineering has been more focused to achieve better software, more quickly and at lower cost, therefore we need to adopt a design process that is based on ***systematic software reuse***.

Reuse-based software engineering (reuse practice)

- Application system reuse
- Component reuse
- Object and function reuse

Benefits of reuse

- Increased reliability - Components exercised in working systems
- Reduced process risk - Less uncertainty in development costs
- Effective use of specialists - Reuse components instead of people
- Standards compliance - Embed standards in reusable components
- Accelerated development - Avoid original development and hence speed-up production

Requirements for design with reuse

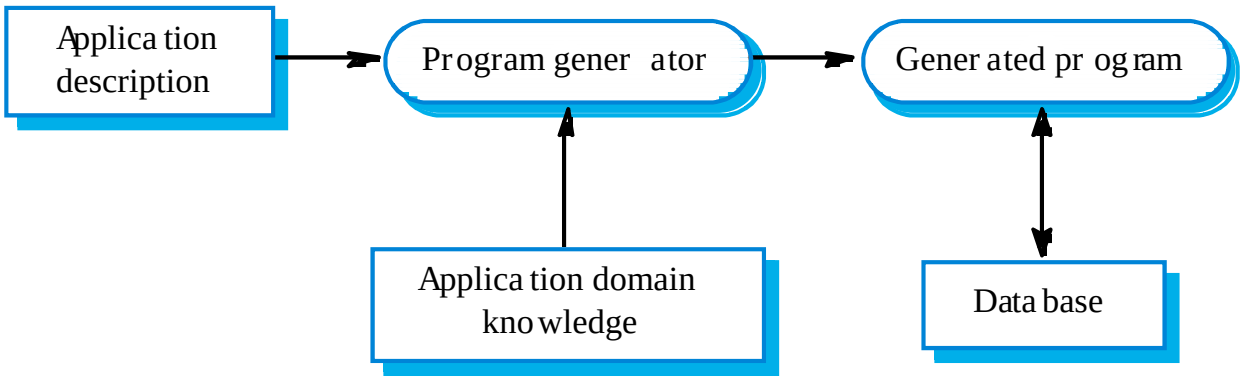
- Need to find appropriate reusable components
- Reuser must have confidence over reliability of the component being reused
- Components must be well documented so that they can be understood and, where appropriate, modified

Reuse problems

- Increased maintenance costs
- Lack of tool support
- Not-invented-here syndrome
- Maintaining a component library
- Finding and adapting reusable components

Generator-based reuse (alternative to the component oriented view of reuse)

- Program generators involve the reuse of standard patterns and algorithms
- These are embedded in the generator and parameterised by user commands. A program is then automatically generated
- Generator-based reuse is possible when domain abstractions and their mapping to executable code can be identified
- A domain specific language is used to compose and control these abstractions

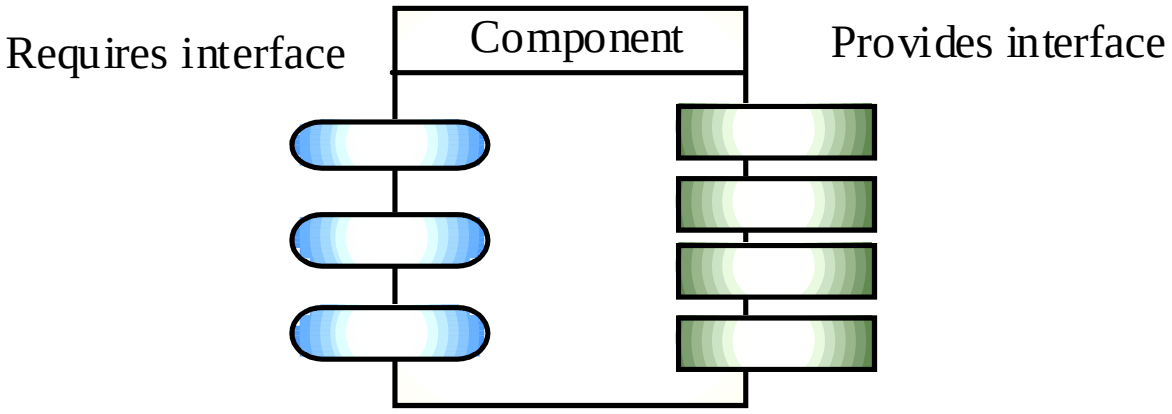


Component-based development (CBSE)

- It emerged from the failure of object-oriented development to support effective reuse. Single object classes are too detailed and specific
- Components are more abstract than object classes and can be considered to be stand-alone service providers

Components

- Components provide a service without regard to where the component is executing or its programming language
    - A component is an independent executable entity that can be made up of one or more executable objects
    - The component interface is published and all interactions are through the published interface
- Components can range in size from simple functions to entire application systems



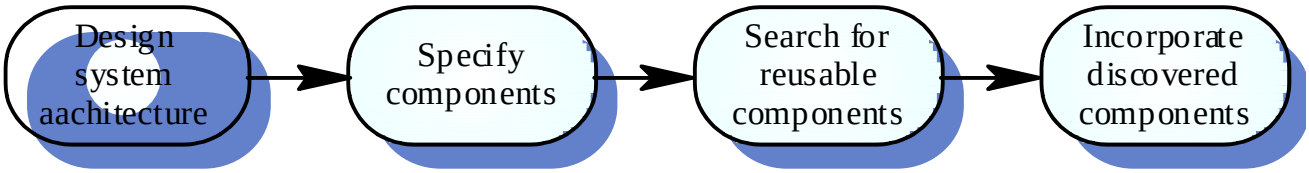
Component abstractions

- *Functional abstraction*
    - The component implements a single function such as a mathematical function
- *Casual groupings*
    - The component is a collection of loosely related entities that might be data declarations, functions, etc.
- *Data abstractions*
    - The component represents a data abstraction or class in an object-oriented language
- *Cluster abstractions*
    - The component is a group of related classes that work together
- *System abstraction*
    - The component is an entire self-contained system

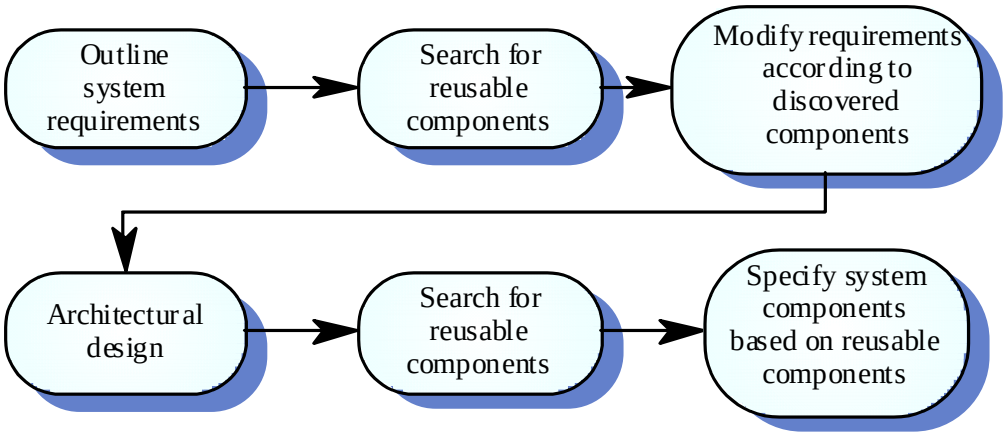
CBSE processes

- Integrated into a standard software process by incorporating a reuse activity
- The system requirements are modified to reflect the components that are available
- CBSE usually involves a prototyping or an incremental development process with components being 'glued together' using a scripting language

An opportunistic reuse process



Development with reuse



CBSE problems

- Component incompatibilities - cost and schedule savings are less then expected
- Finding and understanding components
- Managing evolution as requirements change in situations where it may be impossible to change the system components

Application frameworks

- Frameworks are a sub-system design made up of a collection of abstract and concrete classes and the interfaces between them
- The sub-system is implemented by adding components to fill in parts of the design and by instantiating the abstract classes in the framework
- Frameworks are moderately large entities that can be reused

Framework classes

- System infrastructure frameworks
  - Support the development of system infrastructures such as communications, user interfaces and compilers
- Middleware integration frameworks
  - Standards and classes that support component communication and information exchange
- Enterprise application frameworks
  - Support the development of specific types of application such as telecommunications or financial systems

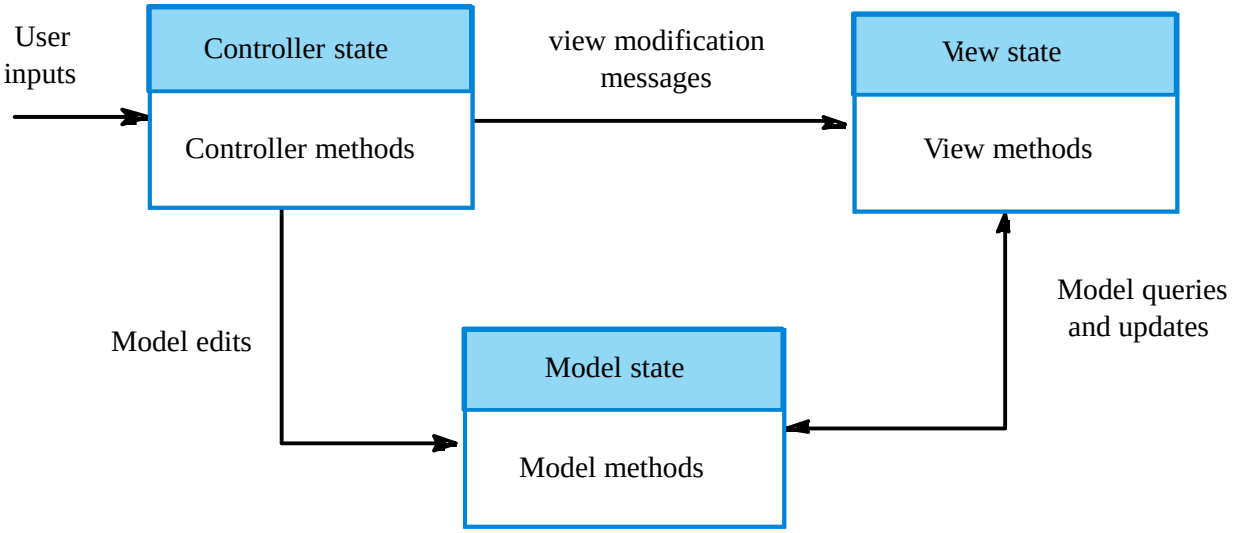
Extending frameworks

- Frameworks are generic and are extended to create a more specific application or sub-system
- Extending the framework involves
  - Adding concrete classes that inherit operations from abstract classes in the framework

    - Adding methods that are called in response to events that are recognised by the framework
- Problem with frameworks is their complexity and the time it takes to use them effectively

Model-view controller

- System infrastructure framework for GUI design
- Allows for multiple presentations of an object and separate interactions with these presentations
- MVC framework involves the instantiation of a number of patterns (discussed later)



COTS product reuse

- COTS - Commercial Off-The-Shelf systems
- COTS systems are usually complete application systems that offer an API (Application Programming Interface)
- Building large systems by integrating COTS systems is now a viable development strategy for some types of system such as E-commerce systems

COTS system integration problems

- Lack of control over functionality and performance
- Problems with COTS system inter-operability
- No control over system evolution
- Support from COTS vendors

Component development for reuse

- Components for reuse may be specially constructed by generalising existing components
- Component reusability
    - Should reflect stable domain abstractions
    - Should hide state representation
    - Should be as independent as possible
    - Should publish exceptions through the component interface
- There is a trade-off between reusability and usability.
    - The more general the interface, the greater the reusability but it is then more complex and hence less usable

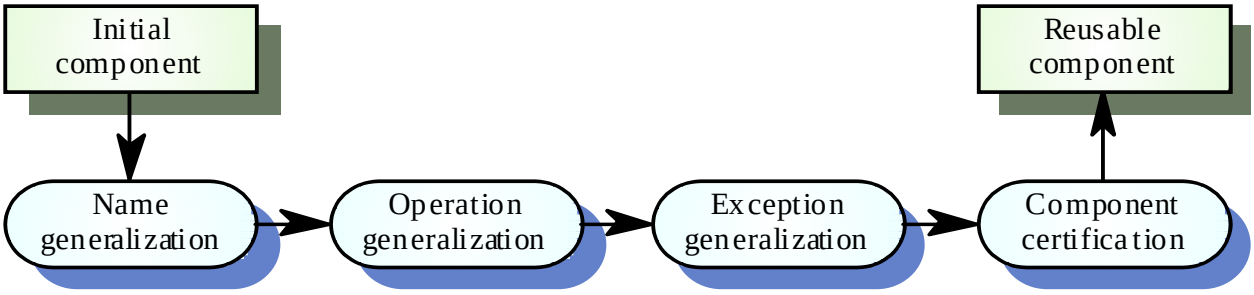
Reusable components

- The development cost of reusable components is higher than the cost of specific equivalents.
- This extra reusability enhancement cost should be an organization rather than a project cost
- Generic components may be less space-efficient and may have longer execution times than their specific equivalents

Reusability enhancement

- Name generalisation
  - Names in a component may be modified so that they are not a direct reflection of a specific application entity
- Operation generalisation
  - Operations may be added to provide extra functionality and application specific operations may be removed
- Exception generalisation
  - Application specific exceptions are removed and exception management added to increase the robustness of the component
- Component certification
  - Component is certified as reusable

Reusability enhancement process



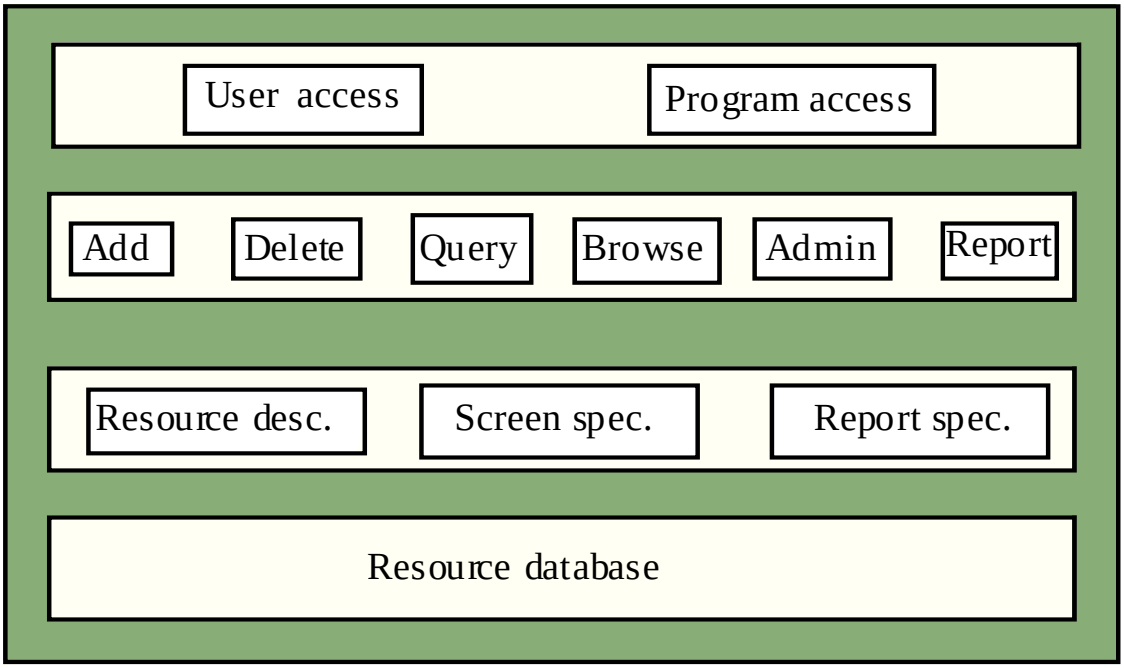
Application families

- An application family or product line is a related set of applications that has a common, domain-specific architecture
- The common core of the application family is reused each time a new application is required
- Each specific application is specialised in some way

Application family specialisation

- Platform specialisation
  - Different versions of the application are developed for different platforms
- Configuration specialisation
  - Different versions of the application are created to handle different peripheral devices
- Functional specialisation
  - Different versions of the application are created for customers with different requirements

A resource management system



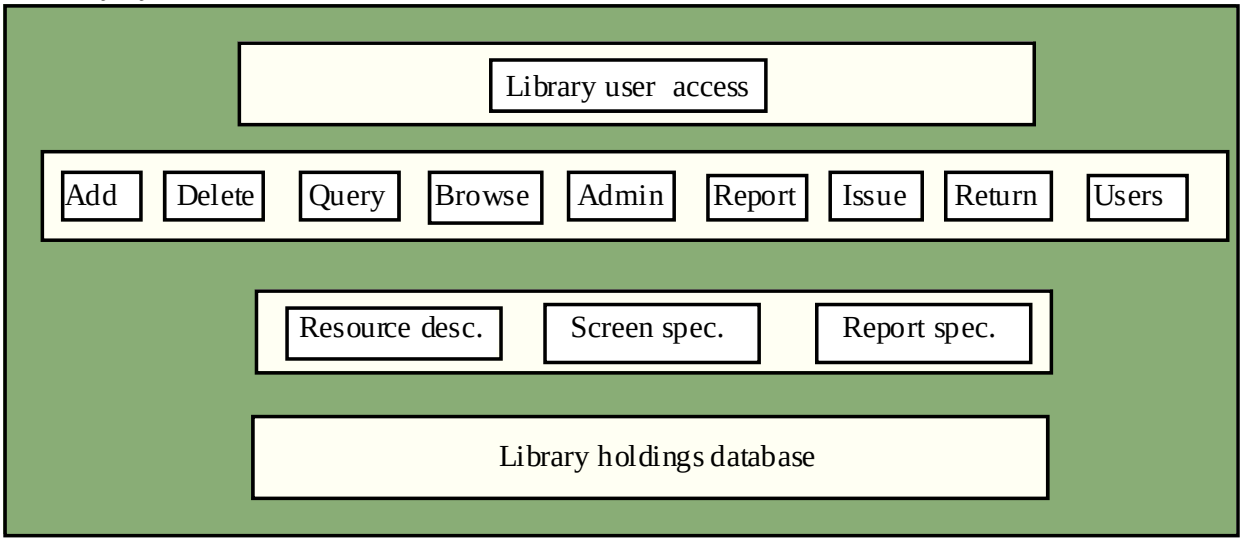
Inventory management systems

- Resource database
  - Maintains details of the things that are being managed
- I/O descriptions
  - Describes the structures in the resource database and input and output formats that are used
- Query level
  - Provides functions implementing queries over the resources
- Access interfaces
  - A user interface and an application programming interface

Application family architectures

- Architectures must be structured in such a way to separate different sub-systems and to allow them to be modified
- The architecture should also separate entities and their descriptions and the higher levels in the system access entities through descriptions rather than directly

A library system



- The resources being managed are the books in the library
- Additional domain-specific functionality (issue, borrow, etc.) must be added for this application

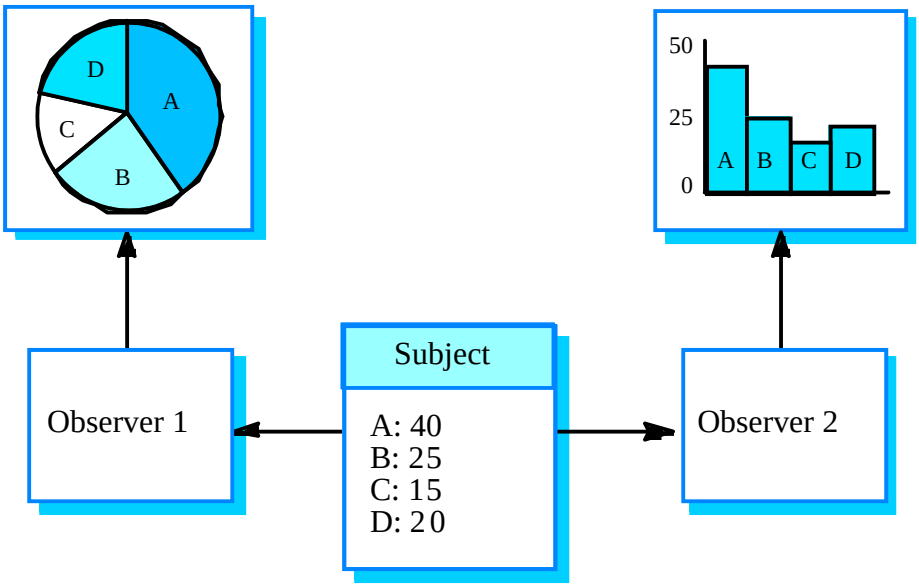
Design patterns

- A design pattern is a way of reusing abstract knowledge about a problem and its solution
- A pattern is a description of the problem and the essence of its solution
- It should be sufficiently abstract to be reused in different settings
- Patterns often rely on object characteristics such as inheritance and polymorphism

Pattern elements

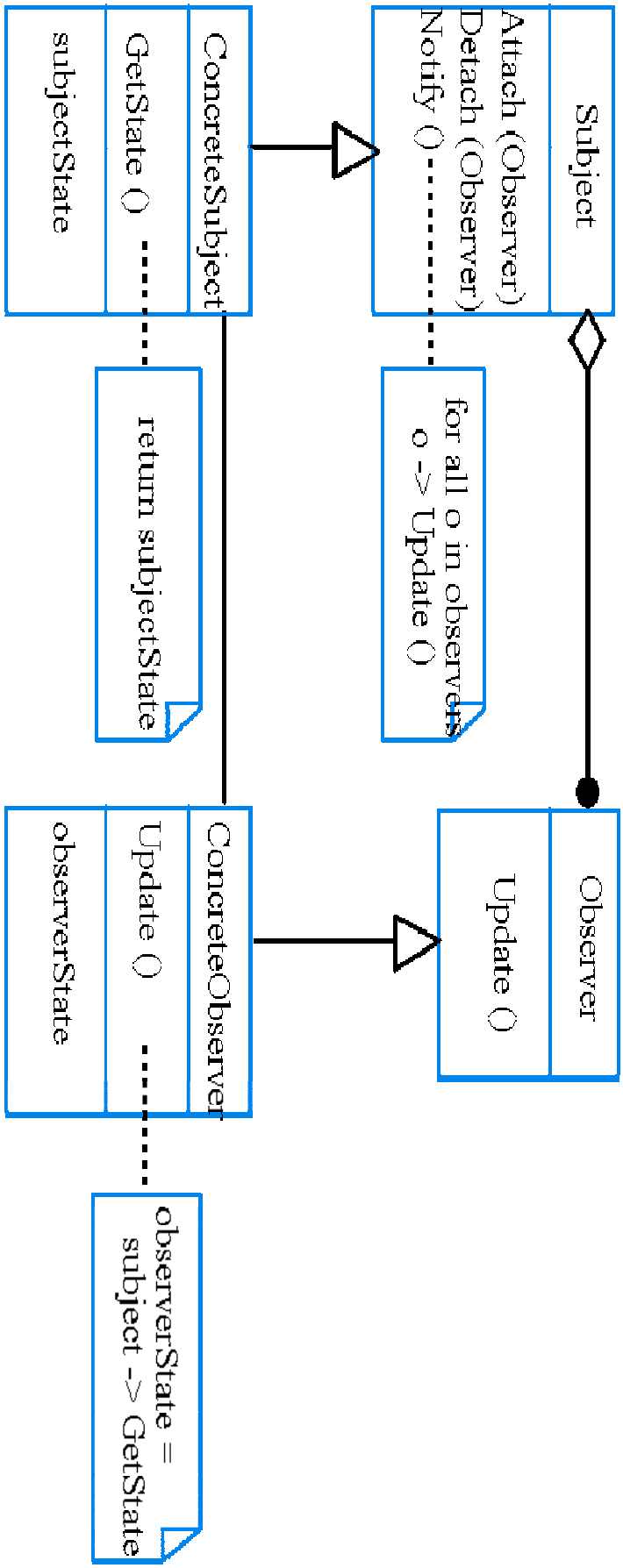
- Name
  - A meaningful pattern identifier
- Problem description
- Solution description
  - Not a concrete design but a template for a design solution that can be instantiated in different ways
- Consequences
  - The results and trade-offs of applying the pattern

Multiple displays



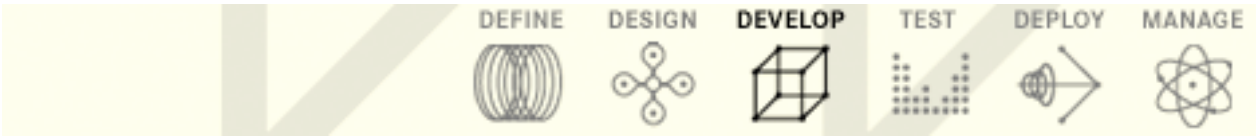
The Observer pattern

- Name
  - Observer.
- Description
  - Separates the display of object state from the object itself.
- Problem description
  - Used when multiple displays of state are needed.
- Solution description
  - See slide with UML description.
- Consequences
  - Optimisations to enhance display performance are impractical.



References: *Sommerville(6) – Chap. 14*

[**Next**] *OOSE*



Software Engineering

Object-Oriented Software Engineering

OO Concepts and Principles
OO Analysis
OO Design
OO Coding
OO Testing

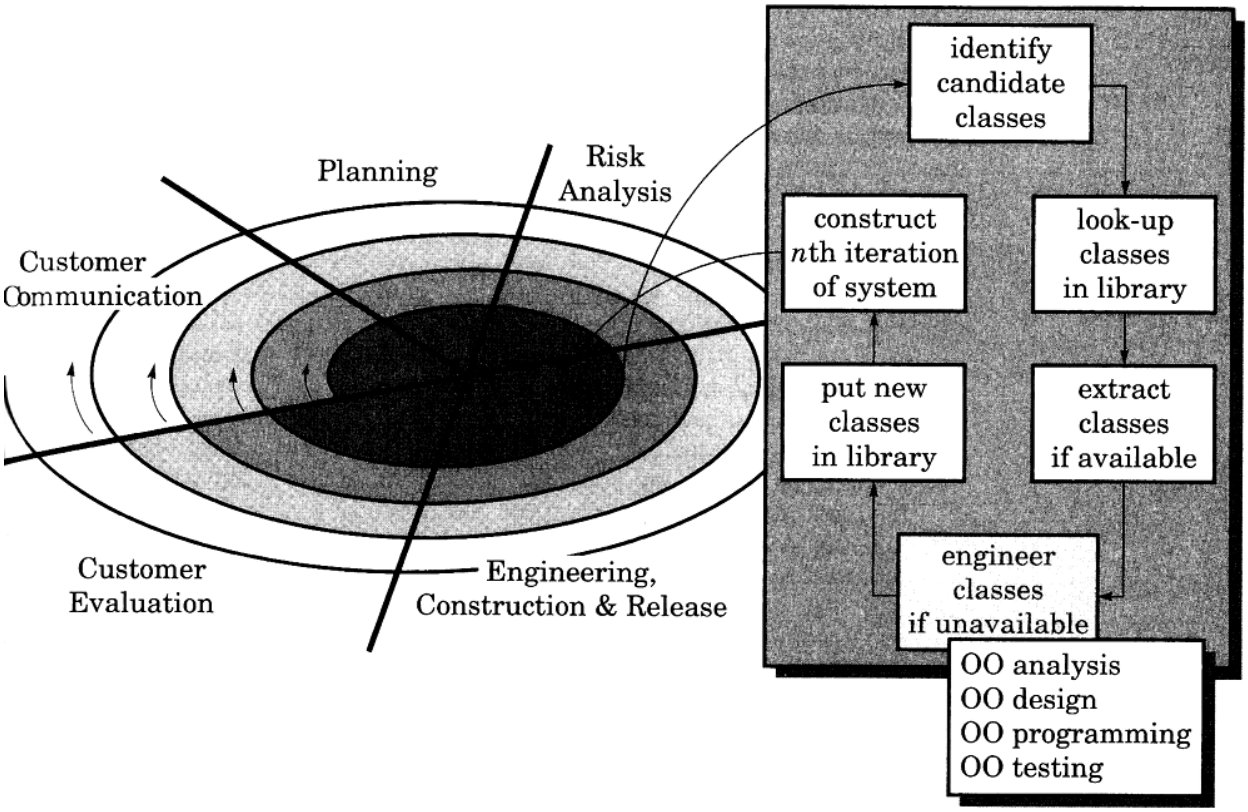
Pramod Parajuli

Object Oriented Concepts and Principles

- Real world – composed of objects
- Automation System/Software is a mapping function of input to the output
- Realization of the mapping is more clear with object representation
- Popular due to the concept of **'reuse'**

Object Oriented Paradigm

Object Oriented process model



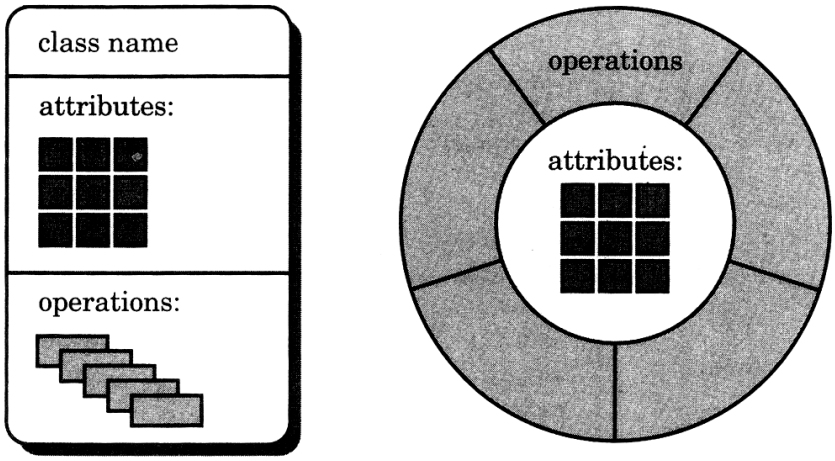
1. Identify the objects
2. Find attributes and methods of the object
3. Combine the general attributes and methods to create a class
4. If specific attributes and/or methods required for particular object, then create the sub-class of the generalized class (inheritance)

Formal definition of Object Oriented

Object-oriented = objects + classification + inheritance + communication

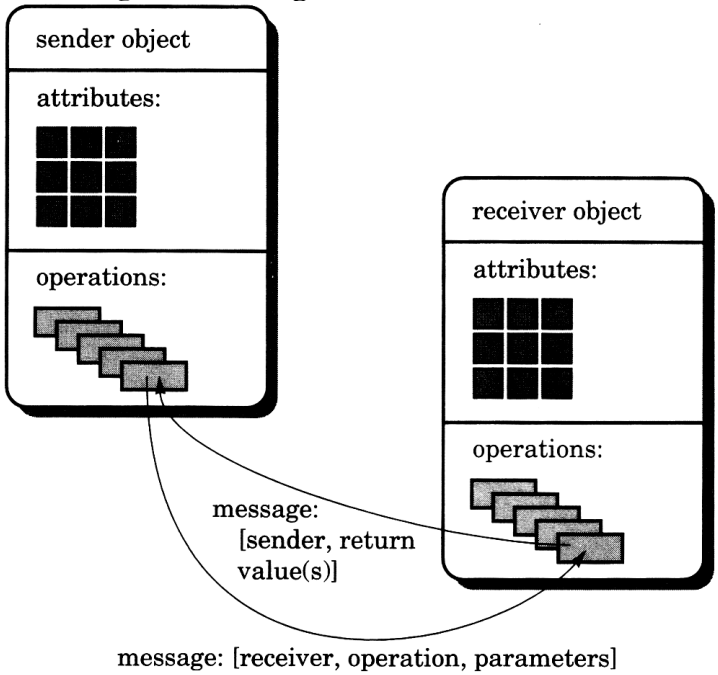
Class and Objects

- Class encapsulates data and procedural abstractions that describe content and behavior of some real world entity
- Class – work as template, pattern, blueprint
- **Superclass** – collection of classes
- **Subclass** – instance of a class
- Superclass and subclass exist only if there is a **class hierarchy**



Operations/Methods/Services

- Provided by the object to the service requestor (another object)
- The service requestor must use proper interface
- Communication through the **messages**



Encapsulation

- Internal implementation details of data and procedures are hidden – reduces the propagation of side effects
- Data structures and operations that manipulate them are highly cohesive – this facilitates component reuse
- Interfaces among encapsulated objects are simplified.

Inheritance

- Inheriting attributes and operations
- Super classes can be extended by sub-classes
- Many types of inheritance
- After creating sub-classes, class hierarchy might need to be restructured

Polymorphism

- Many forms

- Same object serves for many purposes
- Function overriding

Identifying classes and objects
Objects can be;

- External entities – devices, people.. that produce or consume information
- Things – reports, displays, letters.. that are part of information
- Occurrences/events – transaction..
- Roles – engineer, manager, .. played by people
- Organizational units – division, group, ..
- Places – manufacturing block, ..
- Structures – sensors, vehicles, computers ..

While identifying an object, following characteristics must be included;

- Retained information
- Needed services
- Multiple attributes
- Common attributes
- Common operations
- Essential requirements

Specifying Attributes

- Need to derive meaningful set of attributes

e.g. identification info. = system ID + verification phone number + system status

Defining Operations

- Operations define behavior of an object and change the attributes of an object
- Three types of operations
    - Data manipulation
    - Computation
    - Monitoring an object for occurrence of a controlling event

Common process framework for OO

- Do enough analysis to isolate major problem classes
- Do little design to determine implementation of classes and their communication
- Extract reusable classes from the library and build a rough prototype
- Conduct some test to recover errors in prototype
- Get customer feedback on prototype
- Modify the analysis model to enhance the prototype
- Refine the design
- Engineer the special objects (if not available from the library)
- Assemble the objects
- Conduct the test

OO Project Metrics and Estimation

- No. of scenario scripts
- No. of key classes
- No. of supported classes
- Average no. of support classes per key class

References: *Pressman (4) – Chap. 19, OO Analysis & Design – Grady Booch (2)* **Next**: *OOA*

Object-Oriented Analysis

In this phase, a number of tasks must be accomplished;

1. Basic user requirements must be communicated
2. Classes must be identified
3. Class hierarchy must be specified
4. Object-to-object relationships should be presented
5. Object behavior must be modeled
6. Tasks 1 – 5 need to be reapplied iteratively until the model is complete.

OOA Landscape

The Booch method (encompasses both micro and macro development models)

- Identify classes and objects
- Identify the semantics of classes and objects
- Identify relations among classes and objects
- Conduct a series of refinements
- Implement classes and objects

The Coad and Yourdon Method

- Identify objects using 'what to look for' criteria
- Define a generalization-specification structure
- Define a whole-part structure
- Identify subjects (subsystem components)
- Define attributes
- Define services

The Jacobson Method (**standard method**)

- Identify the users of the system and their overall responsibilities
- Build requirements model
    - Define actors and their responsibilities
    - Identify use cases for each actor
    - Prepare initial view of system objects and relationships
    - Review model using use cases as scenarios to determine validity
- Build analysis model
    - Identify interface objects using actor-interaction information
    - Create structural views of interface objects
    - Represent object behavior
    - Isolate subsystems and models for each
    - Review the model using use cases as scenarios to determine validity

The Rambaugh Method

- Develop a statement of scope for the problem
- Build an object model
- Develop a dynamic model
- Construct a functional model for the system

The Wirfs-Brock Method

- Evaluate the customer specification
- Use a grammatical parse to extract candidate classes from the specification
- Group classes in an attempt to identify superclasses

- Define responsibilities for each class
- Assign responsibilities to each class
- Identify relationships between classes
- Define collaboration between classes based on responsibilities
- Build hierarchical representation of classes to show inheritance relationships
- Construction a collaboration graph for the system

The Generic Method

Despite many methods suggested, a software engineer should perform the following generic steps;

- Obtain customer requirements for OO system
    - Identify scenarios or use case
    - Build a requirements model
- Select classes and objects using basic requirements as a guide
- Identify attributes and operations for each system object
- Define structures and hierarchies that organize classes
- Build an object-relationship model
- Build an object-behavior model
- Review the OO analysis model against use cases/scenarios

OOA Process

- Use Cases
- Class-Responsibility-Collaborator (CRC) Modeling

Use Cases

- Define role players – actors (represented using stick figure)
- Use cases – tasks/functions

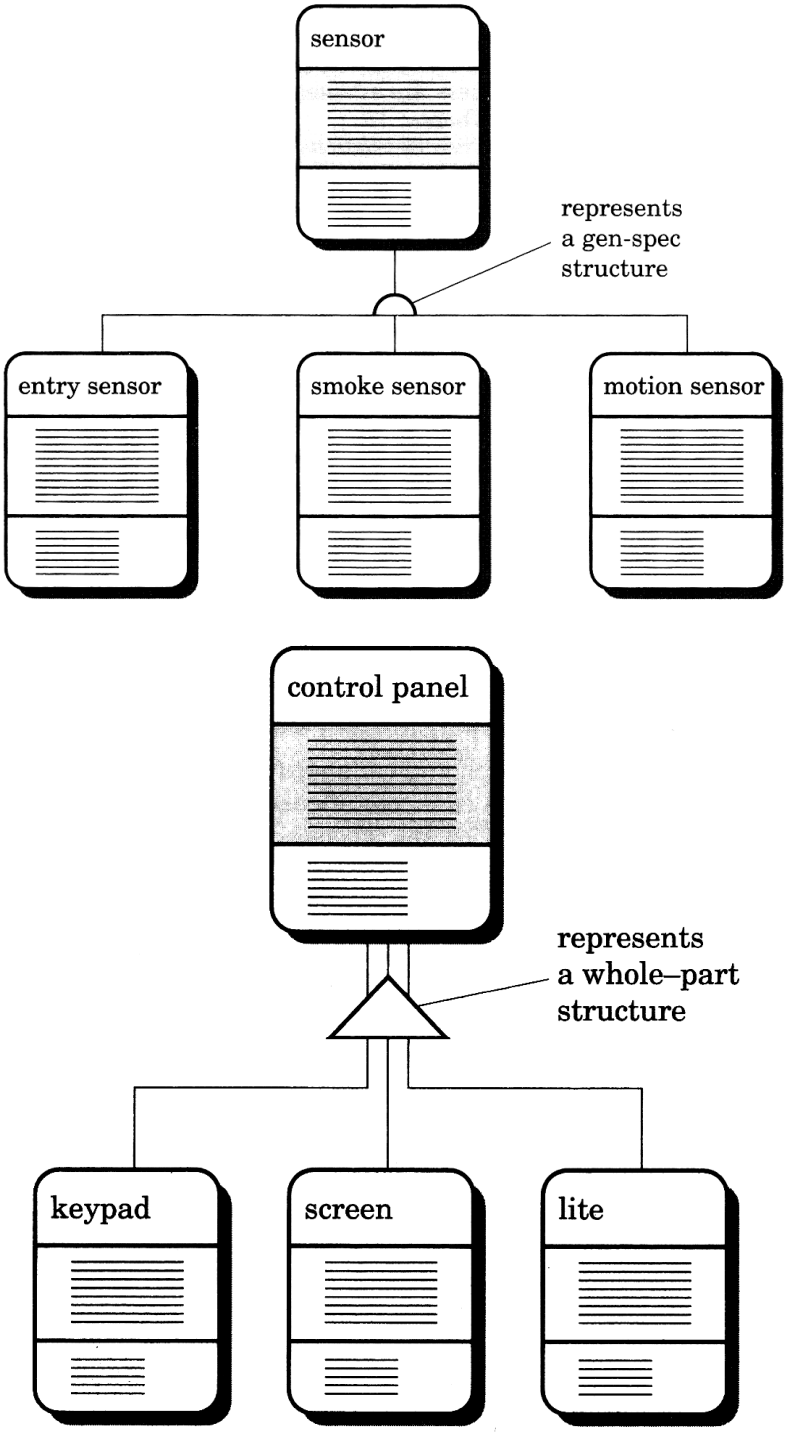
CRC

- Collection of standard index cards that represent classes
- 3"x5"
- Classes
    - Class name
    - Class type
        - Device classes
        - Property classes
        - Interaction classes
    - Class characteristics
        - Tangibility – is the class a tangible thing?
        - Inclusiveness –is the class atomic or aggregate?
        - Sequentiality – is the class concurrent or sequential?
        - Persistence – is the class transient, temporary, or permanent?
        - Integrity – is the class corruptible or guarded?
- Responsibilities
    - The class responsible to maintain the data and methods
- Collaborators
    - Collaborating classes to fulfill its responsibilities

class name:	
class type:	
class characteristics:	
responsibilities:	collaborators:

Defining Structures and Hierarchies

- Generalization-specialization structure (is-a hierarchy)
- Aggregation structure (part-of hierarchy)

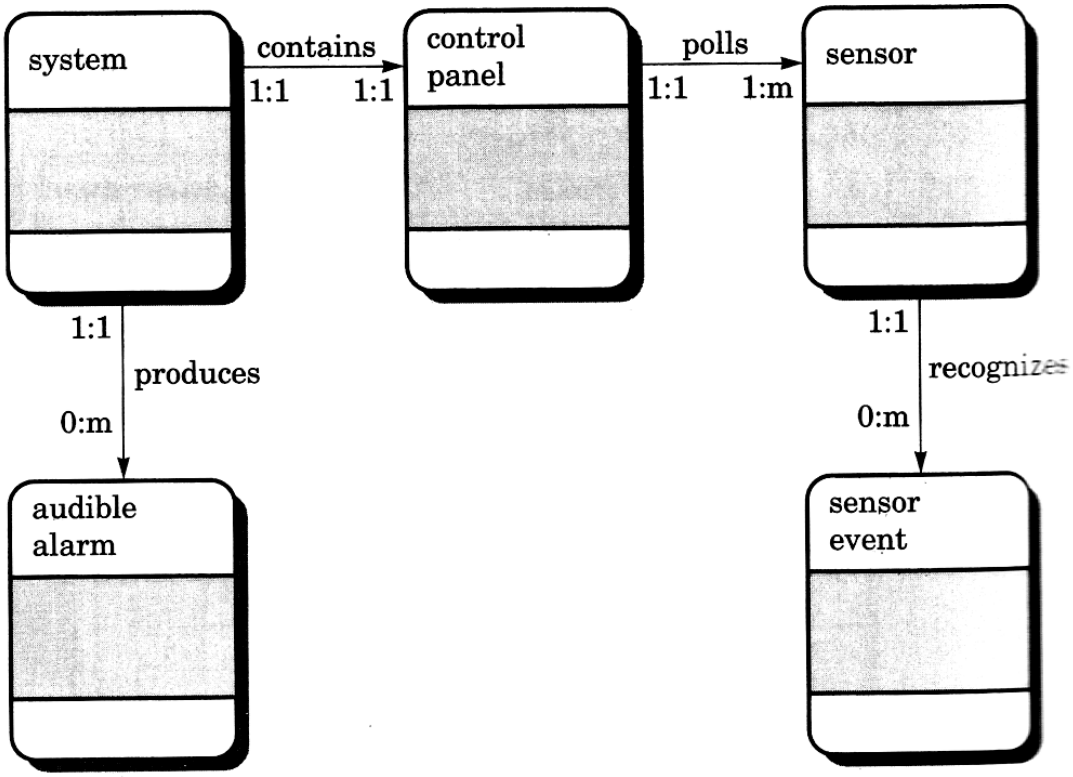


Subjects and references

- Draw a box to represent referencing object for the class

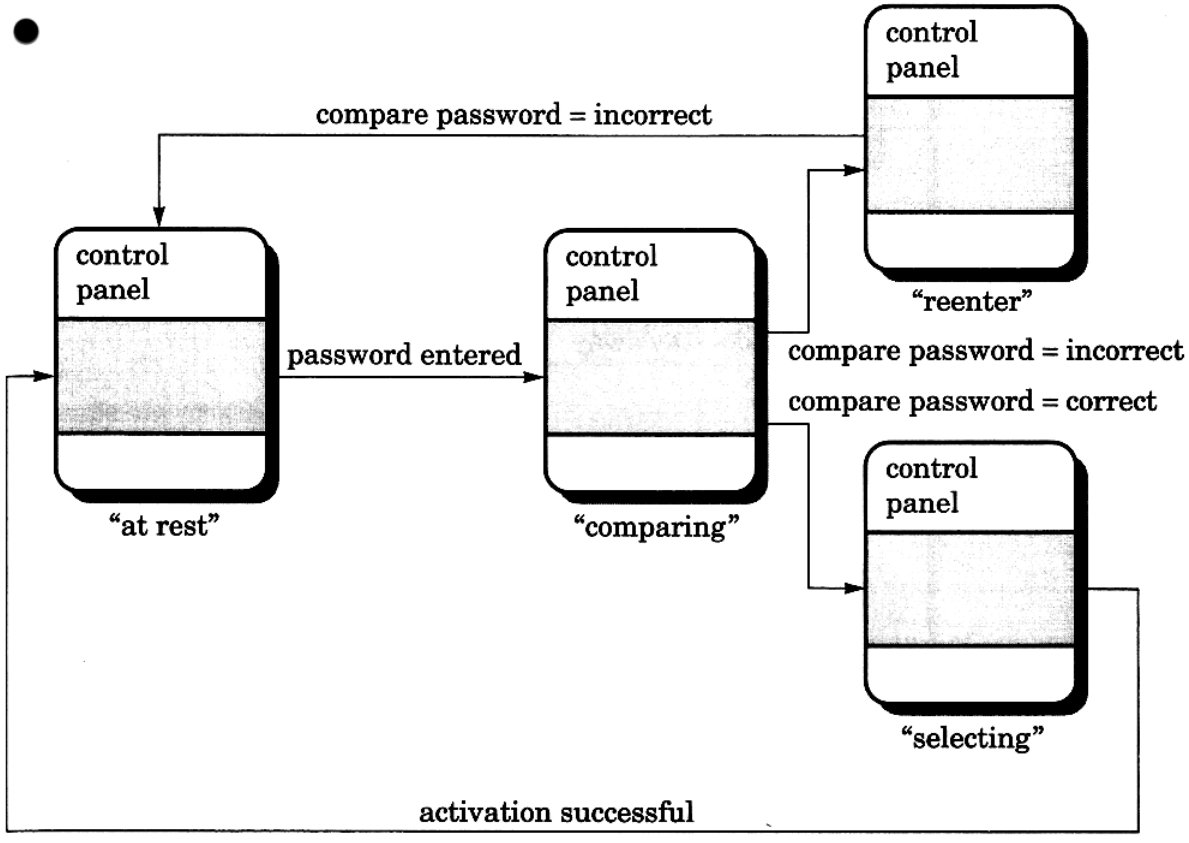
Object-Relationship Model

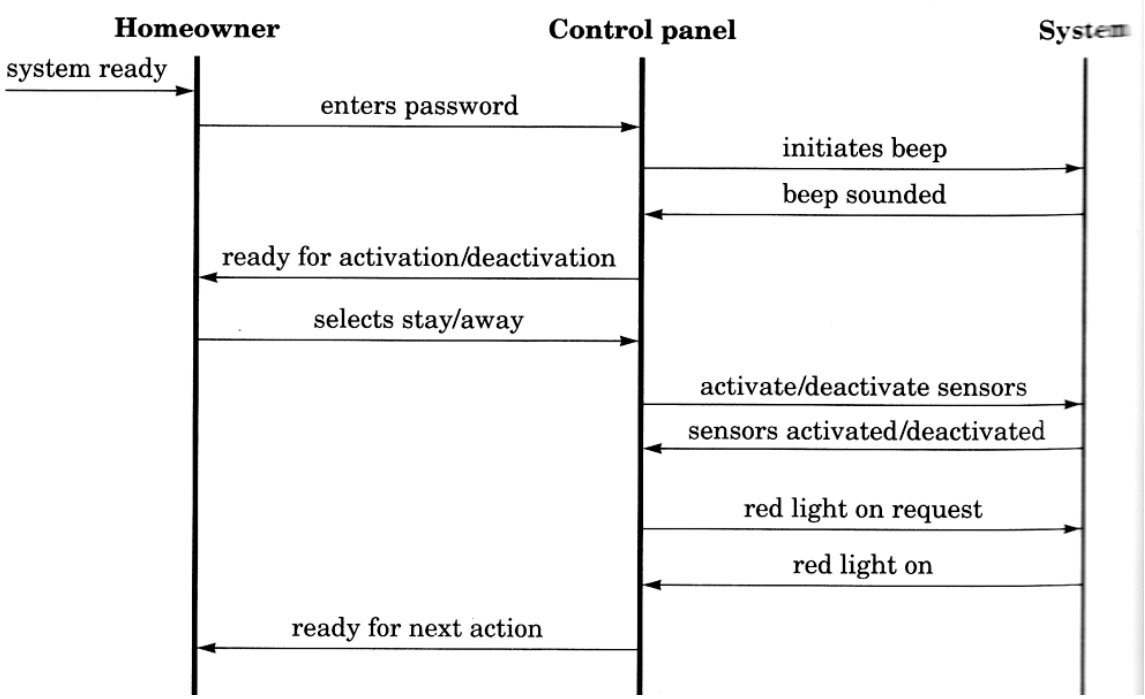
- Using CRC card index, a network of collaborator objects can be drawn
- The relationships are evaluated to determine cardinality (0:1, 1:1, 0:m, m:m)



Object Behavior Model

- Modeled using state representation and event-flow representation

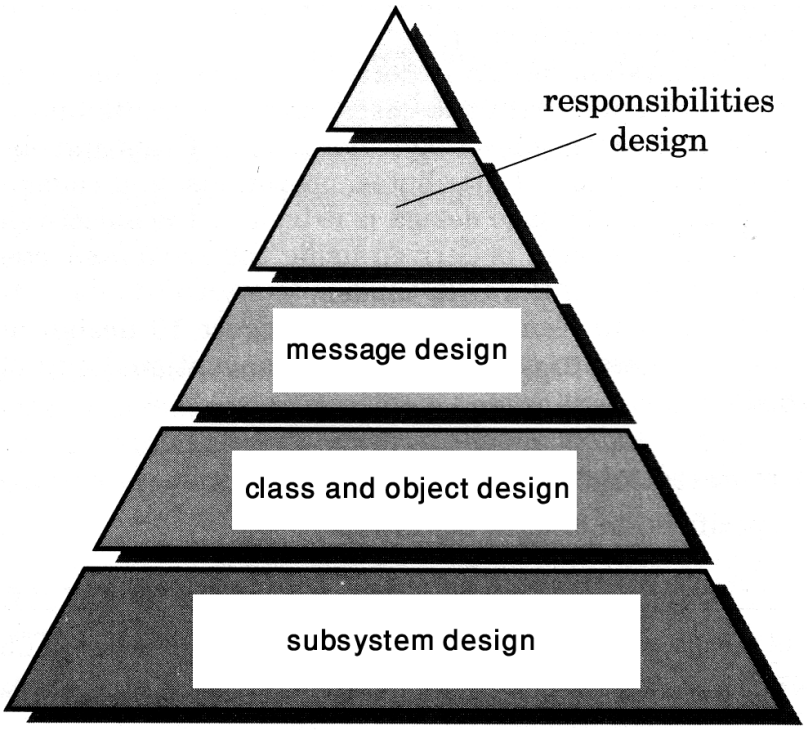



Homeowner
Control panel
System
system ready
enters password
initiates beep
beep sounded
ready for activation/deactivation
selects stay/away
activate/deactivate sensors
sensors activated/deactivated
red light on request
red light on
ready for next action

Object Oriented Design

- Transform analysis model into design model that serves as blueprint for software construction.
- Designing 'reusable' components

OO Design pyramid and translation



Sub-system design: Use cases, object-behavior model
Class and object design: CRC Index Cards, Attributes, Operations, Collaborators
Message design: Object Relationship Model
Responsibilities design: Attributes, Operations, Collaborators, CRC Index Cards

Design Issues

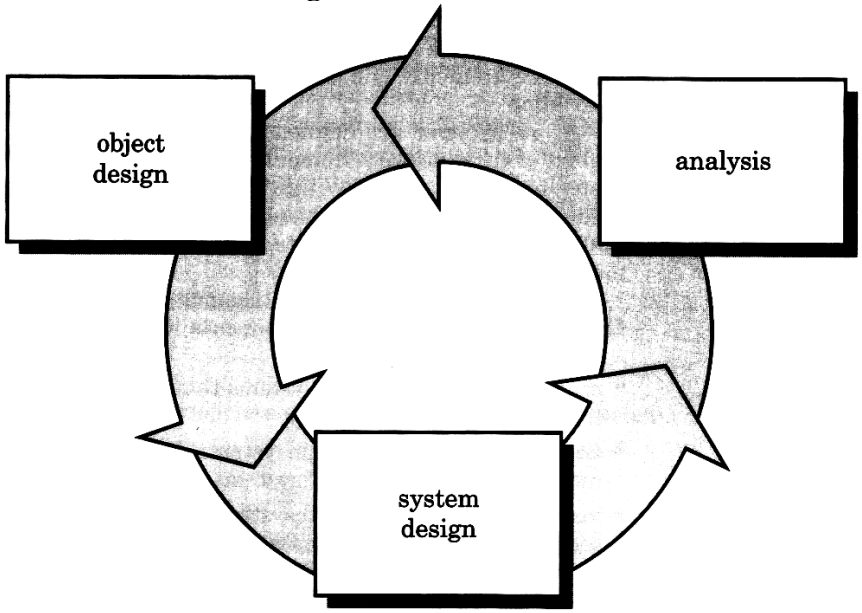
- Decomposability
- Composability
- Understandability
- Continuity
- Protection

OOD landscape

- Many method exist for OOD as for OOA (*Refer to Section 21.1.3 in Pressman(4)*)
- Generic steps for OOD can be defined as;
    - Describe all subsystems in a manner that is implementable
        - Allocate subsystems to processors and tasks
        - Choose a design strategy for implementing data mgmt., interface support, and task management
        - Design an appropriate control mechanism for the system
    - Object design
        - Design each operation at a procedural level
        - Define any internal classes
        - Design internal data structures for class attributes

- Message design
  - Using collaborations between objects and OR, design messaging model
- Review the design model and iterate as required

Generic components of the OO design model



Important design components

- Problem domain
- Human interaction
- Task management
- Data management

Object design

Analysis Model		Design Model
classes	→	objects
attributes	→	data structures
methods	→	algorithms
relationships	→	messaging
behavior	→	control

Use of Design Patterns

Design patterns can be described by four pieces of information;

- The name of pattern
- The problem to which the pattern is generally applied
- The characteristics of the design pattern
- The consequences of applying the design pattern

Composition

- set of complex subsystem
- formed using aggregate objects

References: *Pressman (4) – Chap. 21, OO Analysis & Design – Grady Booch (2)* ***Next**: OOT*

Object Oriented Coding

- All of the coding guidelines and standards do also apply to OO Coding
- Prefer to use pure Object Oriented programming languages such as SmallTalk, Objective C, Java etc.
- If implemented using C++, make the program highly OO

OO Testing

Three tasks must be done;

1. The definition of testing must be broadened to include error discovery techniques applied in OOA, and OOD models
2. The strategy for unit and integration testing must change significantly
3. The design of test cases must account for the unique characteristics of OO software

Review of OO analysis and design models is essential in OOT;

During OOA phase

- Special subclasses may be introduced to accommodate unnecessary attributes or exceptions.
- Misinterpretation of class definition may lead to incorrect or extra class relationship
- Behavior of system/classes may be improperly characterized

Similarly, during OOD phase

- Improper allocation of classes to subsystem may occur
- Unnecessary design work may be expended to create procedural design
- The messaging model might be incorrect

Testing OOA and OOD models

- Correctness of OOA and OOD models
  - semantic correctness must be checked
  - models should be exposed to domain experts
  - class relationships are evaluated to determine the representation level of real world

- Consistency of OOA and OOD models
  - Examine each class and its connection to other classes (using CRC and OR model)

OO Testing Strategies

Unit testing

- Single operation can't be tested in isolation
- Test the operation as part of a class
- Class testing

Integration testing

Two different strategies are there;

- Thread based testing
  - Integrate 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
    - Begins with construction of the system by testing independent classes
    - Then starts testing of dependent classes

Cluster testing is one step in the integration testing of OO software. Cluster of collaborating classes is exercised by designed test cases that attempt to uncover errors in the collaborations.

Validation testing

Conventional validation testing can be applied in OO context.

[Refer to Sections 22.4 – 22.6, Pressman (4) for;
Test Case Design for OO Software
Test cases and class hierarchy
Scenario based test design
Testing surface structure and deep structure
Random testing
Partition testing
Interface testing
Multiple class testing
Testing for behavioral models]

References: *Pressman (4) – Chap. 22, OO Analysis & Design – Grady Booch (2)*