

CASE Tools

What Is CASE Tools

1. CASE: Stands for Computer Aided Software Engineering.
2. Used to support software process activities.
3. Provides information about the software being developed
4. Currently used in every phase/workflow of life cycle

Reasons for Using CASE Tools

1. Improved productivity.
2. Better documentation.
3. Reduced lifetime maintenance.
4. Improved accuracy.
5. Opportunity to non-programmers.
6. Enforce discipline.
7. Help communication between team members.

Layers Of CASE Tools

Upper CASE Tools

Lower CASE Tools

Integrated CASE Tools

Upper CASE Tools

1. Support analysis and design phases of SDLC.
2. Can be further divided as:
 - Diagramming Tools
 - Report Generator
 - Analysis Tool

Lower CASE Tools

1. Support implementation and maintenance phases of SDLC.
2. Focuses on
 - Central Repository
 - Code Generator
 - Configuration Management
3. Hand-coding is still necessary.

Integrated CASE Tools

1. Support activities that occur across multiple phases of SDLC.
2. Facilitate
 - Analysis
 - Design
 - Code
 - Management

Three Perspective Of CASE Tools

1. Functional perspective

- Tools are classified according to their specific function.

2. Process perspective

- Tools are classified according to process activities that are supported.

3. Integration perspective

- Tools are classified according to their organisation into integrated units.

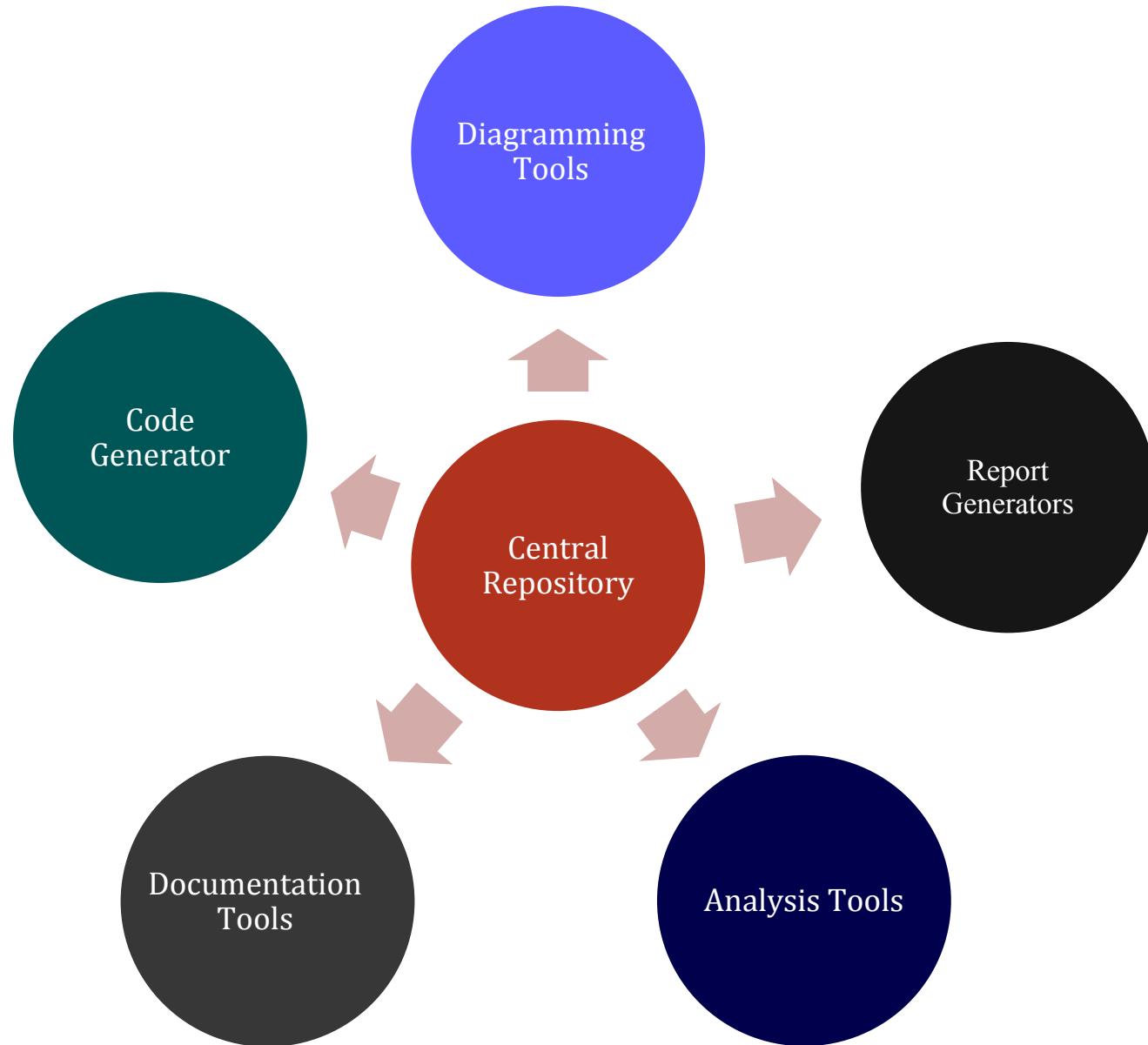
Categories Of CASE Tools

Tools

Workbench

Environment

CASE Environment



Some Other Examples of Case Tools

1. Project planning tools
 - a. for cost & effort estimation, & project scheduling
2. Risk analysis tools
3. Requirements tracing tools
4. Metrics and management tools
5. Software configuration management tools
6. Prototyping & simulation tools
7. Testing tools
8. Reengineering tools

Problems with CASE Tools

1. Limitations in flexibility of documentation.
2. Major danger: completeness and syntactic correctness does NOT mean compliance with requirements
3. Costs associated with the use of the tool
 - Purchase price
 - Training

UML CASE Tools

1. Unified Modeling Language (UML) is a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system.
2. Things to remember while choosing UML tools:
 - Tool should support most of the diagrams.
 - Easy to use, reliable and scalable.
 - Should be open-source or having low cost.

Some Open-Source UML Case Tools

1. Umbrello UML Modeller
2. Umple (By university of ottawa)
3. UML Designer
4. Modelio
5. JetUML
6. Eclipse UML Tools
7. Dia
8. ArgoUML

Formal Specifications

Formal Specification

- Techniques for the unambiguous specification of software

Objectives

- To explain why formal specification techniques help discover problems in system requirements
- To describe the use of algebraic techniques for interface specification
- To describe the use of model-based techniques for behavioural specification

Topics covered

- Formal specification in the software process
- Interface specification
- Behavioural specification

Formal methods

- Formal specification is part of a more general collection of techniques that are known as ‘formal methods’
- These are all based on mathematical representation and analysis of software
- Formal methods include
 - Formal specification
 - Specification analysis and proof
 - Transformational development
 - Program verification

Acceptance of formal methods

- Formal methods have not become mainstream software development techniques as was once predicted
 - Other software engineering techniques have been successful at increasing system quality. Hence the need for formal methods has been reduced
 - Market changes have made time-to-market rather than software with a low error count the key factor. Formal methods do not reduce time to market
 - The scope of formal methods is limited. They are not well-suited to specifying and analysing user interfaces and user interaction
 - Formal methods are hard to scale up to large systems

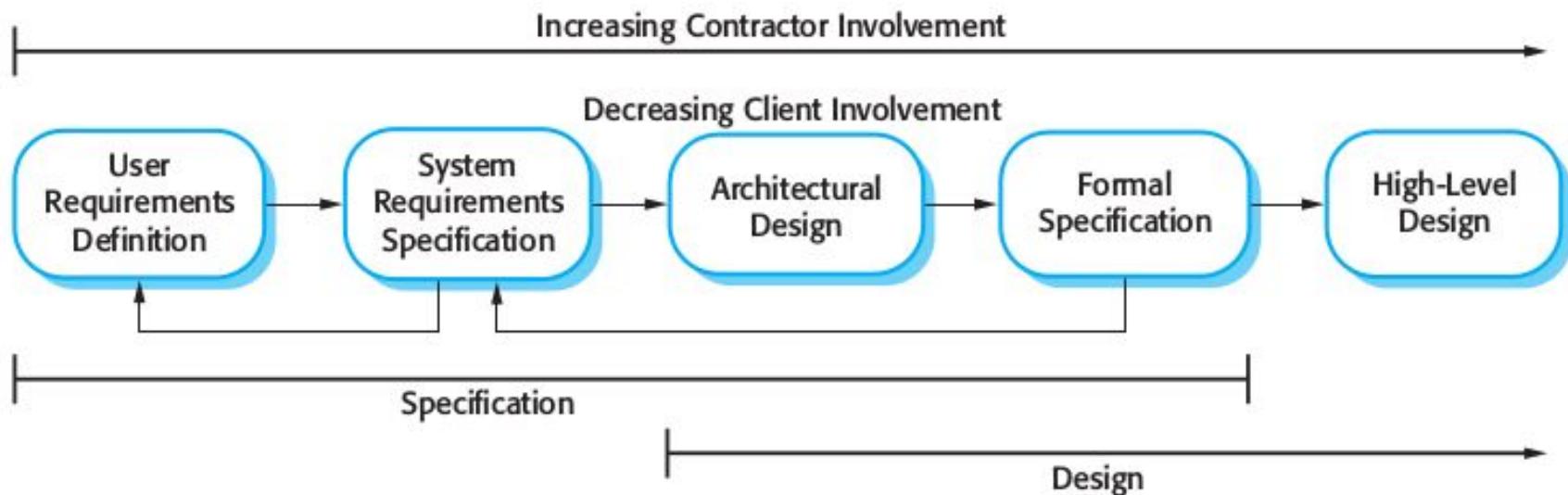
Use of formal methods

- Formal methods have limited practical applicability
- Their principal benefits are in reducing the number of errors in systems so their main area of applicability is critical systems
- In this area, the use of formal methods is most likely to be cost-effective

Specification in the software process

- Specification and design are inextricably intermingled.
- Architectural design is essential to structure a specification.
- Formal specifications are expressed in a mathematical notation with precisely defined vocabulary, syntax and semantics.

Specification and design



Specification techniques

- Algebraic approach
 - The system is specified in terms of its operations and their relationships
- Model-based approach
 - The system is specified in terms of a state model that is constructed using mathematical constructs such as sets and sequences. Operations are defined by modifications to the system's state

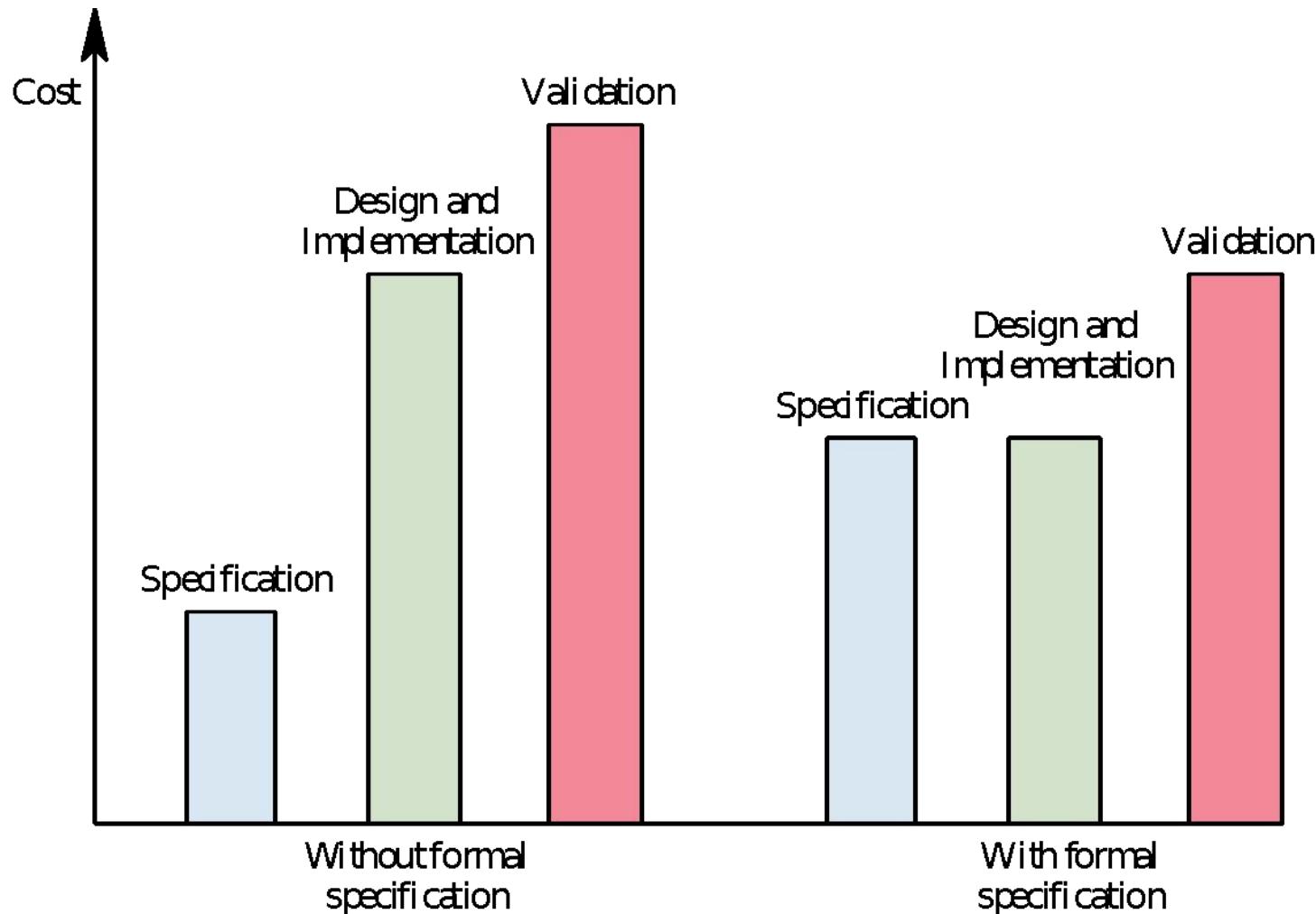
Formal specification languages

	Sequential	Concurrent
Algebraic	Larch (Guttag, Horning et al., 1985; Guttag, Horning et al., 1993), OBJ (Futatsugi, Goguen et al., 1985)	Lotos (Bolognesi and Brinksma, 1987),
Model-based	Z (Spivey, 1992) VDM (Jones, 1980) B (Wordsworth, 1996)	CSP (Hoare, 1985) Petri Nets (Peterson, 1981)

Use of formal specification

- Formal specification involves investing more effort in the early phases of software development
- This reduces requirements errors as it forces a detailed analysis of the requirements
- Incompleteness and inconsistencies can be discovered and resolved
- Hence, savings are made as the amount of rework due to requirements problems is reduced

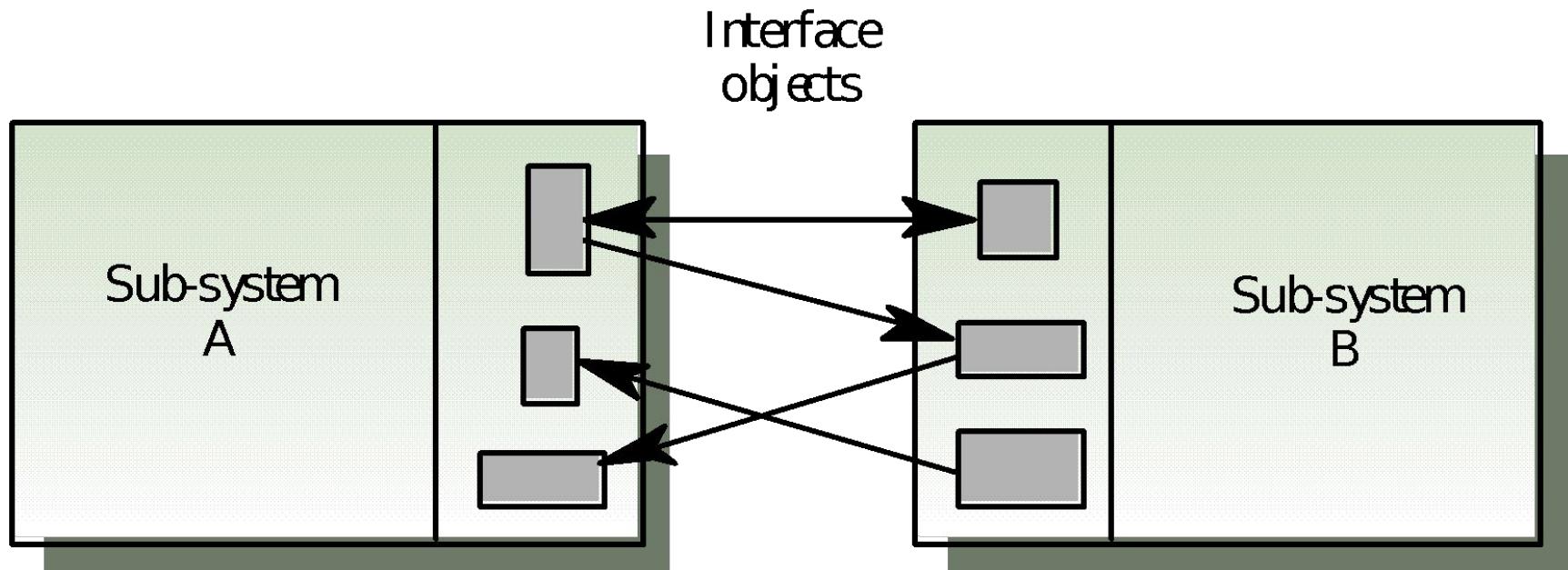
Development costs with formal specification



Interface specification

- Large systems are decomposed into subsystems with well-defined interfaces between these subsystems
- Specification of subsystem interfaces allows independent development of the different subsystems
- Interfaces may be defined as abstract data types or object classes
- The algebraic approach to formal specification is particularly well-suited to interface specification

Sub-system interfaces



Specification components

- **Introduction**
 - Defines the sort (the type name) and declares other specifications that are used
- **Description**
 - Informally describes the operations on the type
- **Signature**
 - Defines the syntax of the operations in the interface and their parameters
- **Axioms**
 - Defines the operation semantics by defining axioms which characterise behaviour

Systematic algebraic specification

- Algebraic specifications of a system may be developed in a systematic way
 - Specification structuring.
 - Specification naming.
 - Operation selection.
 - Informal operation specification
 - Syntax definition
 - Axiom definition

List specification

LIST (Elem)

sort List

imports INTEGER

Defines a list where elements are added at the end and removed from the front. The operations are Create, which brings an empty list into existence, Cons, which creates a new list with an added member, Length, which evaluates the list size, Head, which evaluates the front element of the list, and Tail, which creates a list by removing the head from its input list. Undefined represents an undefined value of type Elem.

Create → List

Cons (List, Elem) → List

Head (List) → Elem

Length (List) → Integer

Tail (List) → List

Head (Create) = Undefined **exception** (empty list)

Head (Cons (L, v)) = **if** L = Create **then** v **else** Head (L)

Length (Create) = 0

Length (Cons (L, v)) = Length (L) + 1

Tail (Create) = Create

Tail (Cons (L, v)) = **if** L = Create **then** Create **else** Cons (Tail (L), v)

Interface specification in critical systems

- Consider an air traffic control system where aircraft fly through managed sectors of airspace
- Each sector may include a number of aircraft but, for safety reasons, these must be separated
- In this example, a simple vertical separation of 300m is proposed
- The system should warn the controller if aircraft are instructed to move so that the separation rule is breached

A sector object

- Critical operations on an object representing a controlled sector are
 - Enter. Add an aircraft to the controlled airspace
 - Leave. Remove an aircraft from the controlled airspace
 - Move. Move an aircraft from one height to another
 - Lookup. Given an aircraft identifier, return its current height

Primitive operations

- It is sometimes necessary to introduce additional operations to simplify the specification
- The other operations can then be defined using these more primitive operations
- Primitive operations
 - Create. Bring an instance of a sector into existence
 - Put. Add an aircraft without safety checks
 - In-space. Determine if a given aircraft is in the sector
 - Occupied. Given a height, determine if there is an aircraft within 300m of that height

Sector specification

SECTOR

sort Sector

imports INTEGER, BOOLEAN

Enter - adds an aircraft to the sector if safety conditions are satisfied

Leave - removes an aircraft from the sector

Move - moves an aircraft from one height to another if safe to do so

Lookup - Finds the height of an aircraft in the sector

Create - creates an empty sector

Put - adds an aircraft to a sector with no constraint checks

In-space - checks if an aircraft is already in a sector

Occupied - checks if a specified height is available

Enter (Sector, Call-sign, Height) → Sector

Leave (Sector, Call-sign) → Sector

Move (Sector, Call-sign, Height) → Sector

Lookup (Sector, Call-sign) → Height

Create → Sector

Put (Sector, Call-sign, Height) → Sector

In-space (Sector, Call-sign) → Boolean

Occupied (Sector, Height) → Boolean

Enter (S, CS, H) =

if In-space (S, CS) **then** S exception (Aircraft already in sector)
elseif Occupied (S, H) **then** S exception (Height conflict)
else Put (S, CS, H)

Leave (Create, CS) = Create exception (Aircraft not in sector)

Leave (Put (S, CS1, H1), CS) =

if CS = CS1 **then** S **else** Put (Leave (S, CS), CS1, H1)

Move (S, CS, H) =

if S = Create **then** Create exception (No aircraft in sector)
elseif not In-space (S, CS) **then** S exception (Aircraft not in sector)
elseif Occupied (S, H) **then** S exception (Height conflict)
else Put (Leave (S, CS), CS, H)

-- NO-HEIGHT is a constant indicating that a valid height cannot be returned

Lookup (Create, CS) = NO-HEIGHT exception (Aircraft not in sector)

Lookup (Put (S, CS1, H1), CS) =

if CS = CS1 **then** H1 **else** Lookup (S, CS)

Occupied (Create, H) = false

Occupied (Put (S, CS1, H1), H) =

if (H1 > H **and** H1 - H ≥ 300) **or** (H > H1 **and** H - H1 ≥ 300) **then** true
else Occupied (S, H)

In-space (Create, CS) = false

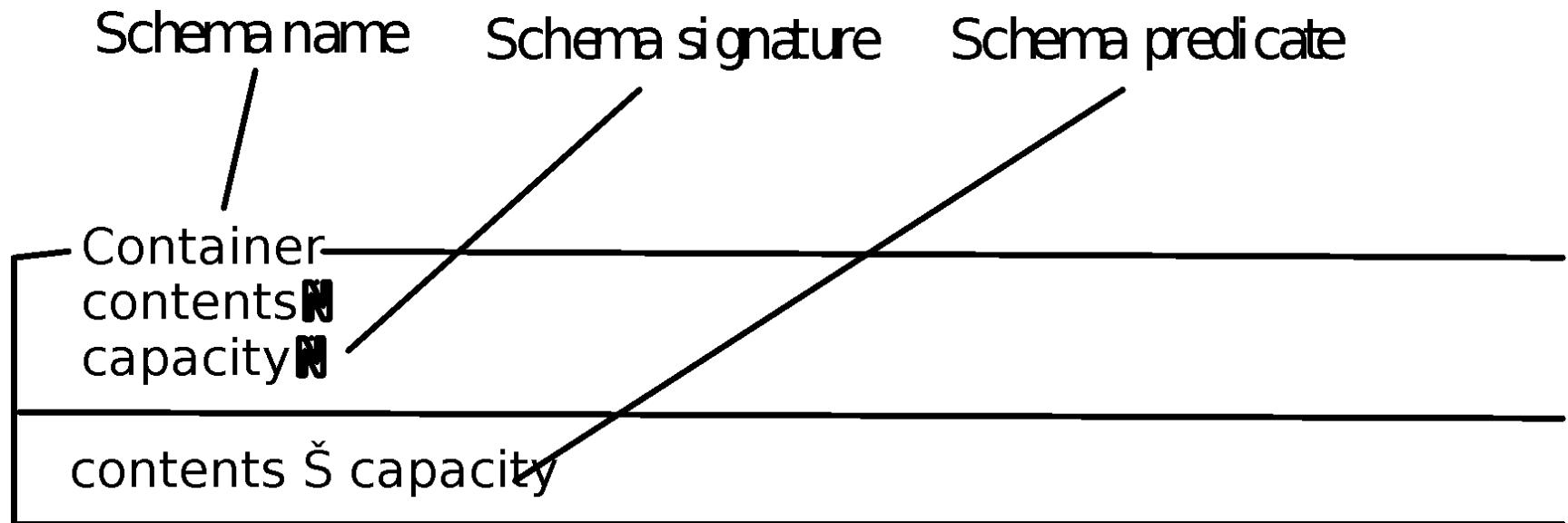
In-space (Put (S, CS1, H1), CS) =

if CS = CS1 **then** true **else** In-space (S, CS)

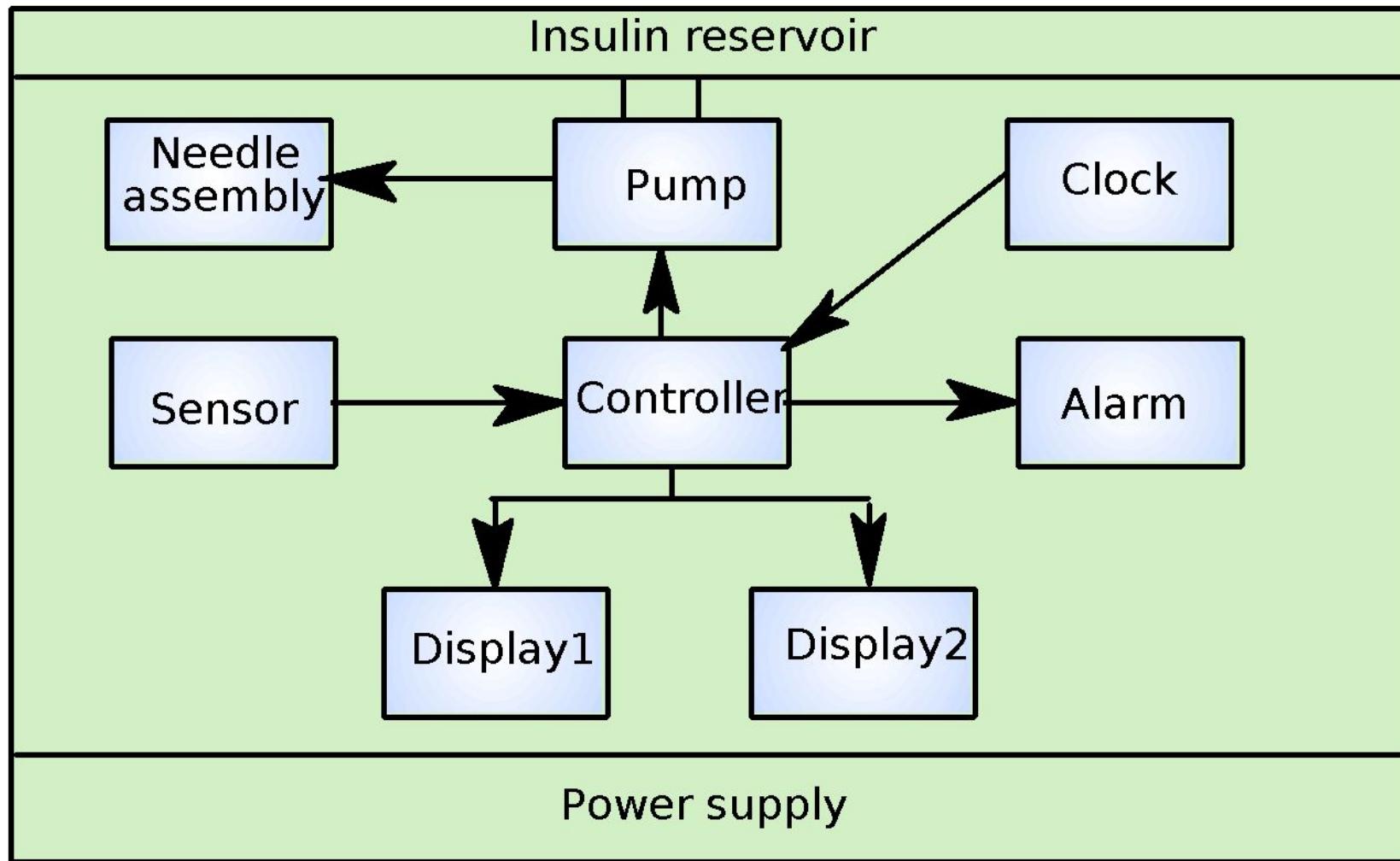
Behavioural specification

- Algebraic specification can be cumbersome when the object operations are not independent of the object state
- Model-based specification exposes the system state and defines the operations in terms of changes to that state
- The Z notation is a mature technique for model-based specification. It combines formal and informal description and uses graphical highlighting when presenting specifications

The structure of a Z schema



An insulin pump



Modelling the insulin pump

- The schema models the insulin pump as a number of state variables
 - reading?
 - dose, cumulative_dose
 - r0, r1, r2
 - capacity
 - alarm!
 - pump!
 - display1!, display2!
- Names followed by a ? are inputs, names followed by a ! are outputs

Schema invariant

- Each Z schema has an invariant part which defines conditions that are always true
- For the insulin pump schema it is always true that
 - The dose must be less than or equal to the capacity of the insulin reservoir
 - No single dose may be more than 5 units of insulin and the total dose delivered in a time period must not exceed 50 units of insulin. This is a safety constraint (see Chapters 16 and 17)
 - `display1!` shows the status of the insulin reservoir.

Insulin pump schema

```
Insulin_pump
reading? : N
dose, cumulative_dose: N
r0, r1, r2: N      // used to record the last 3 readings taken
capacity: N
alarm!: {off, on}
pump!: N
display1!, display2!: STRING
```

```
dose ≤ capacity ∧ dose ≥ 5 ∧ cumulative_dose ≤ 50
capacity ≤ 40 ⇒ display1! = ""
capacity ≤ 39 ∧ capacity ≥ 10 ⇒ display1! = "Insulin low"
capacity ≤ 9 ⇒ alarm! = on ∧ display1! = "Insulin very low"
r2 = reading?
```

DOSAGE schema

```
DO SAGE
  △Insulin_Pump

  (
    dose = 0 ∧
    (
      (( r1 ⊑ r0) ∧ (r2 = r1)) ∨
      ((r1 > r0) ∧ (r2 ⊏ r1)) ∨
      ((r1 < r0) ∧ ((r1-r2) > (r0-r1)))
    ) ∨
    dose = 4 ∧
    (
      ((r1 ⊏ r0) ∧ (r2 = r1)) ∨
      ((r1 < r0) ∧ ((r1-r2) ⊏ (r0-r1)))
    ) ∨
    dose = (r2 - r1) * 4 ∧
    (
      ((r1 ⊏ r0) ∧ (r2 > r1)) ∨
      ((r1 > r0) ∧ ((r2 - r1) ⊑ (r1 - r0)))
    )
  )
  capacity' = capacity - dose
  cumulative_dose' = cumulative_dose + dose
  r0' = r1 ∧ r1' = r2
```

Output schemas

DISPLAY

$\Delta_{\text{Insulin_Pump}}$

```
display2!' = Nat_to_string (dose) ∧  
(reading? < 3 ⇒ display1!' = "Sugar low" ∨  
reading? > 30 ⇒ display1!' = "Sugar high" ∨  
reading? ⊓ 3 and reading? ⊔ 30 ⇒ display1!' = "OK")
```

ALARM

$\Delta_{\text{Insulin_Pump}}$

```
(reading? < 3 ∨ reading? > 30) ⇒ alarm!' = on ∨  
(reading? ⊓ 3 ∧ reading? ⊔ 30) ⇒ alarm!' = off
```

Key points

- Formal system specification complements informal specification techniques
- Formal specifications are precise and unambiguous. They remove areas of doubt in a specification
- Formal specification forces an analysis of the system requirements at an early stage. Correcting errors at this stage is cheaper than modifying a delivered system

Key points

- Formal specification techniques are most applicable in the development of critical systems and standards.
- Algebraic techniques are suited to interface specification and model-based techniques are suited for behavioural specification.

Introduction

- Getting started with software engineering

FAQs about software engineering

- What is software?
- What is software engineering?
- What is the difference between software engineering and computer science?
- When do we call software solution to be successful?
- What is a software process or lifecycle?
- What is a software process model?

What is software?

- Computer programs and associated documentation
- Software products may be developed for a particular customer or may be developed for a general market

Software costs

- Software costs often dominate system costs. The costs of software on a PC are often greater than the hardware cost
- Software costs more to maintain than it does to develop. For systems with a long life, maintenance costs may be several times development costs

What is software engineering?

- Software engineering is an engineering discipline which is concerned with all aspects of software production
- Set of rules

What is the difference between software engineering and computer science?

- Computer science is concerned with theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software

Software Systems

- Ubiquitous, used in variety of applications
 - Business, engineering, scientific applications
- Simple to complex, internal to public, single function to enterprise-wide, informational to mission-critical..
- Generic and Customized Software

What are the attributes of good software?

- The software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable
- Maintainability
 - Software must evolve to meet changing needs
- Dependability
 - Software must be trustworthy
- Efficiency
 - Software should not make wasteful use of system resources
- Usability
 - Software must be usable by the users for which it was designed

Successful Software System

- Software development projects have not always been successful
- When do we consider a software application successful?
 - ✓ Development completed
 - ✓ It is useful
 - ✓ It is usable
 - ✓ It is used
- Cost-effectiveness, maintainability implied

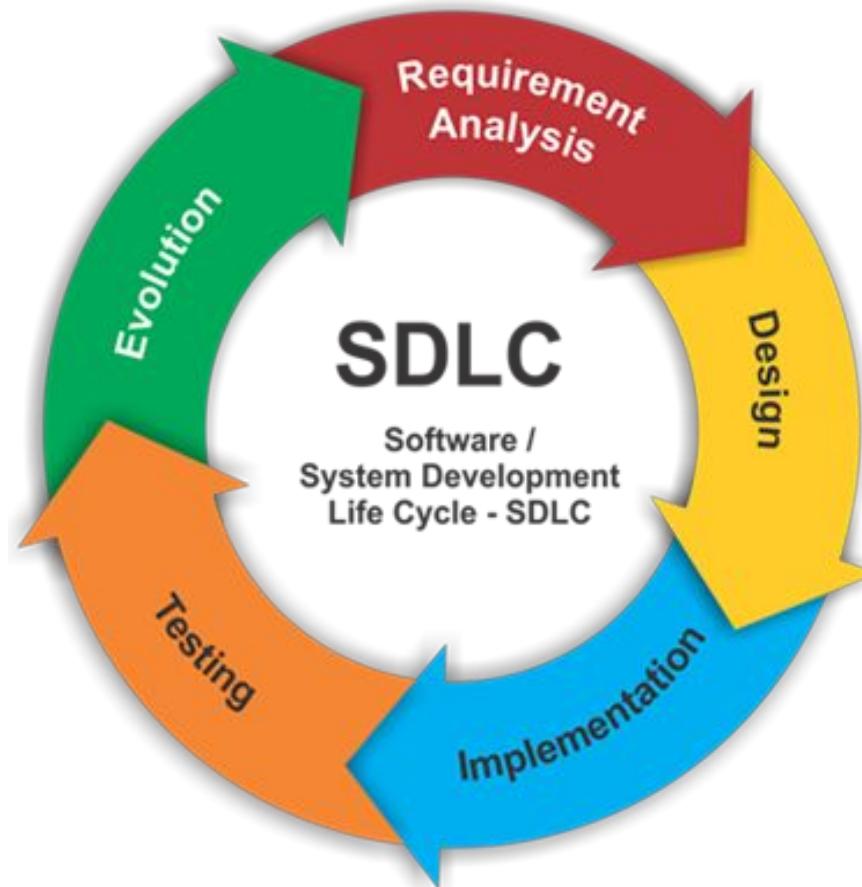
Reason for failure

- Ad hoc software development results in such problems
 - No planning of development work
 - Poor understanding of user requirements
 - No control or review
 - Technical incompetence of developers
 - Poor understanding of cost and effort by both developer and user

What are the key challenges facing software engineering?

- Coping with legacy systems, coping with increasing diversity and coping with demands for reduced delivery times
- Legacy systems
 - Old, valuable systems must be maintained and updated
- Heterogeneity
- Delivery
 - There is increasing pressure for faster delivery of software

Software Development Life Cycle or Software process



Common SDLC models

- Waterfall
- Rapid Prototyping
- Incremental
- Spiral

Software Categories

- Software can be categorized into 6 categories
 - ✓ System Software
 - ✓ Application Software
 - ✓ Engineering Software and Scientific Software
 - ✓ Embedded Software
 - ✓ Web Application
 - ✓ Artificial Intelligence Software

What are the costs of software engineering?

- Roughly 60% of costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs
- Costs vary depending on the type of system being developed and the requirements of system attributes such as performance and system reliability
- Distribution of costs depends on the development model that is used

Characteristics of Good Model

- Should be precisely defined
 - No ambiguity about what is to be done, when, how etc..
- It must be predictable
 - Learn from feedback
 - Can be repeated in other projects with confidence about it's outcome
 - Project-A = done by 3 person in 4 months
 - Project-B = Similar in complexity should also take about same time or with some improvement it should take less time

Advantage of choosing SDLC Model

- Increased development speed
- Increased product quality
- Improved tracking and control
- Improve client relations
- Decreased project risk

No Silver Bullet

- Fredrick Brooks released article Name “No Silver Bullet” on software development
- One of the most famous article in history of software development
- He asserted that there is no single development in either technology or management technique which by itself promise even one order-of-magnitude improvement within a decade in productivity reliability or simplicity

Professional and ethical responsibility

- Software engineering involves wider responsibilities than simply the application of technical skills
- Software engineers must behave in an honest and ethically responsible way if they are to be respected as professionals
- Ethical behaviour is more than simply upholding the law.

Issues of professional responsibility

- *Confidentiality*
 - Engineers should normally respect the confidentiality of their employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.
- *Competence*
 - Engineers should not misrepresent their level of competence. They should not knowingly accept work which is outwith their competence.

Issues of professional responsibility

- *Intellectual property rights*
 - Engineers should be aware of local laws governing the use of intellectual property such as patents, copyright, etc. They should be careful to ensure that the intellectual property of employers and clients is protected.
- *Computer misuse*
 - Software engineers should not use their technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).

ACM/IEEE Code of Ethics

- The professional societies in the US have cooperated to produce a code of ethical practice.
- Members of these organisations sign up to the code of practice when they join.
- The Code contains eight Principles related to the behaviour of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession.

Code of ethics - preamble

• Preamble

- The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.
- Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

Code of ethics - principles

- **1. PUBLIC**
 - Software engineers shall act consistently with the public interest.
- **2. CLIENT AND EMPLOYER**
 - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
- **3. PRODUCT**
 - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.

Code of ethics - principles

- **JUDGMENT**
 - Software engineers shall maintain integrity and independence in their professional judgment.
- **5. MANAGEMENT**
 - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
- **6. PROFESSION**
 - Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.

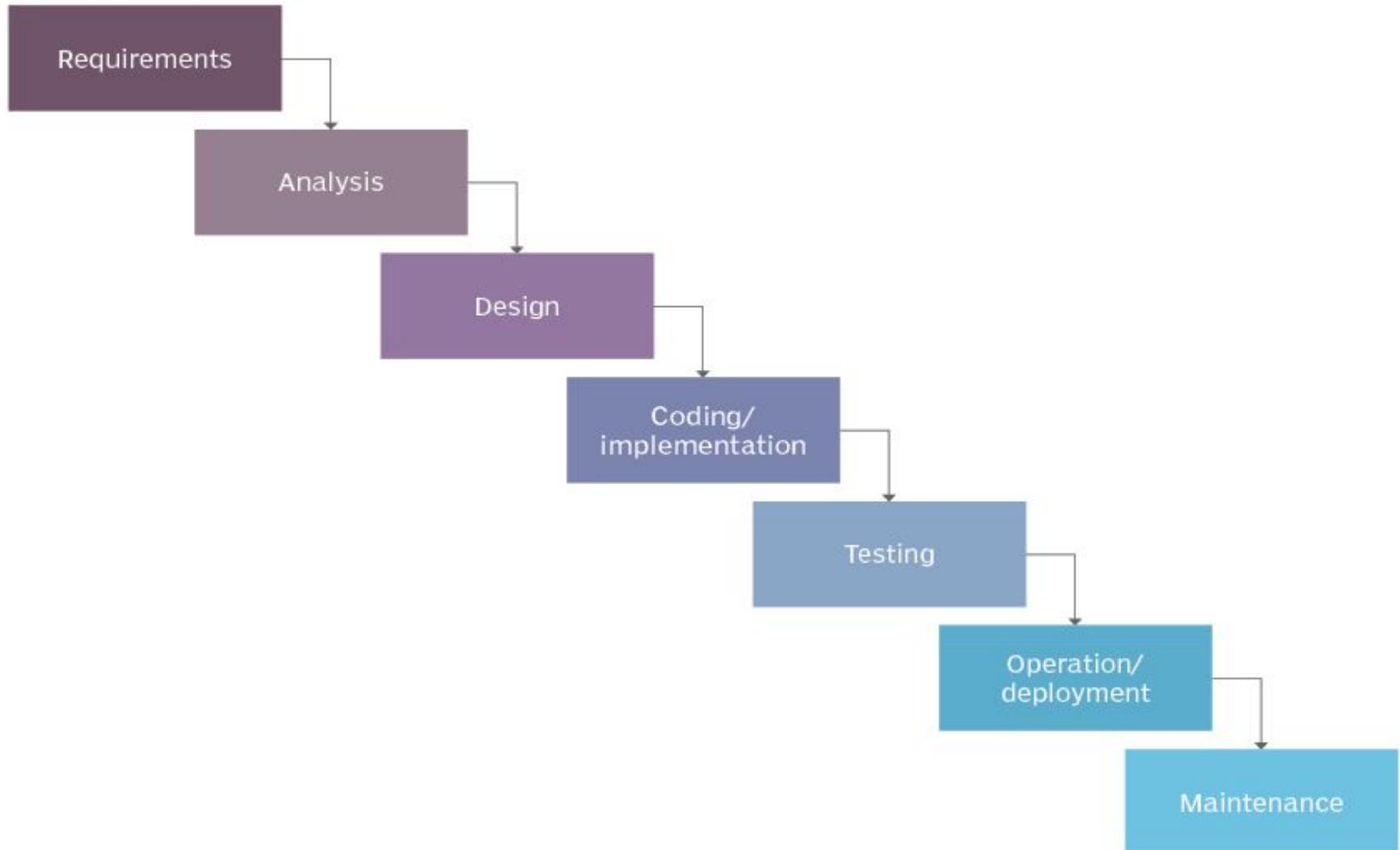
Code of ethics - principles

- **7. COLLEAGUES**
 - Software engineers shall be fair to and supportive of their colleagues.
- **8. SELF**
 - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Ethical dilemmas

- Disagreement in principle with the policies of senior management
- Your employer acts in an unethical way and releases a safety-critical system without finishing the testing of the system
- Participation in the development of military weapons systems or nuclear systems

Waterfall model



..

- Here steps are arranged in linear order
 - A step take inputs from previous step, gives output to next step
 - Exit criteria of a step must match with entry criteria of the succeeding step
- Produces many intermediate deliverable, usually documents
 - Important for quality assurance
- It is widely used when requirements are well understood

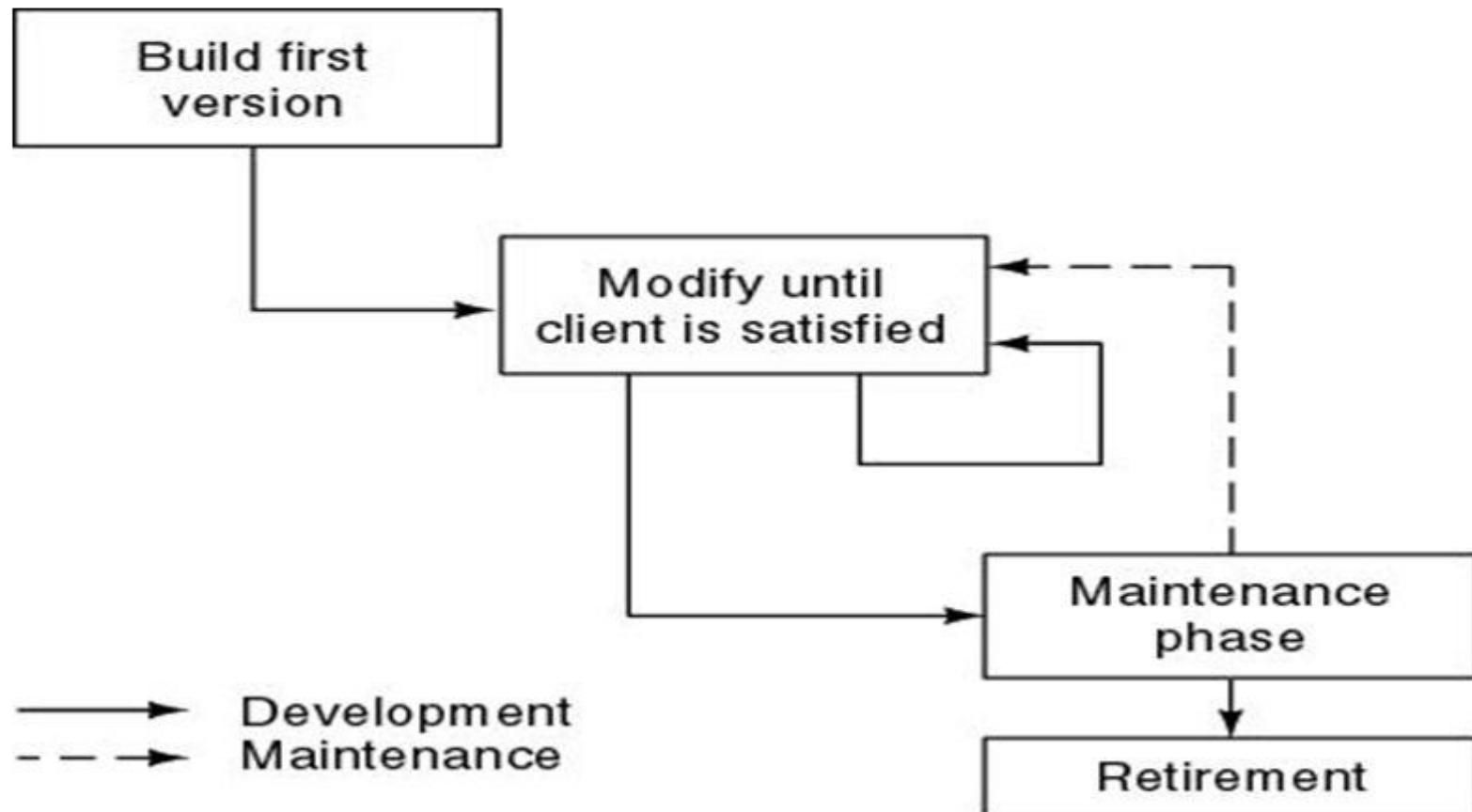
Deliverables in Waterfall model

- Project plan and feasibility report
- Requirements documents
- System design documents
- Test plans and test reports
- Source code
- Software manuals (User manual, installation manual)

Shortcoming of Waterfall model

- Requirements may not be clearly known, especially for applications not having existing manual
- Requirements change with time during project life cycle
 - User may find solution of little use
 - Better to develop in parts in small increments

Build and Fix Model



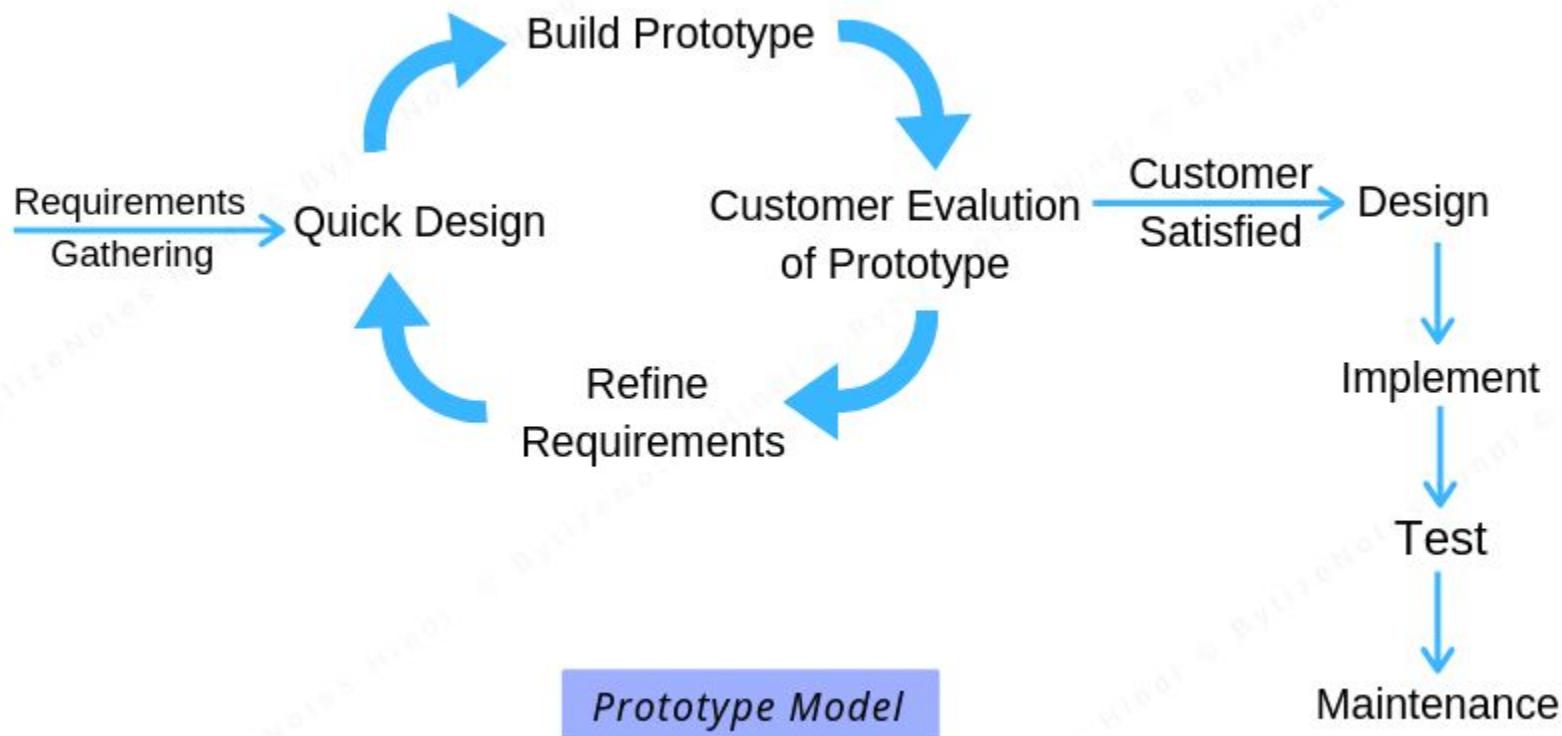
Prototype Model

- When customer or developer is not sure
 - Of requirements
 - Of algorithms, efficiency, human-machine interaction
- A throwaway prototype from currently known user needs
- Working or even paper prototype
- Quick design focuses on aspects visible to user; features clearly understood need not be implemented

Prototype Model

- Prototype is turned to satisfy customer needs
 - Many iterations may be required to incorporate changes and new requirements
- Final product follows usual define-design-build-test life cycle like waterfall mode

Prototype Model



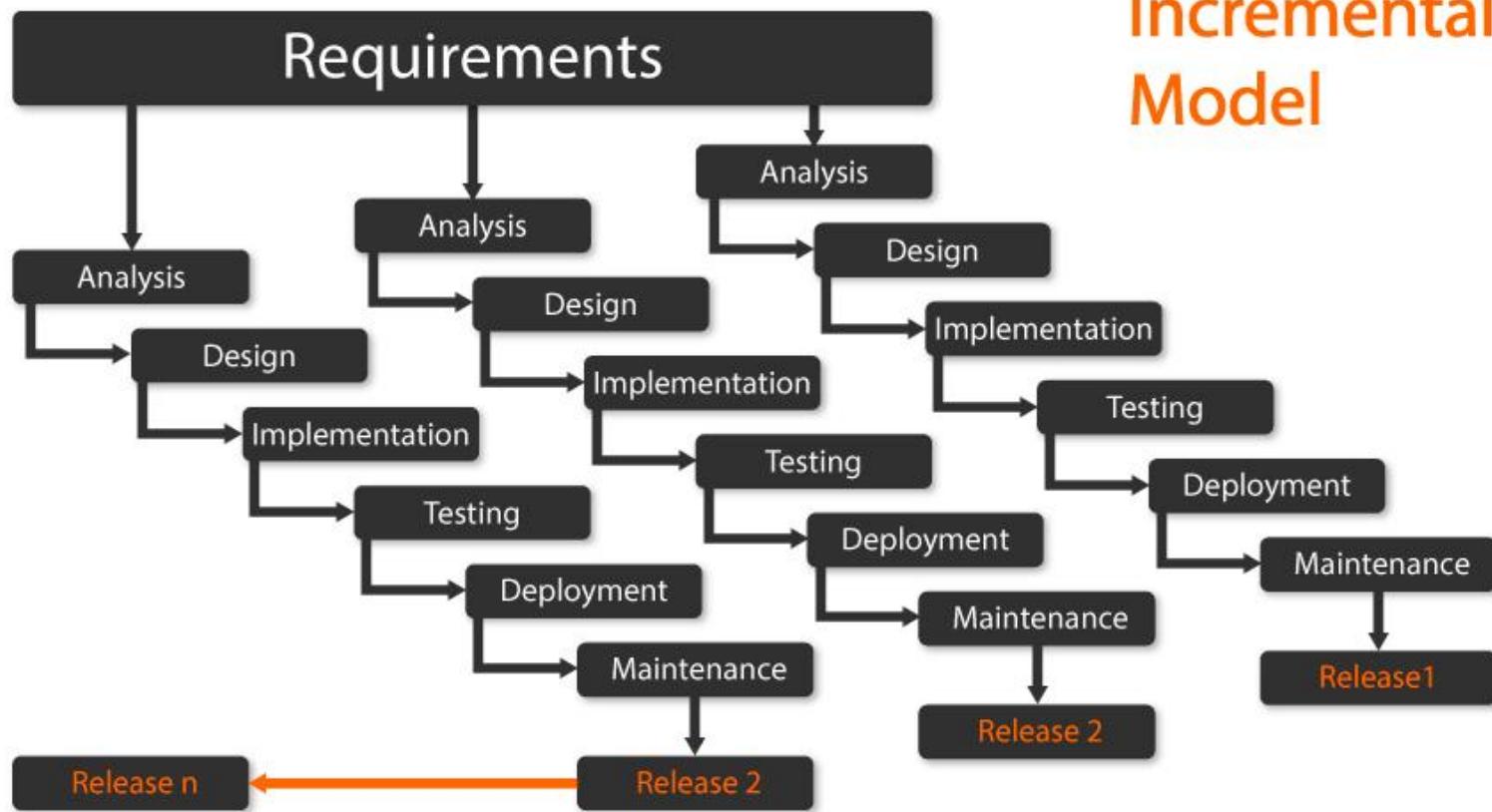
Limitations

- Customer may want prototype itself
- Developer may continue with implementation choices made during prototype
- Good tools need to be acquired for quick development
- May increase project cost due to multiple number of iterations

Incremental Model

- Useful for product development where developers define scope, features to serve many customer
- Early version with limited feature important to establish market and get customer feedback
- A list of features for future versions maintained
- Incremental development reflects the way that we solve problems. We rarely work out complete problem solution in advance but move toward a solution in series of steps

Incremental Model



Advantages

- Generates working software quickly
- Flexible to customer
- Easier to test and debug
- Easier to manage risk

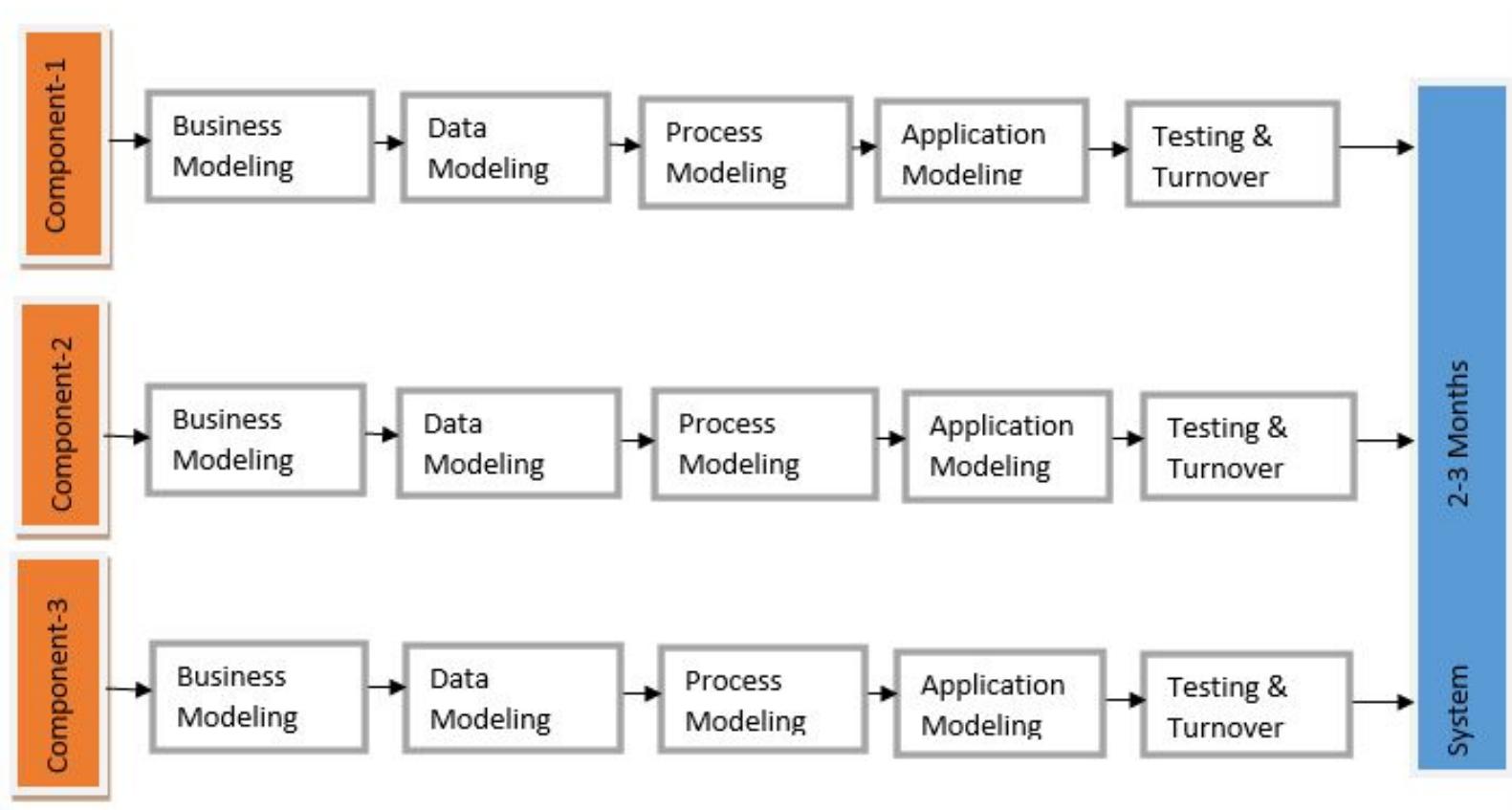
Disadvantages

- Total cost is based on number of iteration
- System structure tends to degrade as new increments are added

RAD

- Extension for the incremental model
- Rapid application development
- The software project which we can break into modules can use this model
- Development of each module requires basic SDLC steps like waterfall steps

RAD



Five Core Elements

- Business Model
- Data Model
- Process Model
- Application Generation
- Testing and Turnover

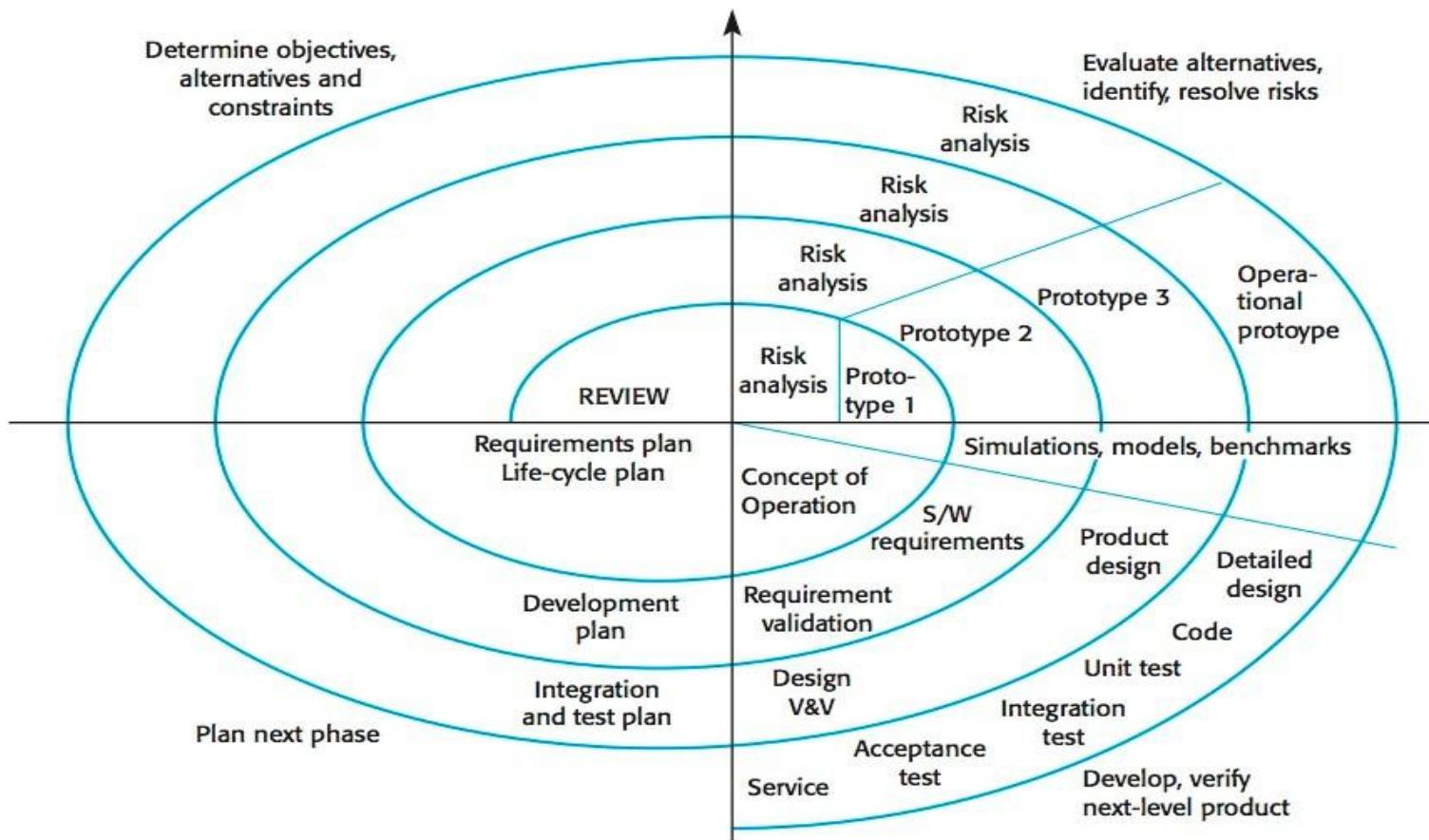
Another Explanation of RAD

- Requirement Planning
- User Description
- Construction
- Cut-out

Merits and Demerits

- Fast
 - Customer satisfaction
-
- Requires active customer
 - Not good for big projects
 - Not efficient

Spiral Model



Spiral Model

- Risk driven approach
- Prototyping, simulations, benchmarking may be done to resolve uncertainty/risk
- Development step depends on remaining risk; e.g.
 - Do prototype for user interface risk
 - Use basic waterfall model when user interface and performance issues are understood but only development is remaining

Advantage

- Critical high risk functions are developed first
- The model provides early indication of risk
- User can be closely tied to all lifecycle steps
- Early and frequent feedback
- Frequent releases
- Suitable for large system
- Possible to add additional functionality in later stage

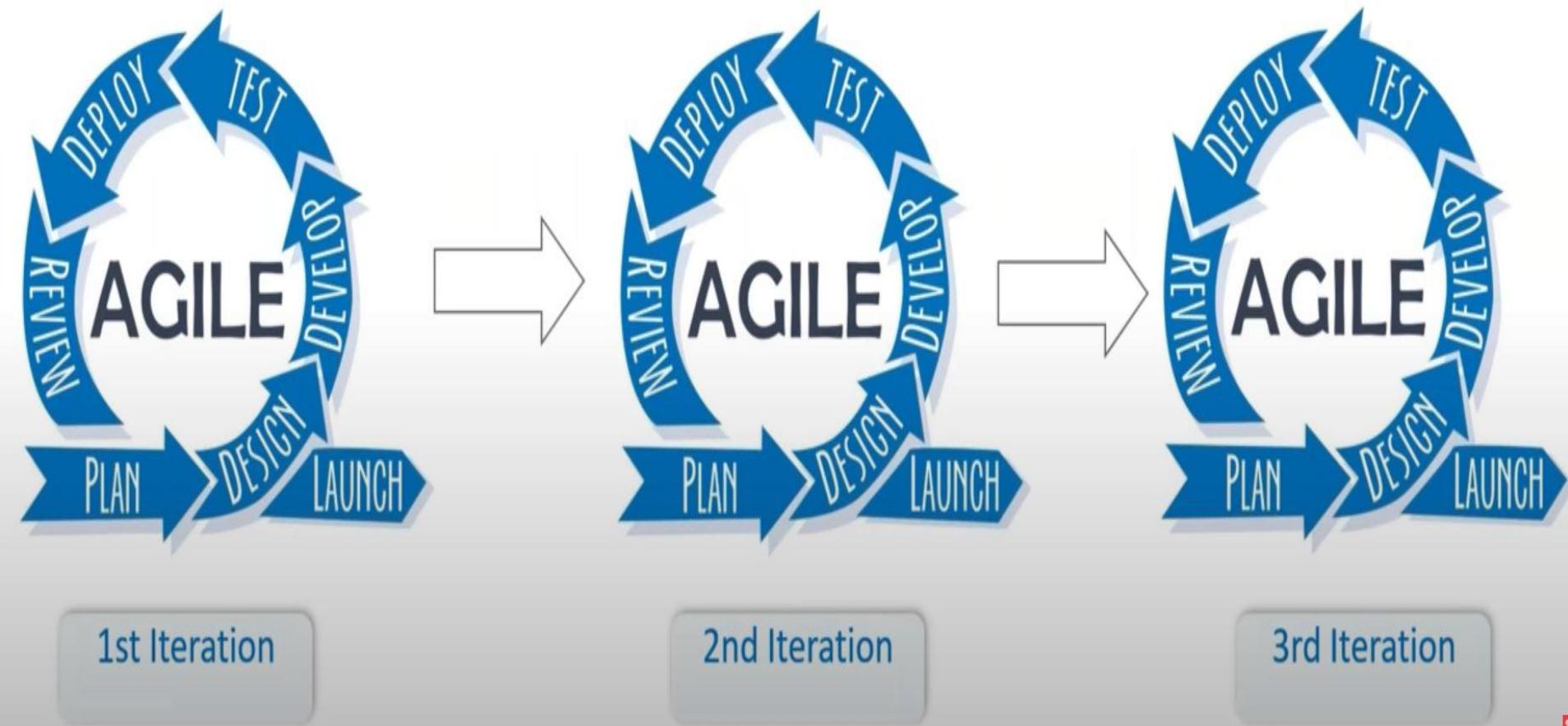
Disadvantage

- What is the impact of using spiral model for small and low risk projects?
 - Time spent for evaluating risks too large for small and low risk projects
 - Time spent for planning, resetting objectives, doing risk analysis and prototyping may be excessive
- Risk is not meeting the schedule on budget
- Expertise are required for risk analysis
- Complex and Costly

Agile

- Main aim of agile is that we build the framework that is nimble enough or agile enough to adjust the changing demands of customer
- It is an iterative and incremental process
- Agile is an idea, based on that idea multiple frameworks are there
 - Scrum, XP, Crystal, FDD, DSDM, Kanban

Agile



1st Iteration

2nd Iteration

3rd Iteration

Terms and Value of Agile

- People over Processes and Tools
- Working software over Comprehensive document
- Customer collaboration over Rigid contract
- Responding to change rather than following a plan

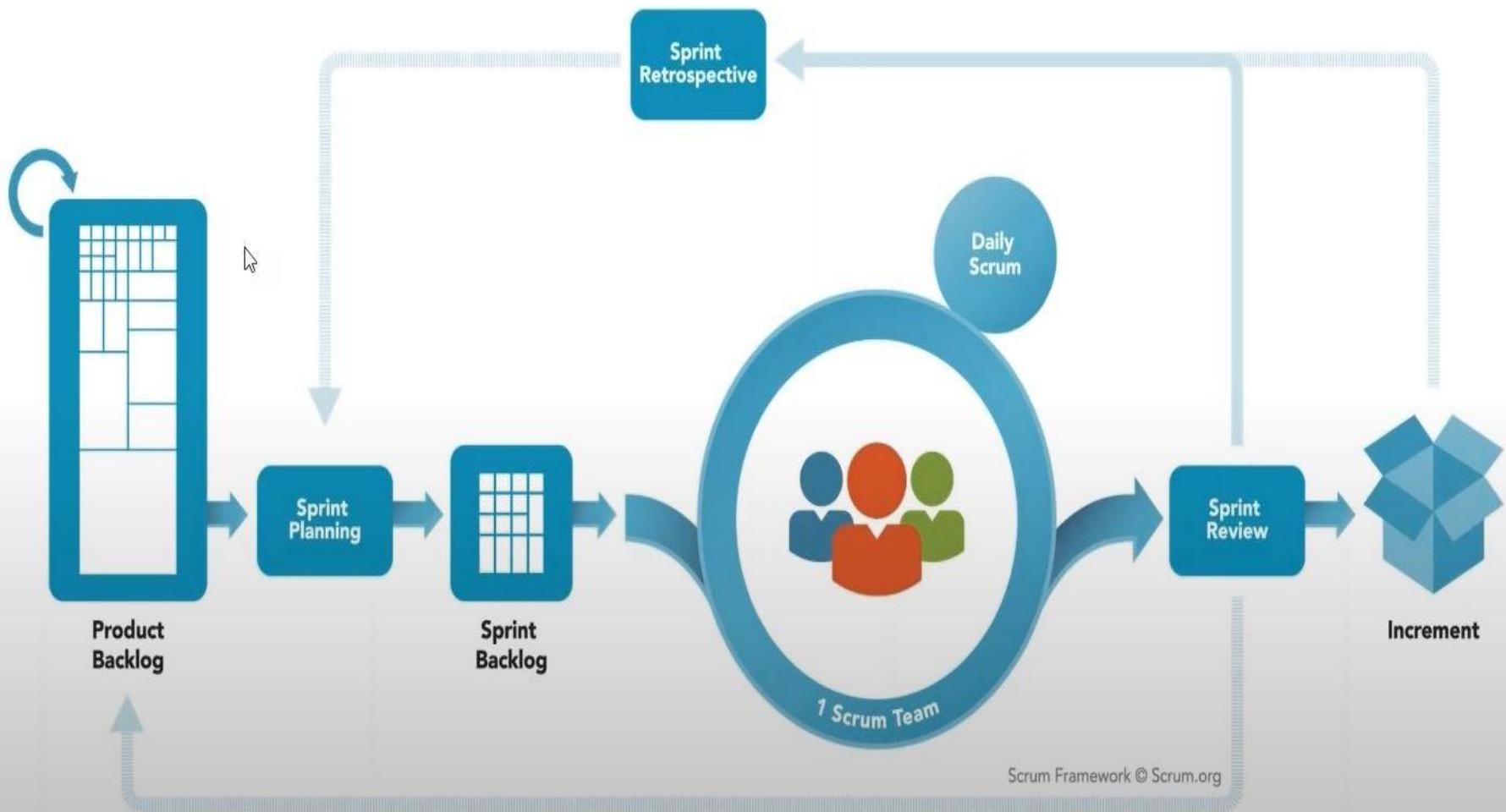
Principle of Agile

- Satisfy the customer
- Welcome changing requirements
- Deliver working software frequently
- Frequent interaction with stockholder
- Motivated individual
- Face-to-face communication
- Measure by working software
- Maintain constant pace
- Sustain technical excellence and good design
- Keep it simple
- Empower self-organization team
- Reflect and adjust continuously

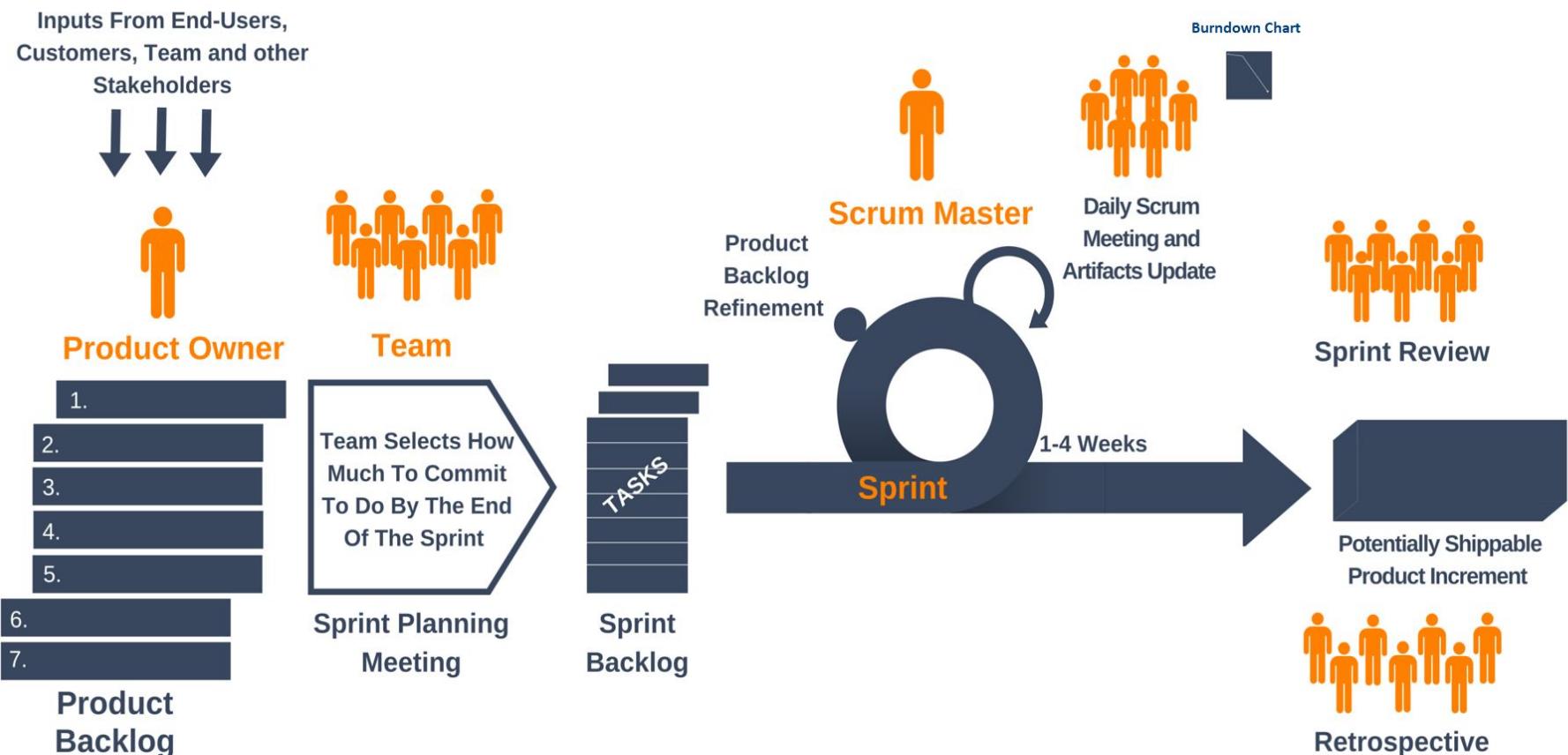
Scrum

- Most popular framework in Agile
- Scrum is a light weight agile project management framework that can be useful to manage creative and incremental projects of all type
- Iterative and Incremental approach
- Inspired by Rugby
- Scrum says we have to have cross-functional team and they have to be focused on advancing the common goal

Scrum Overview



Scrum



ISO-9000

- ISO stands for International Standardization organization
- It is an international body which is recognized by the UN, which sets up the quality and standards for various production services
- Various certifications are there like
 - ISO 3100 = Risk Management
 - ISO 9000 = Quality Management
 - ISO 14000 = Environment Management
 - ISO 26000 = Social Responsibility

ISO-9000

- ISO 9000 is a family of quality based standards
- Quality Management certification category
- What is Quality Management ?
- Consist 4 family member
 - ISO : 9001
 - ISO : 9002
 - ISO : 9003
 - ISO : 9004

ISO-9000 Timeline

1980

- Technical Committee 176 formed

1987

- First edition

1994

- First minor revision

2000

- First major revision

2008

- Second minor revision

2015

- Second major revision

ISO-9000 Quality Principles

- Customer Focus
- Effective Leadership
- Involvement
- Process Approach
- System Approach to Management
- Continual Improvement
- Factual Approach to Decision Making
- Mutually Beneficial Supplier Relationship

CMM

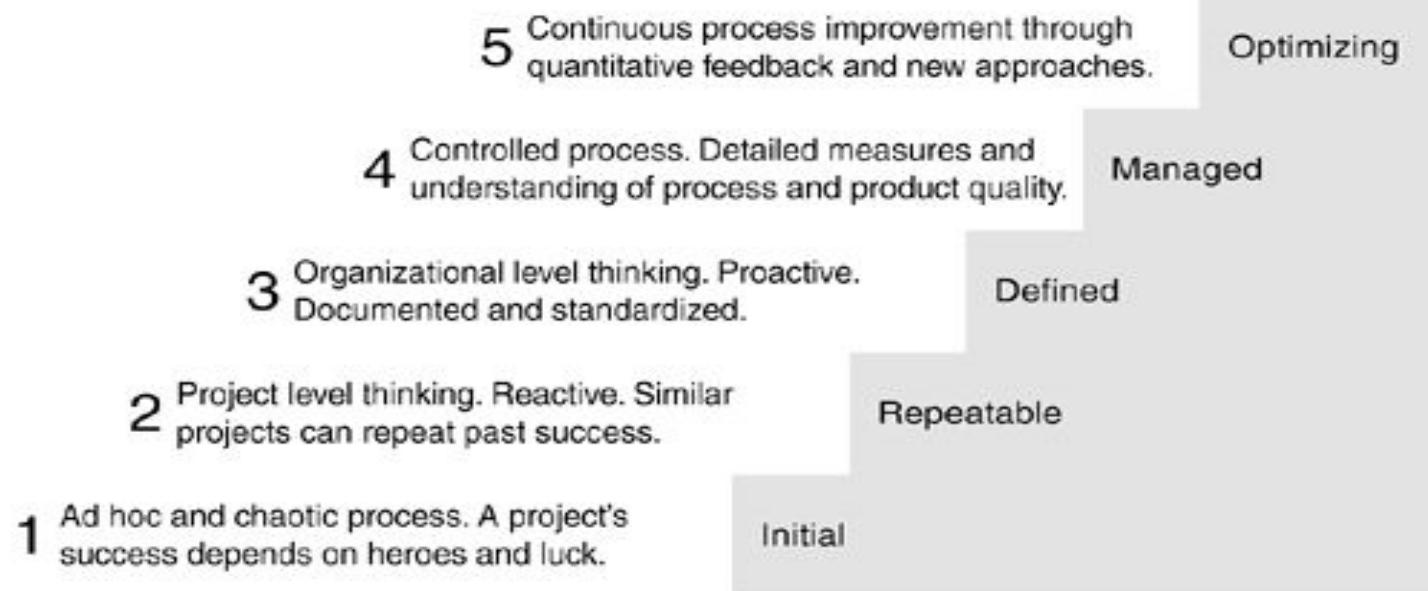
- Capability Maturity Model
- Developed by SEI (Software Engineering Institute)
- CMM is a strategy for improving the software process, irrespective of the actual life cycle model, which is being used
- CMM is used to judge the maturity of software processes of an organization, and to identify key practices that are required to increase the maturity of these processes

CMM Levels

- There are 5 levels in CMM
- Measures a maturity level of an organization based on certain key process area
 - The project
 - Clients
- Each level ranks the organization according to what kind of standard process company following for the particular subject area
- One is minimum, 5 is maximum

CMM Levels

CMM Software Maturity Levels



ISO 9000 is a set of international standards on quality management and quality assurance developed to help companies effectively document the quality system elements needed to an efficient quality system.

Focus is customer supplier relationship, attempting to reduce customer's risk in choosing a supplier.

It is created for hard goods manufacturing industries.

ISO9000 is recognized and accepted in most of the countries.

It specifies concepts, principles and safeguards that should be in place.

This establishes one acceptance level.

Its certification is valid for three years.

It focuses on inwardly processes.

It has no level.

It is basically an audit.

It is open to multi sector.

Follow set of standards to make success repeatable.

SEI (Software Engineering Institute), Capability Maturity Model (CMM) specifies an increasing series of levels of a software development organization.

Focus on the software supplier to improve its internal processes to achieve a higher quality product for the benefit of the customer.

It is created for software industry.

SEICMM is used in USA, less widely elsewhere.

CMM provides detailed and specific definition of what is required for given levels.

It assesses on 5 levels.

It has no limit on certification.

It focus outwardly.

It has 5 levels:

- (a). Initial
- (b). Repeatable
- (c). Defined
- (d). Managed
- (e). Optimized

It is basically an appraisal.

It is open to IT/ITES.

It emphasizes a process of continuous improvement.

Model Checking Using SPIN/PROMELA

Formal Verification :

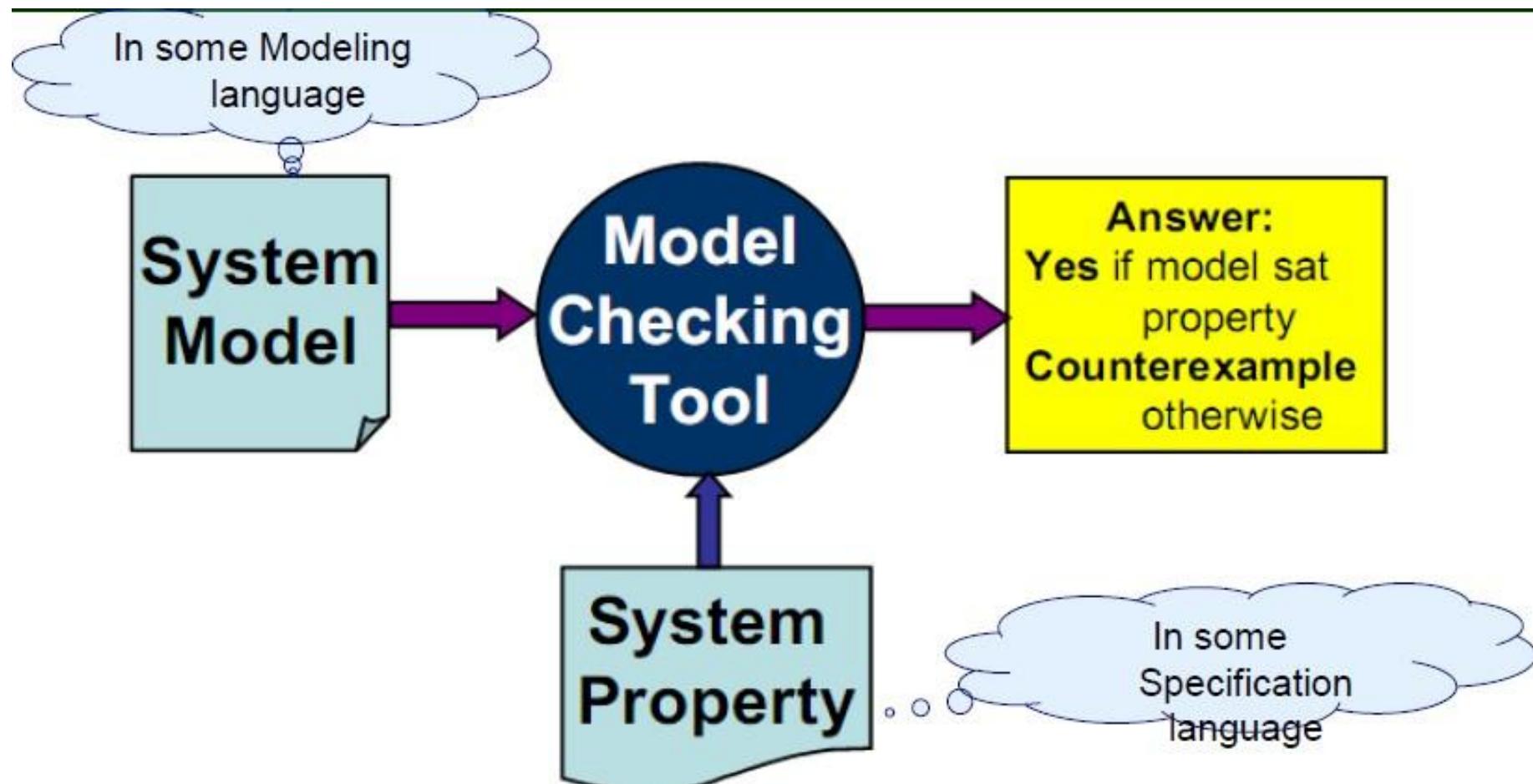
- The process of Formal Verification consists of
 - Requirements
 - Capture Modeling
 - Specification
 - Analysis
 - Documentation
 - n
- In practice, some phases may overlap
The overall process is iterative rather than sequential

Model Checking

- Model checking
 - Is one of the powerful FORMAL VERIFICATION technique
 - Allows one to verify temporal properties of a finite state representation
 - The finite state representation is that of a typical concurrent system
 - The representation is a model of the system
- The basic idea is
 - That a finite state model of a system is systematically explored in order to determine whether or not a given temporal property holds
 - Deliver a counter-example if the specified property does not hold.

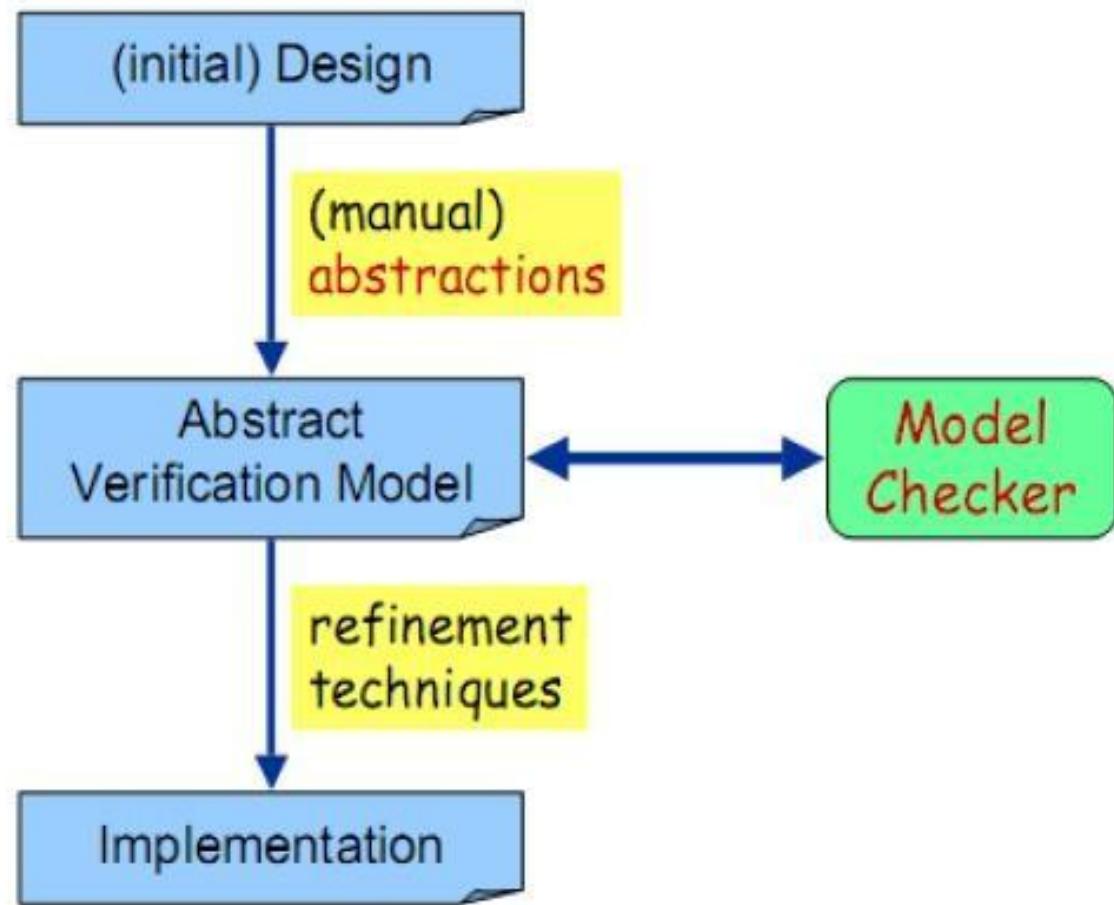
Model Checking

- State space explosion
 - Because a complete state space is generated for a given model, the analysis may fail due to lack of memory, if the model is too large.
 - Can be tackled via abstraction
- Verifying design model
 - Against specification for finite state concurrent systems
- It is an automated technique wherein
 - Inputs
 - Finite state model of the system and properties stated in some standard formalism
 - Outputs
 - Property valid against the model or not



The entire process is automated

Our Focus
here.....



Applicability: Distributed

- Specific concern on the distributed systems
 - Network Applications
 - Data Communication
 - Protocols Multithreaded code
 - Client-Server applications
- Suffer from common design flaws

Common Design Flaws..

- Deadlock

- Livelock,

- Starvation

- Underspecification

- Unexpected reception of messages

- Overspecification

- Dead code

- Violations of constraints

- Buffer overruns

- Array bound violations

Model Checking Definition

- “Model checking is an automated technique that,
 - Given a finite-state model of a system and
 - A logical property, systematically checks whether this property holds for (a given initial state in) that model”

$$M, s \models p$$

- Does system model **M** with initial state **s** satisfy system property **p**
- **M** given as a state machine, that is finite-state
- **p** is usually specified in temporal logic

Model Checking with SPIN

- Involves three steps Modeling
 - Convert the design into a formalism to be accepted by the model checking tool SPIN
- Specification
 - State the properties that the design must satisfy Must be complete
- Verification
 - Normally based on a tool i.e. on
 - SPIN Is automatic
 - Analysis of verification results is, however, manual

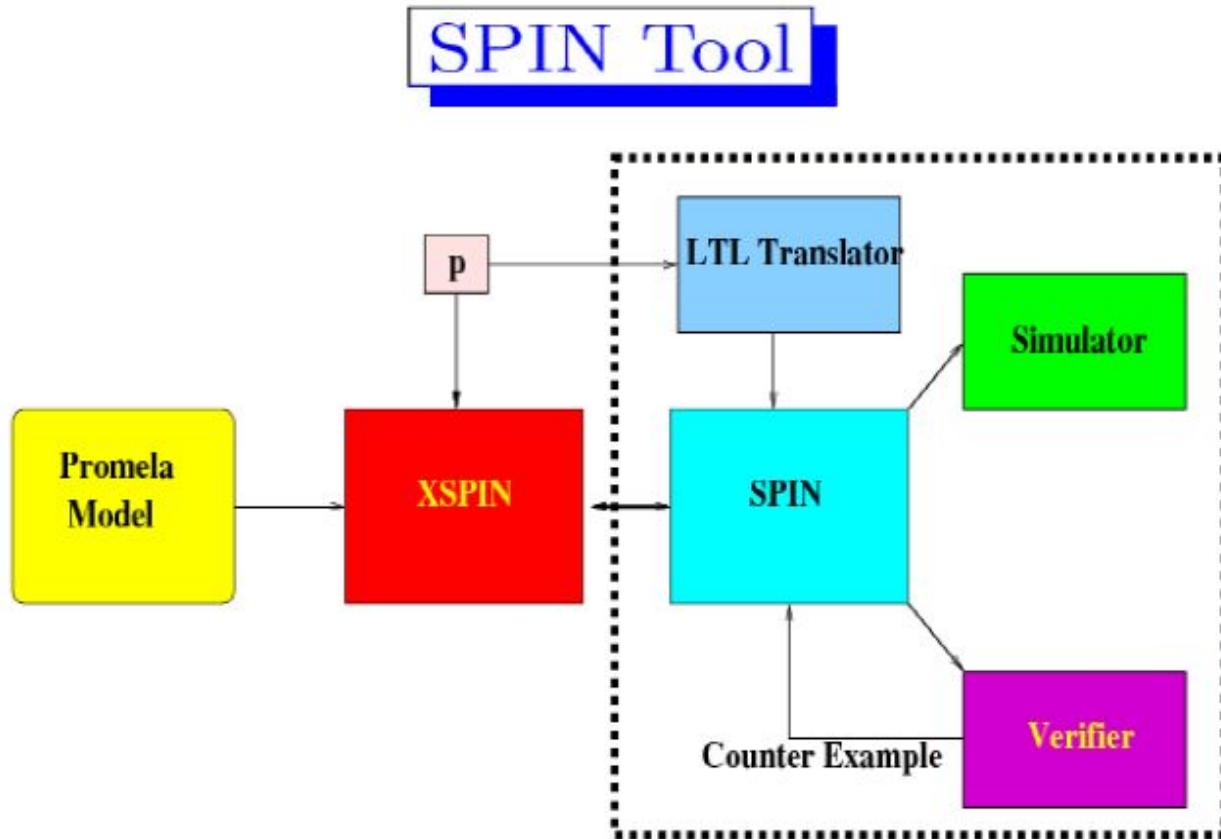
Introduction to

- SPIN – Simple Promela INterpreter
 - A tool for analyzing the logical consistency of concurrent systems, specifically of data communication protocols
 - System model of a concurrent system is described in PROMELA
- PROMELA – PRocess MEta Language
 - Specification language to model finite-state systems & allows dynamic creation of concurrent processes
 - Expressive enough to describe processes and their interactions in Synchronous as well as Asynchronous manner
 - Resembles the programming language C

Introduction to

- PROMELA and SPIN/XSPIN are
 - Developed by Gerard Holzmann at Bell Labs
 - Freeware for non-commercial use
 - Is a State-of-art model checker

Introduction to



Introduction to SPIN

- Major versions:

1.0	Jan 1991	initial version [Holzmann 1991]
2.0	Jan 1995	partial order reduction
3.0	Apr 1997	minimised automaton representation
4.0	late 2002	Ax: automata extraction from C code

- Some success factors of SPIN
 - “Press on the button” verification (model checker)
 - Very efficient implementation (using C)
 - Nice Graphical user interface (Xspin)
 - Not just a research tool, but well supported
 - Contains more than two decades research on advanced computer aided verification (Many optimization algorithms)

Introduction to

- SPIN's starting page:
 - <http://www.spinroot.com>
 - Basic SPIN manual
 - Getting started with
 - Xspin Getting started
 - with SPIN Examples and
 - Exercises
 - Concise Promela Reference
- Proceedings of all SPIN
 - workshops

Gerard Holzmann's website for papers on
SPIN

PROMELA

- Defining a validation model consisting of
 - A set of states incorporating info about values of variables, program counters etc.
 - A transition relation
- A representation of a FSM in terms of
 - Processes
 - Message Channels
 - State Variables
- Only the design of consistent set of rules to govern interaction amongst processes in a Distributed System NOT the implementation details

PROMELA

- PROMELA model consists of
 - Process declarations
 - Channel declarations
 - Variable declarations Type declarations INIT
 - process

As mentioned, PROMELA model = FSM (Usually a very large)

But it is finite and hence has

- No unbounded data
- No unbounded channels

PROMELA

- A process type (proctype) consists of
 - A name, list of formal parameters, declarations of local variables and body
- A process
 - Executes concurrently with all other processes
 - Communicates with other processes using channels & global variables
 - May access shared variables
 - Defined by proctype declarations
- Each process has
 - Its program counter

PROMELA Variables and Basic Data

- PROMELA variables

- Provide the means of storing information about the system being modelled
- May hold global information on the system or information that is local to a particular component
- Supports five basic data types

Name	Size (bits)	Usage	Range
bit	1	unsigned	0...1
bool	1	unsigned	0...1
byte	8	unsigned	0...255
short	16	signed	$-2^{15} - 1 \dots 2^{15} - 1$
int	32	signed	$-2^{31} - 1 \dots 2^{31} - 1$

PROMELA Variables and Basic Data

- PROMELA variables

- Variables must be declared before they can be used
- Variable declarations follow the style of the C programming language
 - A basic data type followed by one or more identifiers and optional initializer
 - byte count, total = 0
- By default all variables of the basic types are initialized to 0.
- As in C, 0 (zero) is interpreted as false while any non-zero value is interpreted as true

The init process

- All PROMELA programs must contain an init process
 - Is similar to the main() function within a C program
 - The execution of a PROMELA program begins with the init process
 - An init process
 - Takes the form:
 - init /* local declarations and statements */
 - init {skip}
 - While a proctype definition declares the behavior of a process, the instantiation and execution of a process definition is coordinated via the init process.

Statement

- A process body consists of sequence of statements
- A statement is either
 - Executable: It can be executed
 - immediately Blocked: It cannot be executed
- An assignment statement is always executable
 - E.g. `x = 2;`

Statement

- An expression is also a statement;
 - It is executable if it evaluates to non-zero
 - skip, $2 < 3$ are always executable
 - $X < 27$ is executable only if the value of x is less than 27
 - A run statement is executable
 - Only if the process can be created
 - Returns 0 if this cannot be done
 - Value otherwise returned is a run-time process ID number
 - `run()` is defined as an operator and so can be embedded in other expressions.

```
proctype A(byte state; short set)
{   (state==1) → state = set
}
init {run A(1,3) }
```

Executability of

• Promela

- Does not make a distinction between a condition and a statement
 - E.g. the simple boolean condition $a == b$ represents a statement in Promela
- Statements are either executable or blocked.
 - The execution of a statement is conditional or it being blocked.
- Notion of statement executability provides the basic means by which process synchronization can be achieved.
- E.g.
 - `while (a != b) skip /*Conventional busy wait */`
 - `(a == b) /* Promela equivalent */`

Hello

- A simple two process system
 - ```
proctype hello() { printf("Hello")}

proctype world() {printf ("World \n")}

init { run hello(); run world(); }
```
- init is the starting point
- run operator is executable only if process instantiation is possible
- If a run is executable than a pid is returned. The pid for a process can be accessed via the predefined local variable \_pid.
- The execution of run does not wait for the associated process to terminate. i.e. further applications of run will be executed concurrently

# Process

- A process can be instantiated also by using “active” in front of proctype definition.
  - i.e. HelloWorld can also be instantiated as

```
active proctype hello() {printf("Hello")}

active proctype world()
{printf("World\n")}
```
- Multiple instances of the same proctype declaration can be generated using an optional array suffix , e.g.

```
active [4] proctype hello() {printf("Hello")}

active [7] proctype world()
{printf("World\n")}
```

```
proctype Foo(byte x) {
 ...
}

init {
 int pid2 = run Foo(2);
 run Foo(27);
}

active [3] proctype Bar() {
 ...
}
```

- a process can be created at any point in the execution (even within any process)
- processes can also be created using active in front of proctype declaration

# Other Data

- Arrays
  - An array type is declared as int table[max]
  - This generates an array of integers i.e. table[0], table[1],...
- Enumerated Types
  - A set of symbolic constants is declared as

```
mtype = {LINE_CLEAR, TRAIN_ON_LINE, LINE_BLOCKED}
```

  - A program can only contain one mtype declaration which must be global
- Structures
  - A record data type is declared as

```
typedef msg {byte data[4], byte checksum}
```

  - Structure access is as in C

```
msg message;
... message.data[0]
```

# PROMELA – An illustration

```
/* a Hello World PROMELA model for SPIN
*/
/* A "Hello World" Promela model for SPIN. */
active proctype Hello() {
 printf("Hello process, my pid is: %d\n", _pid);
}
init {
 int lastpid;
 printf("init process, my pid is: %d\n", _pid);
 lastpid = run Hello();
 printf("last pid was: %d\n", lastpid);
}
```

- How many processes are created,  
here?

# Statement

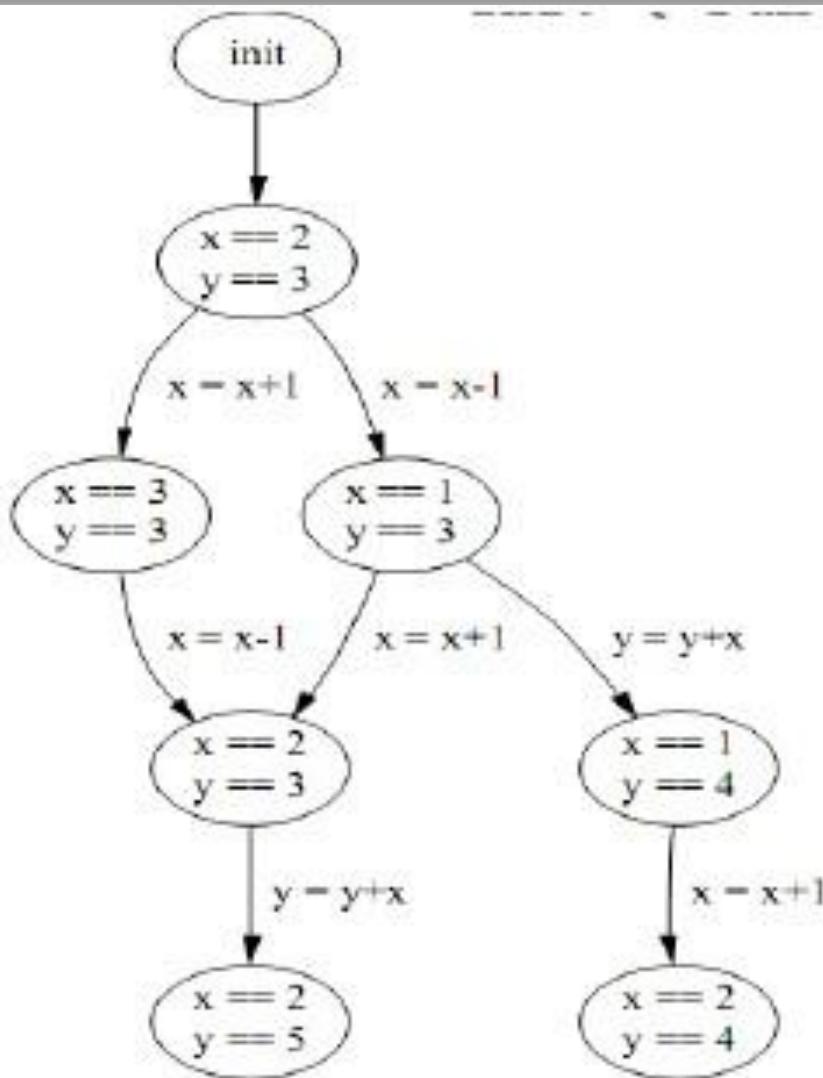
- Two types of statement delimiters
  - ; and ->
  - Use the one that is most appropriate at the given situation Usually ; is used between ordinary statements
  - -> is often used after “guards” in a if OR do statement, pointing at what comes next
  - These can be used interchangeably.

# Interleaving

- PROMELA processes execute concurrently
- Non-deterministic scheduling of the processes Processes are interleaved
  - Statements of different processes do not occur at the same time
  - Exception: rendezvous communication
- All statements are atomic;
  - each statement is executed without interleaving with other processes.
- Each process may have several different possible actions enabled at each point of execution
  - Only one choice is made, non-deterministically i.e. randomly

```
init {run A(); run B();}
```

```
byte x = 2, y = 3;
proctype A(){x = x + 1}
proctype B(){
 x = x - 1;
 y = y + x
}
```



# Deterministic V/s Nondeterministic

## Behaviour

- Deterministic behaviour
  - A process is deterministic if for a given start state, it behaves in exactly the same way ; if supplied with the same stimuli from its environment.
- Non-deterministic behaviour
  - A process is non-deterministic if it need not always behave in exactly the same way ; each time it executes from a given start state with the same stimuli from its environment
  - Hence, race conditions can occur..

# Race conditions

- Solutions

- Use standard mutex
- algorithms Use atomic sequences

```
byte state = 1;
proctype A() { { (state==1) -> state=state + 1 } }
proctype B() { { (state==1) -> state=state - 1 } }
init { run A(); run B() }
```

What could be the output ?

# Race conditions

- Solutions

- Use standard mutex
- algorithms Use atomic sequences

```
byte state = 1;
proctype A() {atomic{ (state==1) → state=state + 1}}
proctype B() {atomic{ (state==1) → state=state - 1}}
init { run A(); run B() }
```



What could be the output ?

# Using atomic

- atomic keyword
  - Helps avoid the undesirable interleaving of the PROMELA execution sequences...
  - Restricts the level of interleaving and so
    - Reduces complexity when it comes to validating a PROMELA model.
  - However, atomic should be used carefully...

# PROMELA Control

## Control structures

- Three ways for achieving control flows
  - Statement sequencing
  - Atomic sequencing
  - Concurrent process execution
- PROMELA supports three additional control flow constructs
  - Case selection
  - Repetition
  - Unconditional Jumps

# Case Selection

```
byte count;
proctype counter()
{
 if
 :: count = count + 1
 :: count = count - 1
 fi
}
```

- Chooses one of the executable choices.
- If no choice is executable, the if-statement is blocked.
  - The executability of the first statement (guard) in each sequence determines whether sequence is executed OR not

# Case Selection

- An example of case selection with

```
if
:: (n % 2 != 0) -> n = n + 1;
:: (n % 2 == 0) -> skip;
fi
```

- If there is at least one choice (guard) executable,
  - The if statement is executable and SPIN non-deterministically choose one of the alternatives.
- The operator `->` is equivalent to ;
  - By convention, it is used within if-statements to separate the guards from the statements that follow the guards.

# Case Selection

- Guards need not be mutually exclusive

**if**

```
:: (x >= y) -> max = x;
```

```
:: (y >= x) -> max = y;
```

**fi**

- If  $x$  and  $y$  are equal then

- The selection of which statement sequence is executed is decided at random, giving rise to non-deterministic choice

# Repetition

- An example of repetition involving two statement sequences

```
do
 :: (x >= y) -> x = x - y; q = q + 1;
 :: (y > x) -> break;
od
```

- do statement is similar to the if statement...
  - However, instead of executing a choice once, it keeps repeating the execution.
  - The (always executable) break statement may be used to exit a do-loop statement and transfers control to the end of the loop.

# Repetition

- The first statement sequence denotes the body of the loop
  - While the second denotes the termination condition
  - Termination, however, is not always a desirable property of these system, in particular, when dealing with reactive systems

```
do
:: (level > max) -> outlet = open;
:: (level < min) -> outlet = close;
od
```

```

byte count;
protoype counter()
{
 do
 :: count = count + 1
 :: count = count - 1
 :: (count == 0) -> break
 od
}

```

| Name  | Size (bits) | Usage    | Range                          |
|-------|-------------|----------|--------------------------------|
| bit   | 1           | unsigned | 0...1                          |
| bool  | 1           | unsigned | 0...1                          |
| byte  | 8           | unsigned | 0...255                        |
| short | 16          | signed   | $-2^{15} - 1 \dots 2^{15} - 1$ |
| int   | 32          | signed   | $-2^{31} - 1 \dots 2^{31} - 1$ |

```
proctype counter()
{
 do
 :: (count != 0) ->
 if
 :: count = count + 1
 :: count = count - 1
 fi
 :: (count == 0) -> break
 od
}
```

# Unconditional Jump

- PROMELA supports the notion of an unconditional jump via the “goto” statement

```
do
 :: (x >= y) -> x = x - y; q = q + 1;
 :: (y > x) -> goto done;
od;
done:
skip
```

- “done” denotes a label
  - A label can only appear after a
  - statement A goto, like a skip, is always executable.

# Assertions

- An assertion is a statement which can be either true or false
- Interleaving assertion evaluation with code execution provides
  - A simple yet very useful mechanism for checking desirable as well as erroneous behavior with respect to our models
- Assertion: Syntax within PROMELA
  - assert( <logical-statement>)
  - E.g. assert(!(doors == open && lift == moving))
- Within PROMELA we can express local assertions as well as global system assertions

# Global Assertions

- A global assertion is also known as a system invariant
  - Is a property that is true in the initial system state and remains true in all possible execution paths.
- To express a system invariant within PROMELA
  - One must define a monitor process that contains the desired system invariant
- To ensure that the global assertion is checked anypoint during the execution
  - An instance of the monitor process has to be run along with the rest of the system model
- In the case of a simulation the checking is not exhaustive, this is achieved within verification mode

```
bit flag1, flag2;
byte mutex;
```

```
active proctype A() {
 flag1 = 1;
 flag2 == 0;
 mutex++;
 mutex--;
 flag1 = 0;
}
```

```
active proctype B() {
 flag2 = 1;
 flag1 == 0;
 mutex++;
 mutex--;
 flag2 = 0;
}
```

```
active proctype monitor() { assert mutex != 2);
```

- What could be the eventual value of mutex ?
- Is it really achieved ?
- Is the assertion preserved or violated, here ?

“Invalid End state” in SPIN

```
bit flag1, flag2; byte mutex, turn;
```

```
active proctype A() {
 flag1 = 1;
 turn = B_TURN;
 flag2 == 0 || turn == A_TURN;
 mutex++;
 mutex--;
 flag1 = 0;
}
```

```
active proctype B() {
 flag2 = 1;
 turn = A_TURN;
 flag1 == 0 || turn == B_TURN;
 mutex++;
 mutex--;
 flag2 = 0;
}
```

```
active proctype monitor() { assert mutex != 2};
```

First software-only solution to the mutex problem for two processes...

# Timeouts

- Reactive systems typically require a means of aborting OR rebooting when a system deadlocks.
- PROMELA provides a primitive statement called timeout for the purpose.

```
proctype watchdog ()
{ do
 :: timeout -> guard!reset
 od
}
```

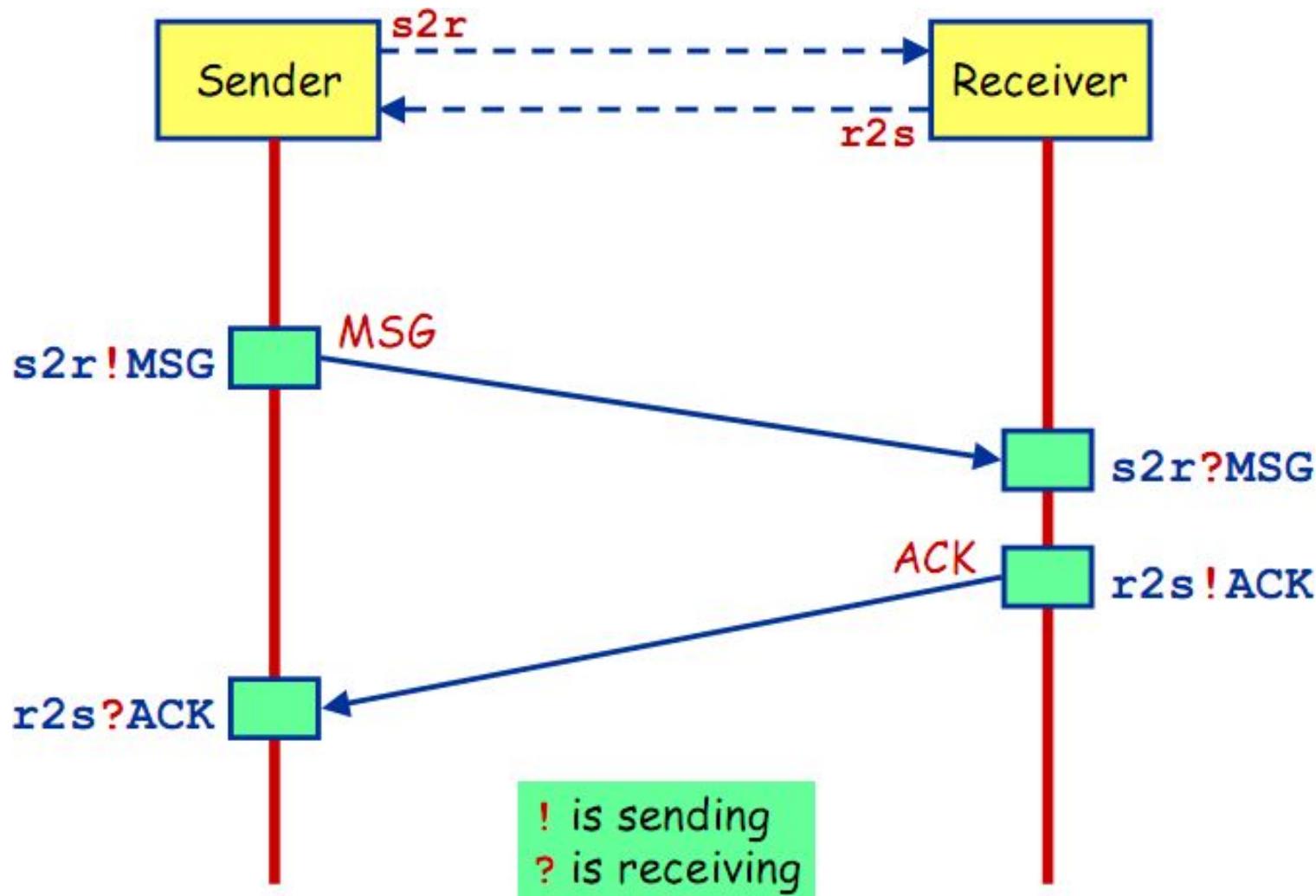
- The timeout condition becomes true when no other statements within the overall system being modeled are executable

# Exceptions

- The unless statement
- A useful exception handling feature
- {statements-1} unless {statements-2}
- Consider an alternate watchdog process.

# Message Channels

- Means to achieve communication between distinct processes?
- PROMELA supports message channels:
  - Provide a more natural and sophisticated means of modelling inter-process communication/data transfer
- Channel declaration:
  - `chan <name> = <dim> of {<t1>,<t2>,<t3> ... <tn>}`
- E.g. `chan q = [1] of {byte}` ; `chan q = [3] of {mtype, int}`
- If `dim = 0` then synchronous. E.g. `chan q = [0] of {bit}`
- A channel can be defined to be either local or global



# Message Channels... Sending-in

- Sending messages through a channel – FIFO buffer ( $\text{dim} > 0$ )
  - Achieved by ! Operator
  - E.g. `in_data ! 4`
  - Type of the channel and variable must match.
- If multiple data values are to be transferred via each message
  - `out_data ! x+1, true, in_data`
- The executability of a send statement is dependent upon the associated channel being non-full

# Message Channels... Receiving from

- Receiving messages is achieved by ? Operator
  - E.g. in\_data ? Msg
- If the channel is not empty, the first message is fetched from the channel and is stored in msg
- Multiple values can also be fetched.
  - E.g. out\_data ? value1, value2, value 3;
- The executability of a receive statement is dependent upon the associated channel being non-empty,

# Message Channels...

- If more data values are sent per message than can be stored by a channel then the extra data values are lost
  - E.g. `in_data ! msg1, msg2 ; msg2` will be lost
- If fewer data values are sent per message than are expected, then the missing data values are undefined.
  - E.g. `out_data ! 4, true` and `out_data? x, y , z`
    - x and y will be assigned the values 4 and true respectively while the value of z will be undefined.

# Message Channels...

- len operator
  - To determine the number of messages in a channel
  - E.g. len (in\_data)
  - If the channel is empty then the statement will block.
- empty, full operators
  - Determine whether or not messages can be received or sent respectively.
    - E.g. empty(in\_data) ; full (in\_data)
- Non-destructive retrieve
  - out\_data ? [x, y, z]
  - Returns 1 if out\_data ? x,y,z is executable otherwise 0.
    - No side-effects. Only evaluation, not execution. No message retrieved.

# Channels as nonlocations

- Output?

```
proctype A(chan q1) {
 chan q2;
 q1?q2;
 q2!123
}
proctype B(chan qforb) {
 int x;
 qforb?x;
 printf("x = %d\n", x)
}
init {
 chan qname[2] = [1] of { chan };
 chan qforb = [1] of { int };
 run A(qname[0]); run B(qforb);
 qname[0]!qforb
}
```

# Communication type

- What was the type of communication pattern observed in the examples till now?
  - Synchronous
  - OR
  - Asynchronous ?
- Why?

# Synchronous

## Communication

- When the channel declaration is
  - `chan ch = [0] of {bit, byte};` i.e. when dim = 0
- If `ch ! x` is enabled and
  - If there is a corresponding receive `ch ? X`
  - that can be executed simultaneously and both the statements are enabled
  - Both statements will handshake and together do the transition
- `chan ch = [0] of {bit, byte}`
- P wants to perform `ch ! 1, 3 +`
- Q wants to perform `ch ? 1, x`
- Then after communication x will be 10.

# Alternating Bit

## Protocol

- To every message, the sender adds a bit
- The receiver acknowledges each message by sending the received bit back.
- The receiver only expects messages with a bit that is expected to receive
- If the sender is sure that the receiver has correctly received the previous message, it sends a new message and it alternates the accompanying bit

# Some Examples

## 1. Hello World!

```
init {
 printf("Hello World!\n")
}
```

## 2. Hello World: Multiple Processes

```
active proctype Hello() {
printf("Hello process, my pid is: %d\n", _pid);
}
```

```
init {
int lastpid;
printf("init process, my pid is: %d\n", _pid);
lastpid = run Hello();
printf("last pid was: %d\n", lastpid);
}
```

## 3. Peterson Algorithm

It is a concurrent programming algorithm for mutual exclusion that allows two or more processes to share a single-use resource without conflict, using only shared memory for communication. It was formulated by Gary L. Peterson in 1981. While Peterson's original formulation worked with only two processes, the algorithm can be generalized for more than two.

The algorithm uses two variables, flag and turn. A flag[n] value of true indicates that the process n wants to enter the critical section. Entrance to the critical section is granted for process P0 if P1 does not want to enter its critical section or vice versa.

```
bool flag[2] = {false, false};
int turn;
```

```
P0: flag[0] = true;
P0_gate: turn = 1;
 while (flag[1] == true &&
turn == 1)
 {
 // busy wait
 }
 // critical section
 ...
 // end of critical section
 flag[0] = false;
```

```
P1: flag[1] = true;
P1_gate: turn = 0;
 while (flag[0] == true &&
turn == 0)
 {
 // busy wait
 }
 // critical section
 ...
 // end of critical section
 flag[1] = false;
```

The algorithm satisfies the three essential criteria: mutual exclusion, progress, and bounded waiting.  
In spin:

```

bool turn, flag[2];
byte ncrit;

active [2] proctype user()
{
 assert(_pid == 0 || _pid == 1);
again:
 flag[_pid] = 1;
 turn = 1 - _pid;
 (flag[1 - _pid] == 0 || turn == _pid);

 ncrit++;
 assert(ncrit == 1); /* critical section */
 ncrit--;

 flag[_pid] = 0;
 goto again
}

```

#### **4. Producer-Consumer Problem**

```

mtype = { P,C }; /* symbols used */
mtype turn = P; /* shared variable */

```

```

active proctype producer(){
do
:: (turn == P) ->
/* Guard */
printf("Produce\n");
turn = C
od
}

```

```

active proctype consumer(){
again:
if
:: (turn == C) ->
/* Guard */
printf("Consume\n");
turn = P;
goto again
fi
}

```

#### **5. Sender and Receiver Process**

##### **5.1: Message with one data item**

```

chan x = [3] of { int };
int turn = 0;

```

```

active proctype sender()
{
 x!3; x!2; x!1;
turn = 1;
 printf("sender process\n");
}

active proctype receiver()
{
 if
 :: (turn == 1) -> int var1, var2, var3;
 x?var1; x?var2; x?var3;
printf("Receiver Process: %d %d %d\n",var1,var2,var3); fi
}

```

## 5.2: Message with multiple data items

```

chan x = [1] of { int , int , int };
int turn = 0;

active proctype sender()
{
 x!3,2,1;
turn = 1;
 printf("sender process\n");
}

active proctype receiver()
{
 if
 :: (turn == 1) -> int var1, var2, var3;
 x?var1, var2, var3;
printf("receiver process: %d %d %d\n",var1,var2,var3); fi
}

```

## 5.2: Other channel operations

```
chan glob = [1] of { chan };
```

```

active proctype A(){
chan loc = [1] of { int }
int var;
glob!loc;
loc?var;
printf("Loc = %d\n",var);
}

```

```

active proctype B(){
chan who;

```

```
glob?who;
who!1000;
}
```

## 6. Alternating Bit Protocol

```
mtype = { msg, ack };

chan to_sndr = [2] of { mtype, bit };
chan to_rcvr = [2] of { mtype, bit };

active proctype Sender ()
{ bit seq_out=1, seq_in;

 do
 :: to_rcvr!msg (seq_out) ->
 to_sndr?ack (seq_in);
 if
 :: seq_in == seq_out ->
 printf("ack received\n"); seq_out = !seq_out
 :: else ->
 printf("ack not received\n"); skip
 fi
 od
}

active proctype Receiver ()
{ bit seq_in;

 do
 :: to_rcvr?msg (seq_in) ->
 printf("msg received\n"); to_sndr!ack (seq_in)
 :: timeout ->
 printf("time out\n"); to_sndr!ack (seq_in)
 od
}
```

---

# **Requirements Engineering Process**

# Requirements Engineering Processes

---

- Processes used to discover, analyse and validate system requirements.

# Topics covered

---

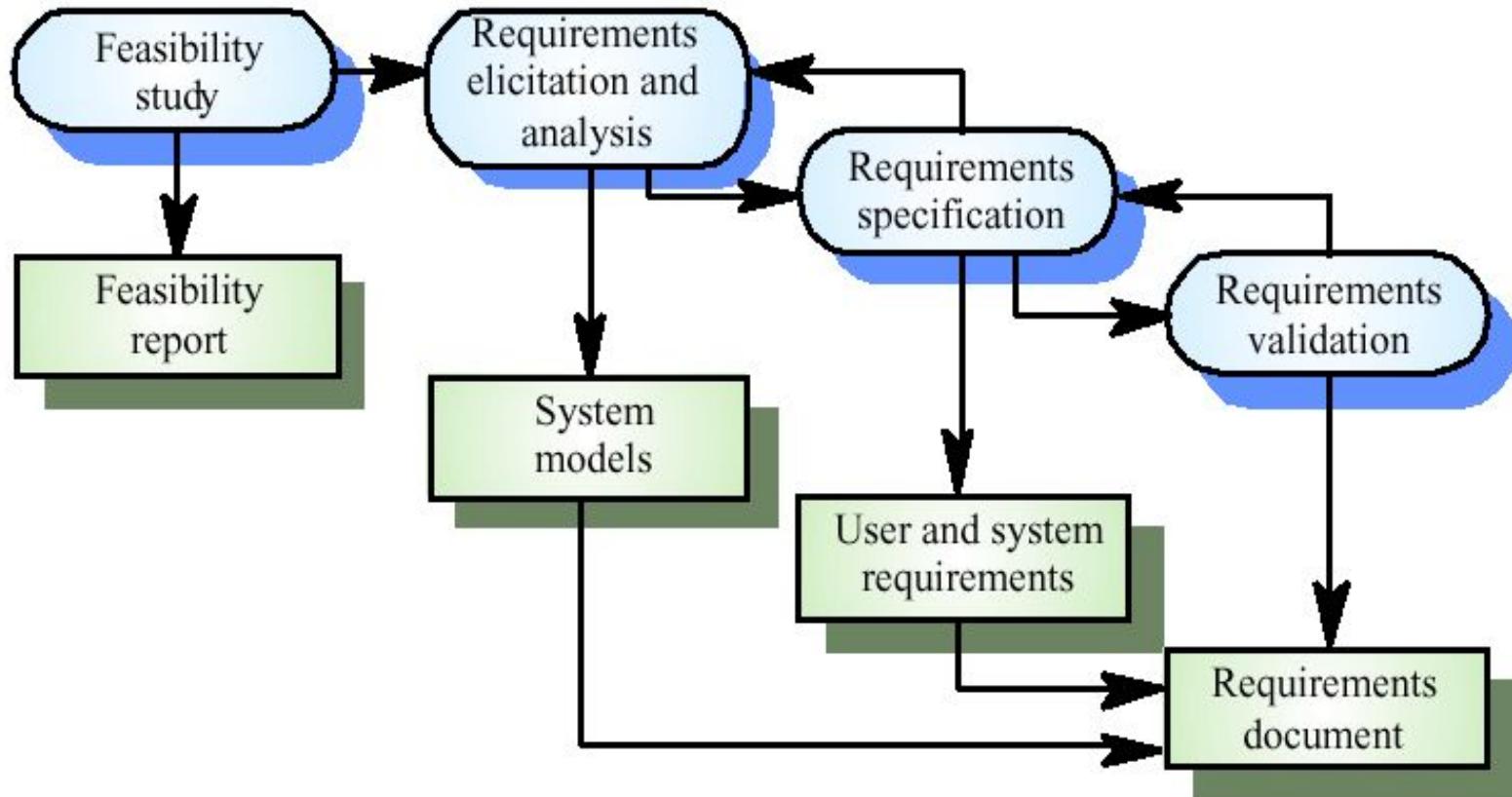
- Feasibility study
- Requirements elicitation and analysis
- Requirements validation
- Requirements management

# Requirements engineering processes

---

- The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements
- However, there are a number of generic activities common to all processes
  - Feasibility Study
  - Requirements elicitation and analysis
  - Requirements validation
  - Requirements management

# The requirements engineering process



# Feasibility study

---

- 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

# Feasibility study implementation

---

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

# Elicitation and analysis

---

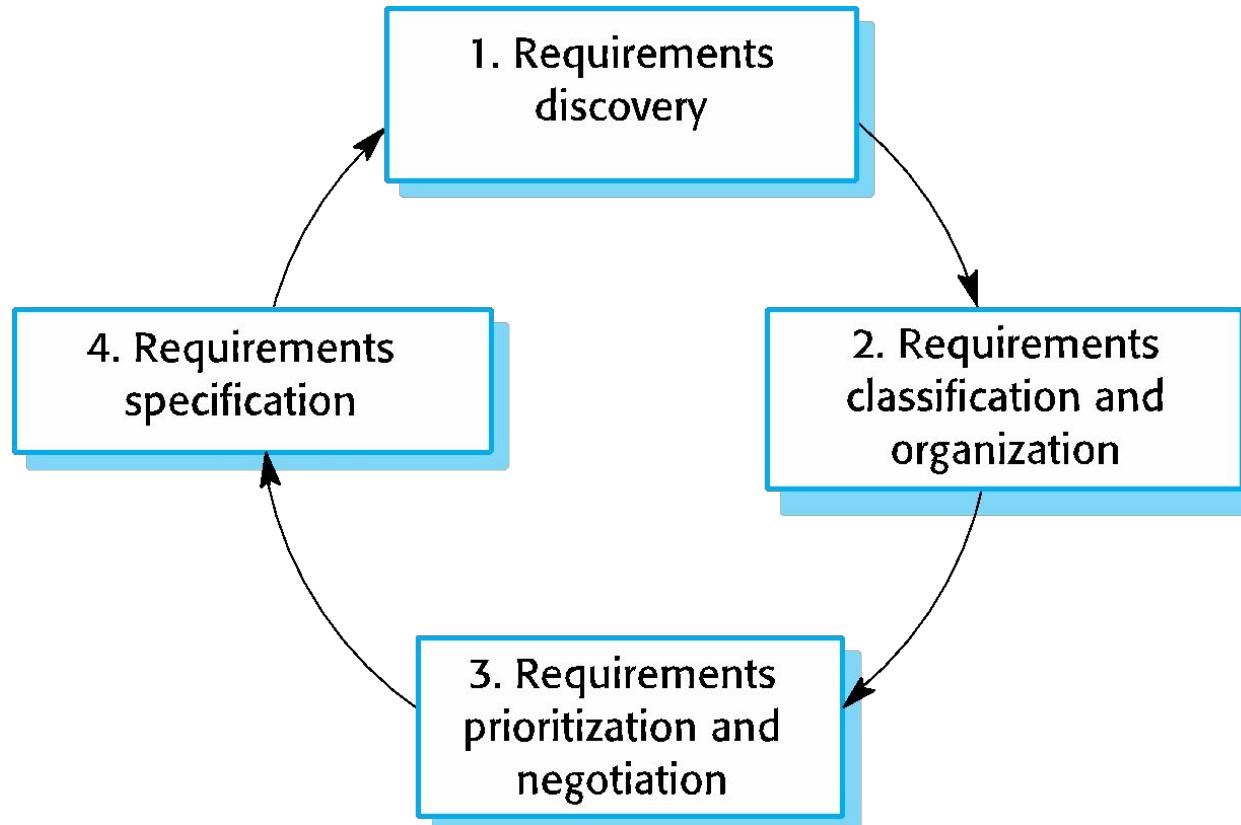
- Sometimes called requirements elicitation or requirements discovery
- Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints
- May involve end-users, managers, engineers involved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

# Problems

---

- 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 elicitation and analysis process



# Process activities

---

- Requirements discovery
  - Interacting with stakeholders to discover their requirements.  
Domain requirements are also discovered at this stage.
- Requirements classification and organization
  - Groups related requirements and organizes them into coherent clusters.
- Prioritisation and negotiation
  - Prioritising requirements and resolving requirements conflicts.
- Requirements specification
  - Requirements are documented and input into the next step.

# Requirements discovery

---

- The process of gathering information about the required and existing systems and distilling the user and system requirements from this information.
- Interaction is with system stakeholders from managers to external regulators.
- Systems normally have a range of stakeholders.

# Interviewing

---

- Formal or informal interviews with stakeholders.
- Types of interview
  - Closed interviews based on pre-determined list of questions
  - Open interviews where various issues are explored with stakeholders.
- Effective interviewing
  - Be open-minded, avoid pre-conceived ideas about the requirements and are willing to listen to stakeholders.
- Prompt the interviewee to get discussions

# Problems with interviews

---

- Application specialists may use language to describe their work that isn't easy for the requirements engineer to understand.
- Usually, interviews are not good for understanding domain requirements.
  - Requirements engineers cannot understand specific domain terminology
  - Some domain knowledge is so familiar that people find it hard to articulate or think that it isn't worth articulating.

# Ethnography

---

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

# Scope of ethnography

---

- Requirements that are derived from the way that people actually work rather than the way in which process definitions say they ought to work
- Ethnography is effective for understanding existing processes but cannot identify new features that should be added to a system.
- Thus, not always appropriate for discovering organizational or domain requirements.

# Requirements specification

---

- The process of writing down the user and system requirements in a requirements document.
- User requirements have to be understandable by end-users and customers who do not have a technical background.
- System requirements are more detailed requirements and may include more technical information.
- The requirements may be part of a contract for the system development

It is therefore important that these are as complete as possible.

# Way of writing requirements specification

---

| Notation                    | Description                                                                                                                                                                                                                                                                                                                                              |
|-----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Natural language            | The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.                                                                                                                                                                                                                                 |
| Structured natural language | The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.                                                                                                                                                                                                     |
| Graphical notations         | Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.                                                                                                                                                                                 |
| Mathematical specifications | These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract |

# Natural language specification

---

- Requirements are written as natural language sentences supplemented by diagrams and tables.
- Used for writing requirements because it is expressive, intuitive and universal. This means that the requirements can be understood by users and customers.

# Guidelines for writing specification

---

- 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.
- Include an explanation (rationale) of why a requirement is necessary.

# 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.

# Structured specification

---

- An approach to writing requirements where the freedom of the requirements writer is limited and requirements are written in a standard way.
- This works well for some types of requirements e.g. requirements for embedded control system but is sometimes too rigid for writing business system requirements.

# Form-based specification

---

- Definition of the function or entity.
- Description of inputs and where they come from.
- Description of outputs and where they go to.
- Information about the information needed for the computation and other entities used.
- Description of the action to be taken.
- Pre and post conditions (if appropriate).
- The side effects (if any) of the function.

# Tabular specification

---

- Used to supplement natural language.
- Particularly useful when you have to define a number of possible alternative courses of action.
- For example, the insulin pump systems bases its computations on the rate of change of blood sugar level and the tabular specification explains how to calculate the insulin requirement for different scenarios.

# Example

---

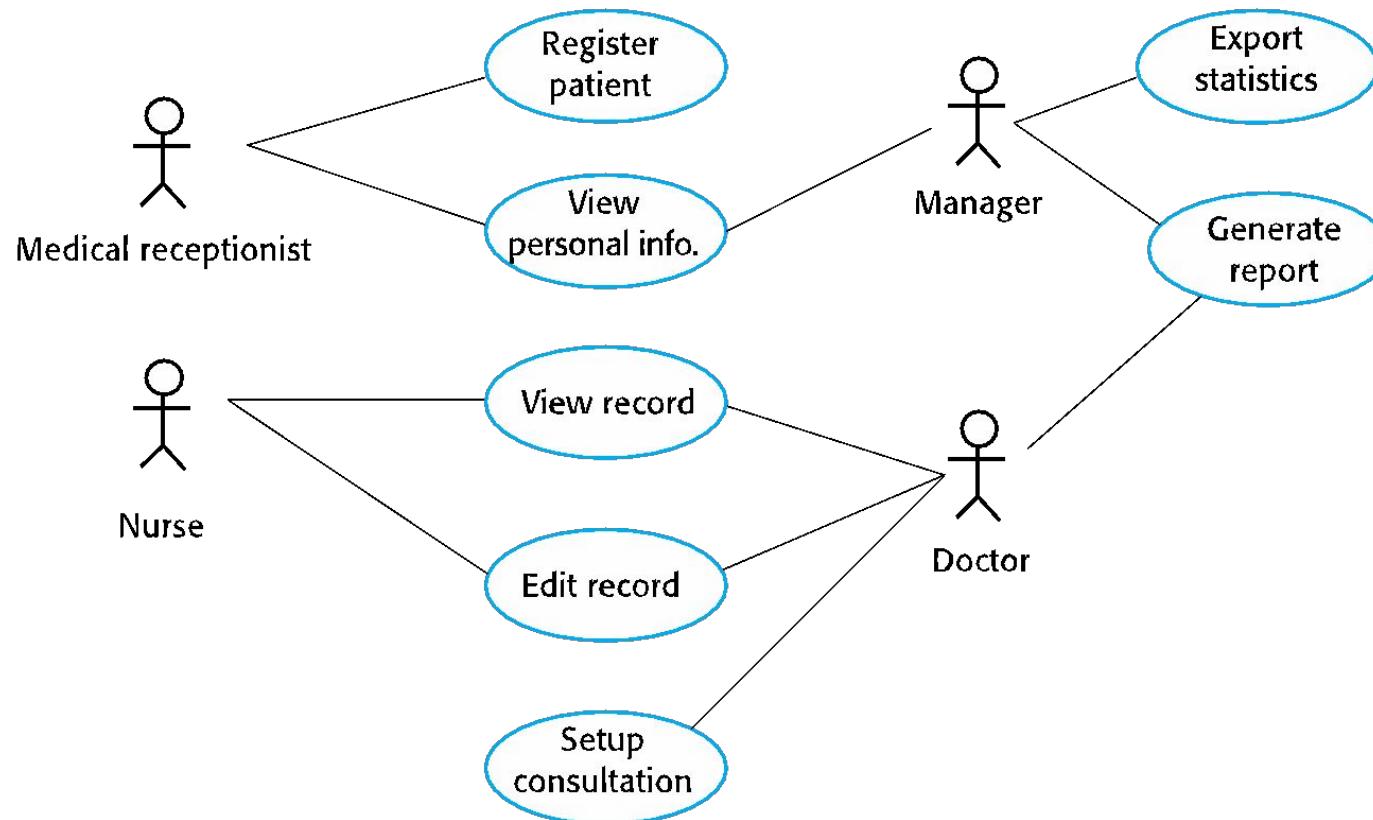
| Condition                                                                                            | Action                                                                                        |
|------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|
| Sugar level falling ( $r_2 < r_1$ )                                                                  | CompDose = 0                                                                                  |
| Sugar level stable ( $r_2 = r_1$ )                                                                   | CompDose = 0                                                                                  |
| Sugar level increasing and rate of increase decreasing<br>$((r_2 - r_1) < (r_1 - r_0))$              | CompDose = 0                                                                                  |
| Sugar level increasing and rate of increase stable or increasing<br>$((r_2 - r_1) \geq (r_1 - r_0))$ | CompDose =<br>round $((r_2 - r_1)/4)$<br>If rounded result = 0 then<br>CompDose = MinimumDose |

# Use cases

---

- Use-cases are a kind of scenario that are included in the UML.
- Use cases 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.
- UML sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system.

# Use cases for Mentcare system



---

# **Requirements validation**

# Requirements validation

---

- ❖
- ❖ It is the process of checking that requirements define the system that the customer really wants.
- ❖
- ❖ Requirements error costs high so validation is important
- ❖
  - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

# Requirements checking

---

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

# Requirements validation techniques

---



## ❖ Requirements reviews

- Systematic manual analysis of the requirements.



## ❖ Prototyping

- Using an executable model of the system to check requirements.



## ❖ Test-case generation

- Developing tests for requirements to check testability.

---

# **Requirements change**

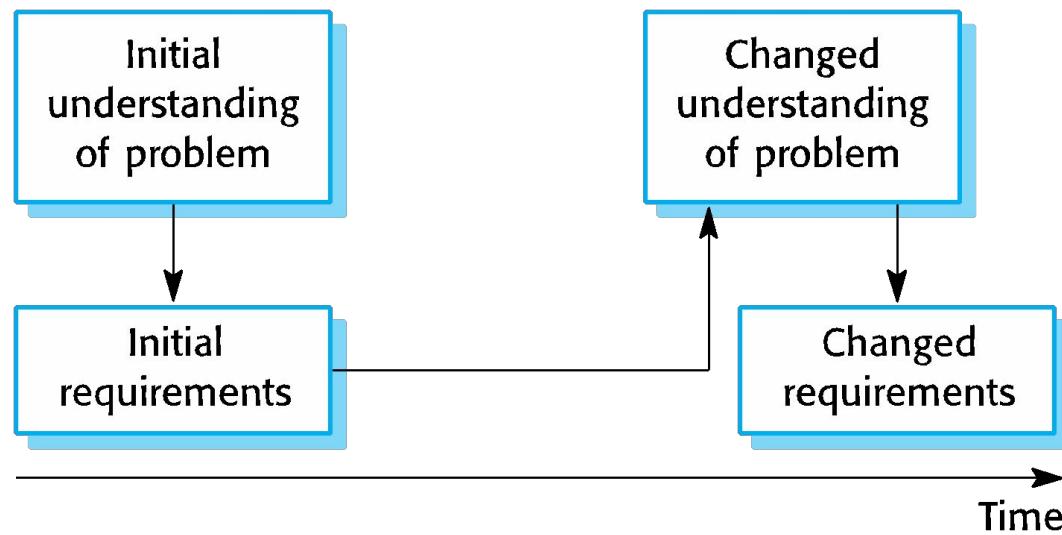
# Changing requirements

---

- ❖ **The business and technical environment of the system always changes after installation.**
  - New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change (with consequent changes in the system support required), and new legislation and regulations may be introduced that the system must necessarily abide by.
- ❖ **The people who pay for a system and the users of that system are rarely the same people.**
  - System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements and, after delivery, new features may have to be added for user support if the system is to meet its goals.

# Requirements evolution

---



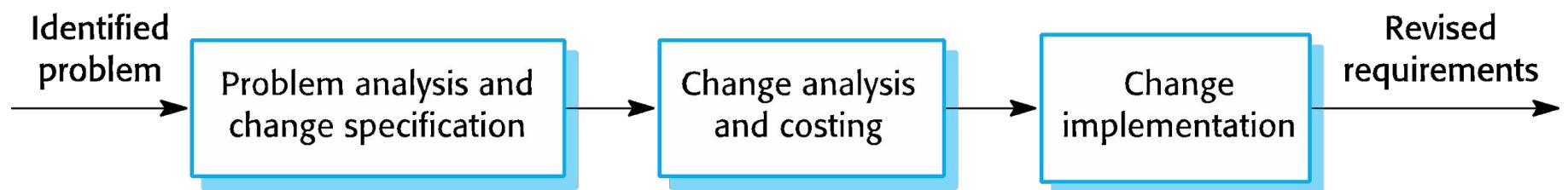
# Requirements management

---

- ❖ Process of managing changing requirements during the requirements engineering process and system development.
- ❖
- ❖ New requirements emerge as a system is being developed and after it has gone into use.
- ❖
- ❖ Need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes. You need to establish a formal process for making change proposals and linking these to system requirements.

# Requirements change management

---



# Requirements change management

---

- ❖ Deciding if a requirements change should be accepted
  - *Problem analysis and change specification*
    - During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.
  - *Change analysis and costing*
    - The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.
  - *Change implementation*
    - The requirements document and, where necessary, the system design and implementation, are modified. Ideally, the document should be organized so that changes can be easily implemented.

# Summary

---

- ❖ **Software requirements:** Services that the system should provide and constraints on its operation and implementation.
- ❖
- ❖ **Types of requirements**
  - User and system
  - Functional and non-functional
- 
- ❖ **Requirements engineering process**
  - Feasibility study; elicitation and analysis; specification; validation; and management

# Structured Vs Object-Oriented Paradigm

---

|              | Structured                                        | Object-Oriented                                      |
|--------------|---------------------------------------------------|------------------------------------------------------|
| Methodology  | SDLC                                              | Iterative/Incremental                                |
| Focus        | Process                                           | Objects                                              |
| Risk         | High                                              | Low                                                  |
| Reuse        | Low                                               | High                                                 |
| Suitable for | Well defined project with stable use requirements | Risky large projects with changing user requirements |

# Structured Vs Object-Oriented Analysis

---

- ❖ **Structured:** Uses DFD's, Decision Table/Tree and E-R diagrams.
- ❖
- ❖ **Object-Oriented:** Use case and object model.
- ❖
- ❖ **Use Case Model:** UML use cases and activity diagrams.
- ❖
- ❖ **Object Model:** Find classes and their relations. Uses sequence and state machine diagrams. Object to ER mapping.

---

# Verification and Validation

---

# Objectives

---

- To introduce software verification and validation and to discuss the distinction between them
- To describe the program inspection process and its role in V & V
- To explain static analysis as a verification technique
- To describe the Cleanroom software development process

# Topics covered

---

- Verification and validation
- Software inspections
- Automated static analysis
- Cleanroom software development

# Verification vs validation

---

- **Verification:**  
"Are we building the product right".
- The software should conform to its specification.
- **Validation:**  
"Are we building the right product".
- The software should do what the user really requires.

# The V & V process

---

- Is a whole life-cycle process - V & V must be applied at each stage in the software process.
- Has two principal objectives
  - The discovery of defects in a system;
  - The assessment of whether or not the system is useful and useable in an operational situation.

# V & V goals

---

- Verification and validation should establish confidence that the software is fit for purpose.
- This does NOT mean completely free of defects.
- Rather, it must be good enough for its intended use and the type of use will determine the degree of confidence that is needed.

# V & V confidence

---

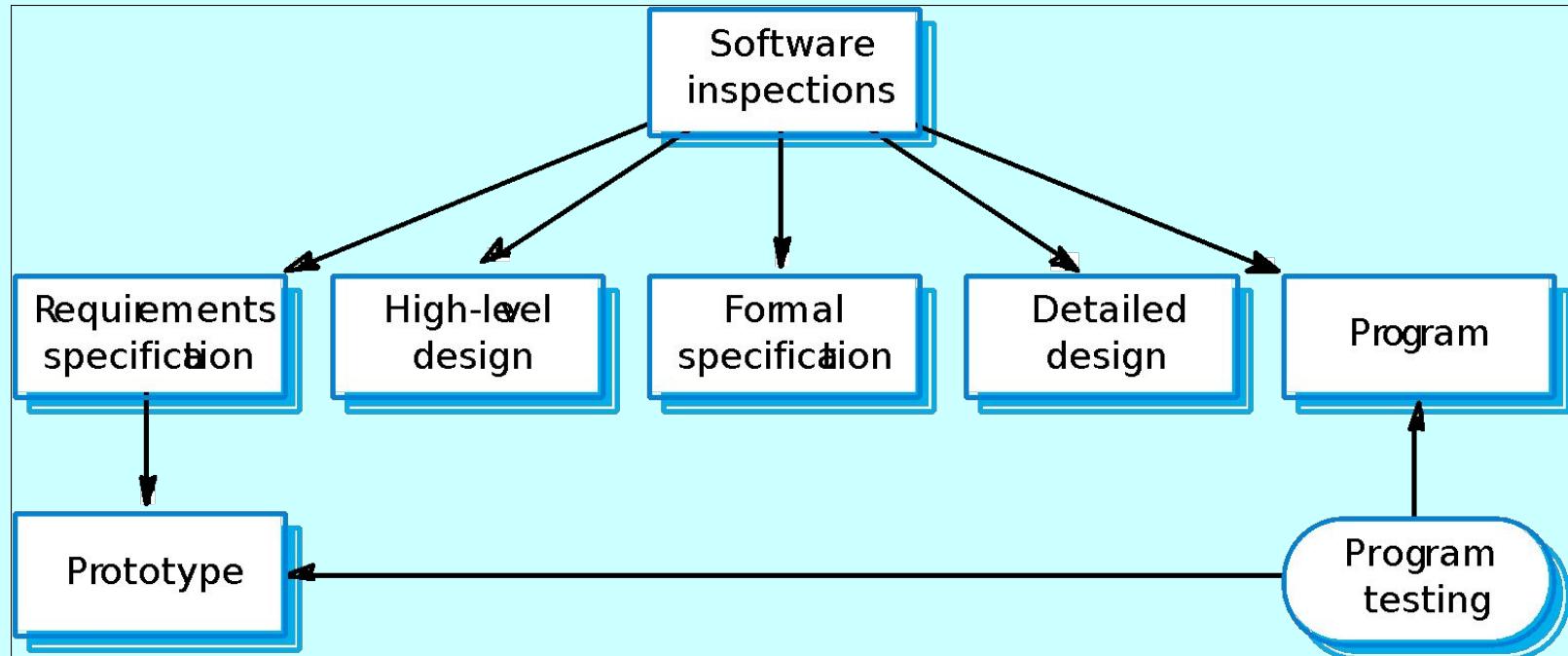
- Depends on system's purpose, user expectations and marketing environment
  - Software function
    - The level of confidence depends on how critical the software is to an organisation.
  - User expectations
    - Users may have low expectations of certain kinds of software.
  - Marketing environment
    - Getting a product to market early may be more important than finding defects in the program.

# Static and dynamic verification

---

- **Software inspections.** Concerned with analysis of the static system representation to discover problems (static verification)
  - May be supplement by tool-based document and code analysis
- **Software testing.** Concerned with exercising and observing product behaviour (dynamic verification)
  - The system is executed with test data and its operational behaviour is observed

# Static and dynamic V & V



# Program testing

---

- Can reveal the presence of errors NOT their absence.
- The only validation technique for non-functional requirements as the software has to be executed to see how it behaves.
- Should be used in conjunction with static verification to provide full V & V coverage.

# Types of testing

---

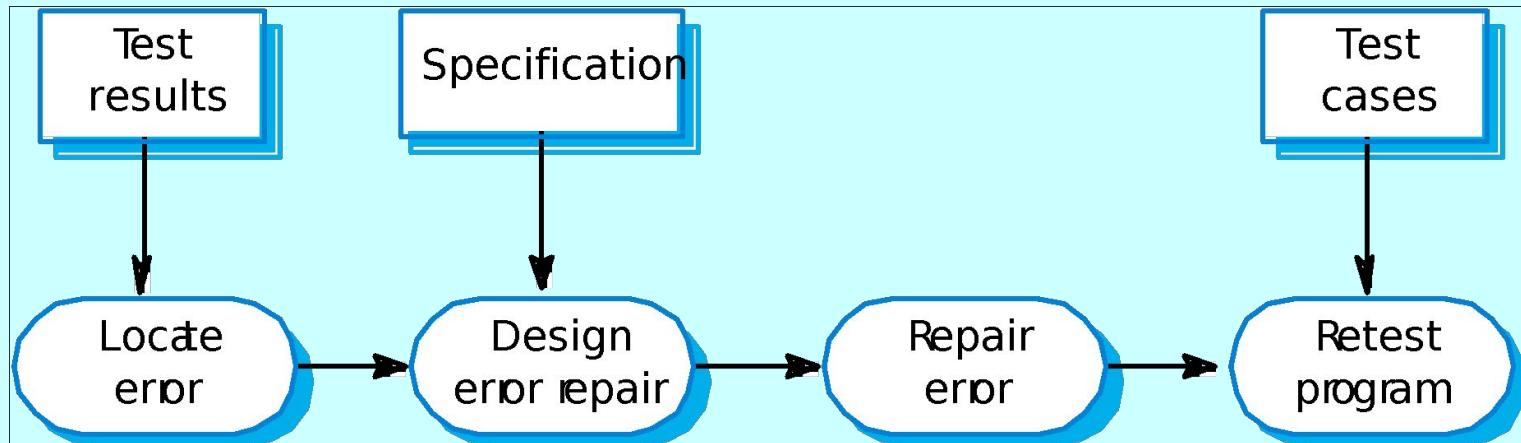
- **Defect testing**
  - Tests designed to discover system defects.
  - A successful defect test is one which reveals the presence of defects in a system.
- **Validation testing**
  - Intended to show that the software meets its requirements.
  - A successful test is one that shows that a requirements has been properly implemented.

# Testing and debugging

---

- Defect testing and debugging are distinct processes.
- Verification and validation is concerned with establishing the existence of defects in a program.
- Debugging is concerned with locating and repairing these errors.
- Debugging involves formulating a hypothesis about program behaviour then testing these hypotheses to find the system error.

# The debugging process

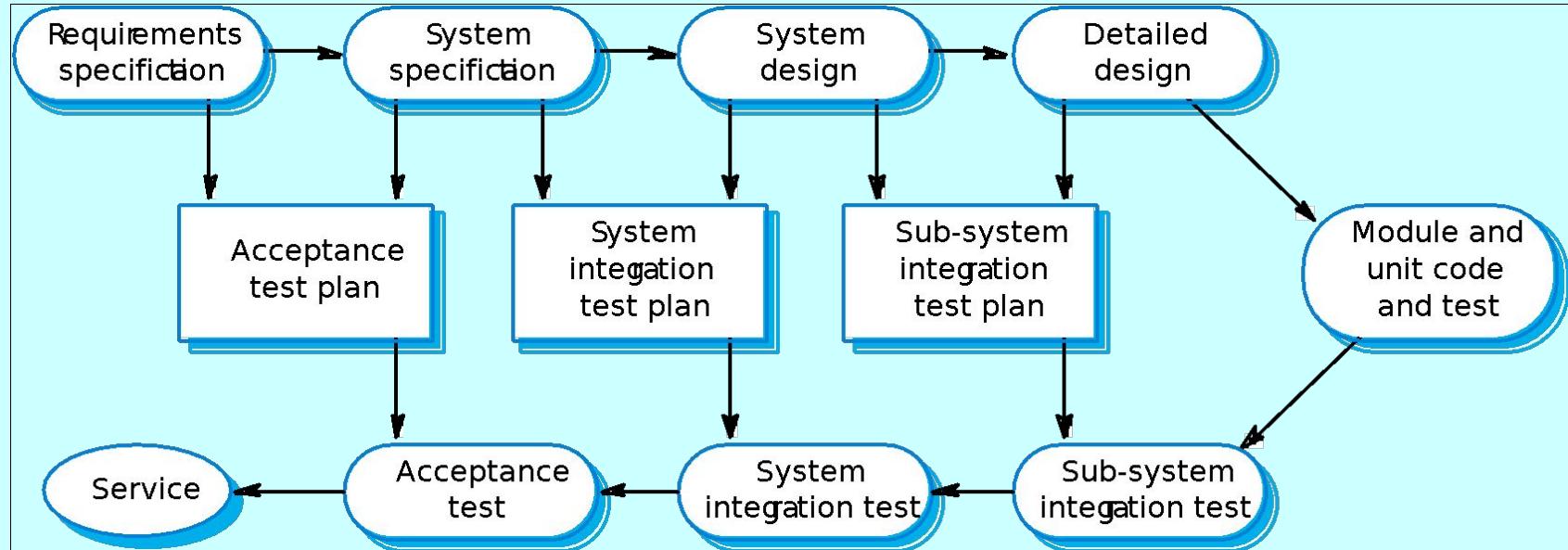


# V & V planning

---

- Careful planning is required to get the most out of testing and inspection processes.
- Planning should start early in the development process.
- The plan should identify the balance between static verification and testing.
- Test planning is about defining standards for the testing process rather than describing product tests.

# The V-model of development



# The structure of a software test plan

---

- The testing process.
- Requirements traceability.
- Tested items.
- Testing schedule.
- Test recording procedures.
- Hardware and software requirements.
- Constraints.

# The software test plan

## **The testing process**

A description of the major phases of the testing process. These might be as described earlier in this chapter.

## **Requirements traceability**

Users are most interested in the system meeting its requirements and testing should be planned so that all requirements are individually tested.

## **Tested items**

The products of the software process that are to be tested should be specified.

## **Testing schedule**

An overall testing schedule and resource allocation for this schedule. This, obviously, is linked to the more general project development schedule.

## **Test recording procedures**

It is not enough simply to run tests. The results of the tests must be systematically recorded. It must be possible to audit the testing process to check that it has been carried out correctly.

## **Hardware and software requirements**

This section should set out software tools required and estimated hardware utilisation.

## **Constraints**

Constraints affecting the testing process such as staff shortages should be anticipated in this section.

# Software inspections

---

- These involve people examining the source representation with the aim of discovering anomalies and defects.
- Inspections not require execution of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.

# Inspection success

---

- Many different defects may be discovered in a single inspection. In testing, one defect may mask another so several executions are required.
- The reuse domain and programming knowledge so reviewers are likely to have seen the types of error that commonly arise.

# Inspections and testing

---

- Inspections and testing are complementary and not opposing verification techniques.
- Both should be used during the V & V process.
- Inspections can check conformance with a specification but not conformance with the customer's real requirements.
- Inspections cannot check non-functional characteristics such as performance, usability, etc.

# Program inspections

---

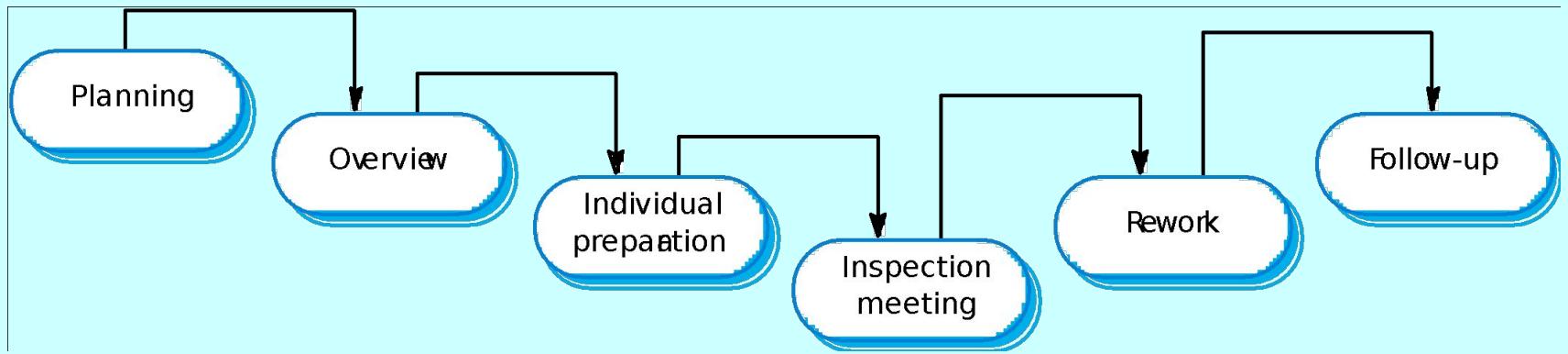
- Formalised approach to document reviews
- Intended explicitly for defect **detection** (not correction).
- Defects may be logical errors, anomalies in the code that might indicate an erroneous condition (e.g. an uninitialised variable) or non-compliance with standards.

# Inspection pre-conditions

---

- A precise specification must be available.
- Team members must be familiar with the organisation standards.
- Syntactically correct code or other system representations must be available.
- An error checklist should be prepared.
- Management must accept that inspection will increase costs early in the software process.
- Management should not use inspections for staff appraisal ie finding out who makes mistakes.

# The inspection process



# Inspection procedure

---

- System overview presented to inspection team.
- Code and associated documents are distributed to inspection team in advance.
- Inspection takes place and discovered errors are noted.
- Modifications are made to repair discovered errors.
- Re-inspection may or may not be required.

# Inspection roles

|                       |                                                                                                                                                            |
|-----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Author or owner       | The programmer or designer responsible for producing the program or document. Responsible for fixing defects discovered during the inspection process.     |
| Inspector             | Finds errors, omissions and inconsistencies in programs and documents. May also identify broader issues that are outside the scope of the inspection team. |
| Reader                | Presents the code or document at an inspection meeting.                                                                                                    |
| Scribe                | Records the results of the inspection meeting.                                                                                                             |
| Chairman or moderator | Manages the process and facilitates the inspection. Reports process results to the Chief moderator.                                                        |
| Chief moderator       | Responsible for inspection process improvements, checklist updating, standards development etc.                                                            |

# Inspection checklists

---

- Checklist of common errors should be used to drive the inspection.
- Error checklists are programming language dependent and reflect the characteristic errors that are likely to arise in the language.
- In general, the 'weaker' the type checking, the larger the checklist.
- Examples: Initialisation, Constant naming, loop termination, array bounds, etc.

# Inspection checks 1

|                     |                                                                                                                                                                                                                                                                                                                                      |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Data faults         | <p>Are all program variables initialised before their values are used?</p> <p>Have all constants been named?</p> <p>Should the upper bound of arrays be equal to the size of the array or Size -1?</p> <p>If character strings are used, is a delimiter explicitly assigned?</p> <p>Is there any possibility of buffer overflow?</p> |
| Control faults      | <p>For each conditional statement, is the condition correct?</p> <p>Is each loop certain to terminate?</p> <p>Are compound statements correctly bracketed?</p> <p>In case statements, are all possible cases accounted for?</p> <p>If a break is required after each case in case statements, has it been included?</p>              |
| Input/output faults | <p>Are all input variables used?</p> <p>Are all output variables assigned a value before they are output?</p> <p>Can unexpected inputs cause corruption?</p>                                                                                                                                                                         |

# Inspection checks 2

|                             |                                                                                                                                                                                                                                                                                        |
|-----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Interface faults            | <p>Do all function and method calls have the correct number of parameters?</p> <p>Do formal and actual parameter types match?</p> <p>Are the parameters in the right order?</p> <p>If components access shared memory, do they have the same model of the shared memory structure?</p> |
| Storage management faults   | <p>If a linked structure is modified, have all links been correctly reassigned?</p> <p>If dynamic storage is used, has space been allocated correctly?</p> <p>Is space explicitly de-allocated after it is no longer required?</p>                                                     |
| Exception management faults | Have all possible error conditions been taken into account?                                                                                                                                                                                                                            |

# Inspection rate

---

- 500 statements/hour during overview.
- 125 source statement/hour during individual preparation.
- 90-125 statements/hour can be inspected.
- Inspection is therefore an expensive process.
- Inspecting 500 lines costs about 40 man/hours effort - about £2800 at UK rates.

# Automated static analysis

---

- Static analysers are software tools for source text processing.
- They parse the program text and try to discover potentially erroneous conditions and bring these to the attention of the V & V team.
- They are very effective as an aid to inspections - they are a supplement to but not a replacement for inspections.

# Static analysis checks

| Fault class               | Static analysis check                                                                                                                                                                               |
|---------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Data faults               | Variables used before initialisation<br>Variables declared but never used<br>Variables assigned twice but never used between assignments<br>Possible array bound violations<br>Undeclared variables |
| Control faults            | Unreachable code<br>Unconditional branches into loops                                                                                                                                               |
| Input/output faults       | Variables output twice with no intervening assignment                                                                                                                                               |
| Interface faults          | Parameter type mismatches<br>Parameter number mismatches<br>Non-usage of the results of functions<br>Uncalled functions and procedures                                                              |
| Storage management faults | Unassigned pointers<br>Pointer arithmetic                                                                                                                                                           |

# Stages of static analysis

---

- **Control flow analysis.** Checks for loops with multiple exit or entry points, finds unreachable code, etc.
- **Data use analysis.** Detects uninitialised variables, variables written twice without an intervening assignment, variables which are declared but never used, etc.
- **Interface analysis.** Checks the consistency of routine and procedure declarations and their use

# Stages of static analysis

---

- **Information flow analysis.** Identifies the dependencies of output variables. Does not detect anomalies itself but highlights information for code inspection or review
- **Path analysis.** Identifies paths through the program and sets out the statements executed in that path. Again, potentially useful in the review process
- Both these stages generate vast amounts of information. They must be used with care.

# Static analysis tools

---

- Splint
- cpplint
- Eclipse
- Infer
- Sparse
- Checkstyle
- FindBugs
- SpotBugs
- PMD

# Use of static analysis

---

- Particularly valuable when a language such as C is used which has weak typing and hence many errors are undetected by the compiler.
- Less cost-effective for languages like Java that have strong type checking and can therefore detect many errors during compilation.

# Verification and formal methods

---

- Formal methods can be used when a mathematical specification of the system is produced.
- They are the ultimate static verification technique.
- They involve detailed mathematical analysis of the specification and may develop formal arguments that a program conforms to its mathematical specification.

# Arguments for formal methods

---

- Producing a mathematical specification requires a detailed analysis of the requirements and this is likely to uncover errors.
- They can detect implementation errors before testing when the program is analysed alongside the specification.

# Arguments against formal methods

---

- Require specialised notations that cannot be understood by domain experts.
- Very expensive to develop a specification and even more expensive to show that a program meets that specification.
- It may be possible to reach the same level of confidence in a program more cheaply using other V & V techniques.

# Cleanroom software development

---

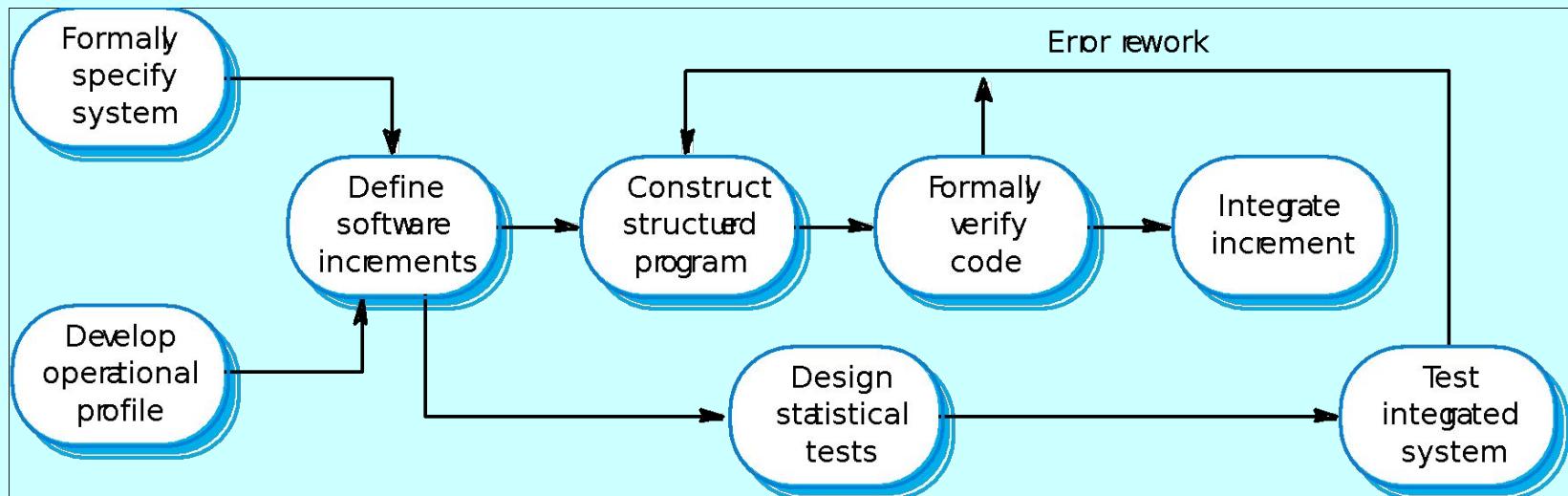
- The name is derived from the 'Cleanroom' process in semiconductor fabrication. The philosophy is defect avoidance rather than defect removal.
- This software development process is based on:
  - Incremental development;
  - Formal specification;
  - Static verification using correctness arguments;
  - Statistical testing to determine program reliability.

# Cleanroom process characteristics

---

- Formal specification using a state transition model.
- Incremental development where the customer prioritises increments.
- Structured programming - limited control and abstraction constructs are used in the program.
- Static verification using rigorous inspections.
- Statistical testing of the system.

# The Cleanroom process



# Formal specification and inspections

---

- The state based model is a system specification and the inspection process checks the program against this model.
- The programming approach is defined so that the correspondence between the model and the system is clear.
- Mathematical arguments (not proofs) are used to increase confidence in the inspection process.

# Cleanroom process teams

---

- **Specification team.** Responsible for developing and maintaining the system specification.
- **Development team.** Responsible for developing and verifying the software. The software is NOT executed or even compiled during this process.
- **Certification team.** Responsible for developing a set of statistical tests to exercise the software after development. Reliability growth models used to determine when reliability is acceptable.

# Cleanroom process evaluation

---

- The results of using the Cleanroom process have been very impressive with few discovered faults in delivered systems.
- Independent assessment shows that the process is no more expensive than other approaches.
- There were fewer errors than in a 'traditional' development process.
- However, the process is not widely used. It is not clear how this approach can be transferred to an environment with less skilled or less motivated software engineers.

# Key points

---

- Verification and validation are not the same thing. Verification shows conformance with specification; validation shows that the program meets the customer's needs.
- Test plans should be drawn up to guide the testing process.
- Static verification techniques involve examination and analysis of the program for error detection.

# Key points

---

- Program inspections are very effective in discovering errors.
- Program code in inspections is systematically checked by a small team to locate software faults.
- Static analysis tools can discover program anomalies which may be an indication of faults in the code.
- The Cleanroom development process depends on incremental development, static verification and statistical testing.

# Specification

# Outline

- Discussion of the term "specification"
- Types of specifications
  - operational
    - Data Flow Diagrams
    - (Some) UML diagrams
    - Finite State Machines
    - Petri Nets
  - descriptive
    - Entity Relationship Diagrams
    - Logic-based notations
    - Algebraic notations

# Specification

- A broad term that means *definition*
- Used at different stages of software development for different purposes
- Generally, a statement of agreement (*contract*) between
  - producer and consumer of a service
  - implementer and user
- All desirable qualities must be specified

# Uses of specification

- Statement of user requirements
  - major failures occur because of misunderstandings between the producer and the user
  - "The hardest single part of building a software system is deciding precisely what to build" (F. Brooks)

# Uses of specification (cont.)

- Statement of the interface between the machine and the controlled environment
  - serious undesirable effects can result due to misunderstandings between software engineers and domain experts about the phenomena affecting the control function to be implemented by software

# Uses of specification (cont.)

- Statement of requirements for implementation
  - design process is a chain of specification (i.e., definition)–implementation–verification steps

# Specification qualities

- Precise, clear, unambiguous
- Consistent
- Complete
  - internal completeness
  - external completeness
- Incremental

# Clear, unambiguous, understandable

- Example: specification fragment for a word-processor

Selecting is the process of designating areas of the document that you want to work on. Most editing and formatting actions require two steps: first you select what you want to work on, such as text or graphics; then you initiate the appropriate action.

*can an area be scattered?*

# Precise, unambiguous, clear

- Another example (from a real safety-critical system)

The message must be triplicated. The three copies must be forwarded through three different physical channels. The receiver accepts the message on the basis of a two-out-of-three voting policy.

*can a message be accepted as soon as we receive 2 out of 3 identical copies of message or do we need to wait for receipt of the 3<sup>rd</sup>?*

# Consistent

- Example: specification fragment for a word-processor

The whole text should be kept in lines of equal length. The length is specified by the user. Unless the user gives an explicit hyphenation command, a carriage return should occur only at the end of a word.

*What if the length of a word exceeds the length of the line?*

# Complete

- Internal completeness
- External completeness

# *Incrementality principle*

- Due to difficulty in achieving complete, precise and unambiguous specifications.
- Referring to the specification process
  - start from a sketchy document and **progressively add details**
- Referring to the specification document
  - document is structured and can be understood in increments

# Classification of specification styles

- Informal, semi-formal, formal
  - Informal: written in natural language , uses figures , tables etc.
  - Formal: created using precise syntax and meaning (formalism)
- Operational: describes intended system by describing its *desired behavior*.
- Descriptive: states the desired properties of the system in purely declarative fashion.

# Specification

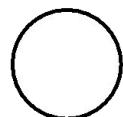
- Types of specifications
  - operational
    - Data Flow Diagrams
    - (Some) UML diagrams
    - Finite State Machines
    - Petri Nets
  - descriptive
    - Entity Relationship Diagrams
    - Logic-based notations
    - Algebraic notations

# Data Flow Diagrams (DFDs)

- A semi-formal operational specification
- System viewed as collection of processing steps or functions
- Represents how data flows through a sequence of processing steps
- DFDs have a graphical notation
- For example: Filtering of duplicate records in a customer database

# Graphical notation

- *bubbles* represent functions
- *arcs* represent data flows
- *open boxes* represent persistent store
- *closed boxes* represent I/O



The function symbol



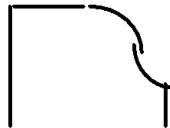
The data flow symbol



The data store symbol

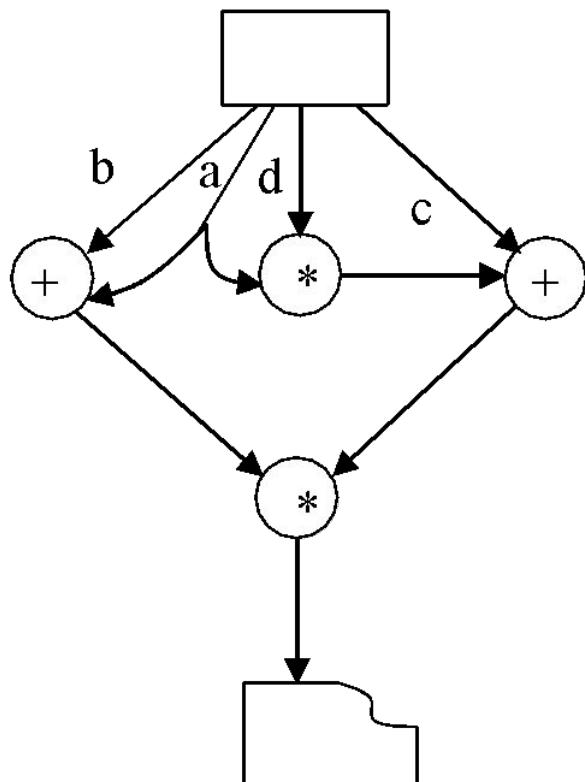


The input device symbol



The output device symbol

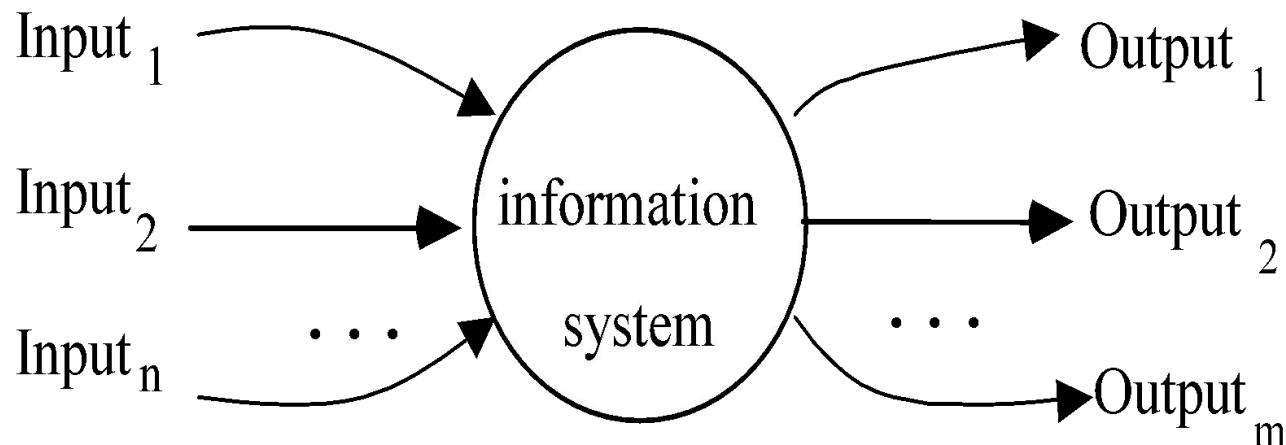
# Example



specifies evaluation of  
 $(a + b) * (c + a * d)$

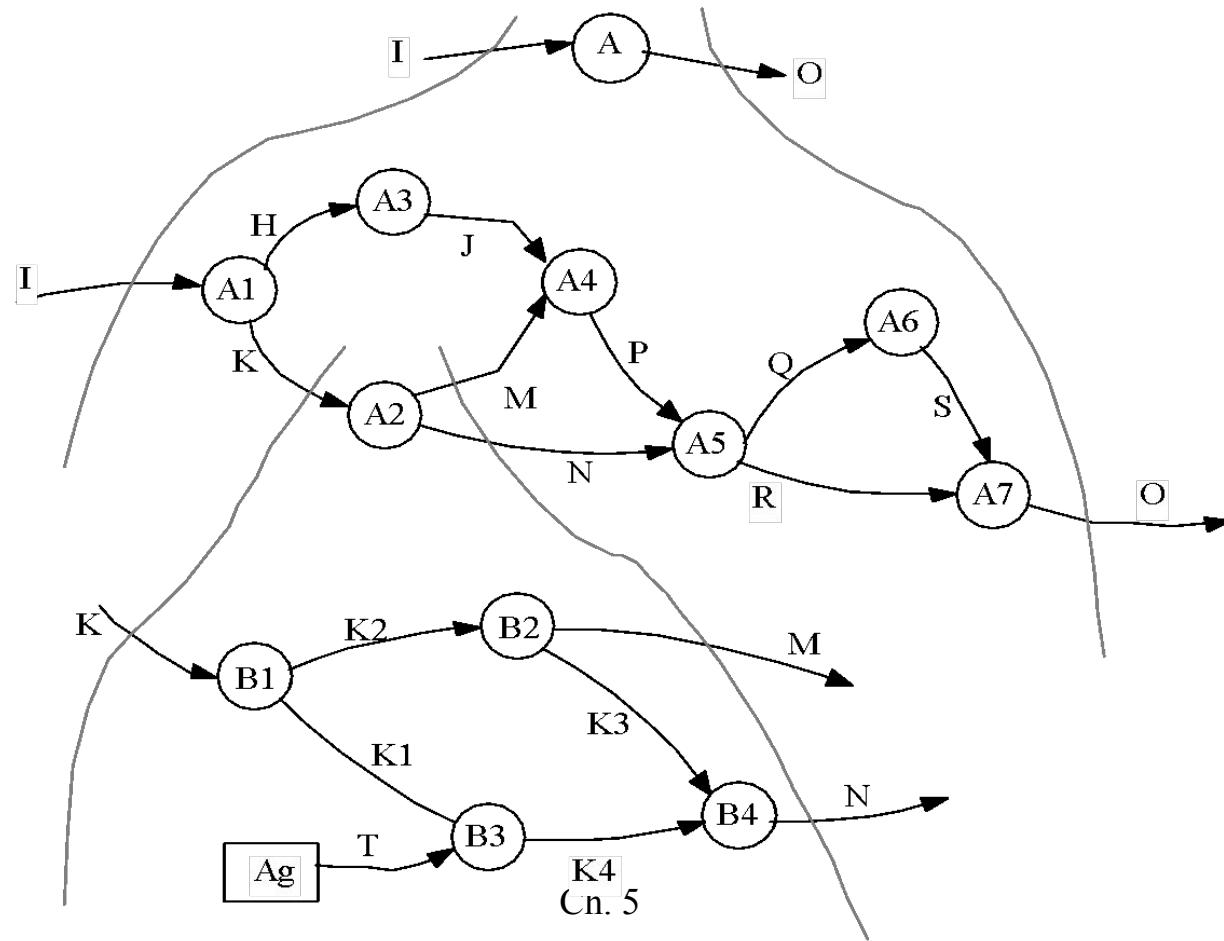
# A construction “method” (1)

1. Start from the “context” diagram

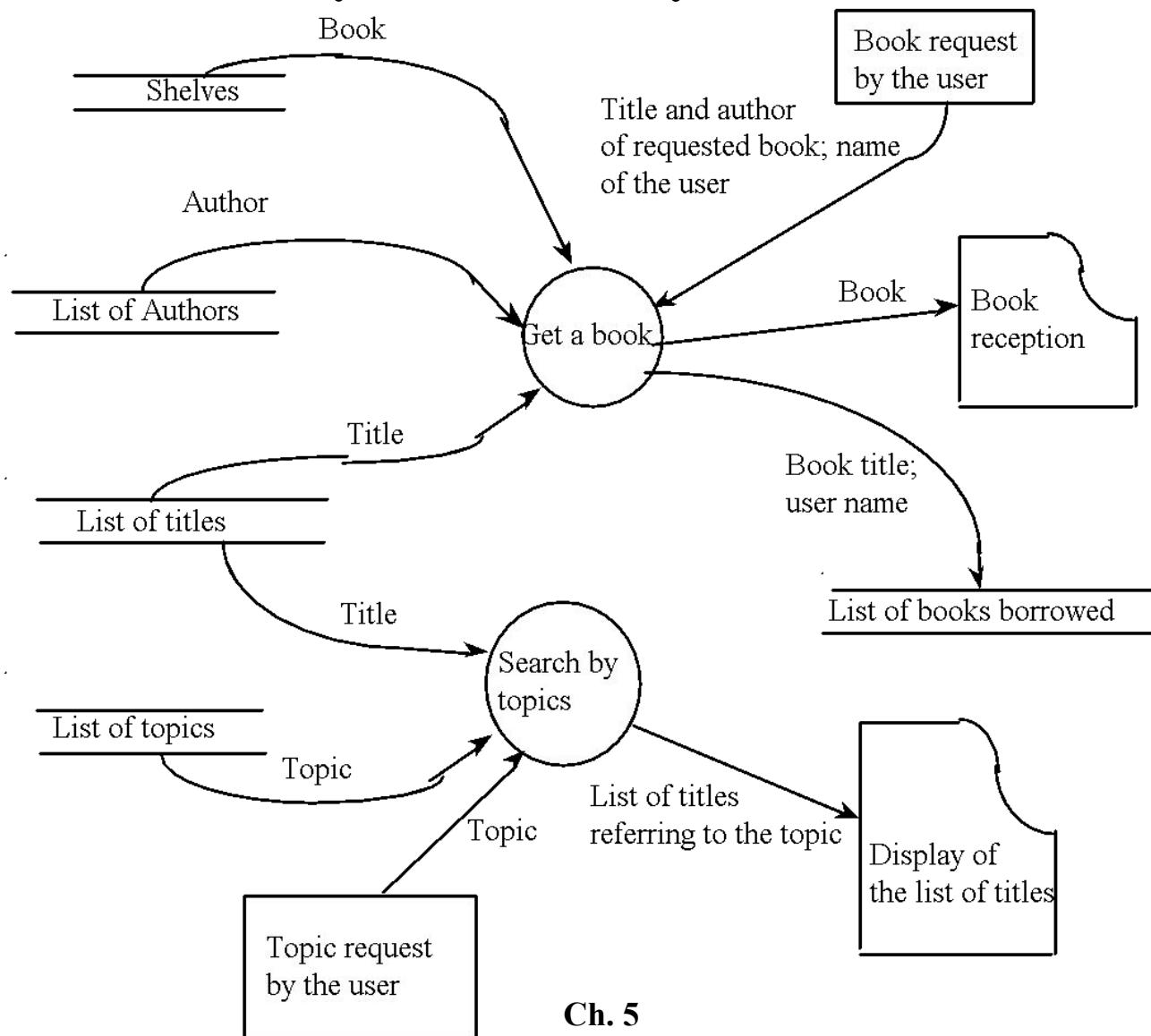


# A construction "method" (2)

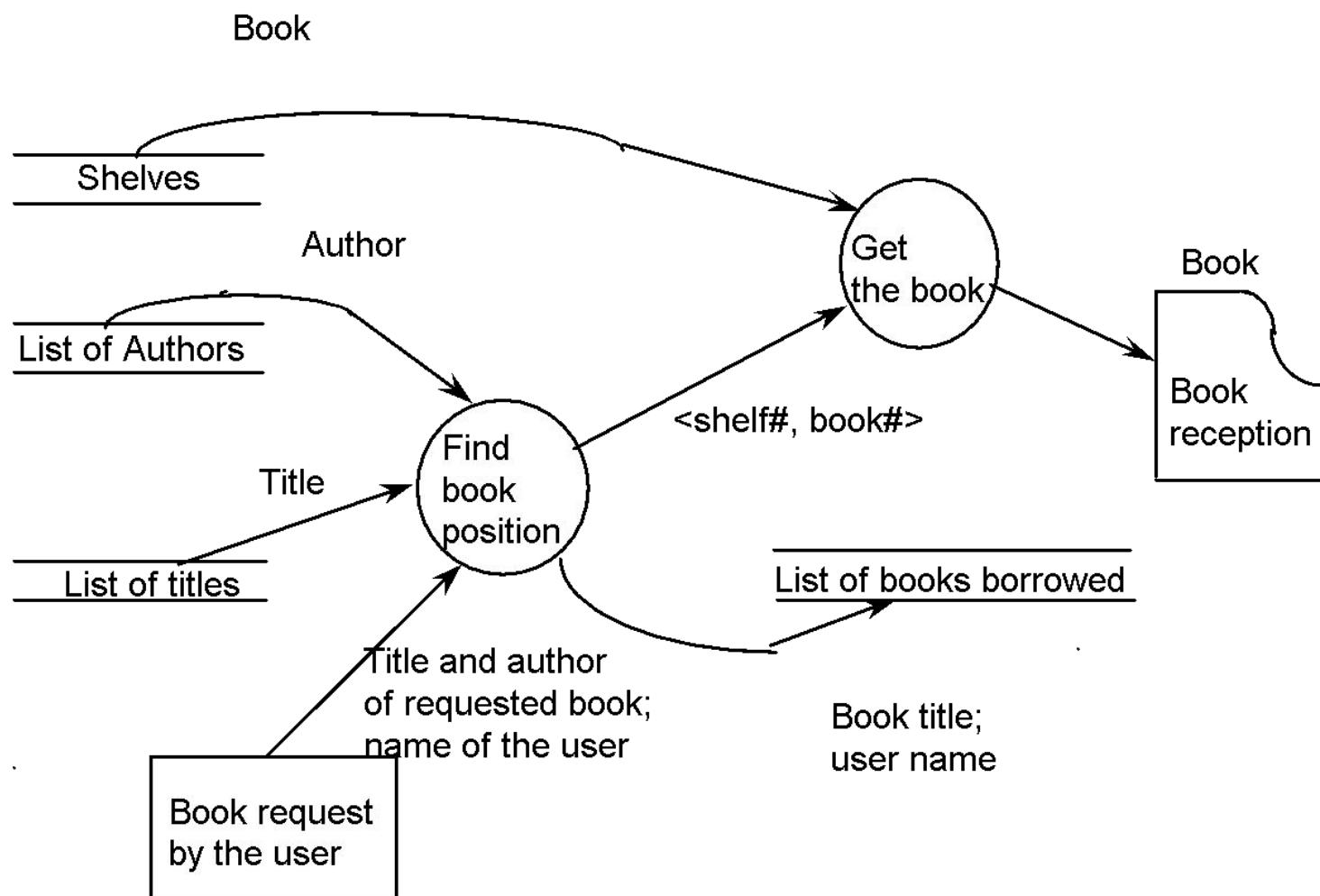
2. Proceed by refinements until you reach "elementary" functions



# A library example (Level 0)

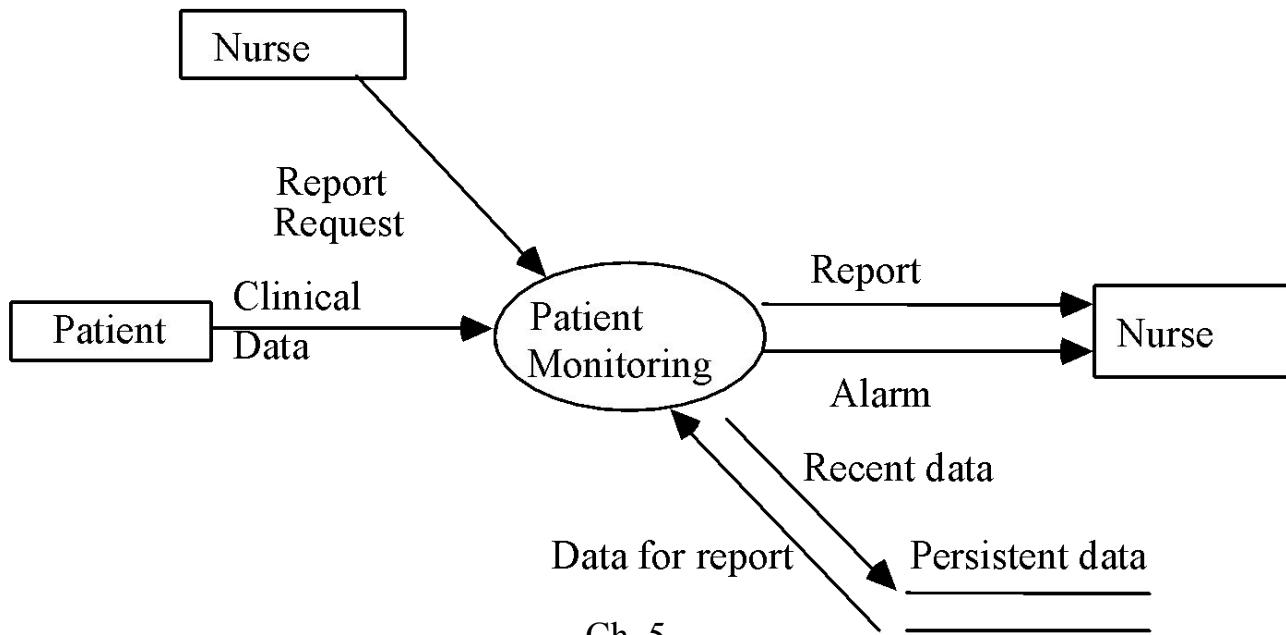


# Refinement of "Get a book" (Level 1)

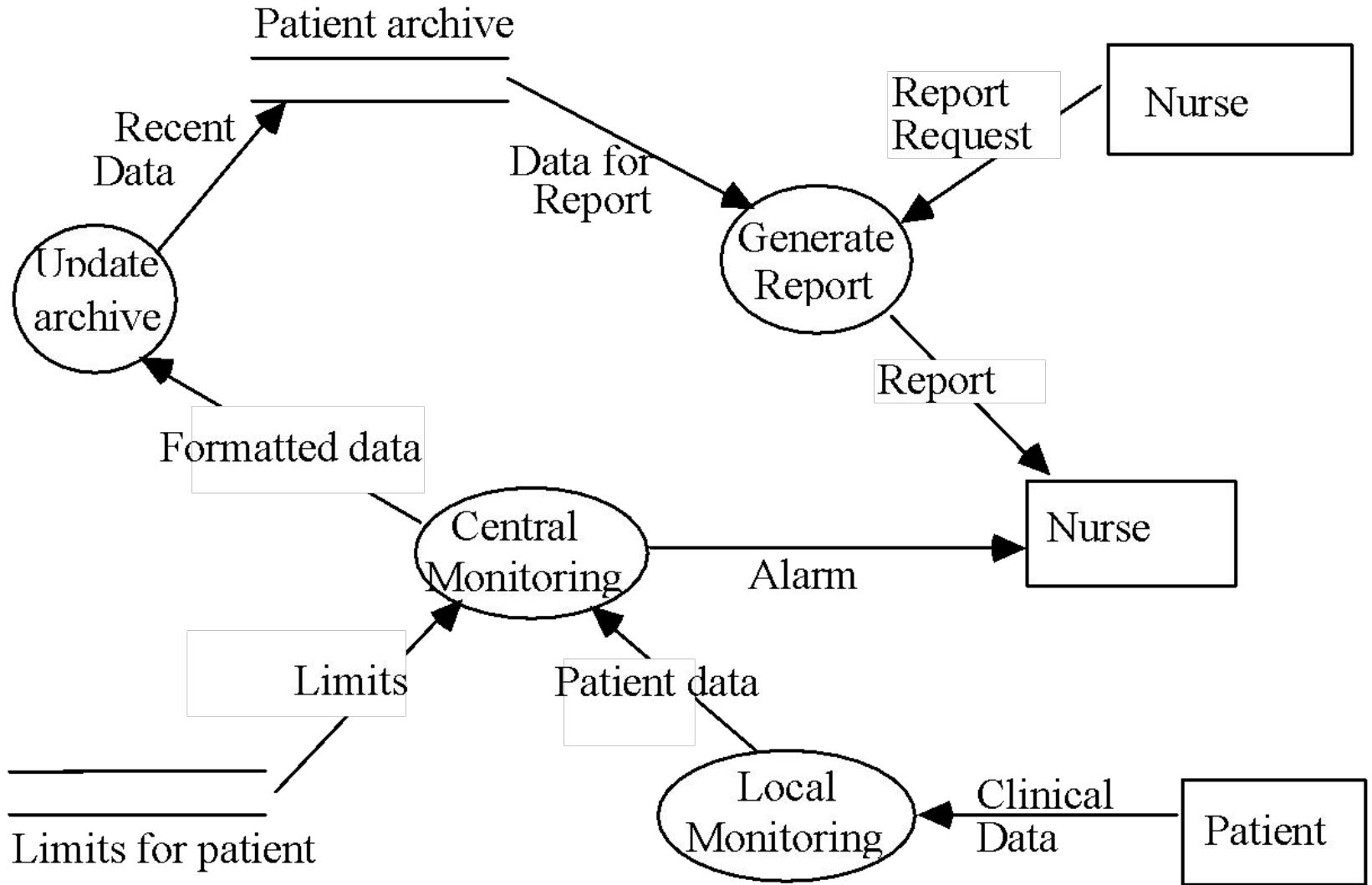


# Patient monitoring systems

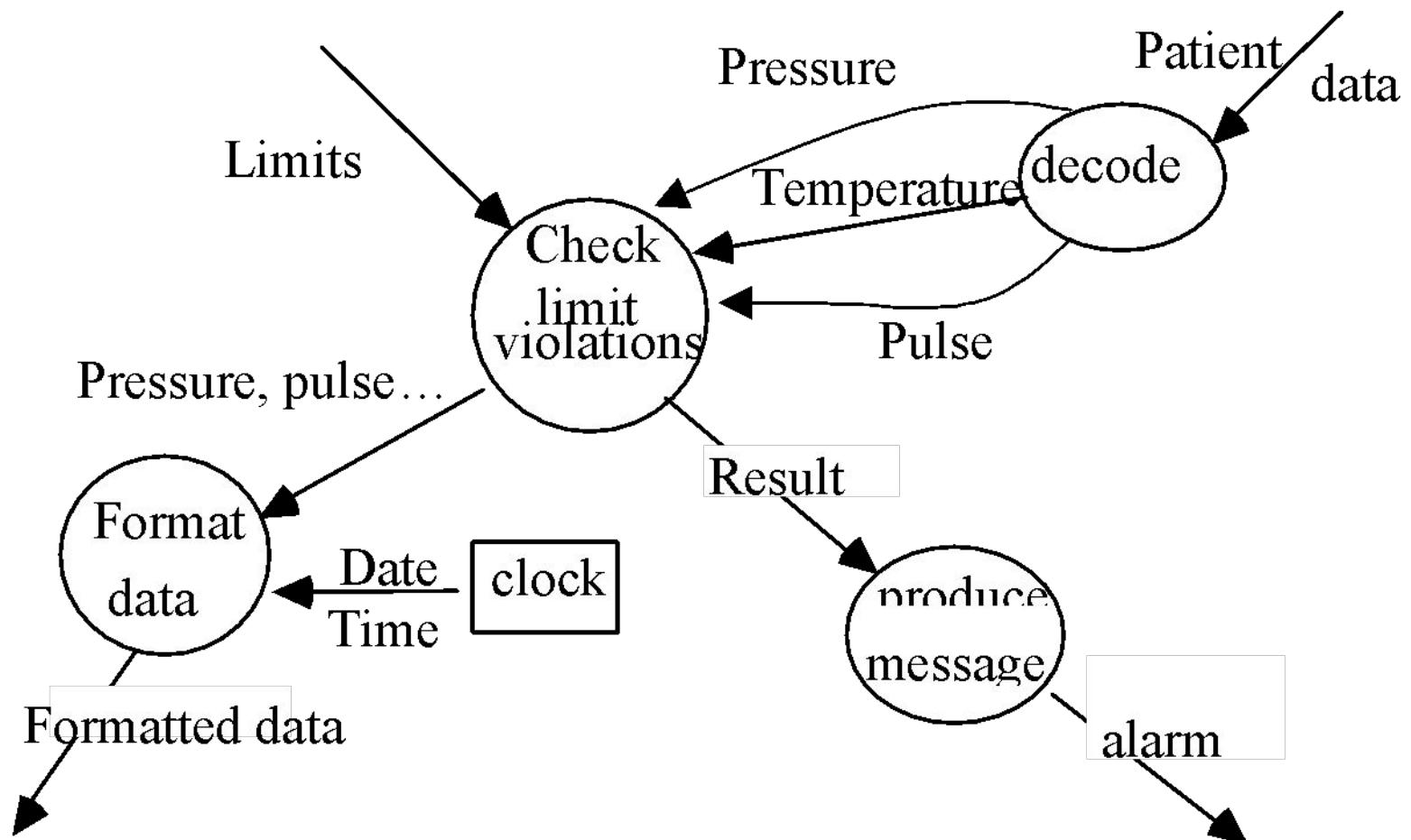
*The purpose is to monitor the patients' vital factors--blood, pressure, temperature, ...--reading them at specified frequencies from analog devices and storing readings in a DB. If readings fall outside the range specified for patient or device fails an alarm must be sent to a nurse. The system also provides reports.*



# A refinement



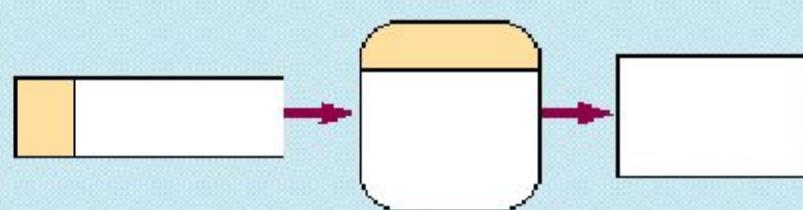
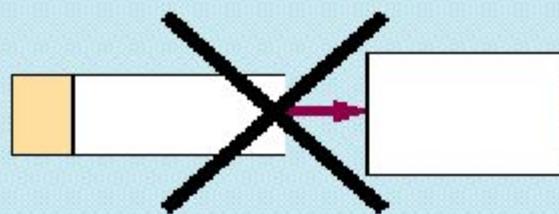
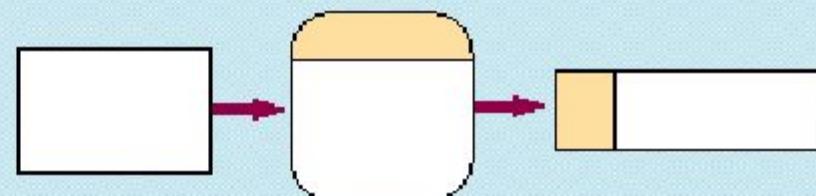
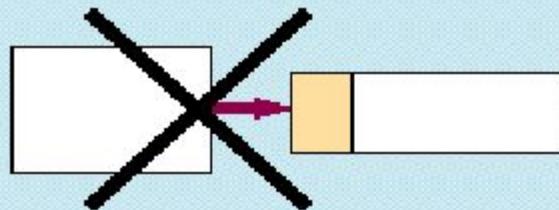
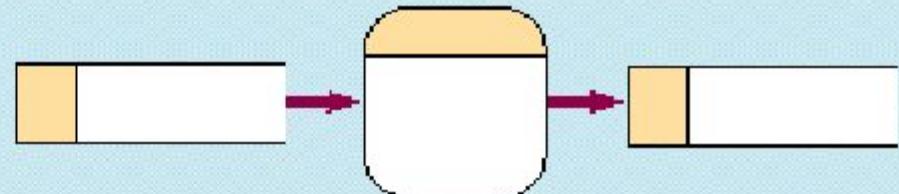
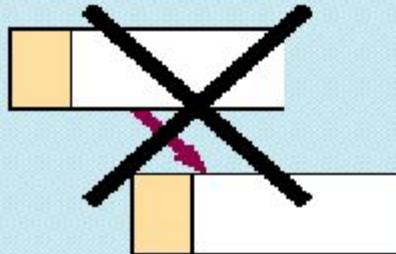
# More refinement



# Basic rules

- Balancing principle: Decomposed DFD (next lower level) should retain the same number of inputs and outputs from its previous higher level DFD
- No process can have only input(s)/output(s)
  - How to define leaf functions?
  - Inherent ambiguities

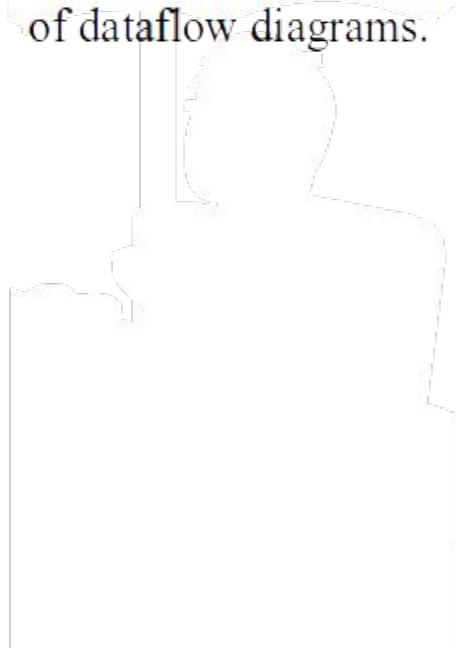
# Basic rules



# Creating DFDs (Lemonade Stand Example)

## Example

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



## Steps:

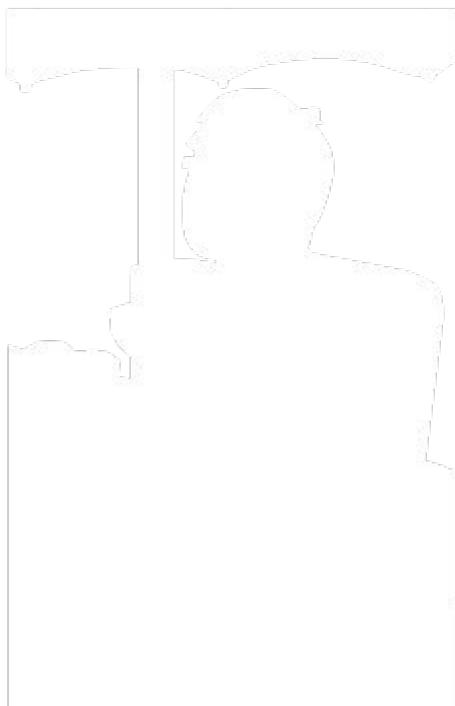
1. Create a list of activities
2. Construct Context Level DFD  
(identifies sources and sink)
3. Construct Level 0 DFD  
(identifies manageable sub processes )
4. Construct Level 1- n DFD  
(identifies actual data flows and data stores )

<number>

# Creating DFDs

## Example

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



### 1. Create a list of activities

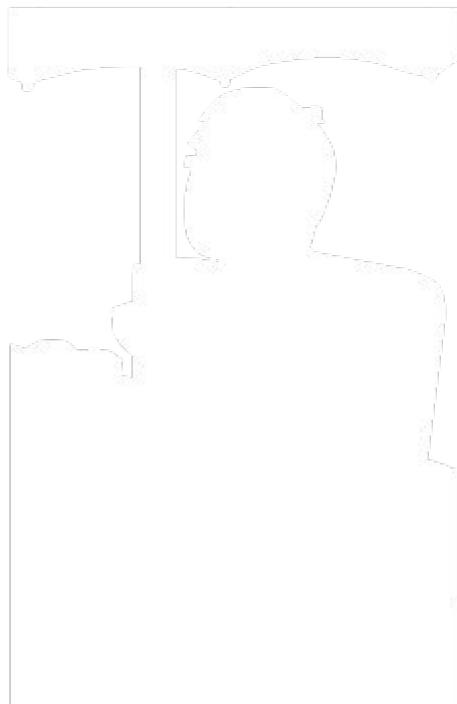
- Customer Order
- Serve Product
- Collect Payment
- Produce Product
- Store Product

<number>

# Creating DFDs

## Example

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



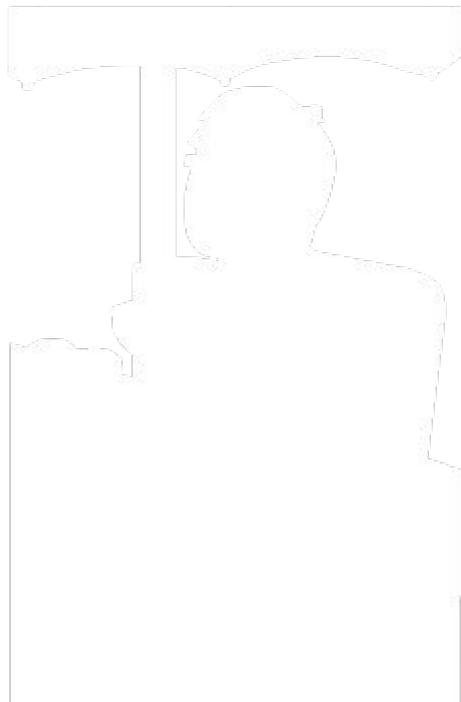
### 1. Create a list of activities

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

# Creating DFDs

## Example

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



### 1. Create a list of activities

Customer Order  
Serve Product  
Collect Payment

Produce Product  
Store Product

Order Raw Materials  
Pay for Raw Materials

Pay for Labor

<number>

# Creating DFDs

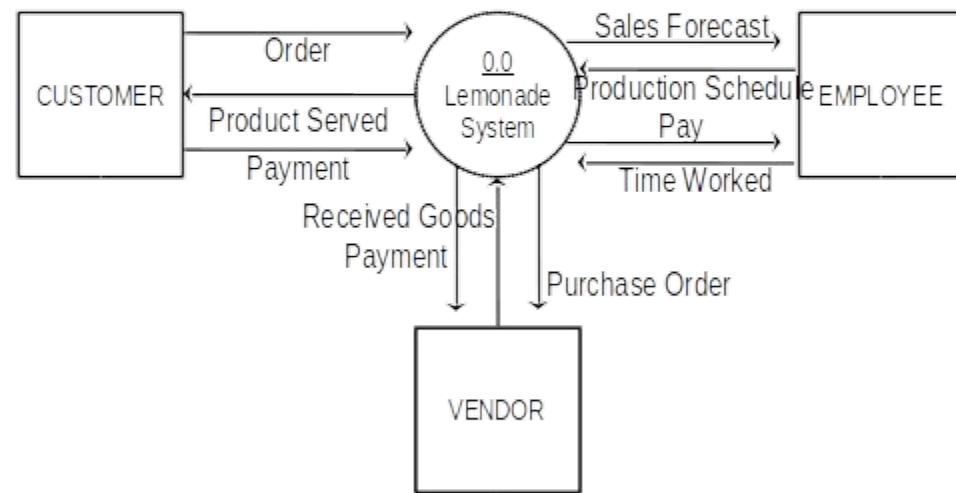
## Example

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



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

## Context Level DFD



Identify manageable subprocesses and refine the DFD

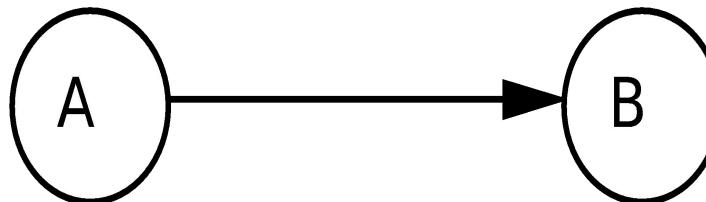
<number>

# Drawbacks

- Time consuming
- It does not provide a complete picture of the system and sometimes leaves vital physical entities.
- A DFD can be confusing and programmers might not differentiate between its levels.

# Drawbacks

- Control information is absent

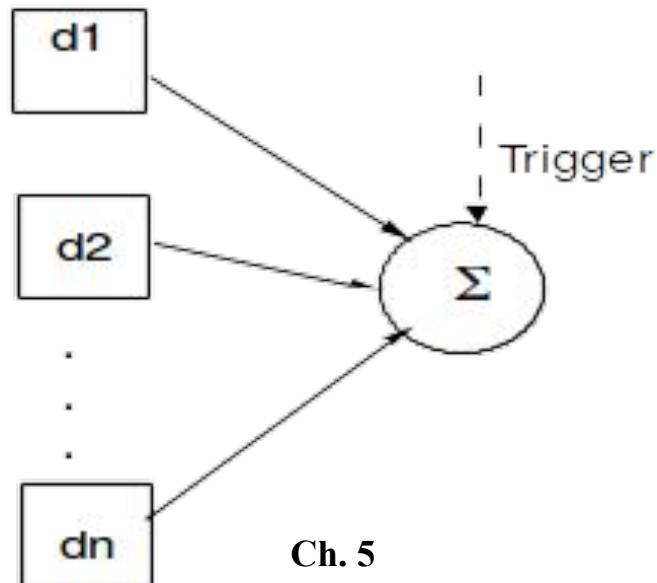


Possible interpretations:

- (a) A produces datum, waits until B consumes it
- (b) B can read the datum many times without consuming it
- (c) a pipe is inserted between A and B

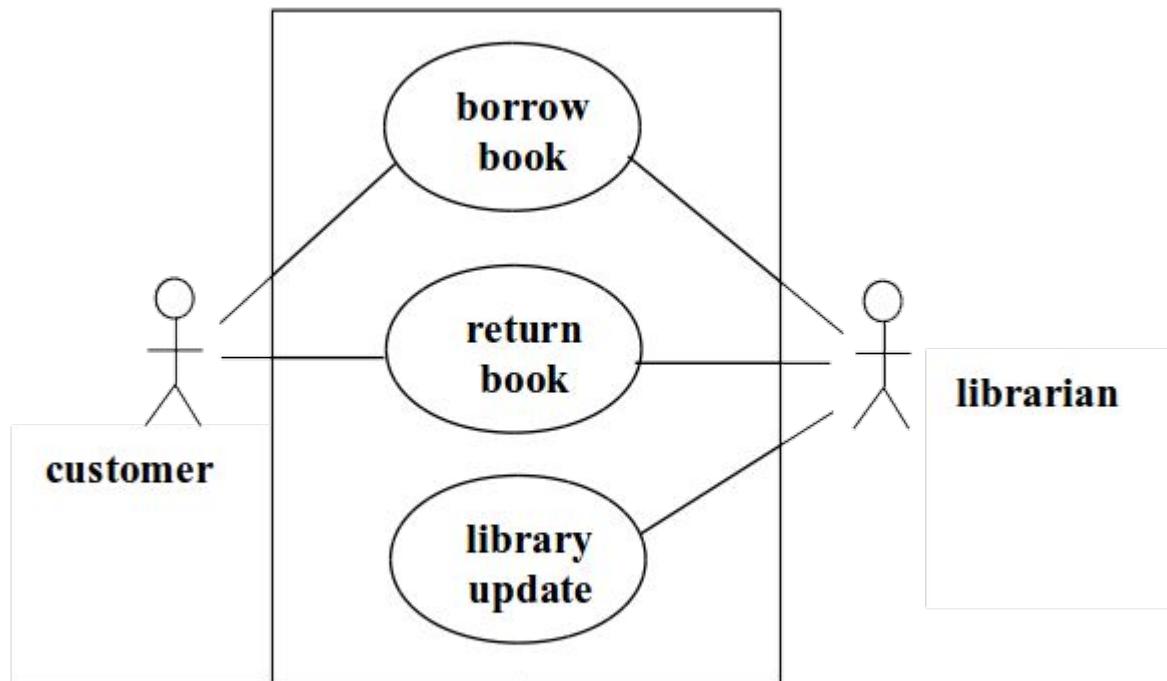
# Solutions

- Formalizations: There have been attempts to *formalize* DFDs
- There have been attempts to *extend* DFDs (e.g., for real-time systems)



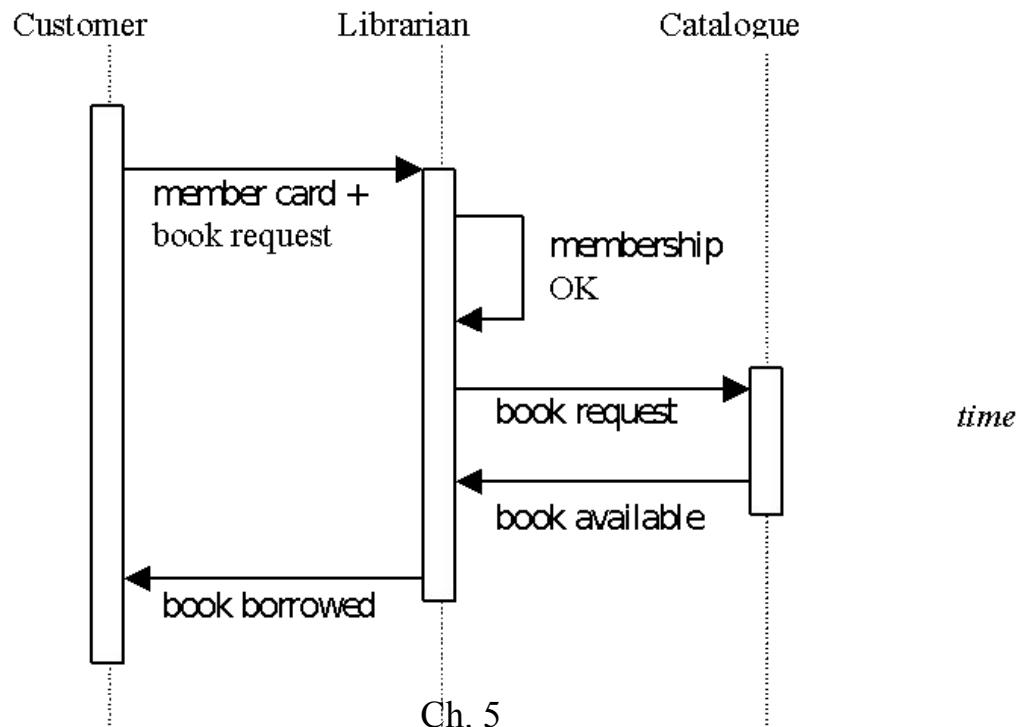
# UML use-case diagrams

- Define functions on basis of actors and actions



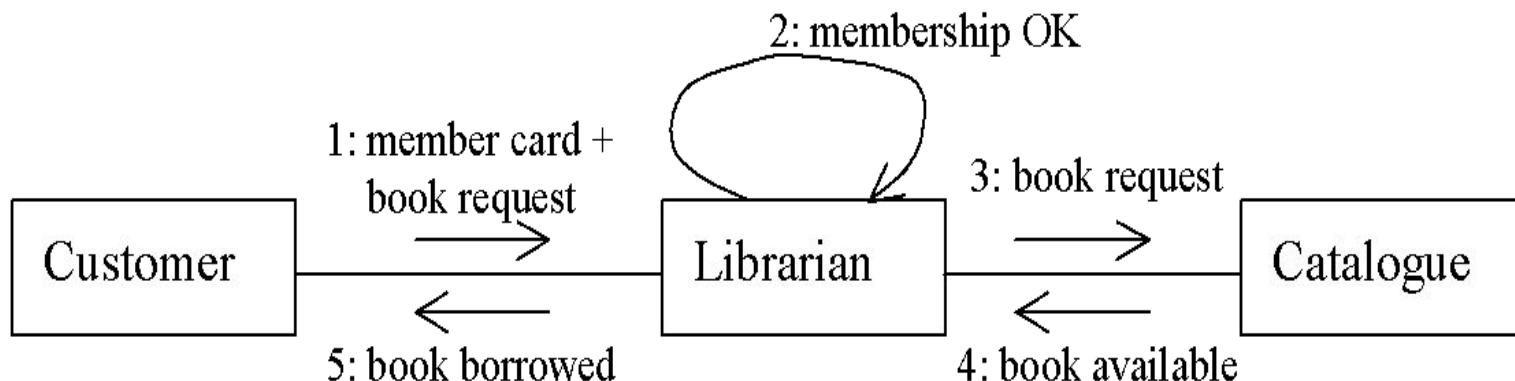
# UML sequence diagrams

- Describe how objects interact by exchanging messages
- Provide a dynamic view



# UML collaboration diagrams

- Give object interactions and their order
- Equivalent to sequence diagrams



# Finite state machines (FSMs)

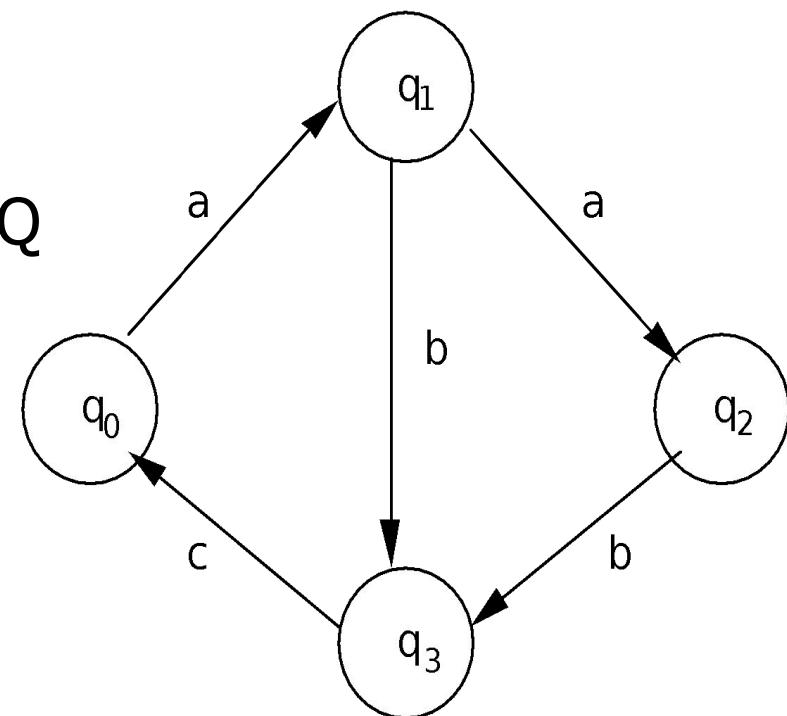
- Can specify control flow aspects
- Defined as

a finite set of states,  $Q$ ;

a finite set of inputs,  $I$ ;

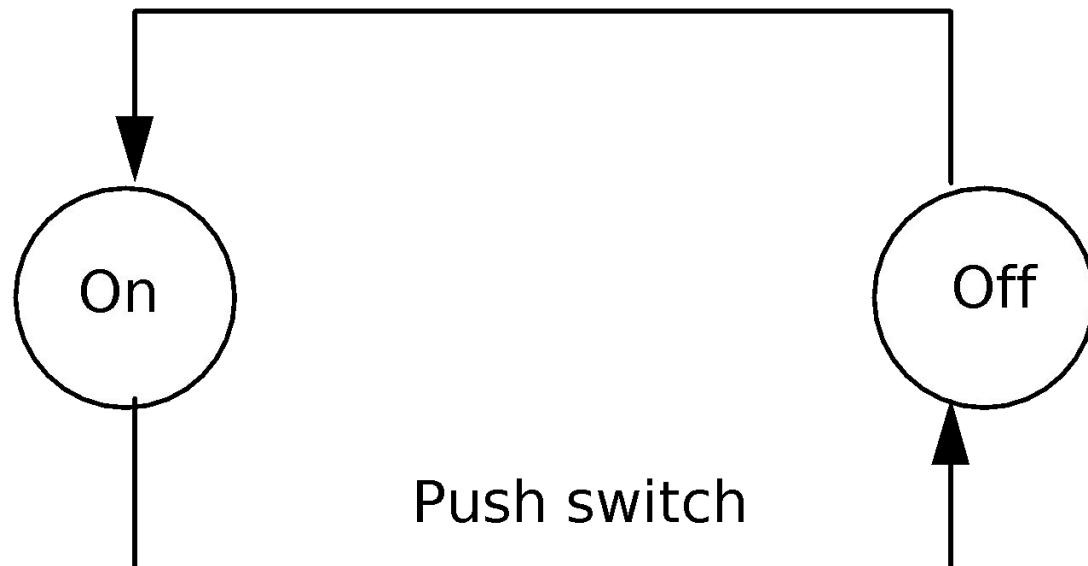
a transition function  $d : Q \times I \rightarrow Q$

( $d$  can be a partial function)

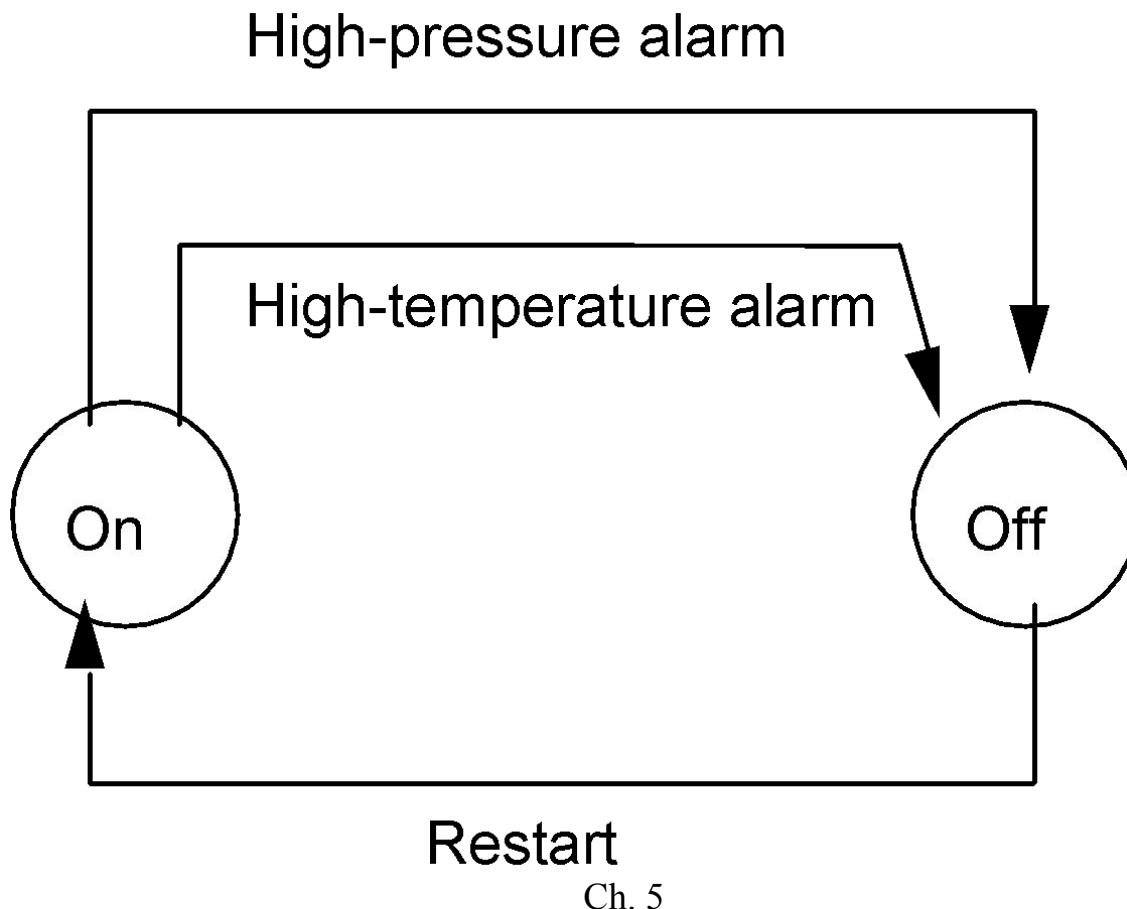


# Example: a lamp

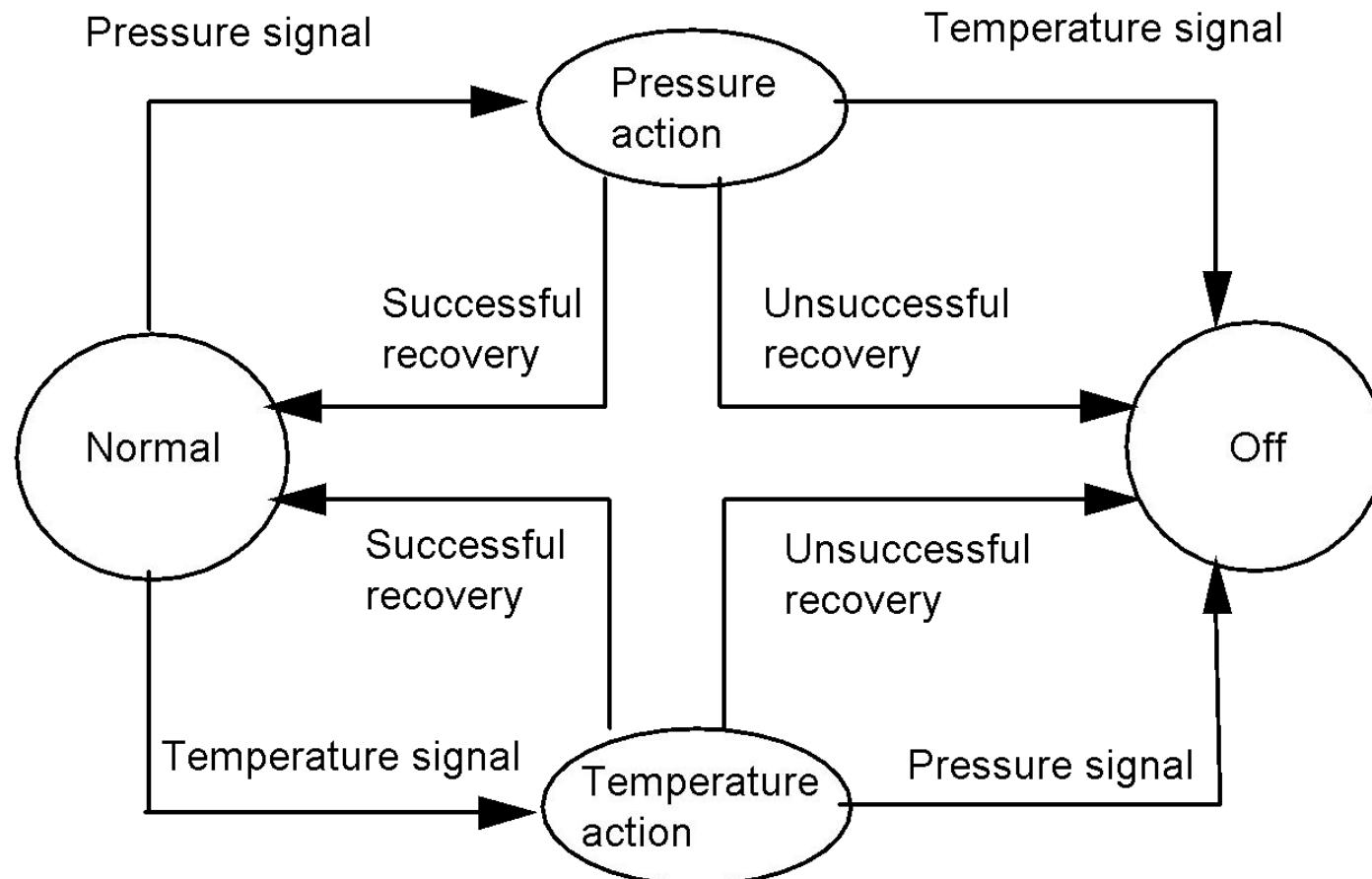
Push switch



# Another example: a plant control system



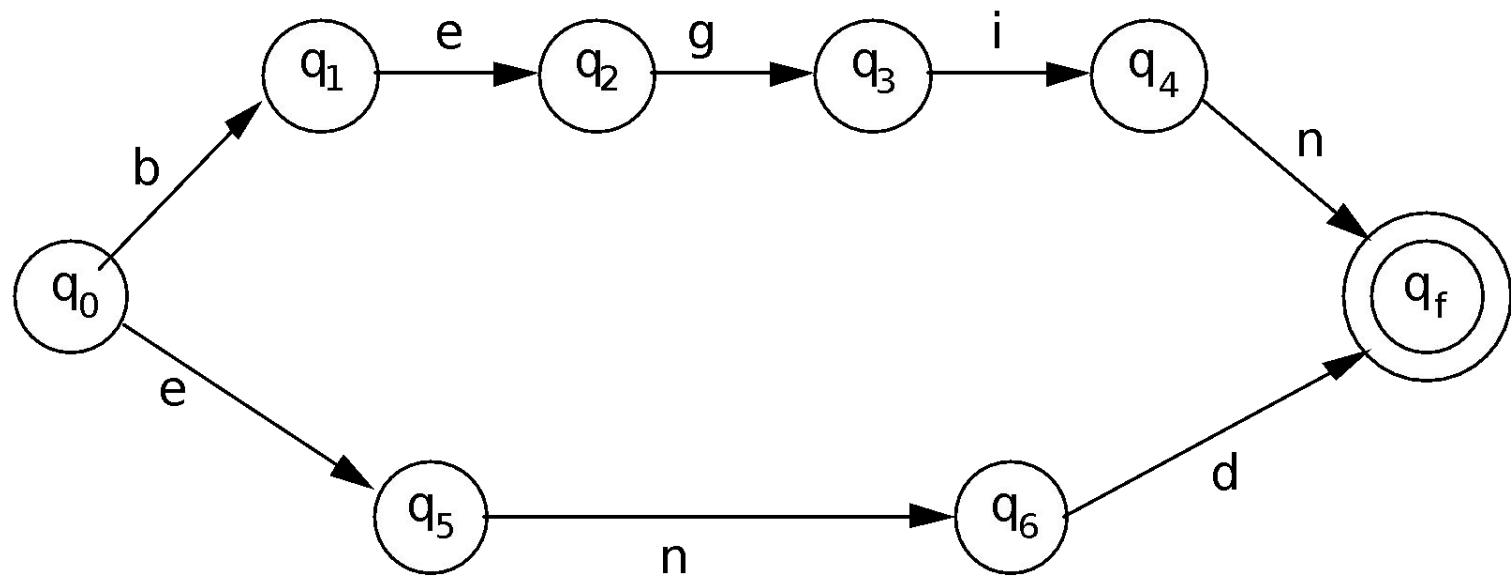
# A refinement



# Classes of FSMs

- Deterministic/nondeterministic
- FSMs as recognizers
  - introduce final states
- FSMs as transducers
  - introduce set of outputs

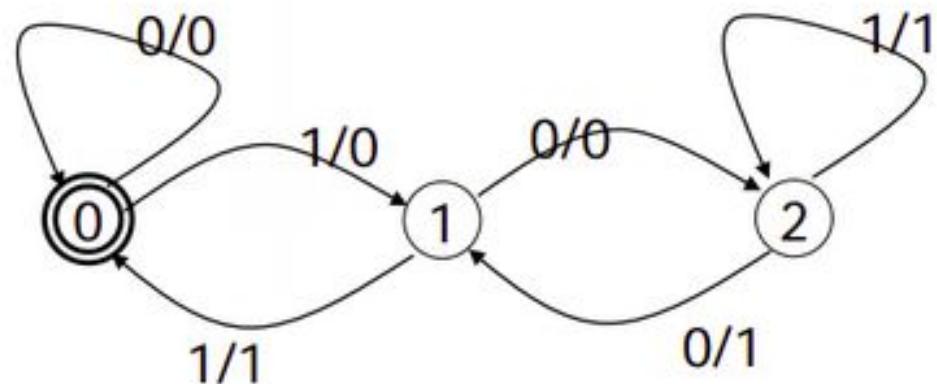
# FSMs as recognizers



$q_f$  is a final state

# FSMs as transducers

| input | output |
|-------|--------|
| 0     | 0      |
| 11    | 01     |
| 110   | 010    |
| 1001  | 0011   |
| 1100  | 0100   |
| 1111  | 0101   |
| 10010 | 00110  |



# Petri net

- Also known as place/transition (PT) net, is one of the several mathematical modeling languages for the description of distributed systems.
- It is a directed bipartite graph, in which the nodes represent transitions (i.e. events that may occur, represented by bars) and places (i.e. conditions, represented by circles). The directed arcs describe which places are pre- and/or postconditions for which transitions (signified by arrows).

# Petri net

A quadruple  $(P, T, F, W)$

P: places    T: transitions ( $P, T$  are finite)

F: flow relation ( $F \subseteq \{P \times T\} \cup \{T \times P\}$ )

W: weight function ( $W: F \rightarrow N - \{0\}$ )

Properties:

(1)  $P \cap T = \emptyset$

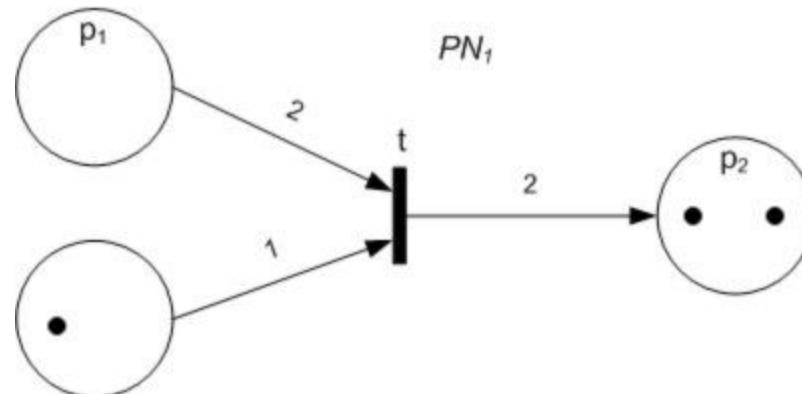
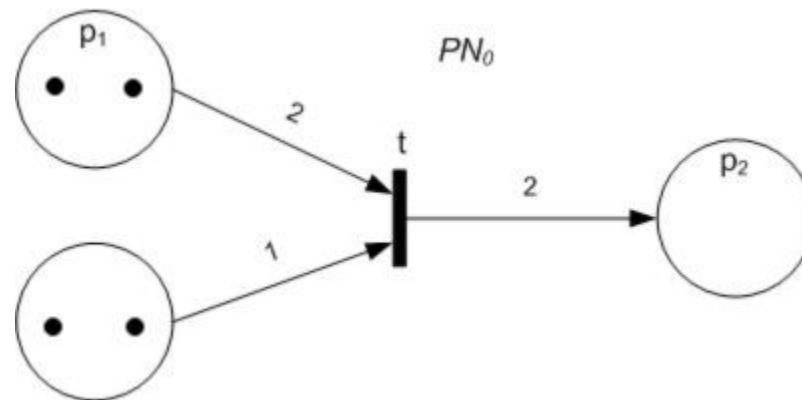
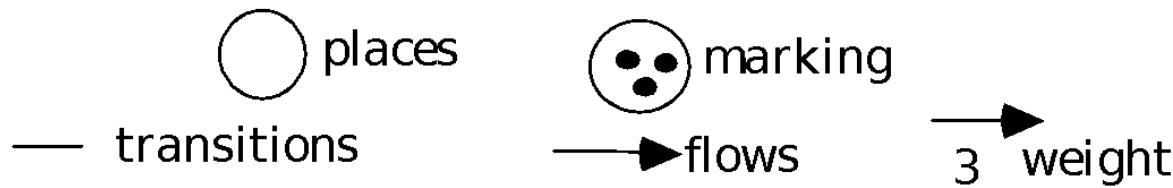
(2)  $P \cup T \neq \emptyset$

(3)  $F \subseteq (P \times T) \cup (T \times P)$

(4)  $W: F \rightarrow N$  is a multiset of arcs, i.e. it assigns to each arc a non-negative integer arc multiplicity (or weight) Default value of  $W$  is 1

State defined by marking:  $M: P \rightarrow N$

# Graphical representation



# Semantics

- Transition  $t$  is enabled iff
  - $\forall p \in t$ 's input places,  $M(p) \geq W(<p,t>)$
- $t$  fires: produces a new marking  $M'$  in places that are either  $t$ 's input or output places or both
  - if  $p$  is an input place:  $M'(p) = M(p) - W(<p,t>)$
  - if  $p$  is an output place:  $M'(p) = M(p) + W(<t,p>)$
  - if  $p$  is both an input and an output place:  
$$M'(p) = M(p) - W(<p,t>) + W(<t,p>)$$

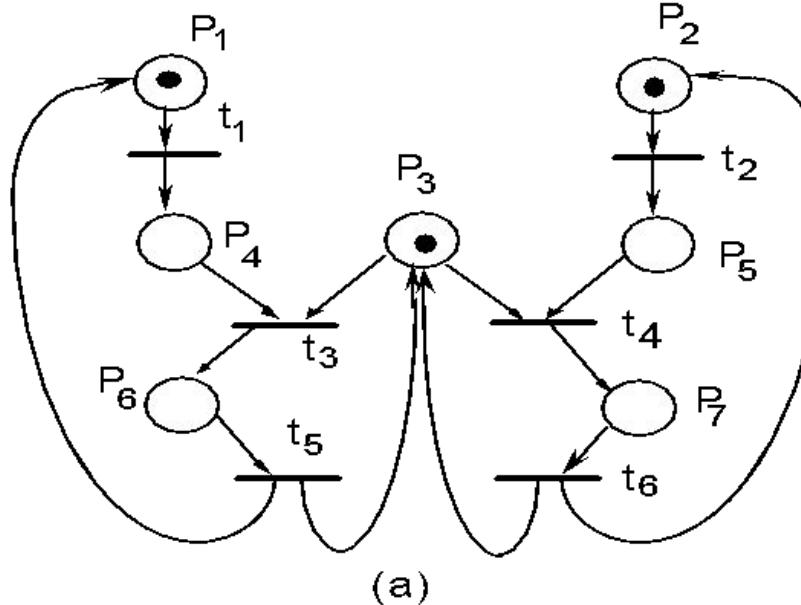
# Modeling with Petri nets

- Places represent distributed states
- Transitions represent actions or events that may occur when system is in a certain state
- They can occur as certain conditions hold on the states

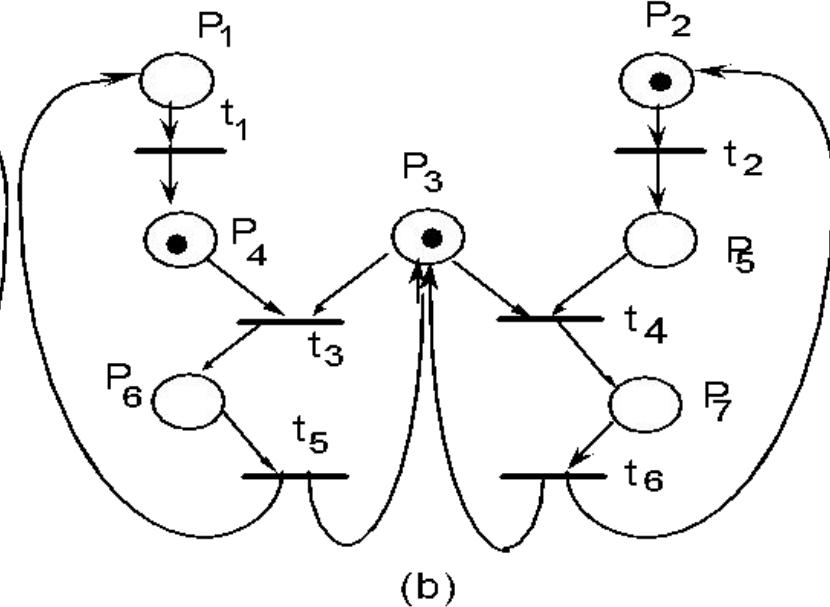
# Nondeterminism

- Any of the enabled transitions may fire
- Model does not specify which fires, nor when it fires

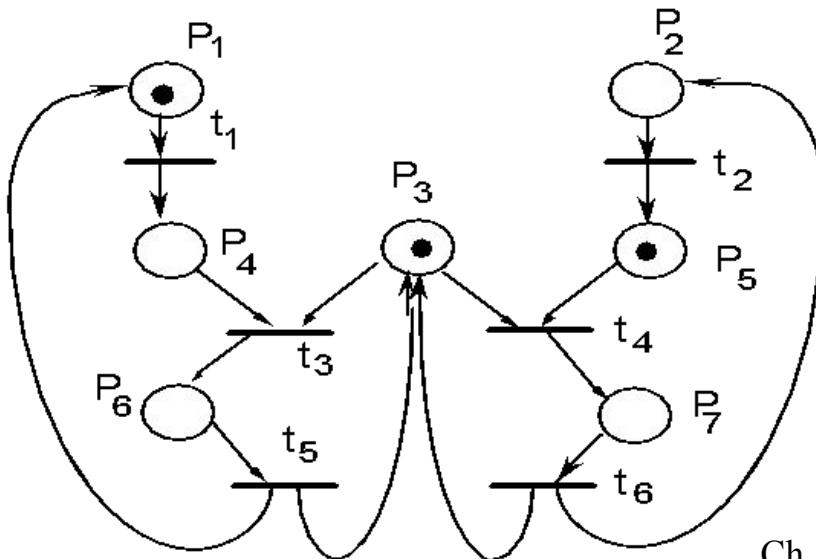
after (a) either (b) or (c) may occur, and then (d)



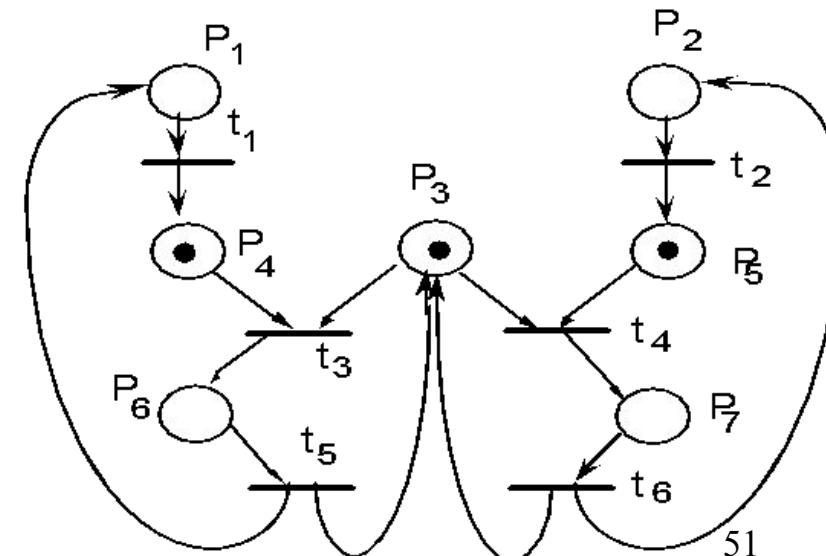
(a)



(b)



(c)



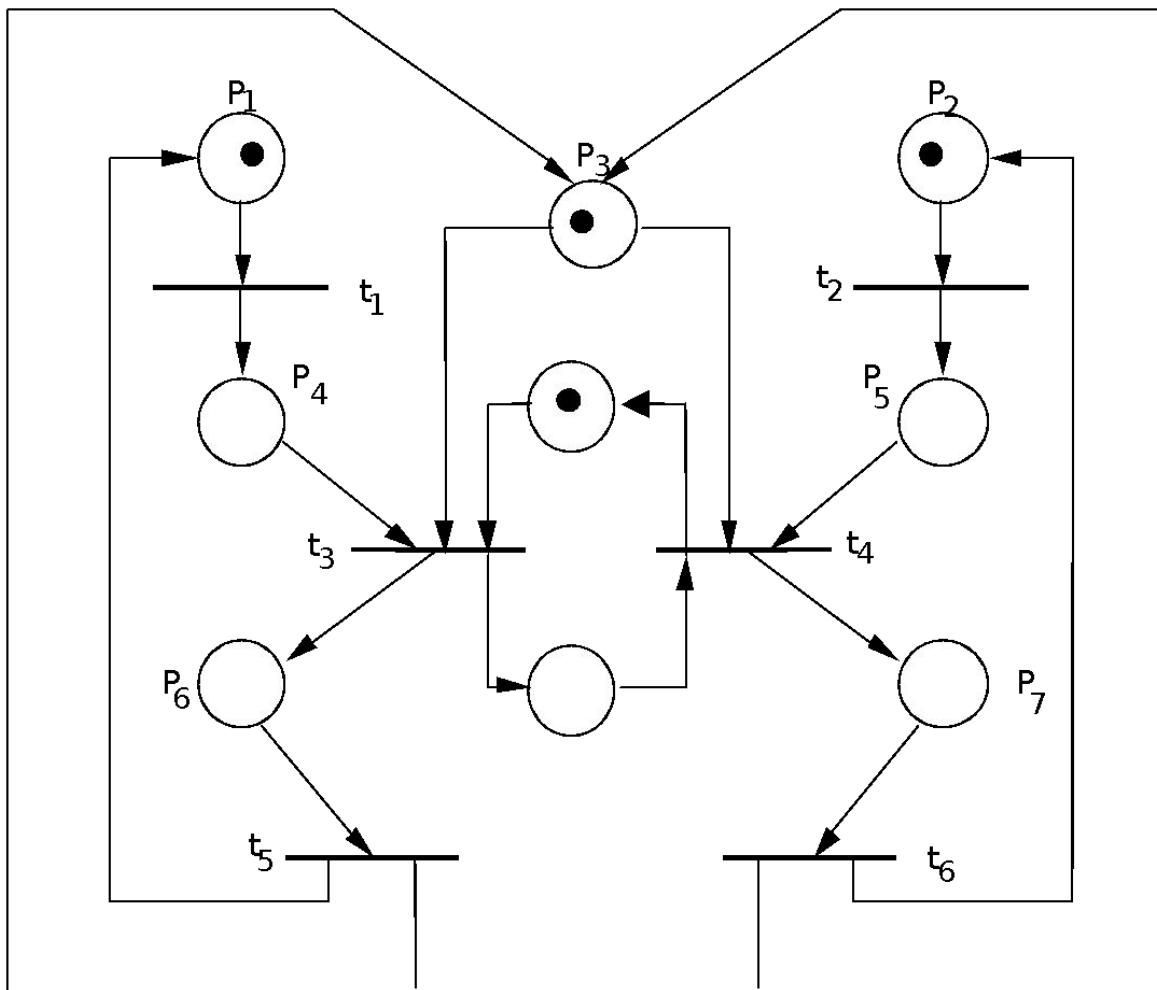
Ch. 5

(d)

# Common cases

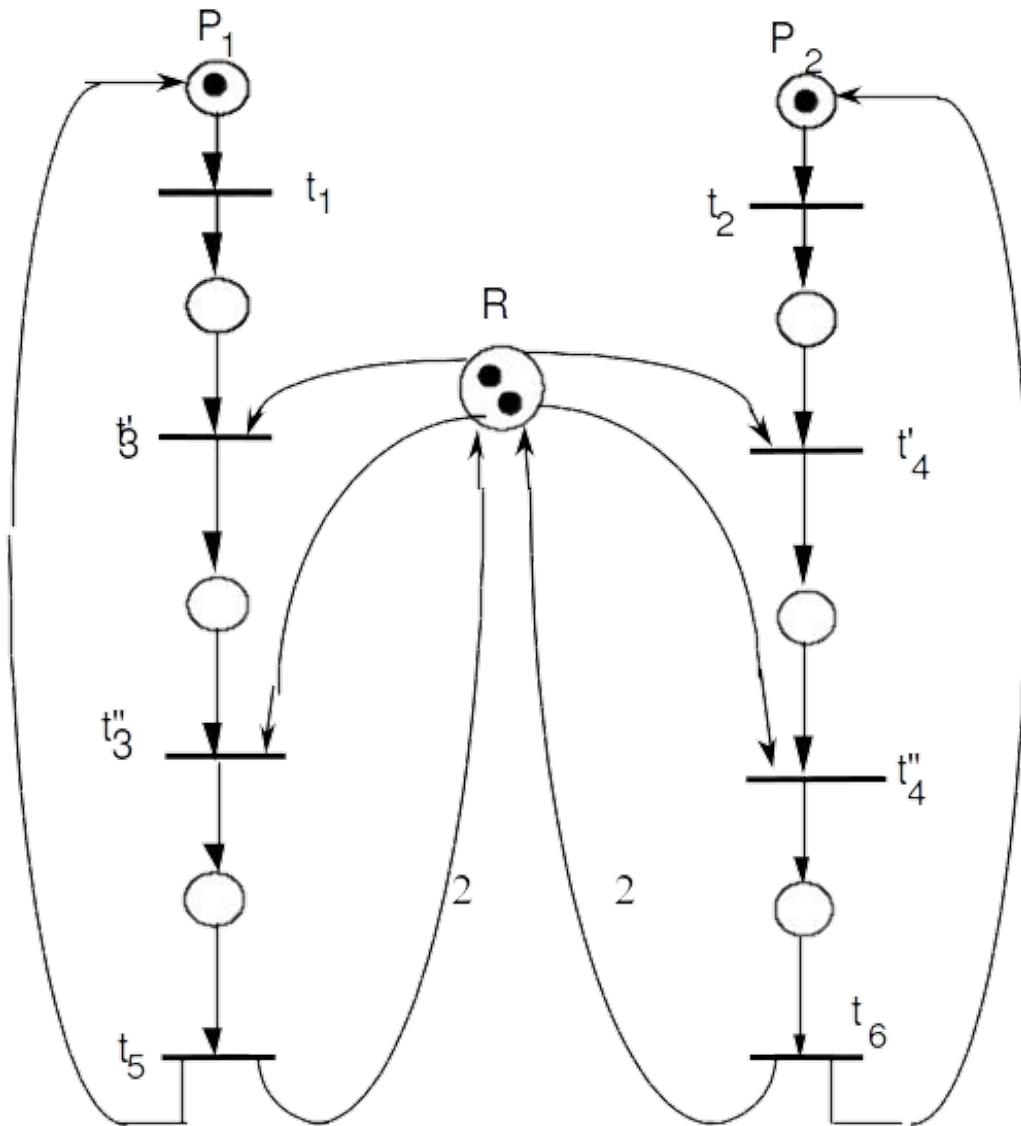
- Concurrency
  - two transitions are enabled to fire in a given state, and the firing of one does not prevent the other from firing
    - see  $t_1$  and  $t_2$  in case (a)
- Conflict
  - two transitions are enabled to fire in a given state, but the firing of one prevents the other from firing
    - see  $t_3$  and  $t_4$  in case (d)
    - place  $P_3$  models a shared resource between two processes
  - no policy exists to resolve conflicts (known as *unfair scheduling*)
  - a process may never get a resource (*starvation*)

# How to avoid starvation



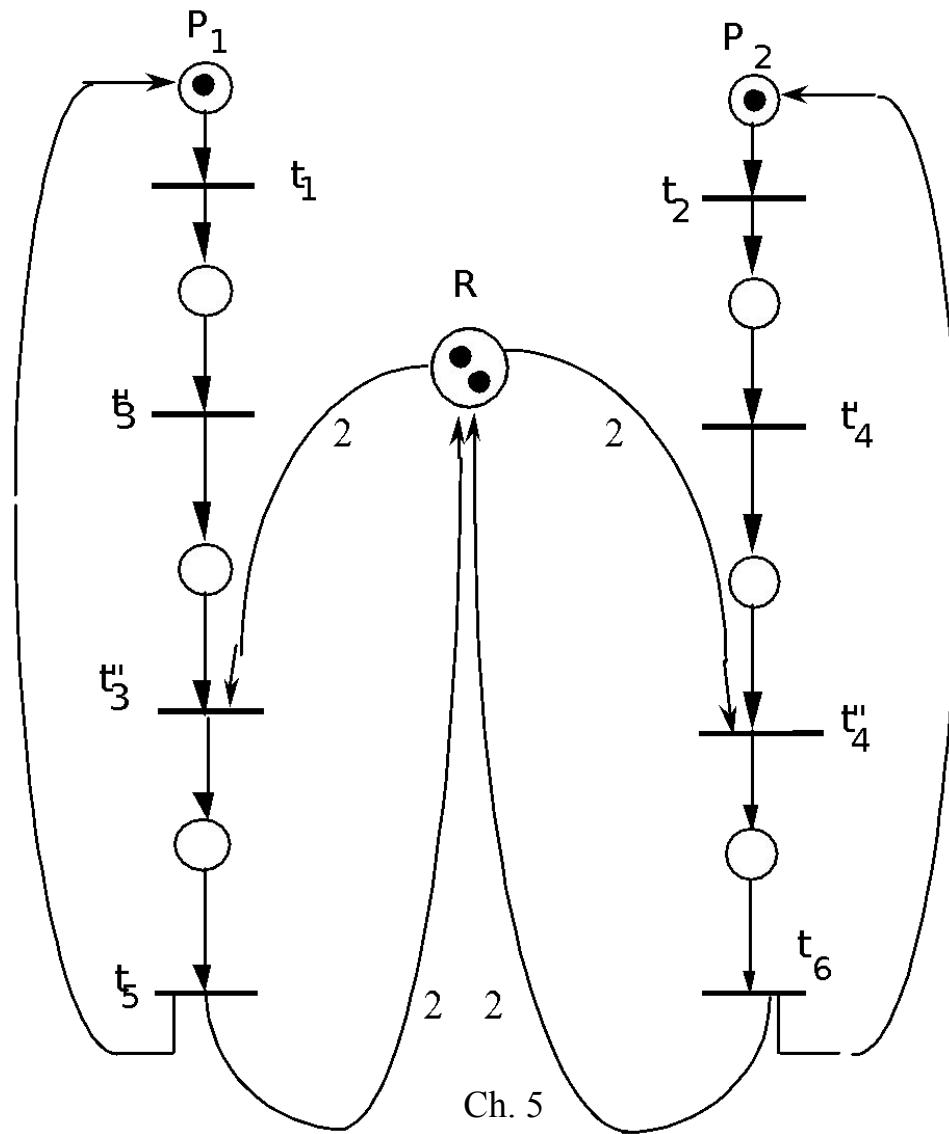
imposes  
alternation

# Deadlock issue

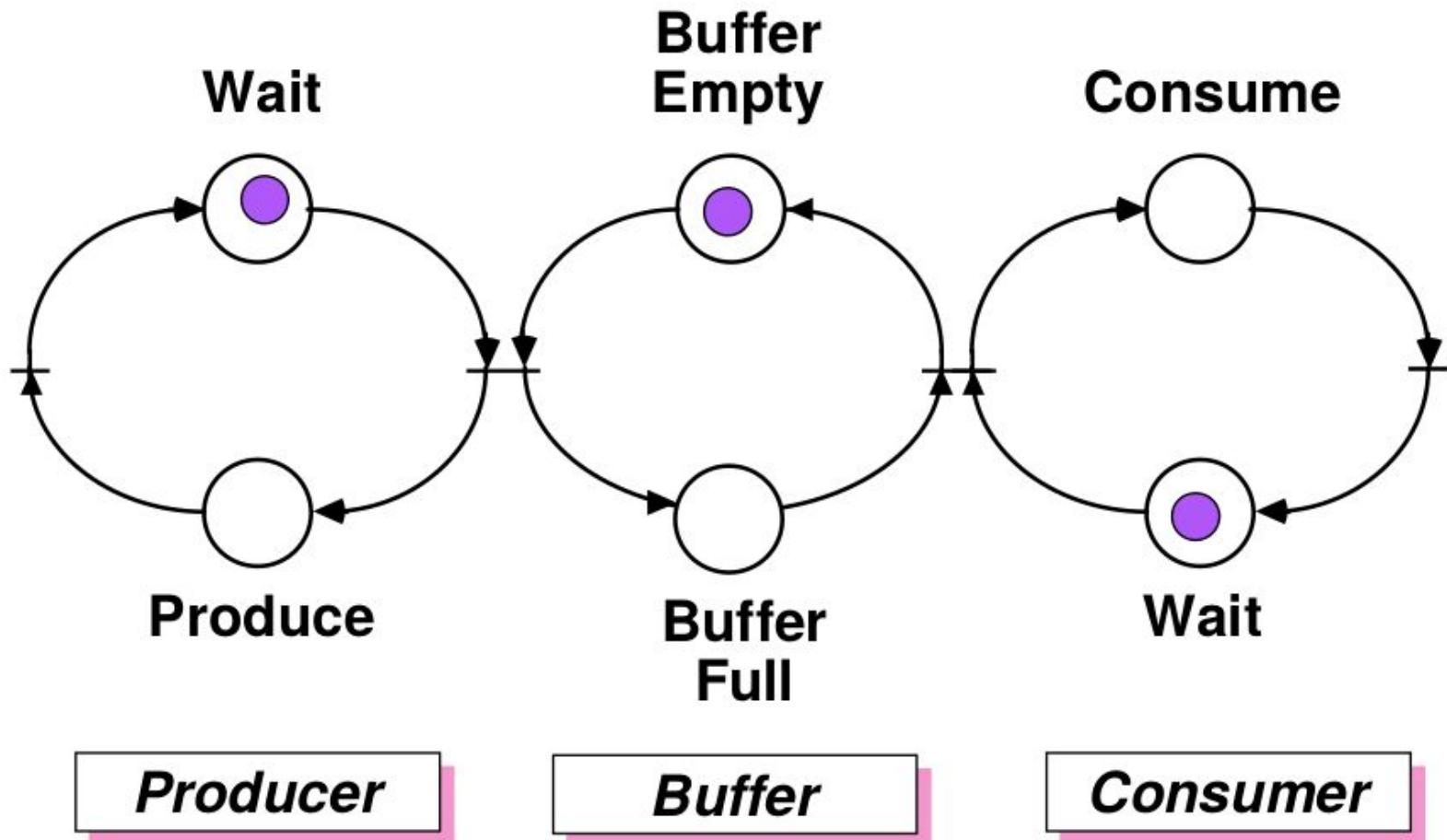


consider  
 $< t_1, t'_3, t_2, t'_4 >$

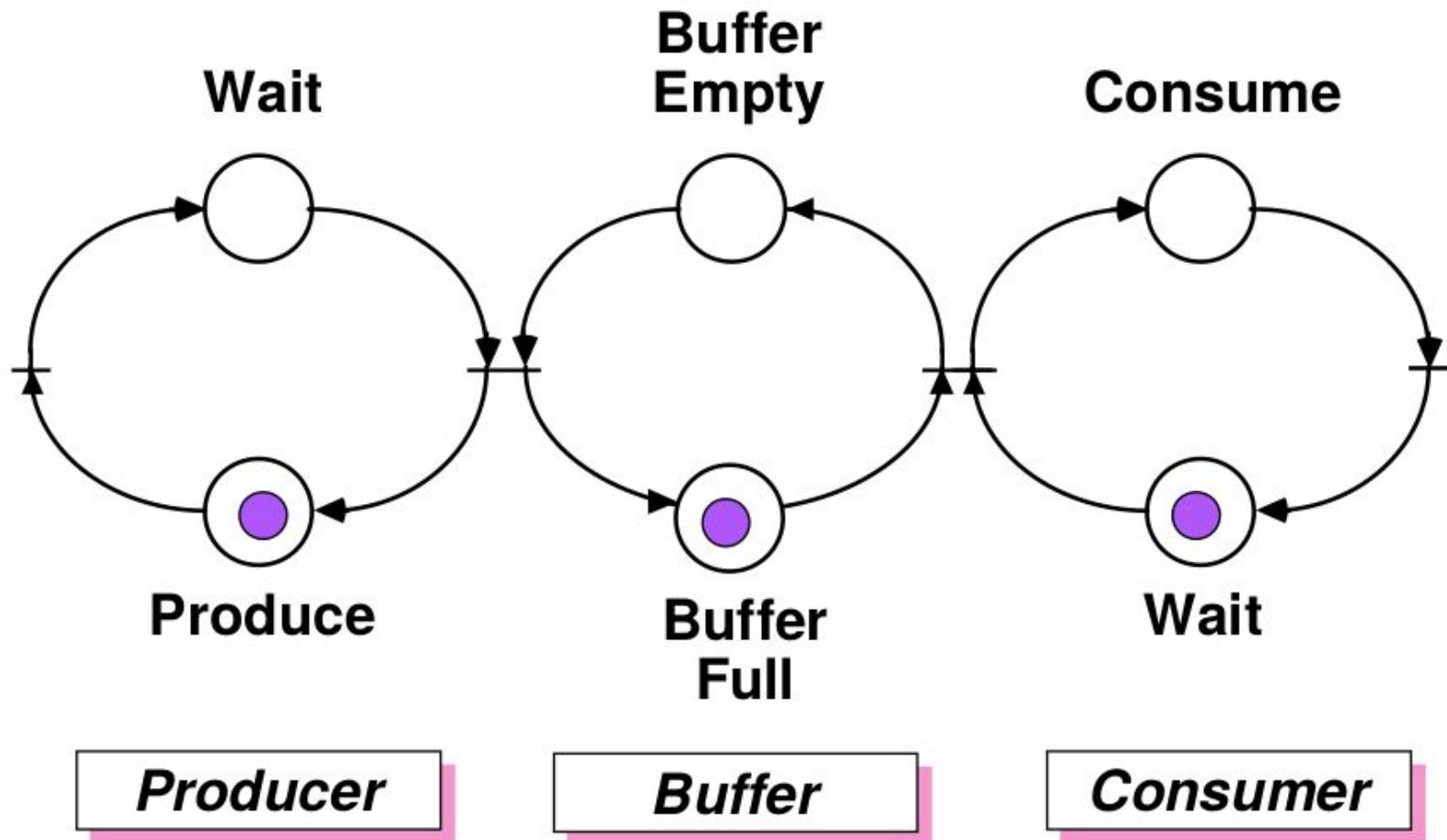
# A deadlock-free net



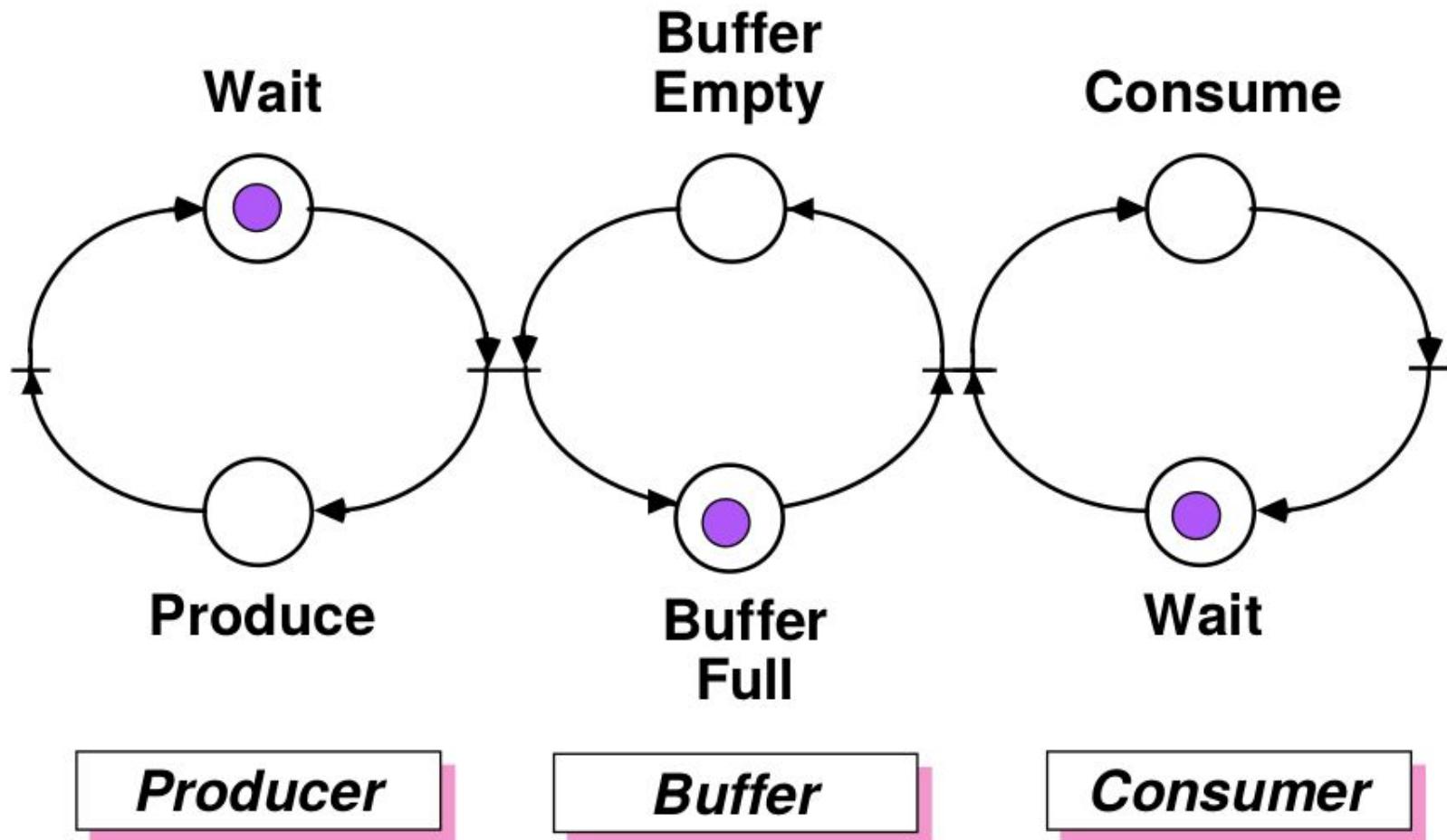
# Producer-consumer example (1)



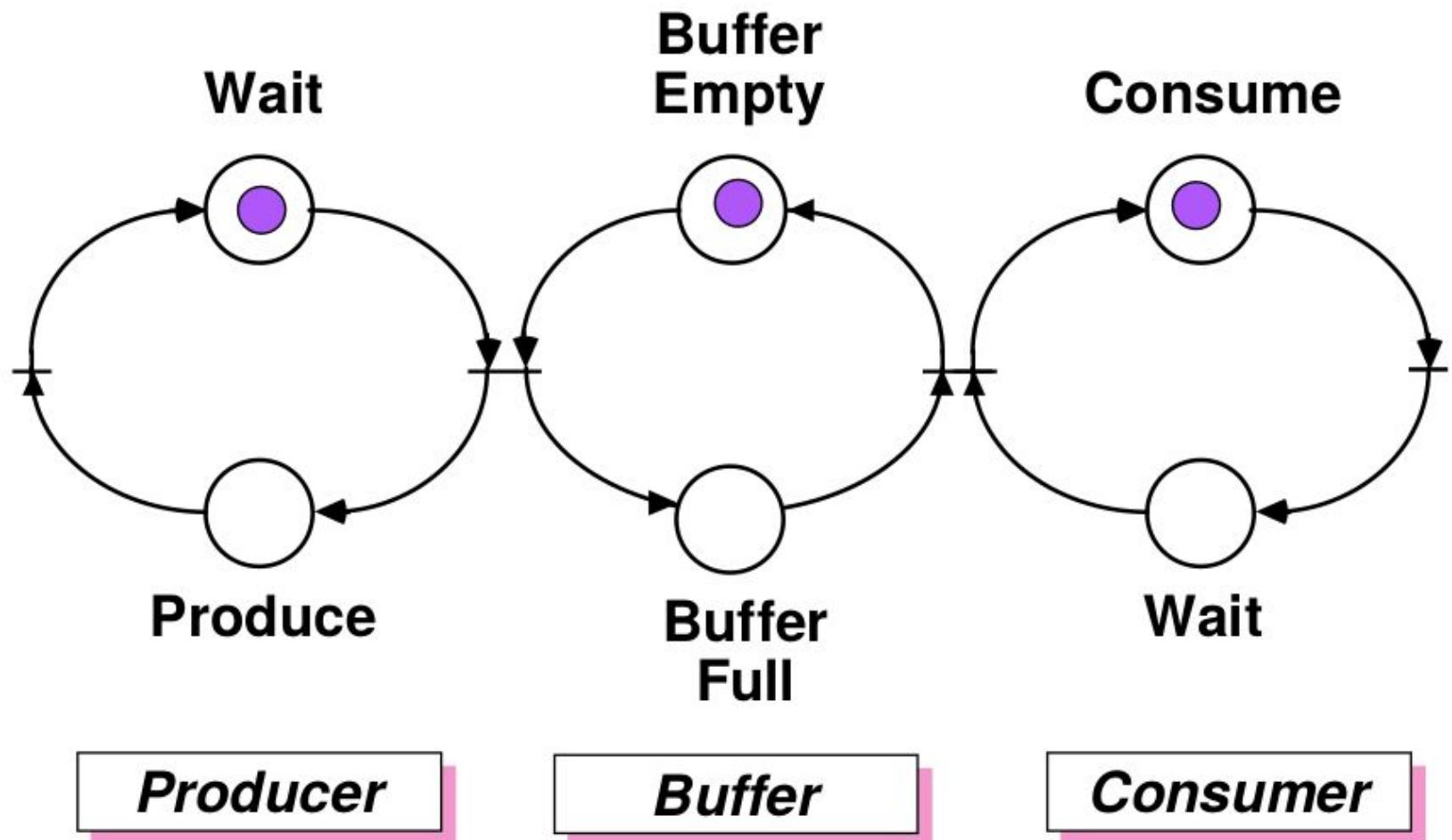
# Producer-consumer example (2)



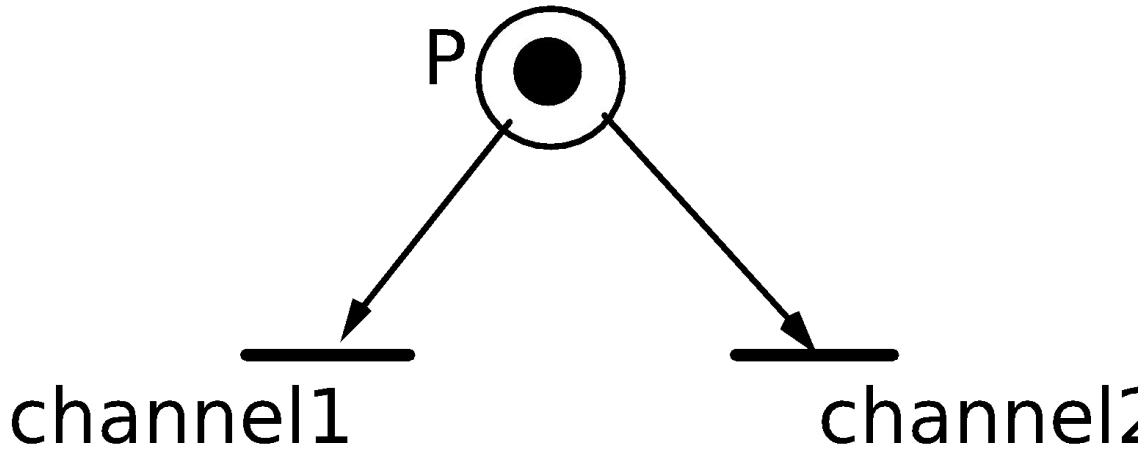
# Producer-consumer example (3)



# Producer-consumer example (4)



# Limitations and extensions



Token represents a message.

You wish to say that the delivery channel depends on contents.

How?

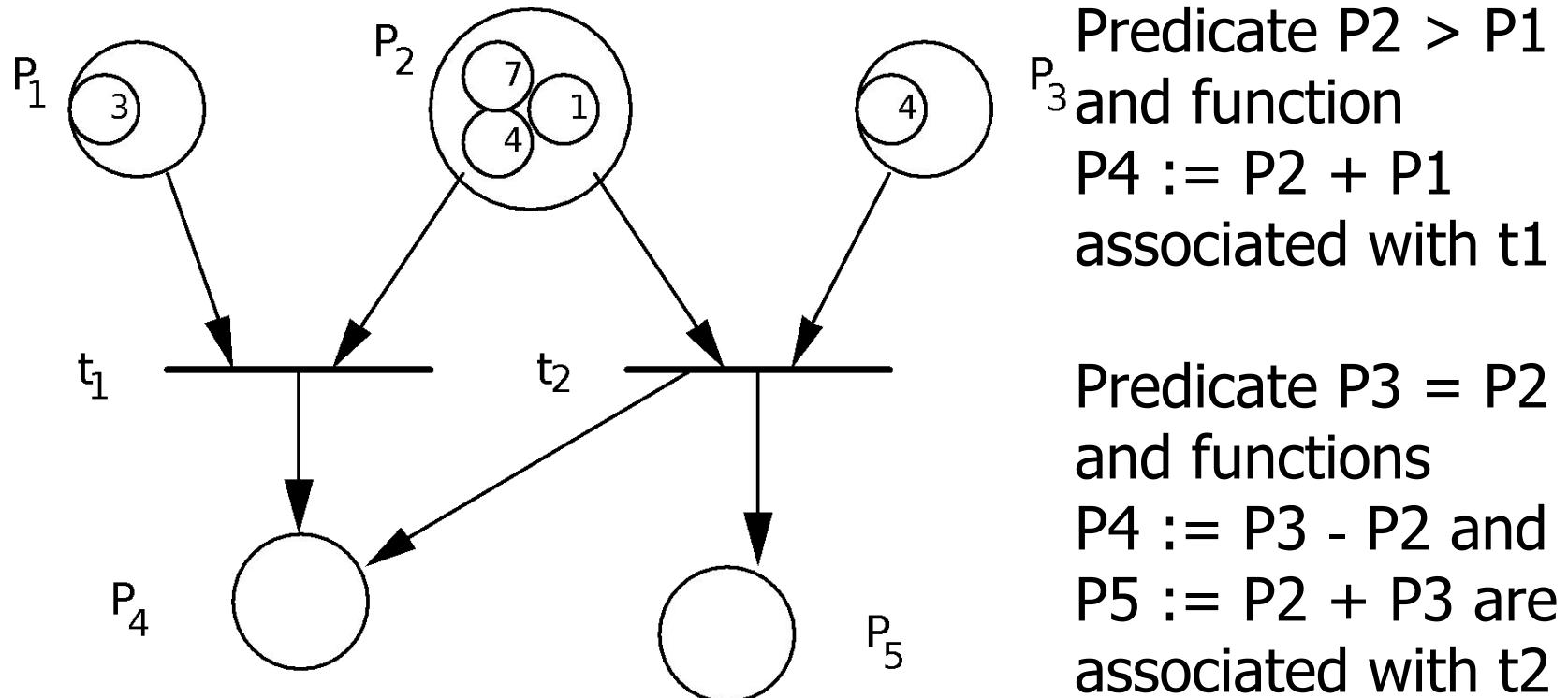
*Petri nets cannot specify selection policies.*

# Extension 1

## assigning values to tokens

- Transitions have associated predicates and functions
- Predicate refers to values of tokens in input places selected for firing
- Functions define values of tokens produced in output places

# Example



The firing of  $t_1$  by using  $\langle 3, 7 \rangle$  would produce the value 10 in  $P_4$ .  $t_2$  can then fire using  $\langle 4, 4 \rangle$

# Extension 2

## specifying priorities

- A priority function  $\text{pri}$  from transitions to natural numbers:
- $\text{pri}: T \rightarrow N$
- When several transitions are enabled, only the ones with maximum priority are allowed to fire
- Among them, the one to fire is chosen nondeterministically

# Extension 3

## Timed Petri nets

- A pair of constants  $\langle t_{\min}, t_{\max} \rangle$  is associated with each transition
- Once a transition is enabled, it must wait for at least  $t_{\min}$  to elapse before it can fire
- If enabled, it *must* fire before  $t_{\max}$  has elapsed, unless it is disabled by the firing of another transition before  $t_{\max}$

# Declarative specifications

ER diagrams

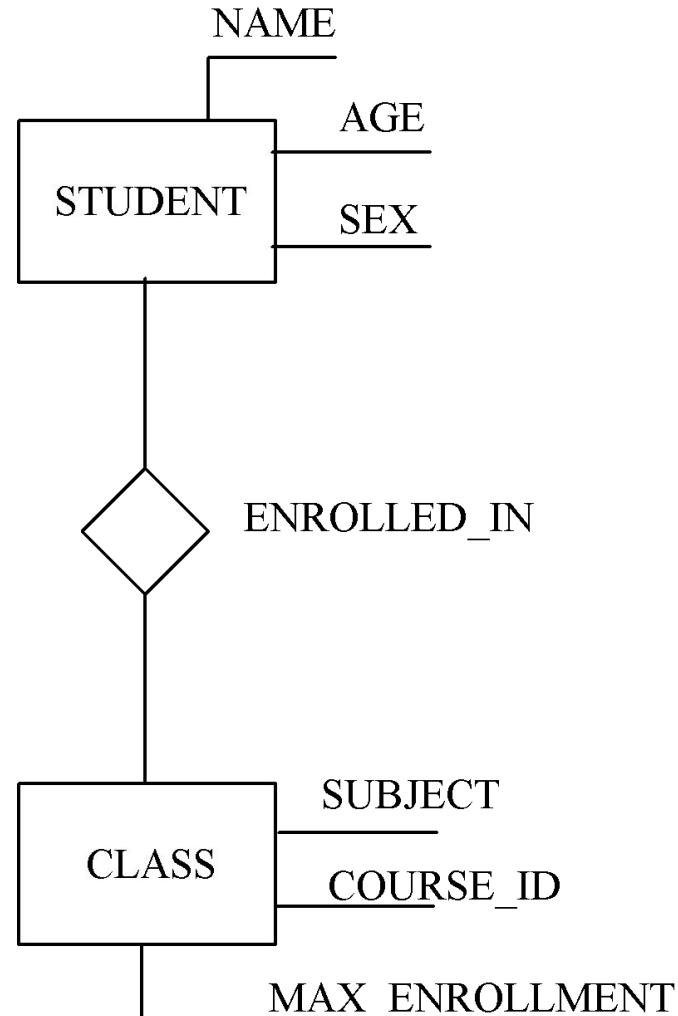
Logic specifications

Algebraic specifications

# ER diagrams

- Often used as a complement to DFD to describe conceptual data models
- Based on entities, relationships, attributes
- They are the ancestors of class diagrams in UML

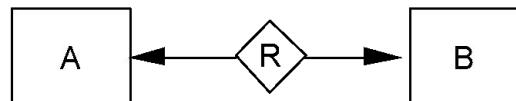
# Example



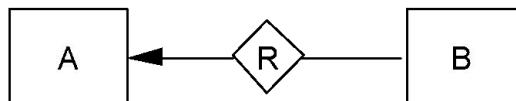
# Relations

- Relations can be partial
- They can be annotated to define

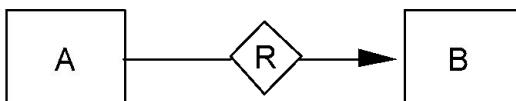
- one to one



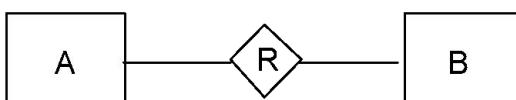
- one to many



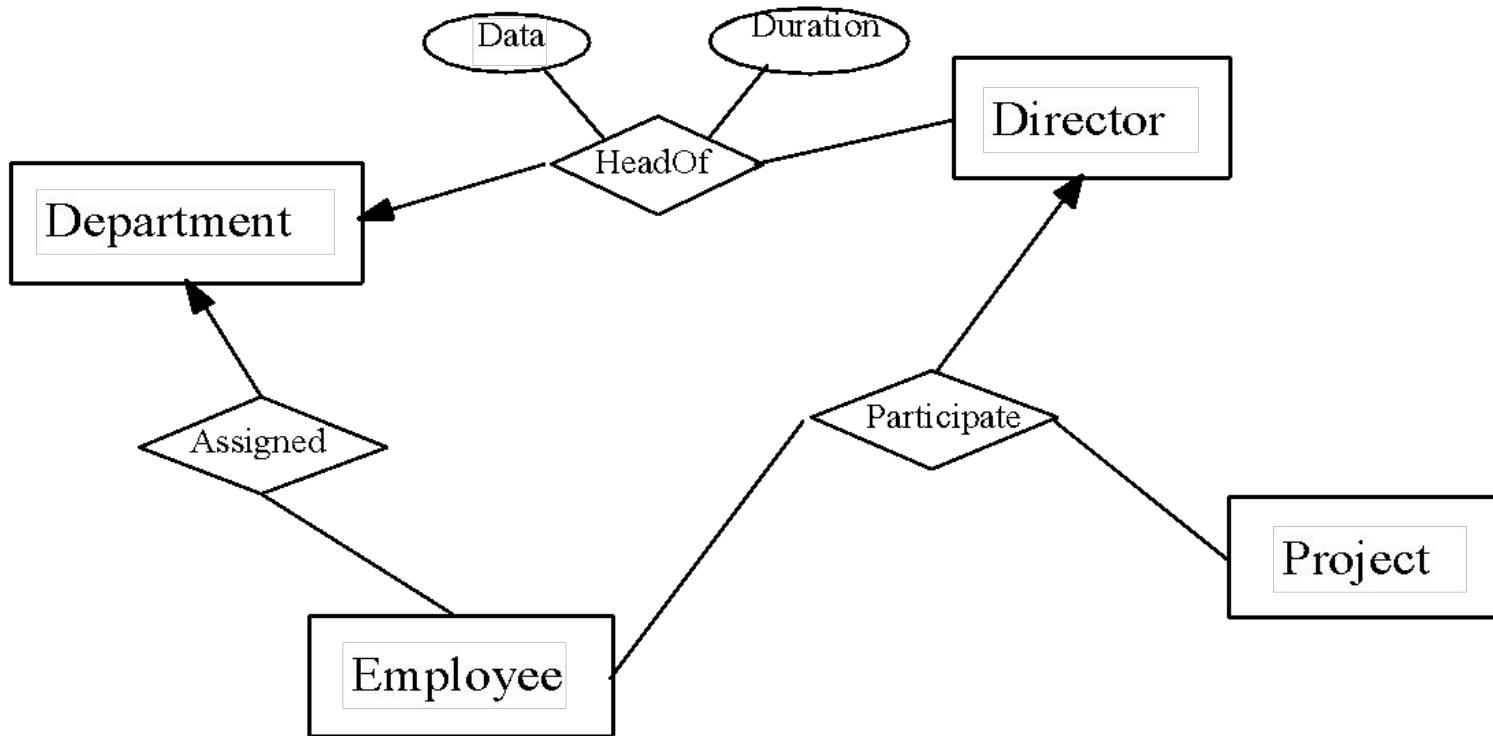
- many to one



- many to many



# Non binary relations



# Logic Specification techniques

# Testing and Proofs

---

| Testing                                      | Proof                                            |
|----------------------------------------------|--------------------------------------------------|
| Observable Property                          | Any program property                             |
| Verify program for one execution             | Verify program for all executions                |
| Manual development with automated regression | Manual development with automated proof checkers |
| Most practical approach now                  | Practical for small critical programs            |

- So why learn about proofs if they are not practical?
  - Proofs tell us how to think about program correctness
  - Foundation for static analysis tools



# How would you argue that this program is correct?

---

```
float sum(float *array, int length) {
 float sum = 0.0;
 int i = 0;
 while (i < length) {
 sum = sum + array[i];
 i = i + 1;
 }
 return sum;
}
```

- ? Mathematical Logic is the solution.....
- ? Descriptive specification technique for specifying software...



# Various Logic techniques

---

- ? Aristotelian logic
- ? Euclidean geometry
- ? Propositional logic
- ? First order logic
- ? Peano axioms
- ? Zermelo Fraenkel set theory
- ? Higher order logic



---

# Propositional Logic



# Propositional (Boolean) Logic

---

- ? In Propositional Logic (a.k.a Propositional Calculus or Sentential Logic), the objects are called **propositions**
- ? Definition
  - ? A proposition is a statement that is either true or false, but not both
  - ? We usually denote a proposition by a letter: p, q, r, s, ...



# Introduction: Proposition

---

- ? The value of a proposition is called its **truth value**;  
denoted by
  - ?  $T$  or 1 if it is true or
  - ?  $F$  or 0 if it is false
- ? **Truth table**

| $p$ |
|-----|
| 0   |
| 1   |



# Introduction: Proposition...

---

- ? The following are propositions
  - ? Today is Monday  $M$
  - ? The grass is wet  $W$
  - ? It is raining  $R$
- ? The following are not propositions
  - ? C++ is the best language *Opinion*
  - ? When is the pretest? *Interrogative*
  - ? Do your homework *Imperative*



# Logical connectives

---

- ? Connectives are used to create a compound proposition
  - ? Negation (denote  $\neg$  or !)
  - ? And or logical conjunction (denoted  $\wedge$ )
  - ? Or or logical disjunction (denoted  $\vee$ )
  - ? XOR or exclusive or (denoted  $\oplus$ )
  - ? Implication (denoted  $\Rightarrow$  or  $\rightarrow$ )
  - ? Biconditional (denoted  $\Leftrightarrow$  or  $\leftrightarrow$ )
- ? We define the meaning (semantics) of the logical connectives using **truth tables**



# Logical Connective: Implication

---

- ? Let  $p$  and  $q$  be two propositions. The implication  $p \rightarrow q$  is the proposition that is false when  $p$  is true and  $q$  is false and true otherwise
  - ?  $p$  is called the hypothesis, antecedent, premise
  - ?  $q$  is called the conclusion, consequence
- ? Truth table

| $p$ | $q$ | $p \rightarrow q$ |
|-----|-----|-------------------|
| 0   | 0   | 1                 |
| 0   | 1   | 1                 |
| 1   | 0   | 0                 |
| 1   | 1   | 1                 |



# Logical Connective: Implication...

---

## ? Examples

- ? If you buy your air ticket in advance, it is cheaper.
- ? If  $x$  is an integer, then  $x^2 \geq 0$ .
- ? If  $2+2=5$ , then all unicorns are pink.



# Logical Connective: Biconditional

---

- ? The biconditional  $p \leftrightarrow q$  is the proposition that is true when  $p$  and  $q$  have the same truth values. It is false otherwise.
- ? Note that it is equivalent to  $(p \rightarrow q) \wedge (q \rightarrow p)$
- ? Truth table

| $p$ | $q$ | $p \leftrightarrow q$ |
|-----|-----|-----------------------|
| 0   | 0   | 1                     |
| 0   | 1   | 0                     |
| 1   | 0   | 0                     |
| 1   | 1   | 1                     |



# Logical Connective: Biconditional...

---

- ? The biconditional  $p \leftrightarrow q$  can be equivalently read as
  - ?  $p$  if and only if  $q$
  - ?  $p$  is a necessary and sufficient condition for  $q$
  - ? if  $p$  then  $q$ , and conversely
- ? Examples
  - ? The alarm goes off if and only if a burglar breaks in
  - ? "if I'm breathing, then I'm alive" and "if I'm alive, then I'm breathing"



# Example : Informal statement

---

- ? A book can either be in stack, on reserve or loaned out.
- ? A book on loan can't be requested
- ? We want to,
  - ? Formalize the concept and statements
  - ? Prove the theorems to gain confidence that the spec. is correct.



# Formalization

---

? Let's first formalize some concepts:

- ? S: the book is in the stack
- ? R: the book is on reserve
- ? L: the book is on loan
- ? Q: the book can be requested



# Formalization...

---

- ? A book can either be in stack, on reserve or loaned out
  - ?  $S \wedge (\neg (R \vee L))$
  - ?  $R \wedge (\neg (S \vee L))$
  - ?  $L \wedge (\neg (S \vee R))$
- ? “A book on loan can’t be requested”
  - ?  $L \Rightarrow (\neg Q)$



# Disadvantage of Propositional Logic

---

- ? Propositional logic has **limited expressive power**
  - ? unlike natural language
  - ? E.g., cannot say “Heavy snow-fall in Himalayas causes cold breeze in some regions of Gujarat “
    - ? except by writing one sentence for each region !!



---

# First Order Logic



# First Order Logic (FOL)

---

- ? Propositional logic assumes the world contains facts.
- ? First-order logic (like natural language) assumes the world contains
  - ? **Objects**: people, houses, wars, ...
  - ? **Relations**: brother of, bigger than, part of, comes between, ...
  - ? **Functions**: one more than, plus, power set ...



# Syntax of FOL: Basic elements

---

## ? Constant Symbols:

- ? Stand for objects
- ? e.g., Abdul Kalam, 2, NIT,...

## ? Predicate Symbols

- ? Stand for relations
- ? E.g., Brother(Ram, Bharat), greater\_than(3,2)...

## ? Function Symbols

- ? Stand for functions
- ? E.g., Sqrt(3), mul(x,y),...



# Syntax of FOL: Basic elements...

---

? Constants Abdul Kalam, 2, NIT, ...

? Predicates Brother, >, ...

? Functions Sqrt, mul, ...

? Variables x, y, a, b, ...

? Connectives  $\neg$ ,  $\Rightarrow$ ,  $\wedge$ ,  $\vee$ ,  $\Leftrightarrow$

? Equality =

? Quantifiers  $\forall$ ,  $\exists$



---

# First Order Logic in Software Specification



# Check validity of address

## Example – Saving addresses

```
// name must not be empty
// state must be valid
// zip must be 5 numeric digits
// street must not be empty
// city must not be empty
```

## Rewriting to logical expression

```
name != "" \wedge state in stateList \wedge zip >= 00000 \wedge zip <= 99999 \wedge street != "" \wedge city != ""
```



# Specifying complete programs: Hoare Triple

---

A *property*, or *requirement*, for  $P$  is specified as a formula of the type

$$\{\text{Pre } (i_1, i_2, \dots, i_n)\}$$
$$P$$
$$\{\text{Post } (o_1, o_2, \dots, o_m, i_1, i_2, \dots, i_n)\}$$

Pre: precondition

Post: postcondition



# Specifying complete programs

---

- ? PRE: FOT formula having  $i_1, i_2, \dots, i_n$  as free variables
- ? POST: FOT formula having  $o_1, o_2, \dots, o_m$ , and possibly  $i_1, i_2, \dots, i_n$  as free variables
- ? PRE :Precondition of P
- ? POST :Post condition of P
- ? *The preceding formula is intended to mean that if PRE holds for the given input values before P's execution, then, after P finishes executing, POST must hold for the output and input values*



# Examples

---

? Simple requirement of the division

$$\{\text{exists } z \ (i_1 = z * i_2)\}$$

P

$$\{o_1 = i_1/i_2\}$$



# Examples...

---

- ? Stronger requirement of the division

$$\{i_1 > i_2\}$$

P

$$\{i_1 = i_2 * o_1 + o_2 \text{ and } o_2 \geq 0 \text{ and } o_2 < i_2\}$$

- ? Imposes more constraints on output values less on input values
- ? A precondition {true} does not place any constraint on input values



# Examples...

---

? Requires that P produce greater of  $i_1$  and  $i_2$

{true}

P

$\{(o = i_1 \text{ and } o \geq i_2) \parallel (o = i_2 \text{ and } o \geq i_1)\}$

? Program to compute sum of the input sequence

{ $n > 0$ }

P

{  $O = \sum_{k=1}^n i_k$  }



# Algebraic specifications

- For formally specifying system behavior.
- Formally define types of data and mathematical operations on those data types.
- Abstracting implementation details, such as the size of representations (in memory) and the efficiency of obtaining outcome of computations.

# Example

- A system for strings, with operations for
  - creating new, empty strings (operation new)
  - concatenating strings (operation append)
  - adding a new character at the end of a string (operation add)
  - checking the length of a given string (operation length)
  - checking whether a string is empty (operation isEmpty)
  - checking whether two strings are equal (operation equal)

# Specification: syntax

algebra StringSpec;

introduces

    sorts String, Char, Nat, Bool;

    operations

        new: () → String;

        append: String, String → String;

        add: String, Char → String;

        length: String → Nat;

        isEmpty: String → Bool;

        equal: String, String → Bool

# Specification: properties

constrains new, append, add, length, isEmpty, equal so that  
for all [s, s1, s2: String; c: Char]

isEmpty (new ()) = true;

isEmpty (add (s, c)) = false;

length (new ()) = 0;

length (add (s, c)) = length (s) + 1;

append (s, new ()) = s;

append (s1, add (s2,c)) = add (append (s1,s2),c);

equal (new (),new ()) = true;

equal (new (), add (s, c)) = false;

equal (add (s, c), new ()) = false;

equal (add (s1, c), add (s2, c)) = equal (s1,s2);

end StringSpec.

# Example: editor

- newF
  - creates a new, empty file
- isEmptyF
  - states whether a file is empty
- addF
  - adds a string of characters to the end of a file
- insertF
  - inserts a string at a given position of a file (the rest of the file will be rewritten just after the inserted string)
- appendF
  - concatenates two files

algebra TextEditor;

introduces

sorts Text, String, Char, Bool, Nat;

operations

newF: () → Text;

isEmptyF: Text → Bool;

addF: Text, String → Text;

insertF: Text, Nat, String → Text;

appendF: Text, Text → Text;

deleteF: Text → Text;

lengthF : Text → Nat;

equalF : Text, Text → Bool;

addFC: Text, Char → Text;

{This is an auxiliary operation that will be needed  
to define addF and other operations on files.}

constrains newF, isEmptyF, addF, appendF, insertF, deleteF  
so that TextEditor generated by [newF, addFC]  
for all [f, f1,f2: Text; s: String; c: Char; cursor: Nat]

```
isEmptyF (newF ()) = true;
isEmptyF (addFC (f, c)) = false;
addF (f, newS ()) = f;
addF (f, addS (s, c)) = addFC (addF (f, s), c);
lengthF (newF ()) = 0;
lengthF (addFC (f, c)) = lengthF (f) + 1;
appendF (f, newF ()) = f;
appendF (f1, addFC (f2, c)) =
 addFC (appendF (f1, f2), c);
equalF (newF (),newF ()) = true;
equalF (newF (), addFC (f, c)) = false;
equalF (addFC (f, c), new ()) = false;
equalF (addFC (f1, c1), addFC (f2, c2)) =
equalF (f1, f2) and equalC (c1, c2);
insertF (f, cursor, newS ()) = f;
end TextEditor.
```

# Incremental specification of an ADT

- We want to target stacks, queues, sets
- We start from "container" and then progressively specialize it
- We introduce another structuring clause
  - assumes
    - defines inheritance relation among algebras

# Container algebra

```
algebra Container;
imports DataType, BoolAlg, NatNumb;
introduces
 sorts Cont;
operations
 new: () → Cont;
 insert: Cont, Data → Cont;
 {Data is the sort of algebra DataType, to which
 elements to be stored in Cont belong}
 isEmpty: Cont → Bool;
 size: Cont → Nat;
constrains new, insert, isEmpty, size so that
Cont generated by [new, insert]
for all [d: Data; c: Cont]
 isEmpty (new ()) = true;
 isEmpty (insert (c, d)) = false;
 size (new ()) = 0;
end Container.
```

# Queue specializes Container

```
algebra QueueContainer;
assumes Container;
introduces
 sorts Queue;
 operations
 last: Queue → Data;
 first: Queue → Data;
 equalQ : Queue , Queue → Bool;
 delete:Queue → Queue;
constrains last, first, equalQ, delete, isEmpty, new, insert so that
for all [d: Data; q, q1, q2: Queue]
 last (insert (q, d)) = d;
 first (insert (new(), d) = d
 first (insert (q, d)) = if not isEmpty (q) then first (q);
 equalQ (new (), new ()) = true;
 equalQ (insert (q, d), new ()) = false;
 equalQ (new (), insert (q, d)) = false;
 equalQ (insert (q1, d1), insert (q2, d2)) = equalD (d1, d2) and
 equalQ (q1,q2);
 delete (new ()) = new ();
 delete (insert (new (), d)) = new ();
end QueueContainer.
```

# From specs to an implementation

- Algebraic spec language described so far is based on the "Larch shared language"
- Several "interface languages" are available to help transitioning to an implementation
  - Larch/C++, Larch/Pascal

# Languages for modular specifications

Statecharts

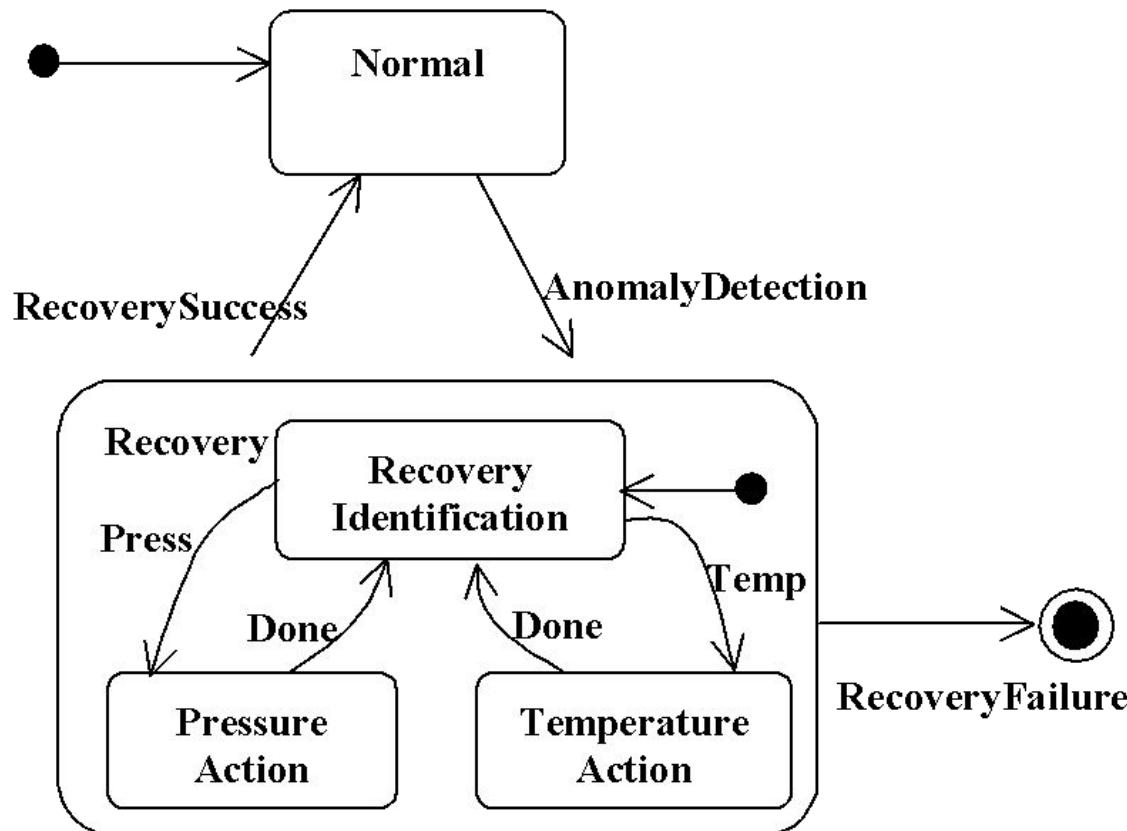
Z

# Modularizing finite state machines

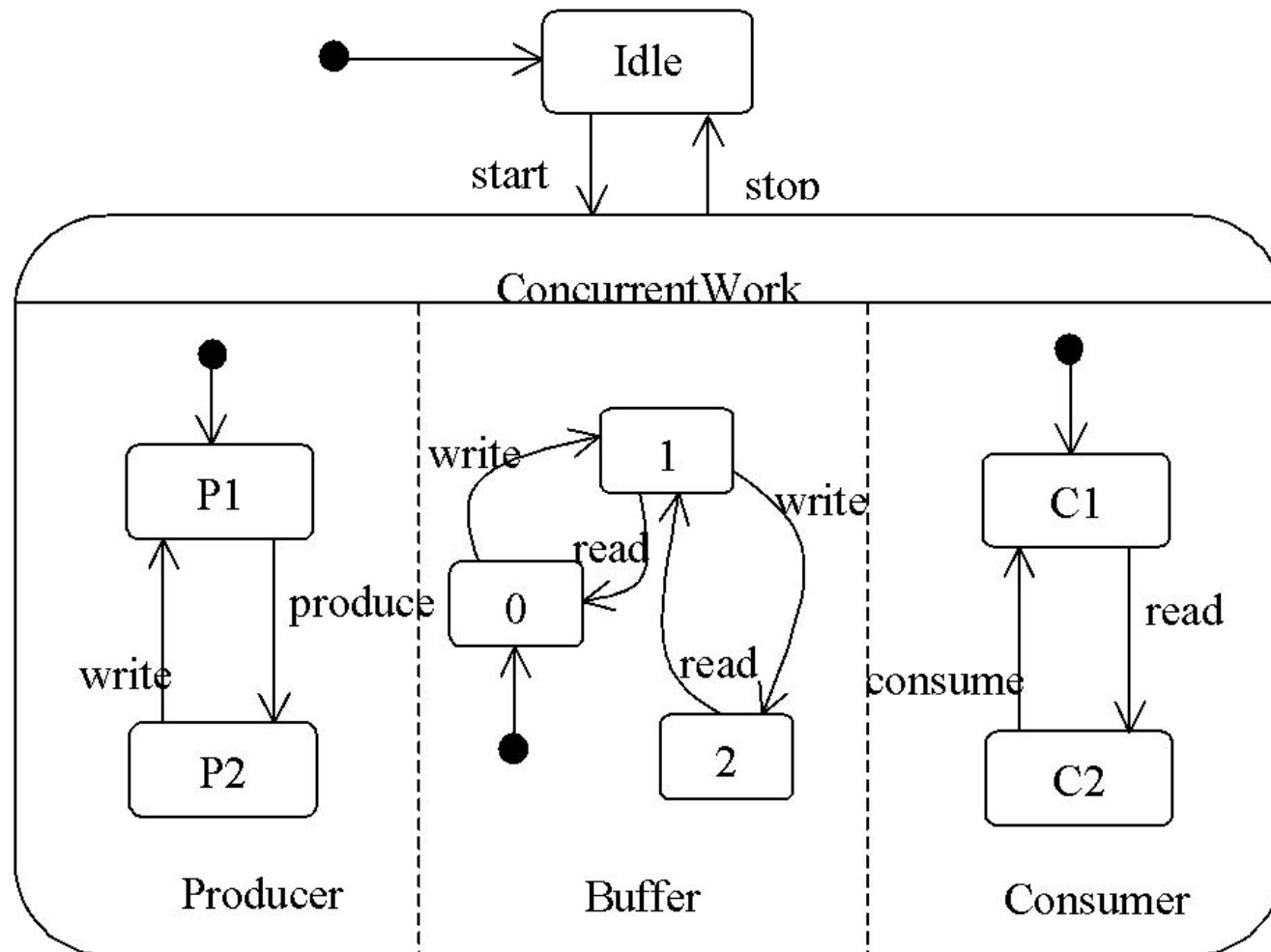
- Statecharts do that
- They have been incorporated in UML
- They provide the notions of
  - superstate
  - state decomposition

# Sequential decomposition

--chemical plant control example--



# Parallel decomposition



# Modularizing logic specifications: Z

- System specified by describing state space, using *Z schemas*
- Properties of state space described by *invariant* predicates
  - predicates written in first-order logic
- Operations define state transformations

# The elevator example in Z

*SWITCH ::= on | off*

*MOVE ::= up | down*

*FLOORS : N*

*FLOORS > 0*

*IntButtons*

*IntReq : 1 .. FLOORS → SWITCH*

*FloorButtons*

*ExtReq : 1 .. FLOORS → P MOVE*

*down*  $\notin$  *ExtReq(1)*

*up*  $\notin$  *ExtReq(FLOORS)*

*Scheduler*

*NextFloorToServe : 0 .. FLOORS*

*Elevator*

*CurFloor : 1 .. FLOORS*

*CurDirection : MOVE*

# Complete state space attempt #1

*System* \_\_\_\_\_

*Elevator*

*IntButtons*

*FloorButtons*

*Scheduler*

*NextFloorToServe*  $\neq 0$

$\Rightarrow \text{IntReq}(\text{NextFloorToServe}) = \text{on} \vee \text{ExtReq}(\text{NextFloorToServe}) \neq \emptyset$

# Complete state space attempt #2

*System* \_\_\_\_\_

*Elevator*

*IntButtons*

*FloorButtons*

*Scheduler*

*NextFloorToServe*  $\neq 0 \Rightarrow$

$\text{IntReq}(\text{NextFloorToServe}) = \text{on} \vee \text{ExtReq}(\text{NextFloorToServe}) \neq \emptyset$

*NextFloorToServe* = 0  $\Rightarrow$

$(\forall f : 1 \dots \text{FLOORS} \bullet (\text{IntReq}(f) = \text{off} \wedge \text{ExtReq}(f) = \emptyset))$

# Complete state space final

*System* —

*Elevator*

*IntButtons*

*FloorButtons*

*Scheduler*

$\exists \text{Pri1}, \text{Pri2}, \text{Pri3} : \mathbb{PN}_1 \bullet$   
 $\text{CurDirection} = \text{up} \Rightarrow$   
 $(\text{Pri1} = \{f : 1..FLOORS \mid f \geq \text{CurFloor} \wedge (\text{IntReq}(f) = \text{on} \vee \text{up} \in \text{ExtReq}(f))\}) \wedge$   
 $\text{Pri2} = \{f : 1..FLOORS \mid \text{down} \in \text{ExtReq}(f) \vee (f < \text{CurFloor} \wedge \text{IntReq}(f) = \text{on})\} \wedge$   
 $\text{Pri3} = \{f : 1..FLOORS \mid f < \text{CurFloor} \wedge \text{up} \in \text{ExtReq}(f)\} \wedge$   
 $((\text{Pri1} \neq \emptyset \wedge \text{NextFloorToServe} = \min(\text{Pri1})) \vee$   
 $(\text{Pri1} = \emptyset \wedge \text{Pri2} \neq \emptyset \wedge \text{NextFloorToServe} = \max(\text{Pri2})) \vee$   
 $(\text{Pri1} = \emptyset \wedge \text{Pri2} = \emptyset \wedge \text{Pri3} \neq \emptyset \wedge \text{NextFloorToServe} = \min(\text{Pri3}))$   
 $\vee (\text{Pri1} = \emptyset \wedge \text{Pri2} = \emptyset \wedge \text{Pri3} = \emptyset \wedge \text{NextFloorToServe} = 0)) \wedge$   
 $\text{CurDirection} = \text{down} \Rightarrow$   
 $(\text{Pri1} = \{f : 1..FLOORS \mid f \leq \text{CurFloor} \wedge$   
 $(\text{IntReq}(f) = \text{on} \vee \text{down} \in \text{ExtReq}(f))\} \wedge$   
 $\text{Pri2} = \{f : 1..FLOORS \mid \text{up} \in \text{ExtReq}(f) \vee$   
 $(f > \text{CurFloor} \wedge \text{IntReq}(f) = \text{on})\} \wedge$   
 $\text{Pri3} = \{f : 1..FLOORS \mid f > \text{CurFloor} \wedge \text{down} \in \text{ExtReq}(f)\} \wedge$   
 $((\text{Pri1} \neq \emptyset \wedge \text{NextFloorToServe} = \max(\text{Pri1})) \vee$   
 $(\text{Pri1} = \emptyset \wedge \text{Pri2} \neq \emptyset \wedge \text{NextFloorToServe} = \min(\text{Pri2})) \vee$   
 $(\text{Pri1} = \emptyset \wedge \text{Pri2} = \emptyset \wedge \text{Pri3} \neq \emptyset \wedge \text{NextFloorToServe} = \max(\text{Pri3}))$   
 $\vee (\text{Pri1} = \emptyset \wedge \text{Pri2} = \emptyset \wedge \text{Pri3} = \emptyset \wedge \text{NextFloorToServe} = 0))$

# Operations (1)

|                                                                                                                                                                                                                                                                                                                                                                       |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>MoveToNextFloor</i>                                                                                                                                                                                                                                                                                                                                                |
| $\Delta System$                                                                                                                                                                                                                                                                                                                                                       |
| $NextFloorToServe \neq 0$<br>$CurFloor \neq NextFloorToServe$<br>$CurFloor > NextFloorToServe \Rightarrow$<br>$CurFloor' = CurFloor - 1 \wedge CurDirection' = down$<br>$CurFloor < NextFloorToServe \Rightarrow$<br>$CurFloor' = CurFloor + 1 \wedge CurDirection' = up$<br>$\theta IntButtons' = \theta IntButtons$<br>$\theta FloorButtons' = \theta FloorButtons$ |
| <i>InternalPush</i>                                                                                                                                                                                                                                                                                                                                                   |
| $\Delta System$                                                                                                                                                                                                                                                                                                                                                       |
| $f? : 1..FLOORS$<br>$IntReq' = IntReq \cup \{f? \rightarrow on\}$<br>$\theta Elevator' = \theta Elevator$<br>$\theta FloorButtons' = \theta FloorButtons$                                                                                                                                                                                                             |
| <i>ExternalPush</i>                                                                                                                                                                                                                                                                                                                                                   |
| $\Delta System$                                                                                                                                                                                                                                                                                                                                                       |
| $f? : 1..FLOORS$<br>$dir? : MOVE$<br>$ExtReq' = ExtReq \cup \{(f? \rightarrow (ExtReq(f?) \cup \{dir?\}))\}$<br>$\theta Elevator' = \theta Elevator$<br>$\theta IntButtons' = \theta IntButtons$                                                                                                                                                                      |
| <i>ServeIntRequest</i>                                                                                                                                                                                                                                                                                                                                                |
| $\Delta System$                                                                                                                                                                                                                                                                                                                                                       |
| $NextFloorToServe = CurFloor$<br>$IntReq(CurFloor) = on$<br>$IntReq' = IntReq \cup \{(CurFloor \rightarrow off)\}$<br>$ExtReq' = ExtReq$<br>$CurFloor' = CurFloor$<br>$CurDirection' = CurDirection$                                                                                                                                                                  |

# Operations (2)

|                                                                                                |       |
|------------------------------------------------------------------------------------------------|-------|
| <i>ServeExtRequestSameDir</i>                                                                  | <hr/> |
| $\Delta System$                                                                                |       |
| $NextFloorToServe = CurFloor$                                                                  |       |
| $IntReq(CurFloor) = off$                                                                       |       |
| $CurDirection \in ExtReq(CurFloor)$                                                            |       |
| $IntReq' = IntReq$                                                                             |       |
| $ExtReq' = ExtReq \oplus \{(CurFloor \mapsto (ExtReq(CurFloor) \setminus \{CurDirection\}))\}$ |       |
| $CurFloor' = CurFloor$                                                                         |       |
| $CurDirection' = CurDirection$                                                                 |       |
| <hr/>                                                                                          |       |
| <i>ServeExtRequestOtherDir</i>                                                                 | <hr/> |
| $\Delta System$                                                                                |       |
| $NextFloorToServe = CurFloor$                                                                  |       |
| $IntReq(CurFloor) = off$                                                                       |       |
| $CurDirection \notin ExtReq(CurFloor)$                                                         |       |
| $IntReq' = IntReq$                                                                             |       |
| $ExtReq' = ExtReq \odot \{(CurFloor \mapsto \emptyset)\}$                                      |       |
| $CurFloor' = CurFloor$                                                                         |       |
| $CurDirection' = CurDirection$                                                                 |       |
| <hr/>                                                                                          |       |
| <i>SystemInit</i>                                                                              | <hr/> |
| $System'$                                                                                      |       |
| $\forall i : 1..FLOORS \bullet IntReq'(i) = off \wedge ExtReq'(i) = \emptyset$                 |       |
| $NextFloorToServe' = 0$                                                                        |       |
| $CurFloor' = 1$                                                                                |       |
| $CurDirection' = up$                                                                           |       |

# Conclusions (1)

- Specifications describe
  - what the users need from a system (requirements specification)
  - the design of a software system (design and architecture specification)
  - the features offered by a system (functional specification)
  - the performance characteristics of a system (performance specification)
  - the external behavior of a module (module interface specification)
  - the internal structure of a module (internal structural specification)

# Conclusions (2)

- Descriptions are given via suitable notations
  - There is no “ideal” notation
- They must be modular
- They support communication and interaction between designers and users

---

# **Software Requirements**

# Topics covered

---

- User requirements
- System requirements
- Functional requirements
- Non-functional requirements
- Domain requirements

# Requirements engineering

---

- Requirements for a system are the descriptions of the services that a system should provide and the constraints on its operation.
- The process of finding out, analyzing, documenting and checking these services and constraints is called requirements engineering (RE).

# What is a requirement?

---

- The term ‘requirement’ is not consistently defined in software industry.
- It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification

# Requirements abstraction (Davis)

---

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

# Types of requirement

---

- User requirements
  - Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers
- System requirements
  - A structured document setting out detailed descriptions of the system services. Written as a contract between client and contractor

# User and system requirements

---

## User requirements definition

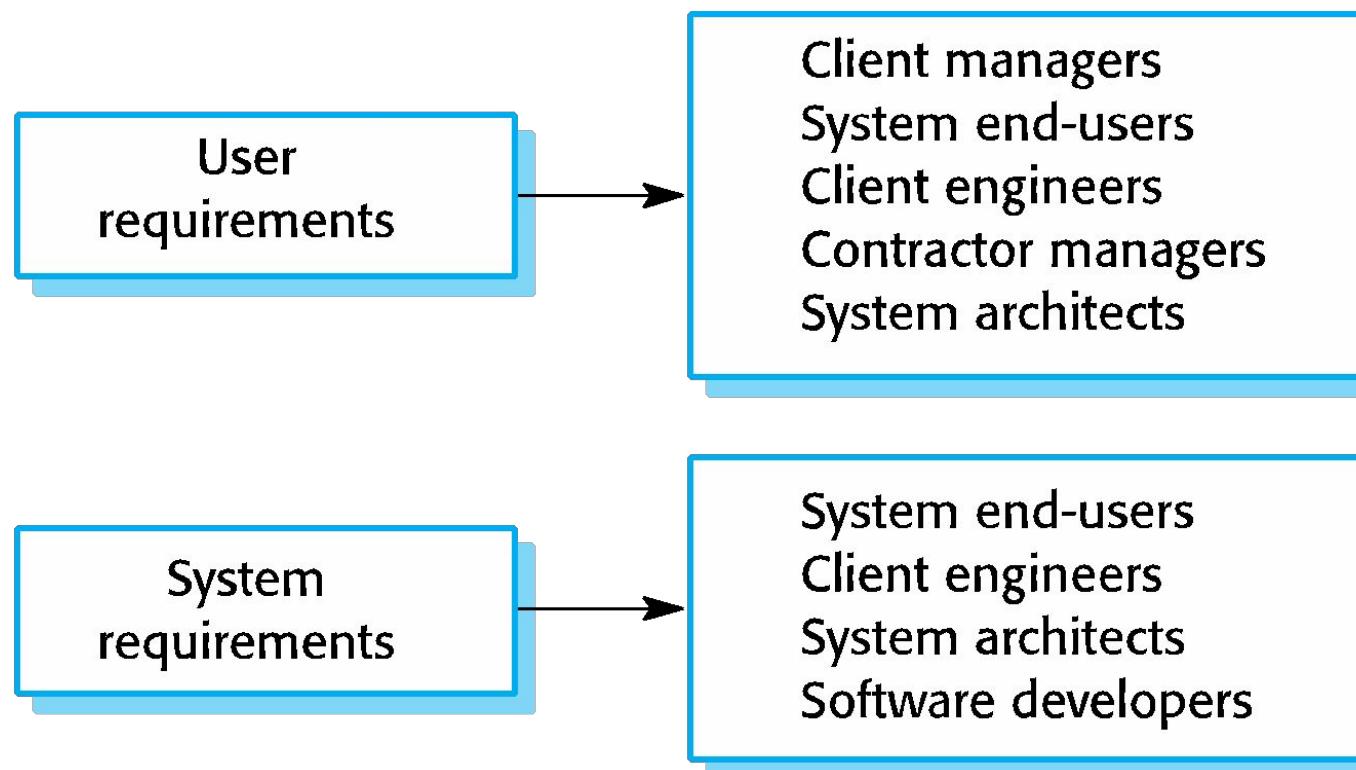
1. The Mentcare system shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

## System requirements specification

- 1.1 On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
- 1.2 The system shall generate the report for printing after 17.30 on the last working day of the month.
- 1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed and the total cost of the prescribed drugs.
- 1.4 If drugs are available in different dose units (e.g. 10mg, 20mg, etc) separate reports shall be created for each dose unit.
- 1.5 Access to drug cost reports shall be restricted to authorized users as listed on a management access control list.

# Requirements readers

---



# **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
- Depend on the type of software, expected users and the type of system where the software is used
- 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

# Example of functional requirements

---

- User shall be able to search appointment lists for all the clinics.
- The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
- Each staff member using the system shall be uniquely identified by his or her 8-digit employee number.

# 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 ‘search’ in requirement 1
  - User intention: search for a patient name across all appointments in all clinics
  - Developer interpretation: search for a patient name in an individual clinic. User chooses clinic then search.

# **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

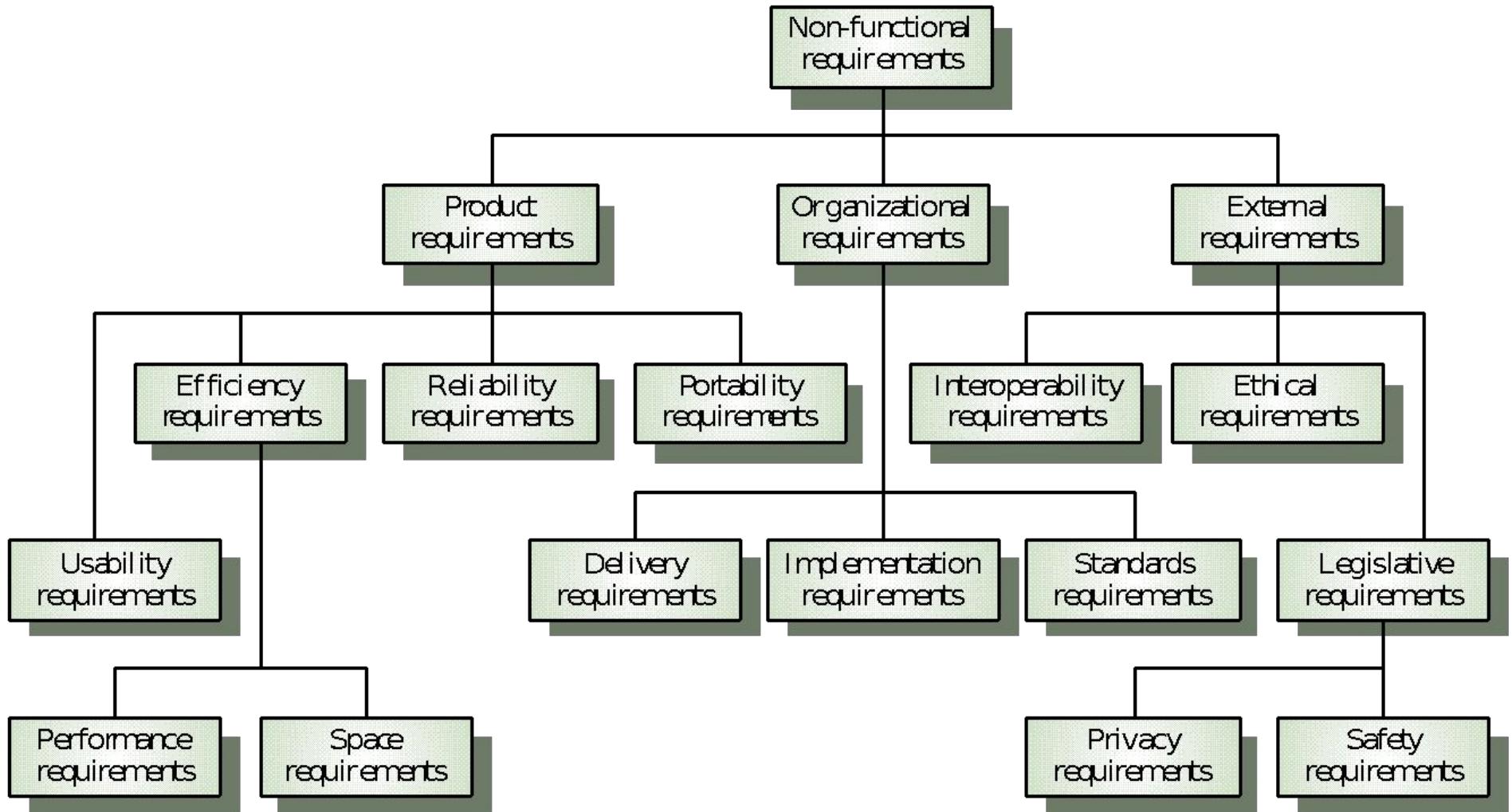
# Non-functional requirements

---

- 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 IDE, programming language or development method
- Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless

# Non-functional requirement types

---



# Non-functional classifications

---

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

# Non-functional requirements examples

---

## **Product requirement**

The Mentcare system shall be available to all clinics during normal working hours (Mon–Fri, 0830–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.

## **Organizational requirement**

Users of the Mentcare system shall authenticate themselves using their health authority identity card.

## **External requirement**

The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

# 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
- Goals are helpful to developers as they convey the intentions of the system users

# Examples

---

- **A system goal**
  - The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized.
- **A verifiable non-functional requirement**
  - Medical staff shall be able to use all the system functions after a total of four hours training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.

# Requirements measures

---

| Property    | Measure                                                                                                            |
|-------------|--------------------------------------------------------------------------------------------------------------------|
| Speed       | Processed transactions/second<br>User/Event response time<br>Screen refresh time                                   |
| Size        | K Bytes<br>Number of RAM chips                                                                                     |
| Ease of use | Training time<br>Number of help frames                                                                             |
| Reliability | Mean time to failure<br>Probability of unavailability<br>Rate of failure occurrence<br>Availability                |
| Robustness  | Time to restart after failure<br>Percentage of events causing failure<br>Probability of data corruption on failure |
| Portability | Percentage of target dependent statements<br>Number of target 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 how specific computations must be carried out
- If domain requirements are not satisfied, the system may be unworkable

# Library system domain requirements

---

- There shall be a standard user interface to all databases which shall be based on the Z39.50 standard.
- Because of copyright restrictions, some documents must be deleted immediately on arrival. Depending on the user's requirements, these documents will either be printed locally on the system server for manually forwarding to the user or routed to a network printer.

# 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

# Requirements engineering process

---

- The processes used for RE vary widely depending on the application domain, the people involved and the organization developing the requirements.
- However, there are a number of generic activities common to all processes
  - Feasibility Study
  - Requirements elicitation
  - Requirements analysis
  - Requirements validation
  - Requirements management