

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

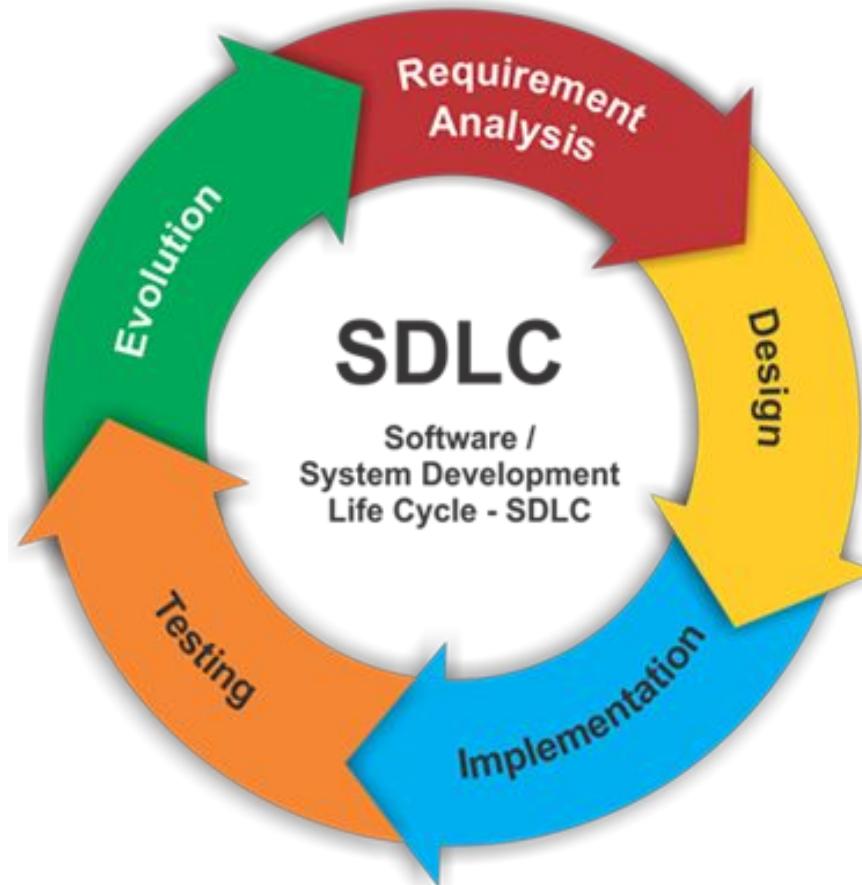
Reason for failure

- 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.

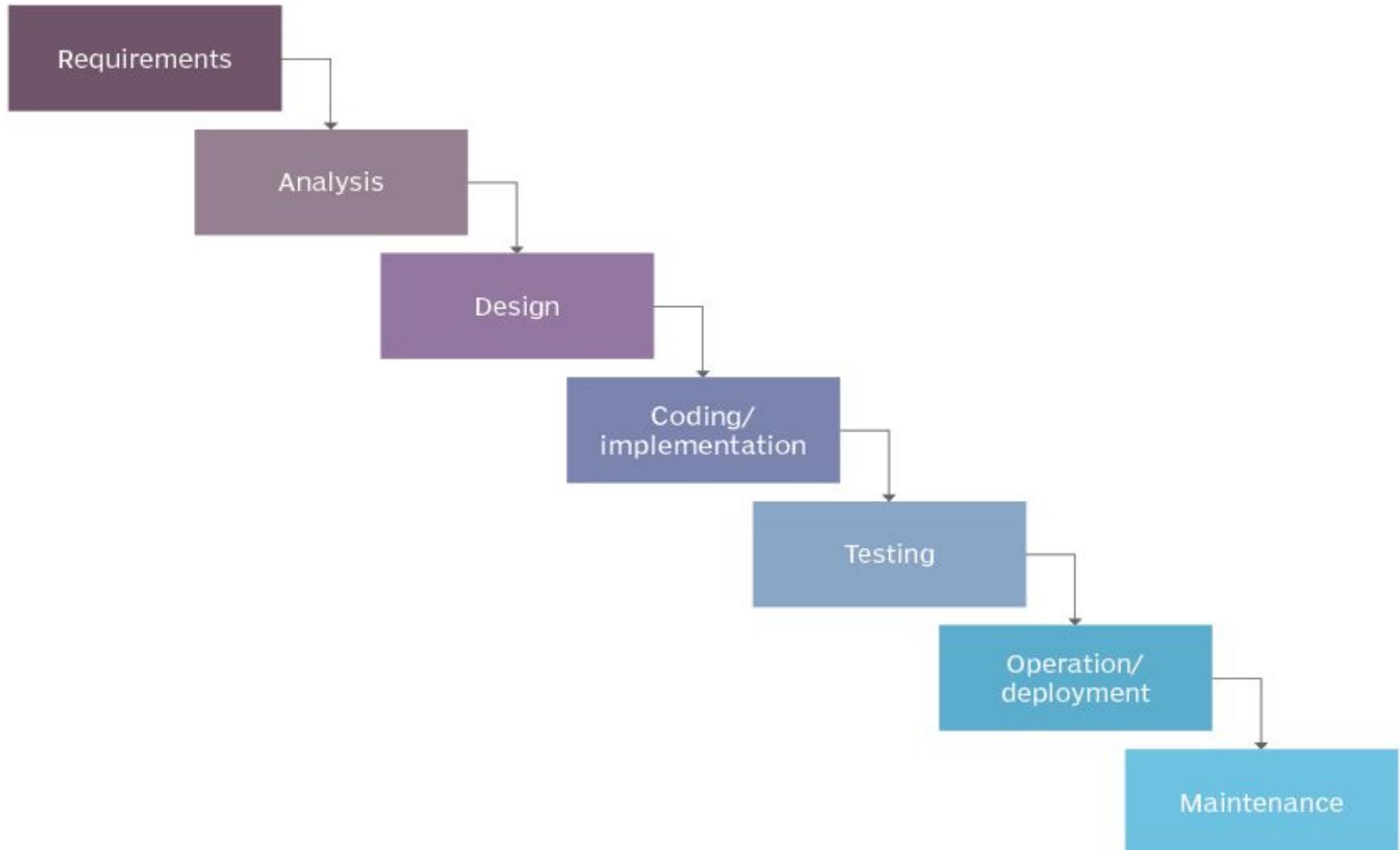
Code of ethics - principles

- **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

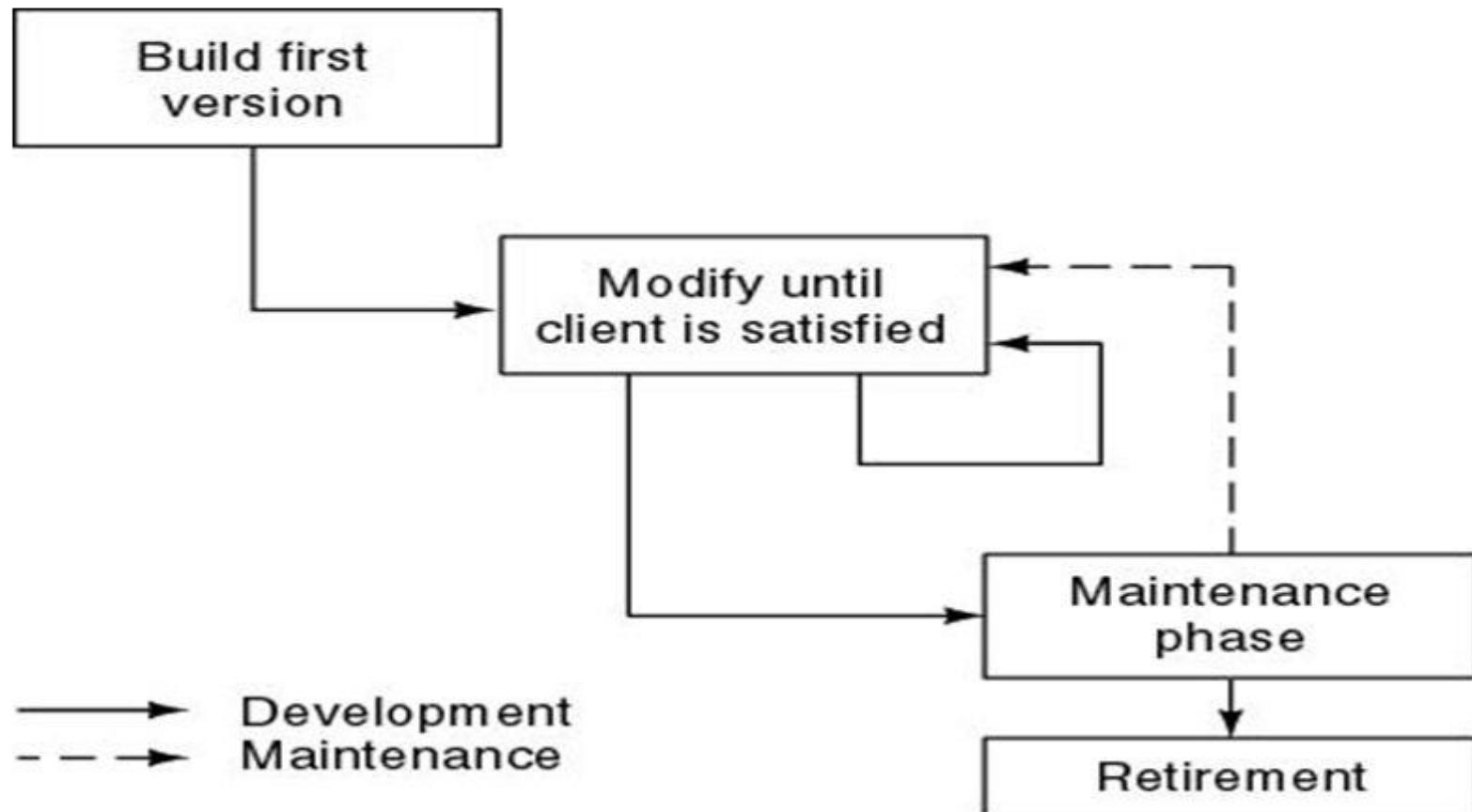
Deliverables in Waterfall model

- 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



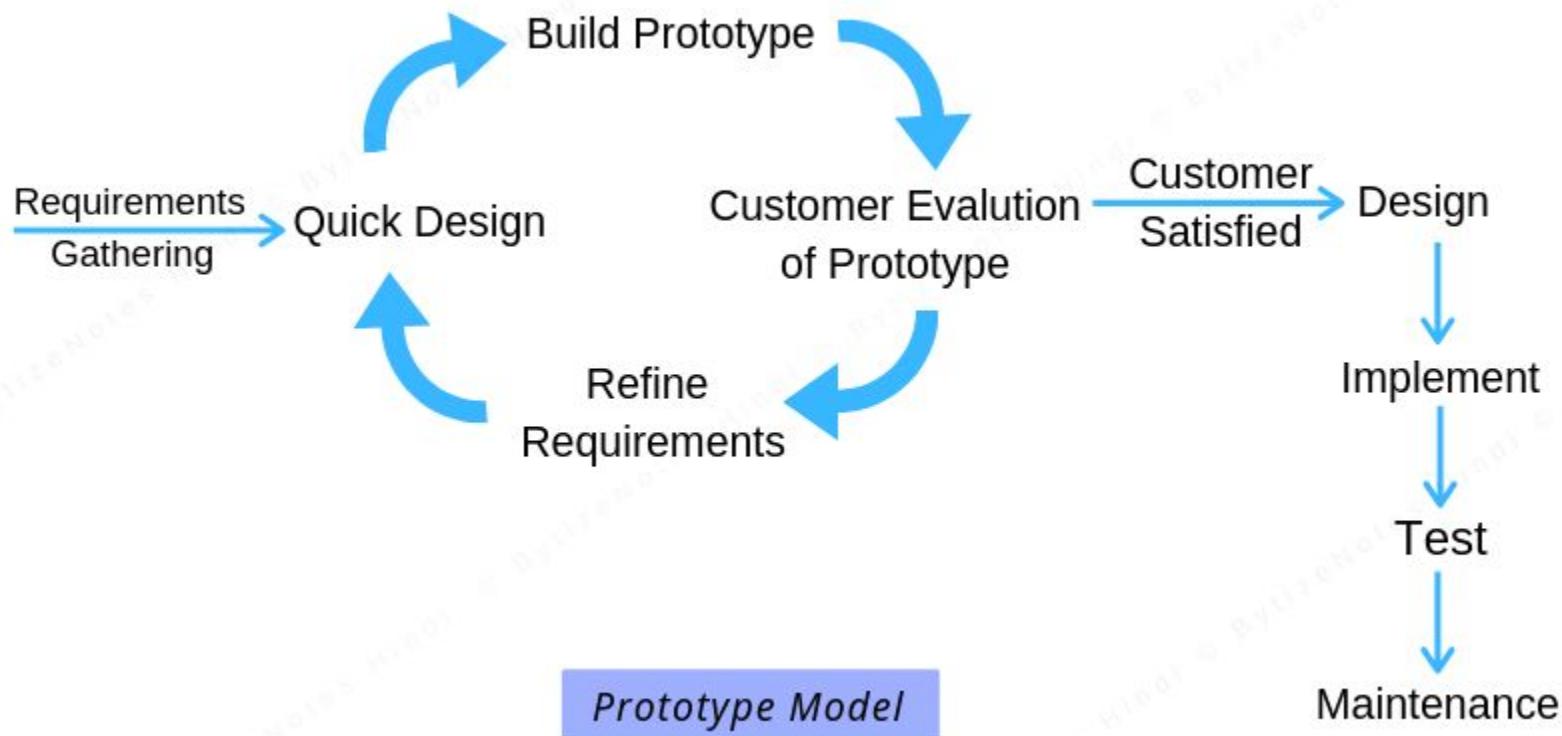
Prototype Model

- 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



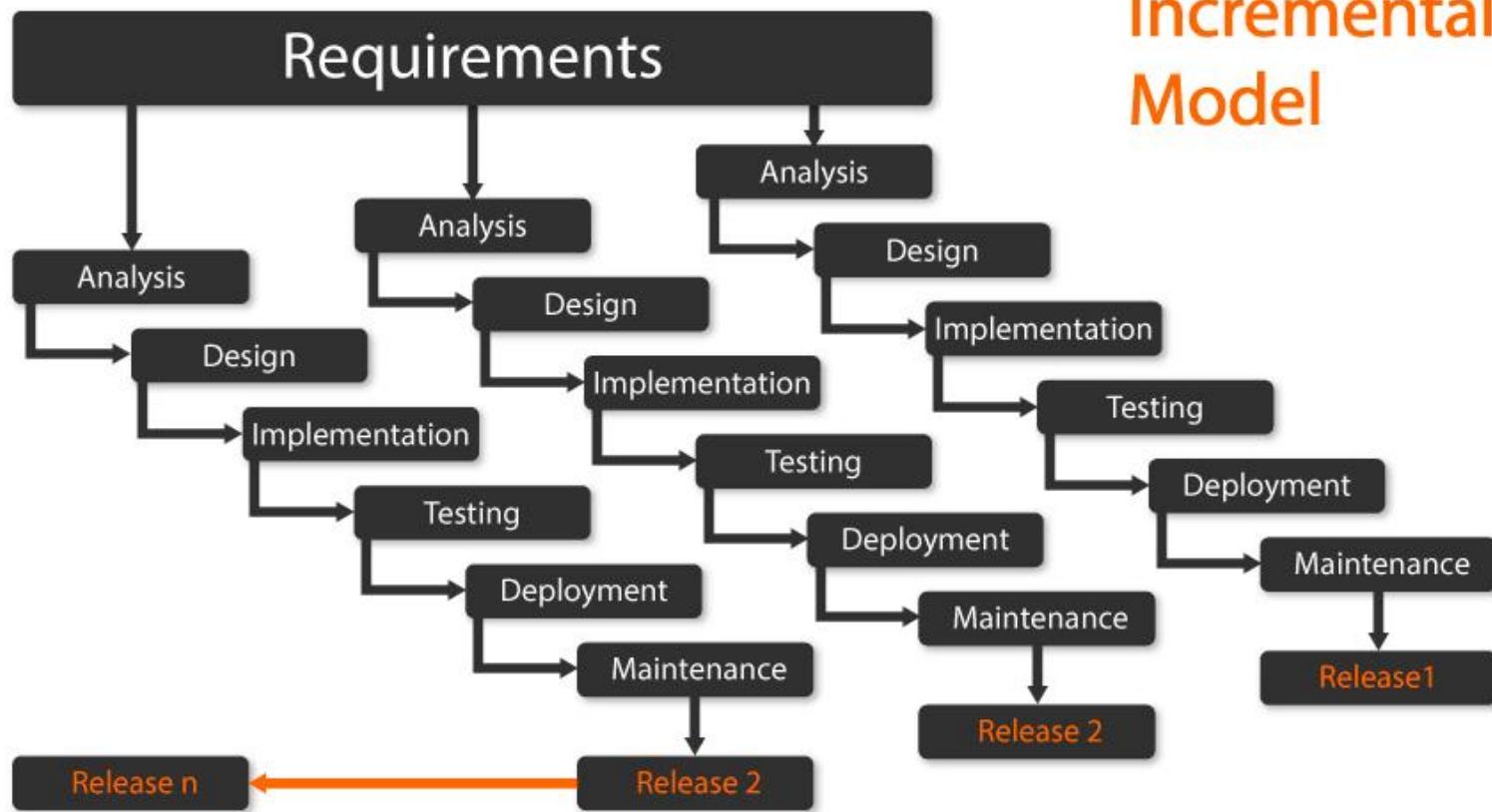
Limitations

- 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

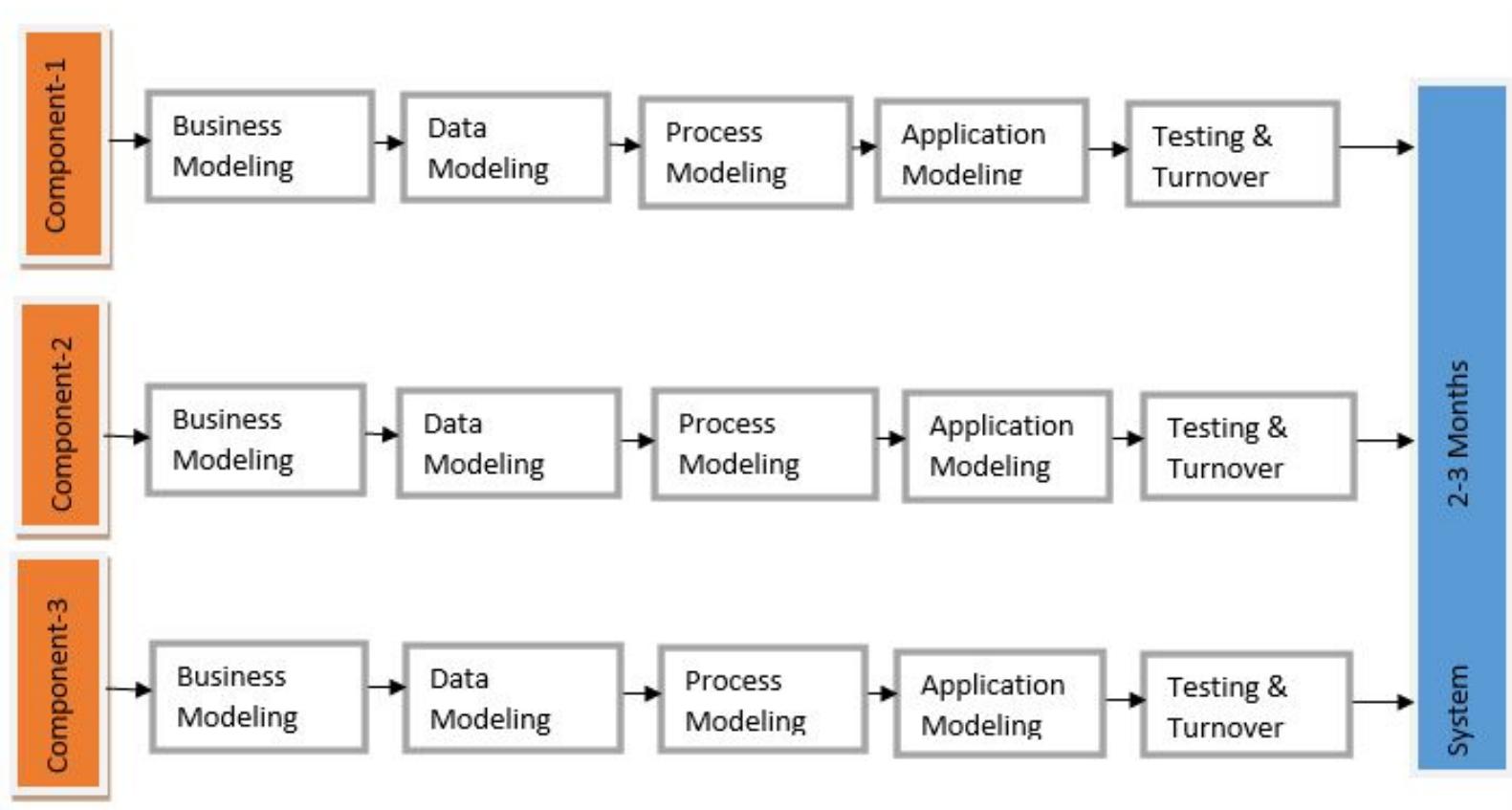
Disadvantages

- 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

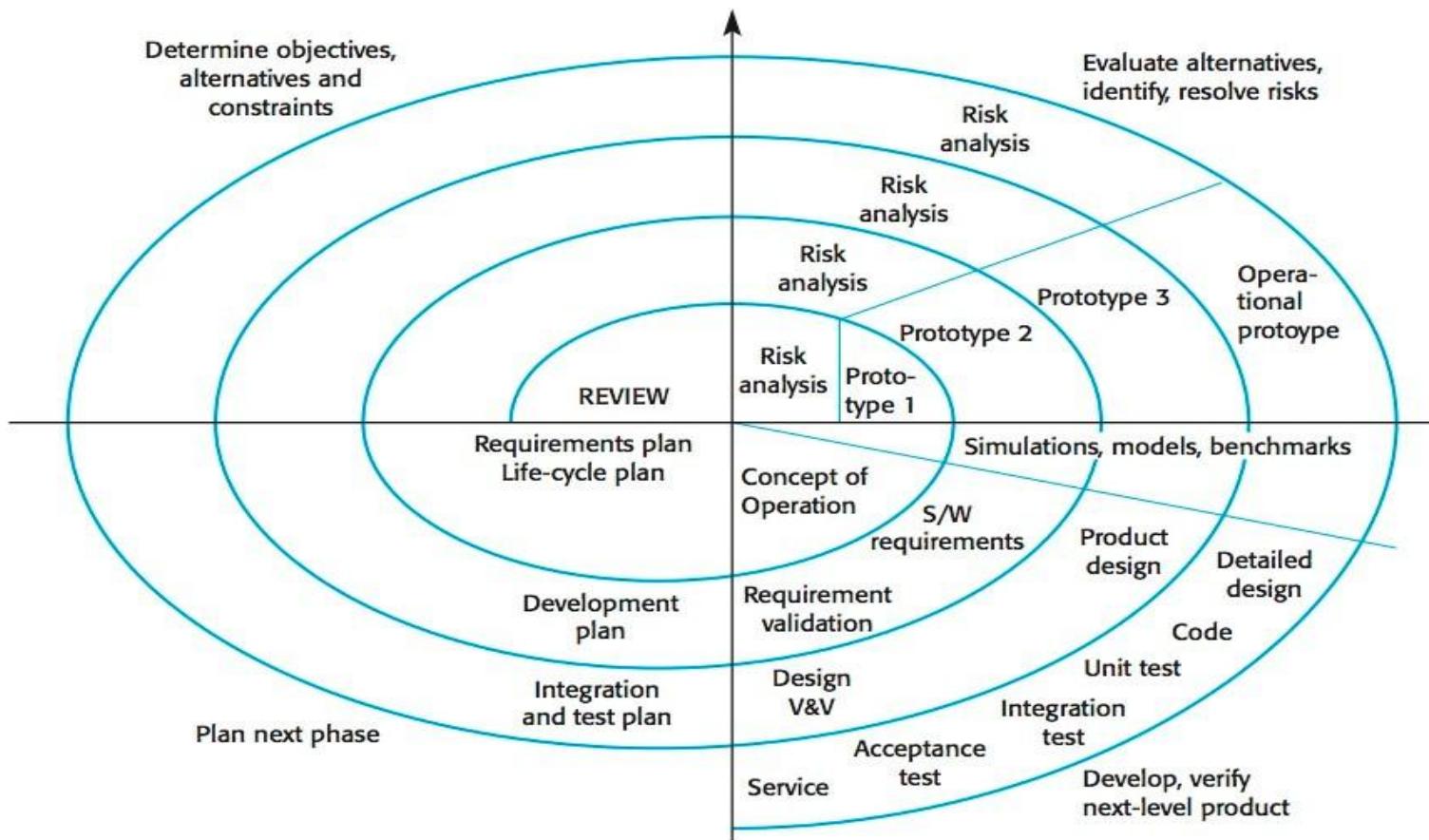
Another Explanation of RAD

- 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

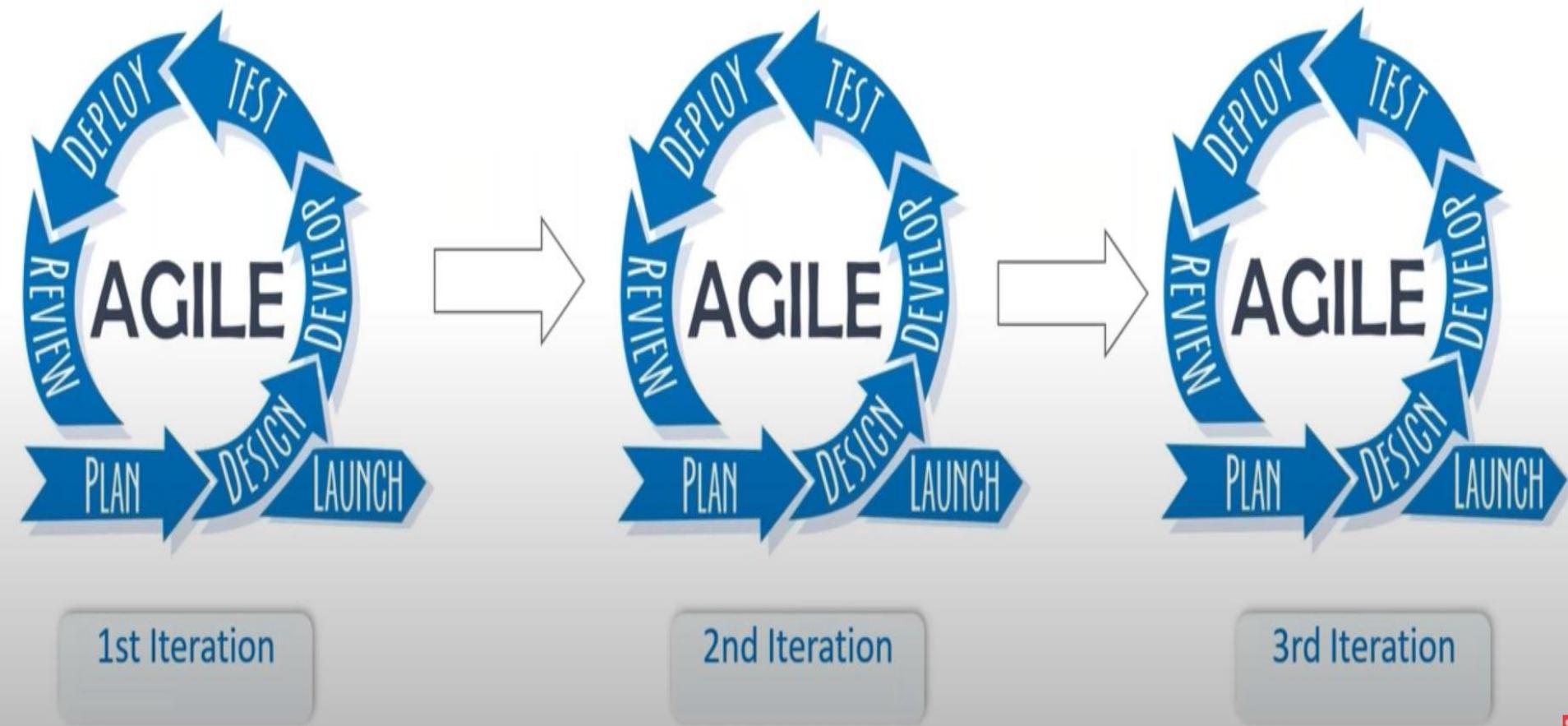
Disadvantage

- 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



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

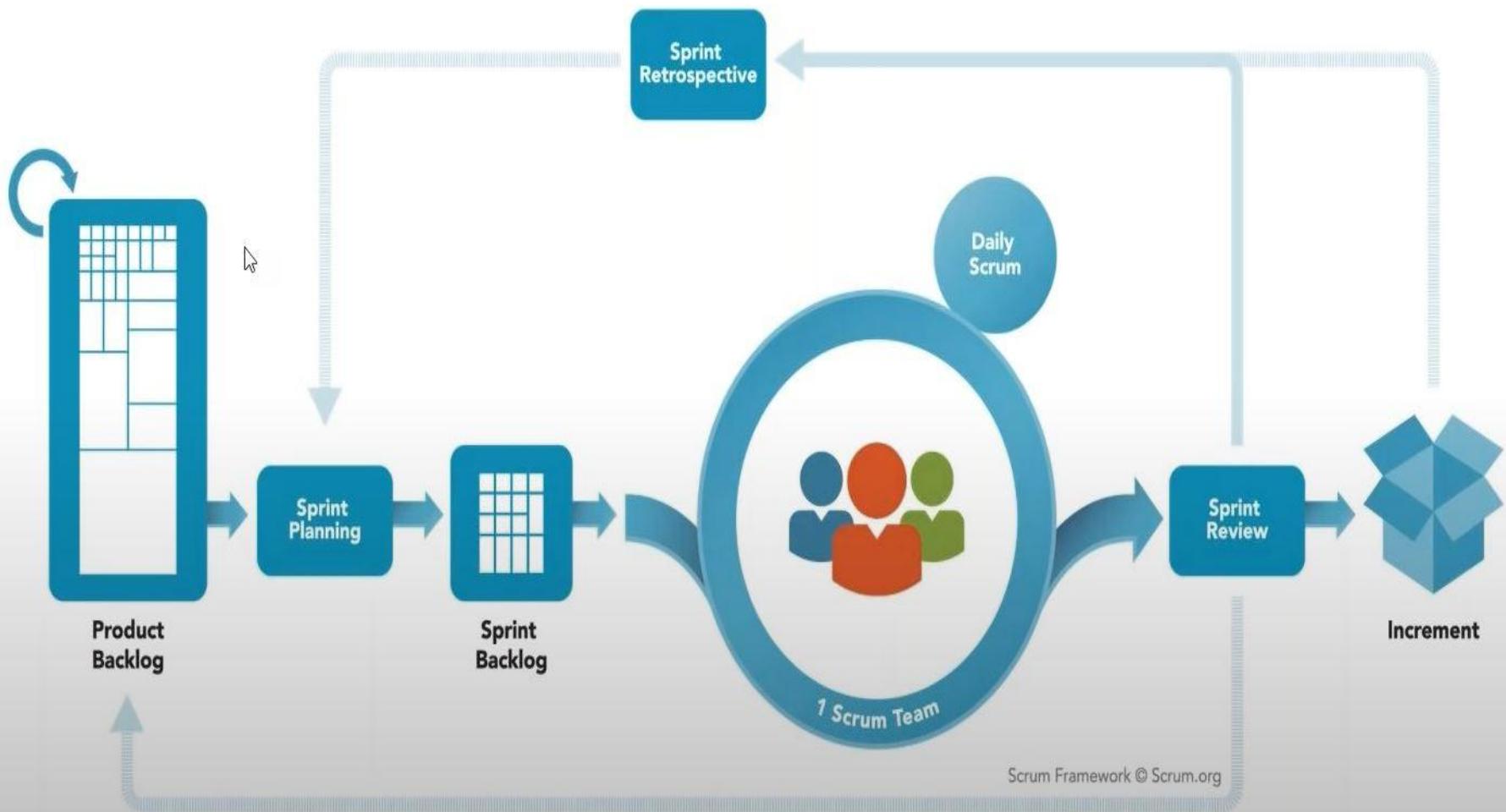
Principle of Agile

- 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

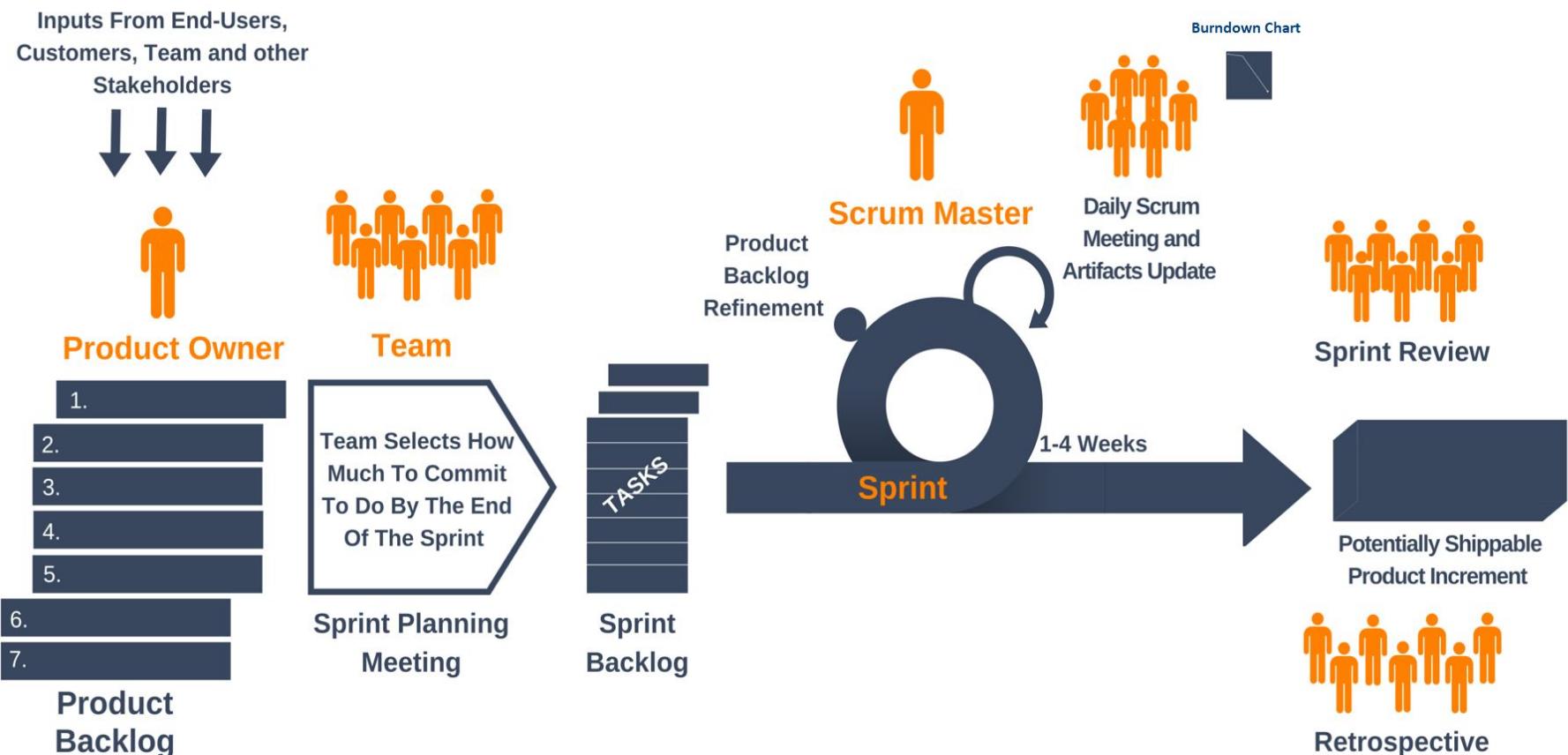
Scrum

- 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

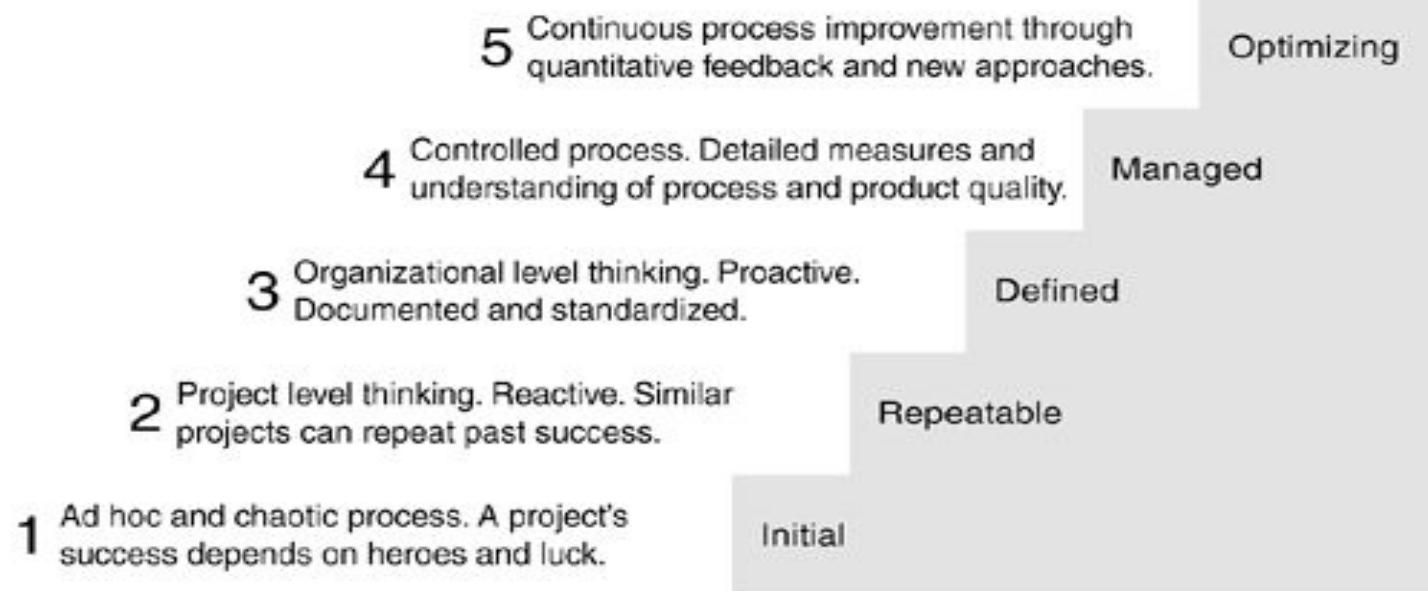
- Capability Maturity Model
- Developed by SEI (Software Engineering Institute)
- CMM is a strategy for improving the software process, irrespective of the actual life cycle model, which is being used
- CMM is used to judge the maturity of software processes of an organization, and to identify key practices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

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

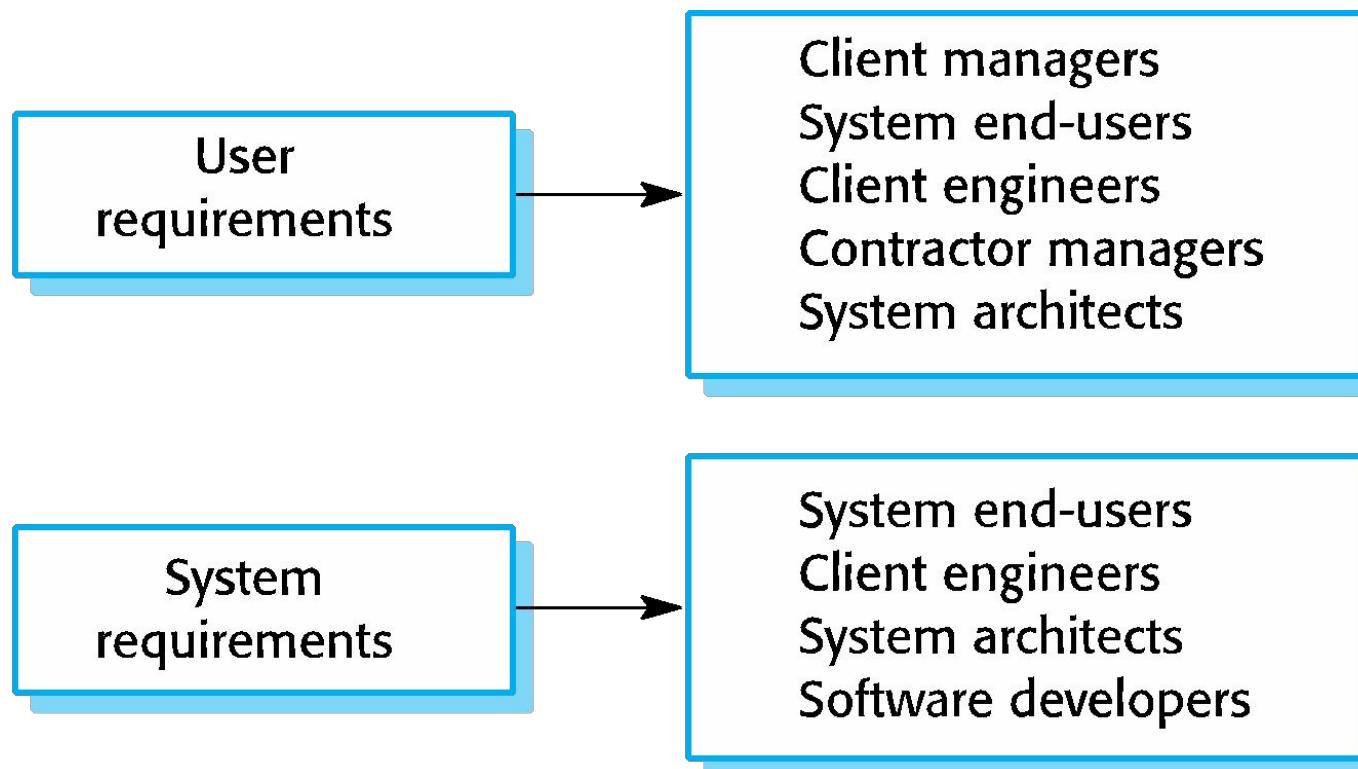
User requirements definition

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.

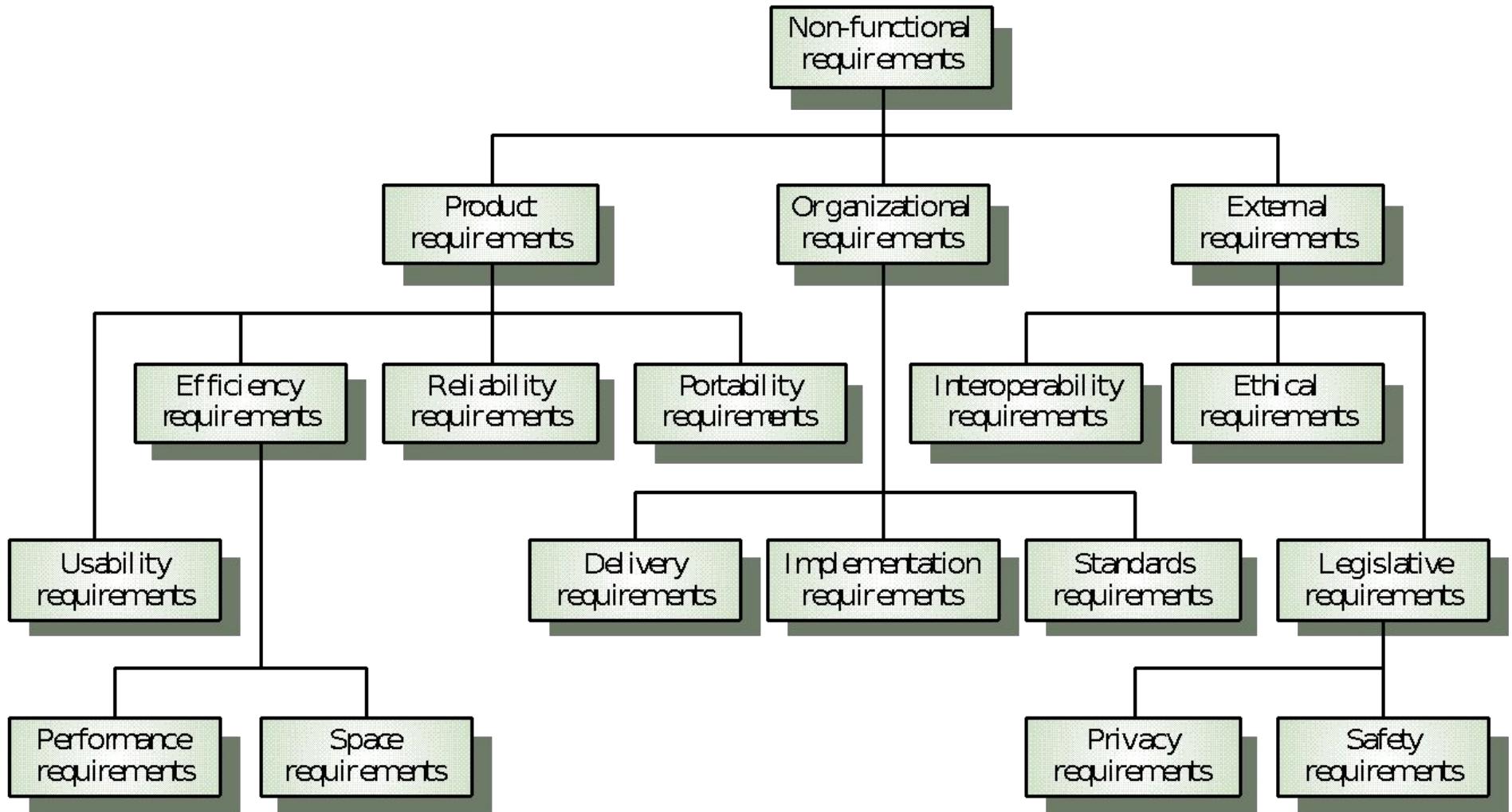
Requirements completeness and consistency

- 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 User/Event response time Screen refresh time
Size	K Bytes Number of RAM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Domain requirements

- 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

Requirements Engineering Process

Requirements Engineering Processes

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

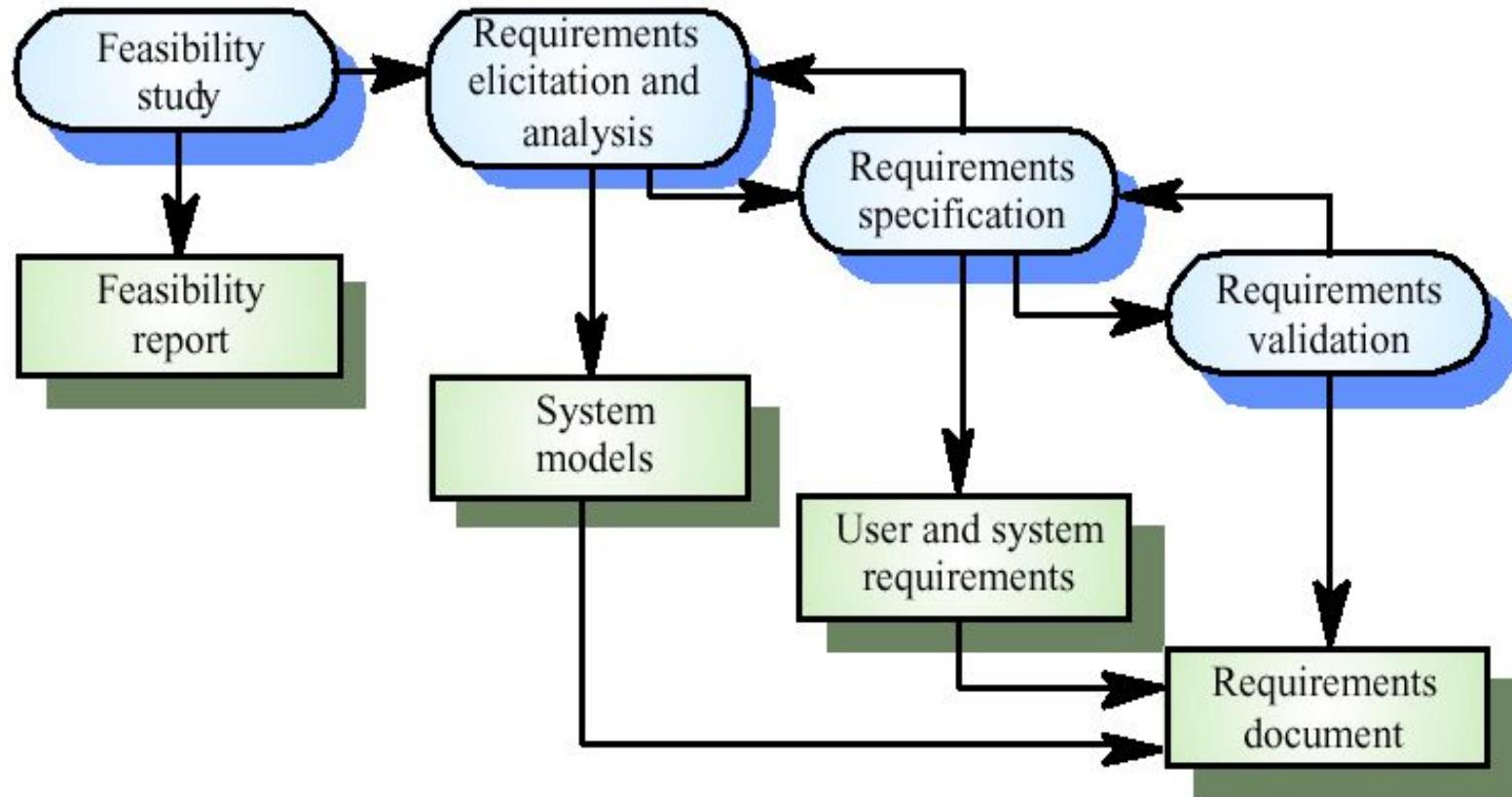
Topics covered

- 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?

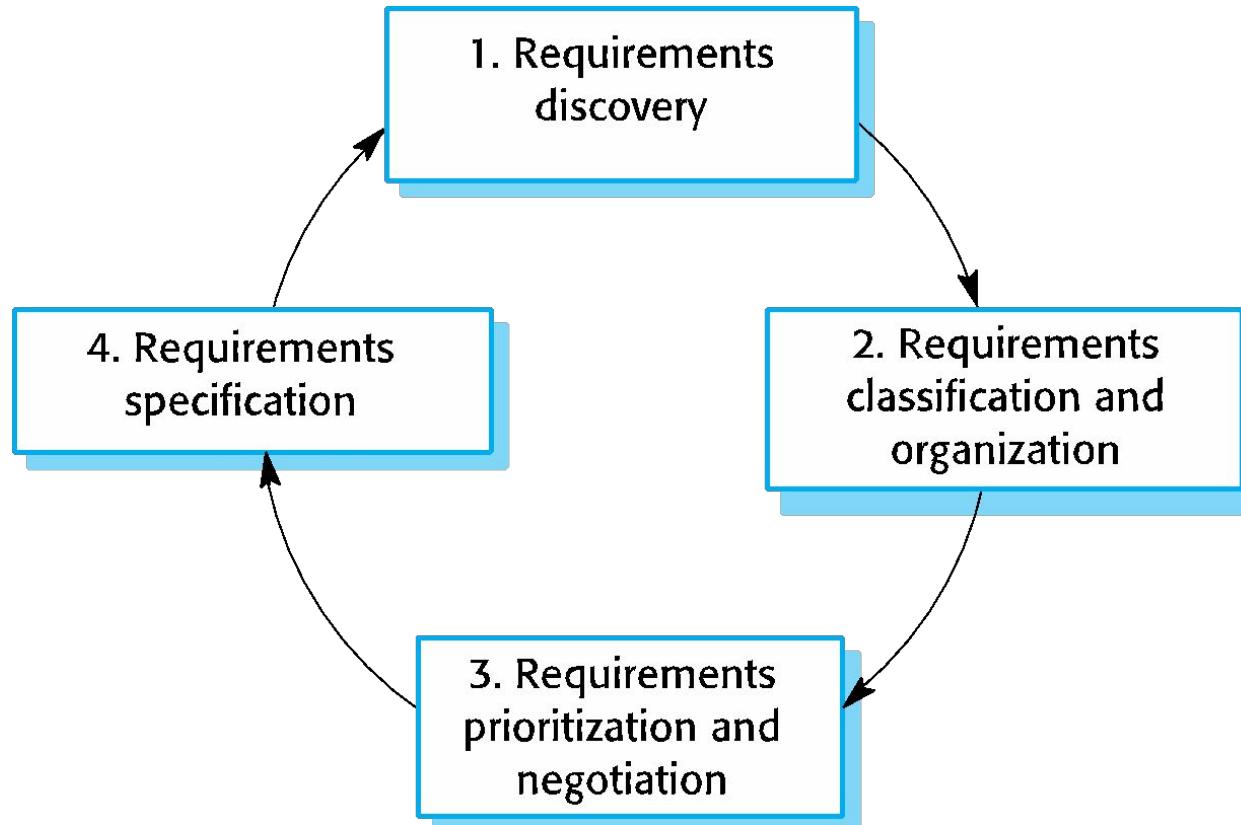
Elicitation and analysis

- 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.

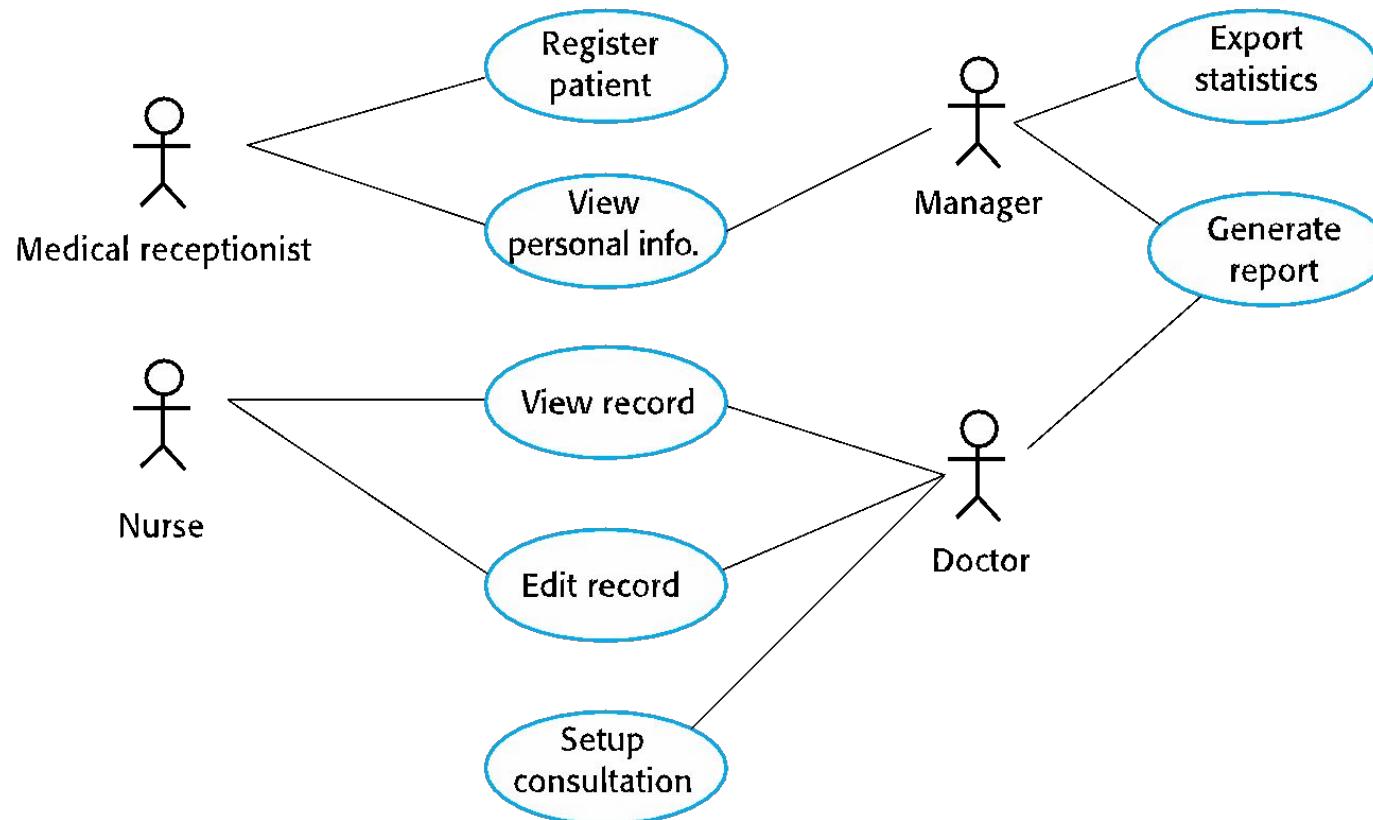
Example

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

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

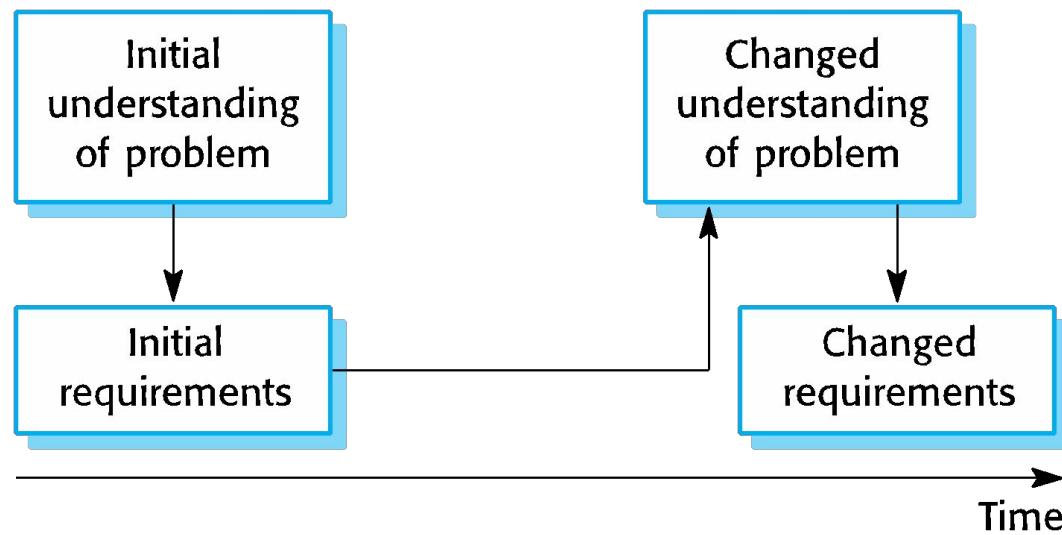
- Developing tests for requirements to check testability.

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.

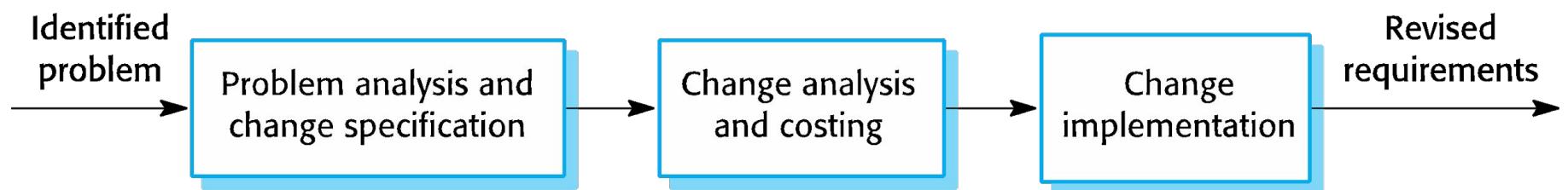
Requirements evolution



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.

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 3rd?

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

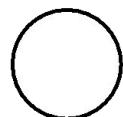
- Types of specifications
 - operational
 - Data Flow Diagrams
 - (Some) UML diagrams
 - Finite State Machines
 - Petri Nets
 - descriptive
 - Entity Relationship Diagrams
 - Logic-based notations
 - Algebraic notations

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



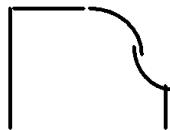
The data flow symbol



The data store symbol

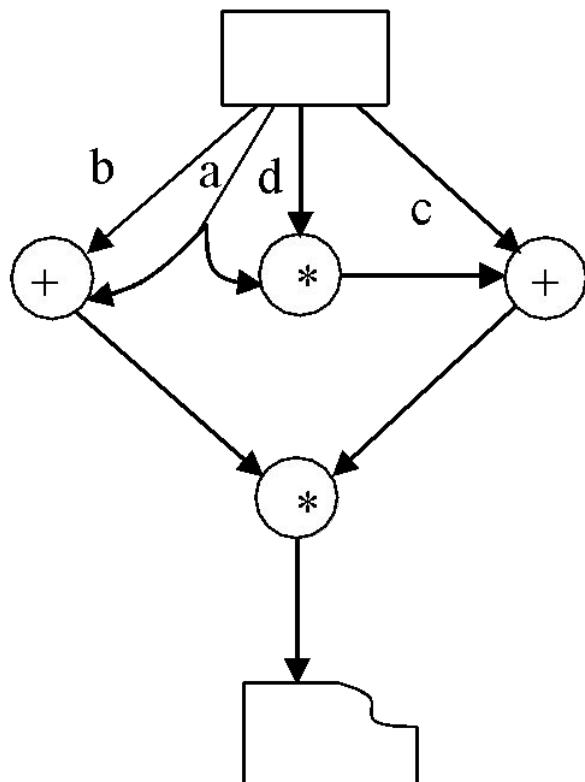


The input device symbol



The output device symbol

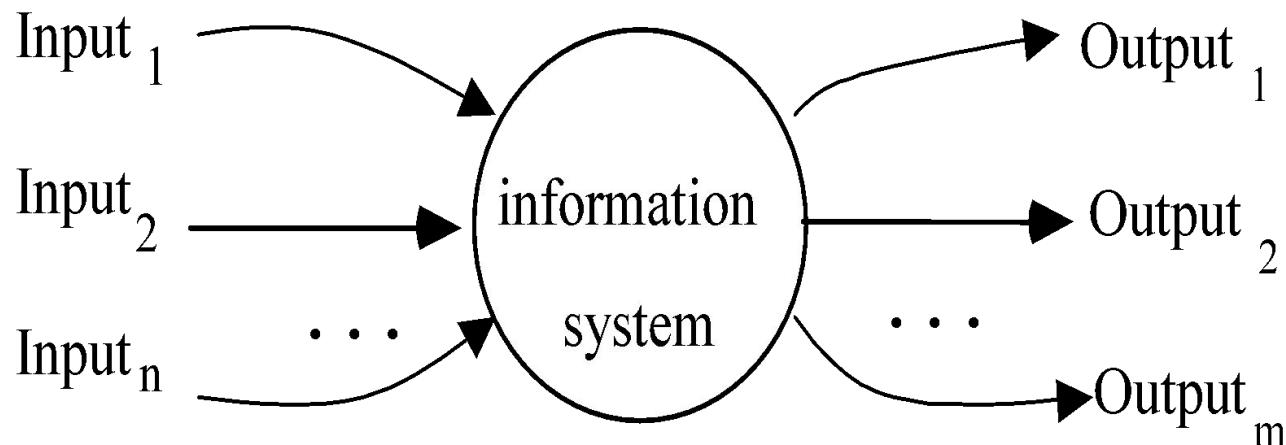
Example



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

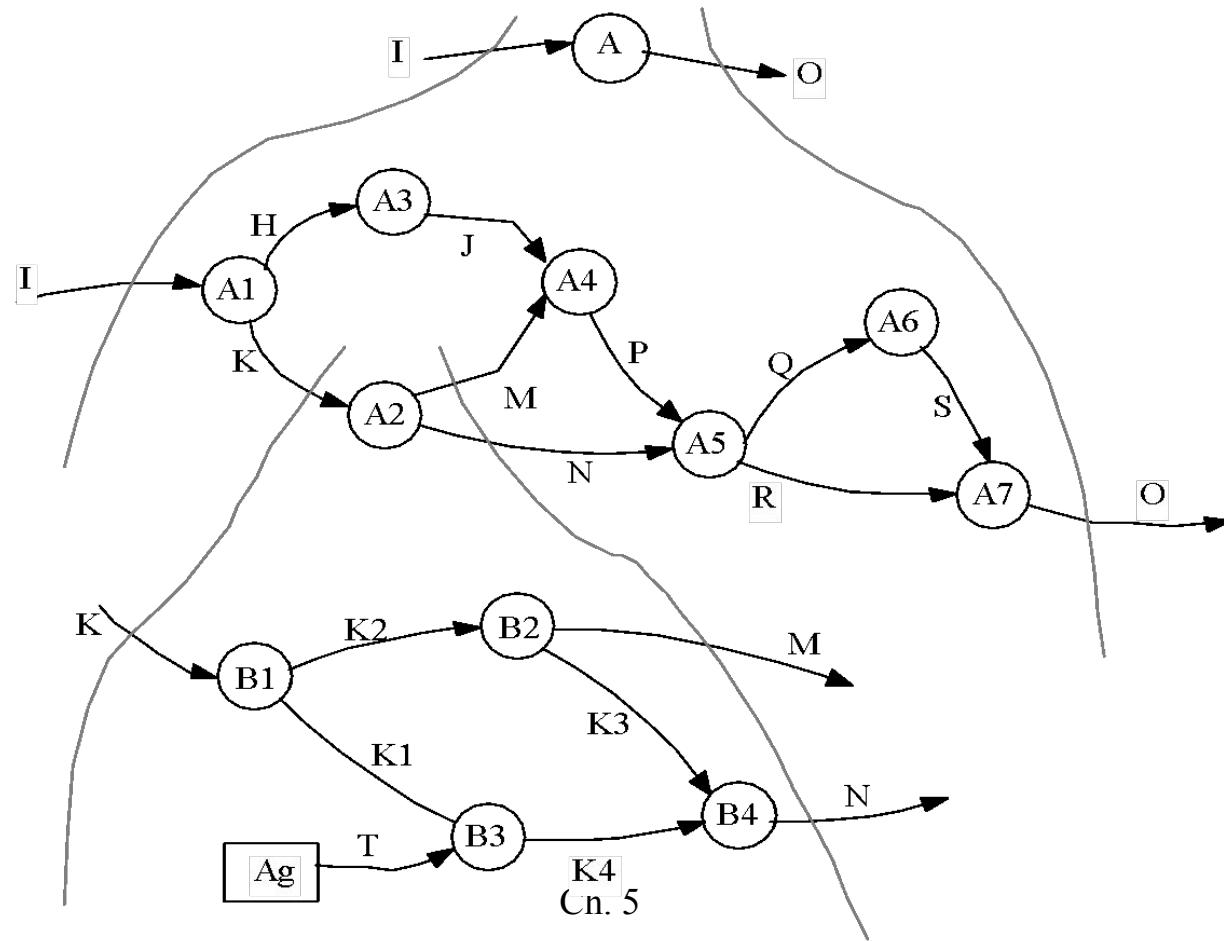
A construction “method” (1)

1. Start from the “context” diagram

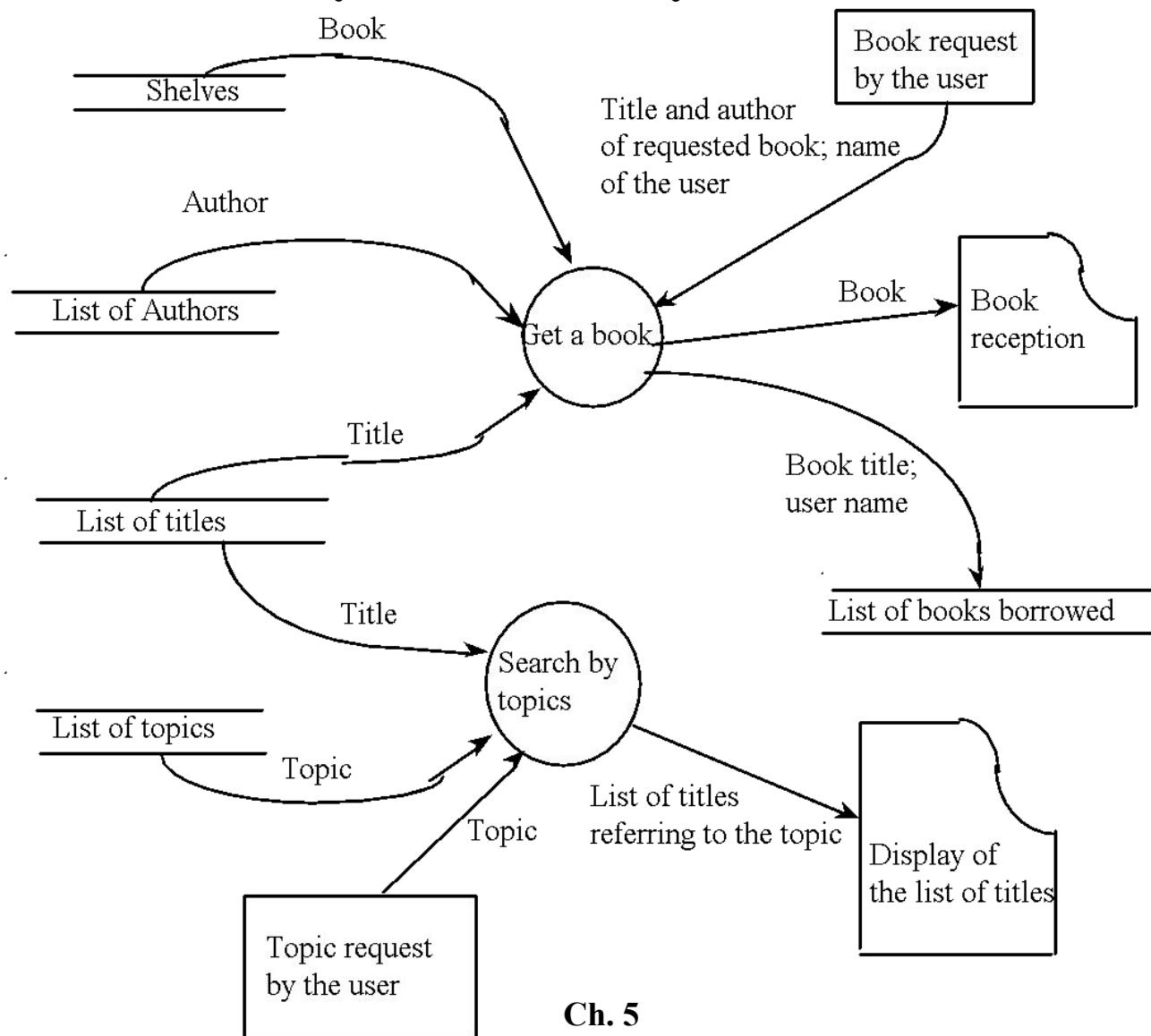


A construction "method" (2)

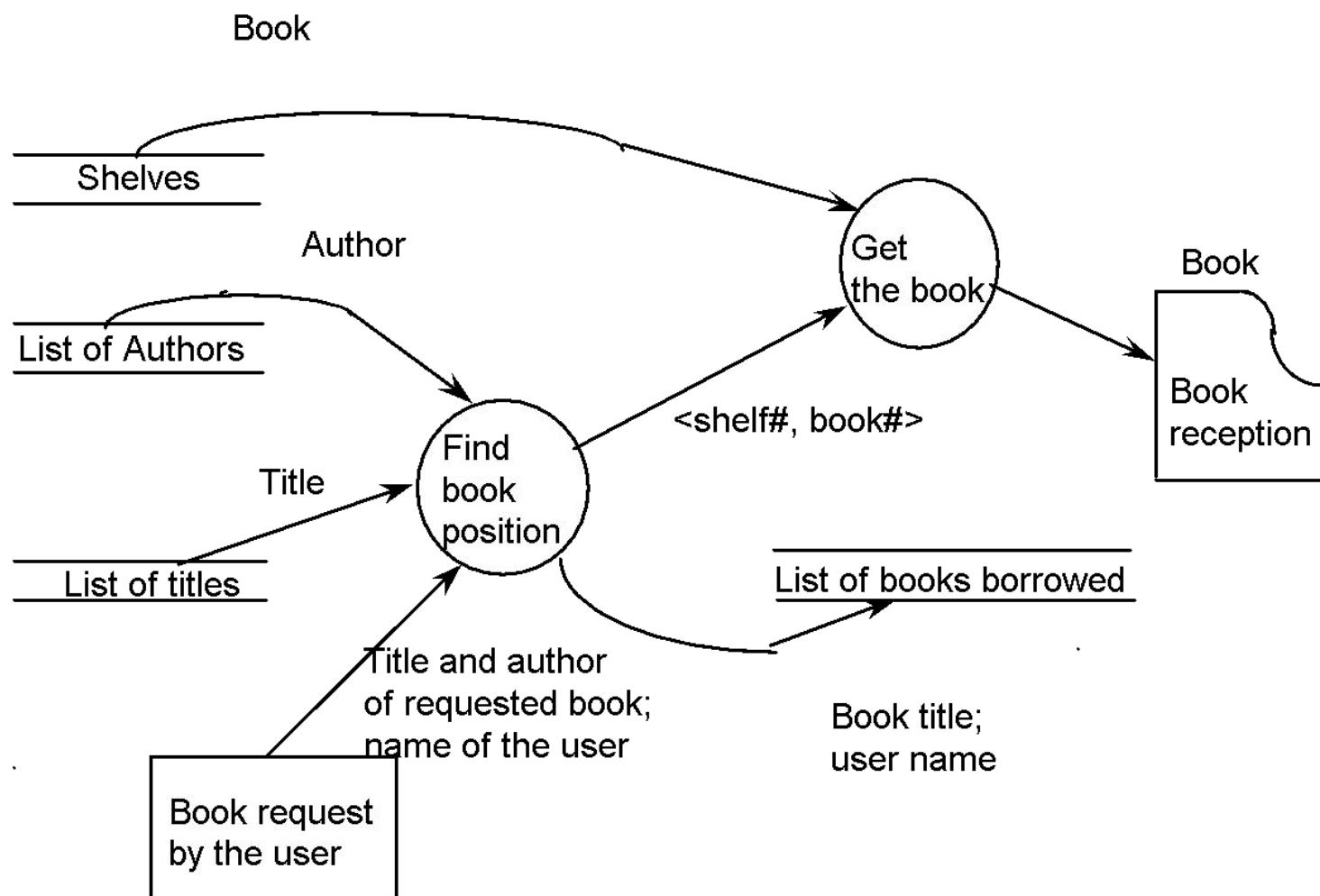
2. Proceed by refinements until you reach "elementary" functions



A library example (Level 0)

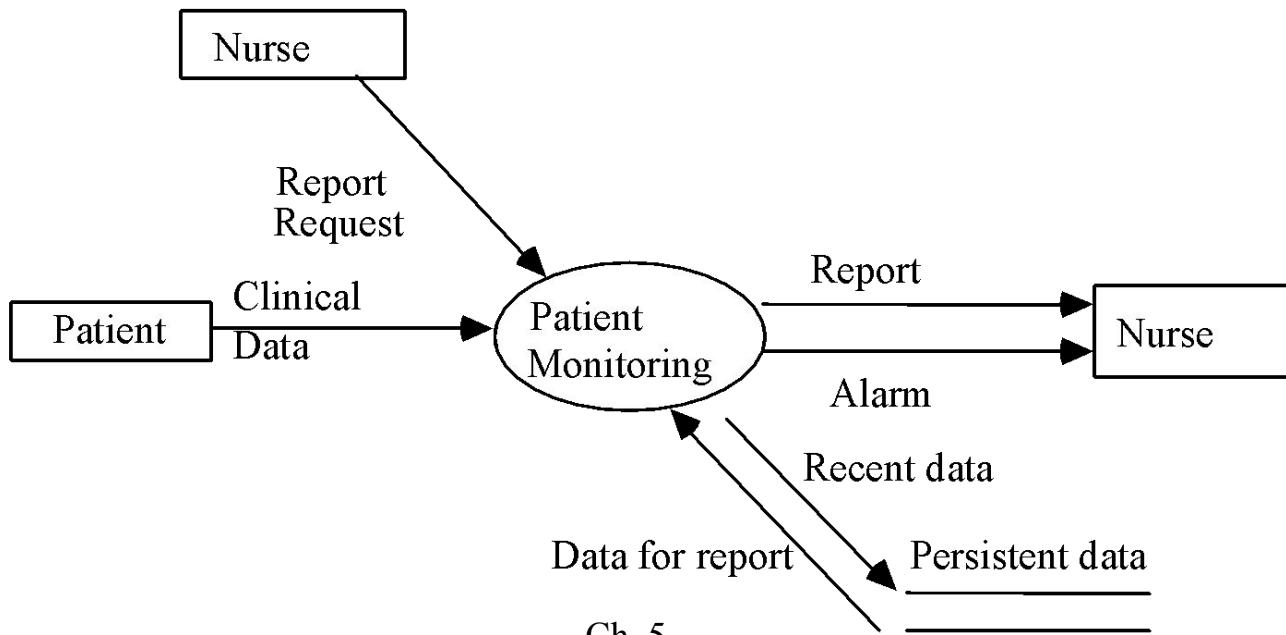


Refinement of "Get a book" (Level 1)

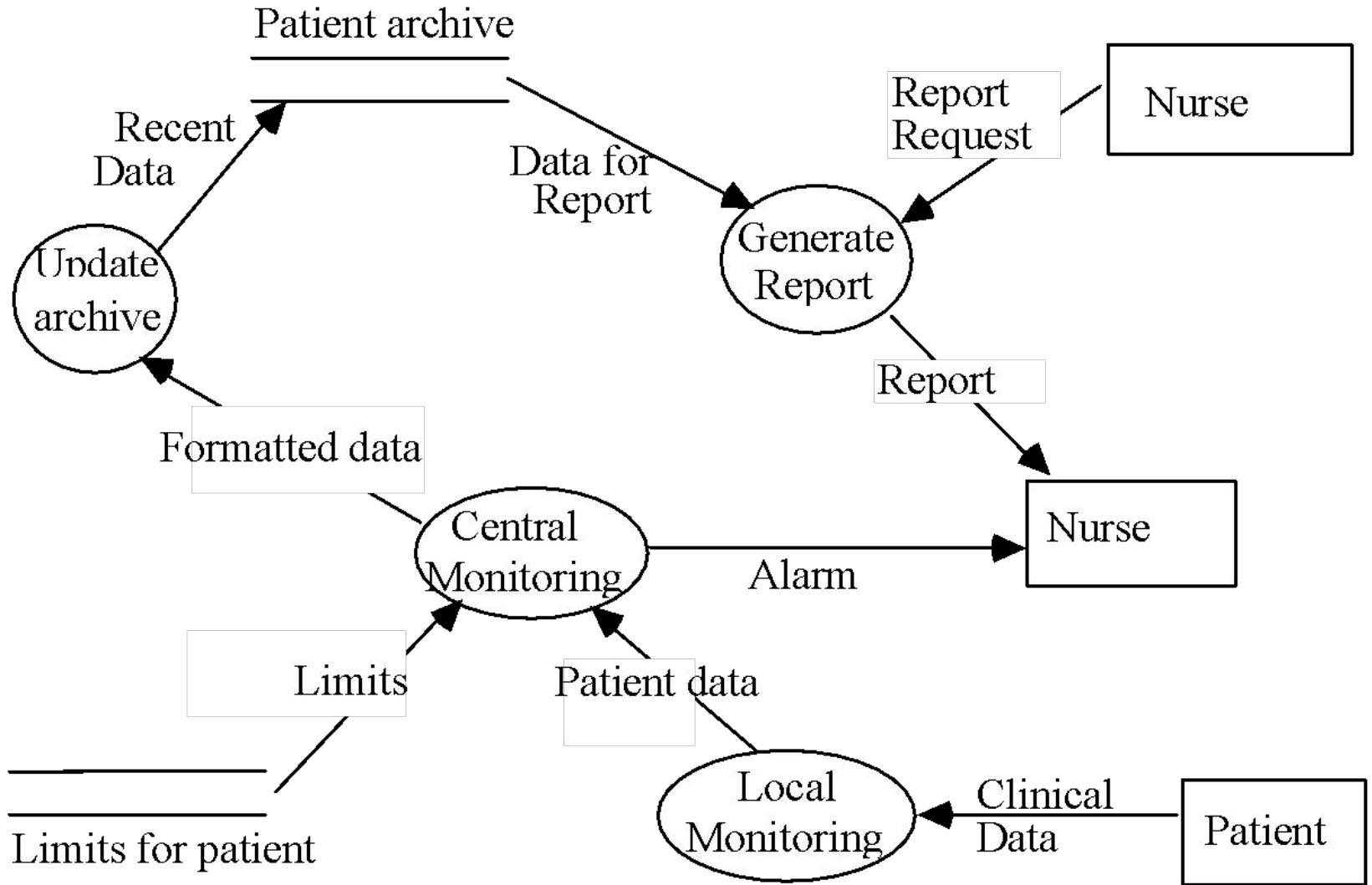


Patient monitoring systems

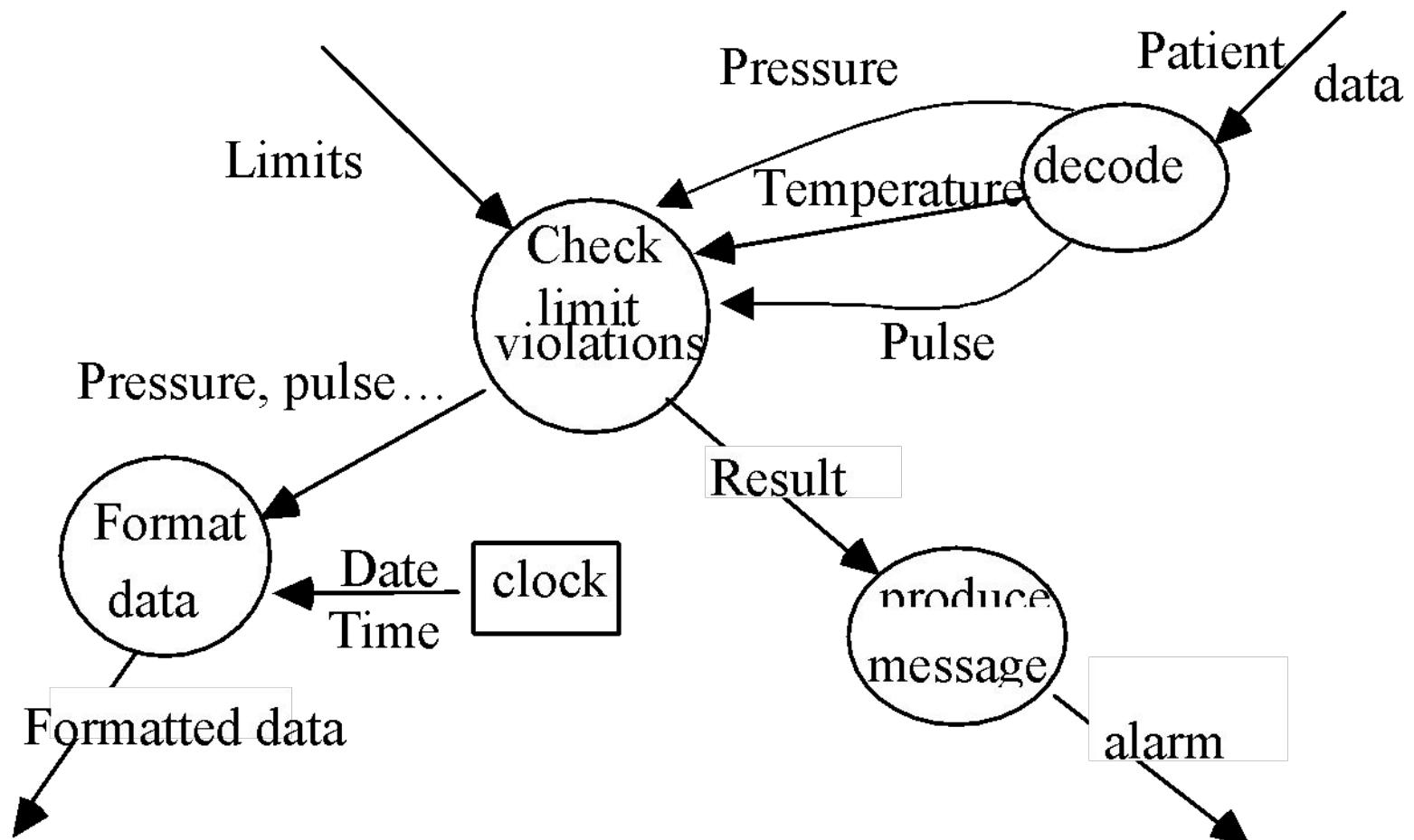
The purpose is to monitor the patients' vital factors--blood, pressure, temperature, ...--reading them at specified frequencies from analog devices and storing readings in a DB. If readings fall outside the range specified for patient or device fails an alarm must be sent to a nurse. The system also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



A refinement



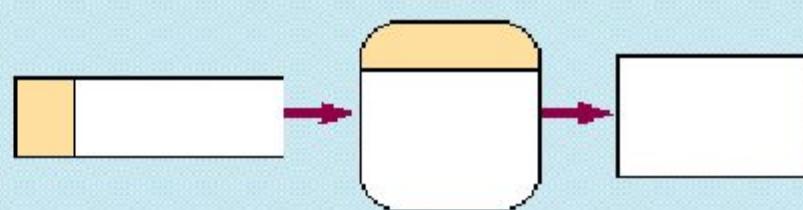
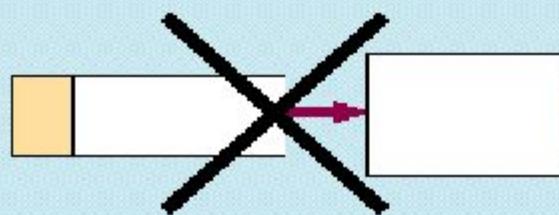
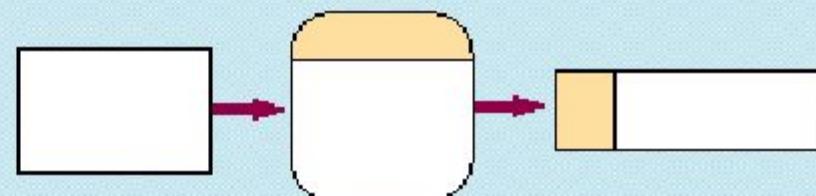
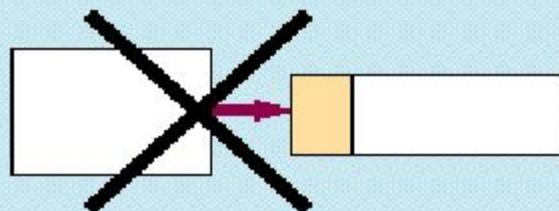
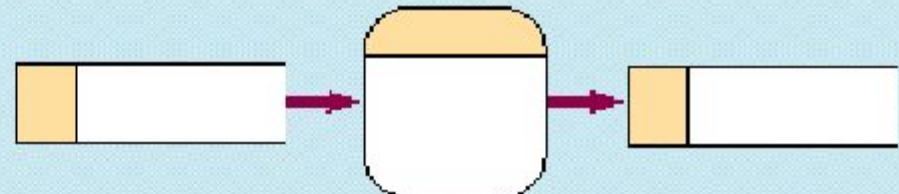
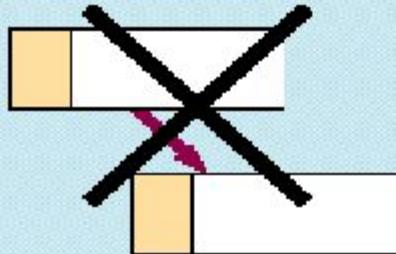
More refinement



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

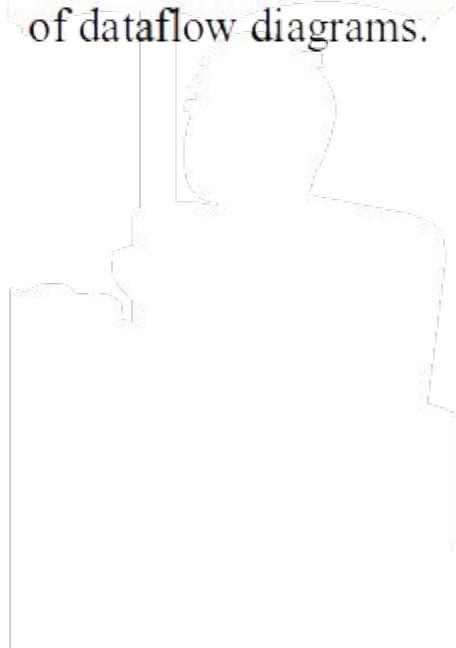
Basic rules



Creating DFDs (Lemonade Stand Example)

Example

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



Steps:

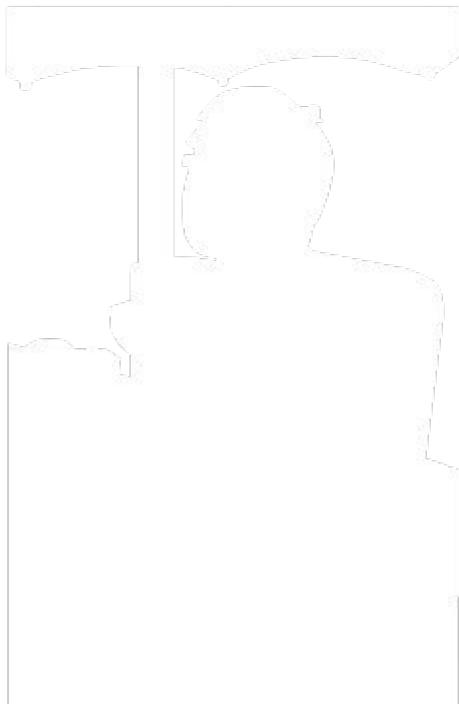
1. Create a list of activities
2. Construct Context Level DFD
(identifies sources and sink)
3. Construct Level 0 DFD
(identifies manageable sub processes)
4. Construct Level 1- n DFD
(identifies actual data flows and data stores)

<number>

Creating DFDs

Example

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



1. Create a list of activities

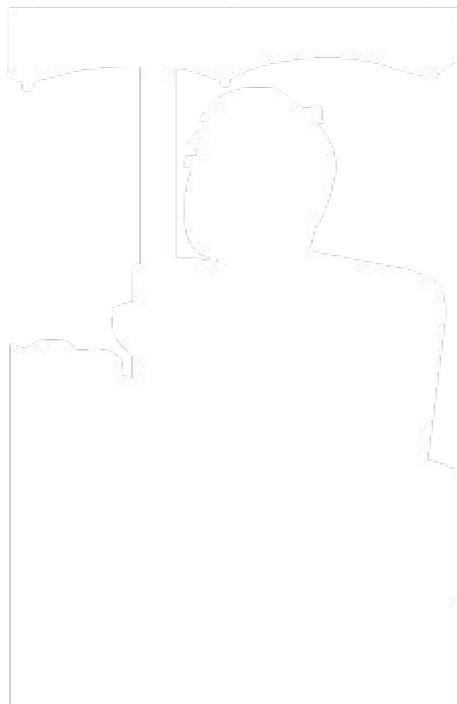
- Customer Order
- Serve Product
- Collect Payment
- Produce Product
- Store Product

<number>

Creating DFDs

Example

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



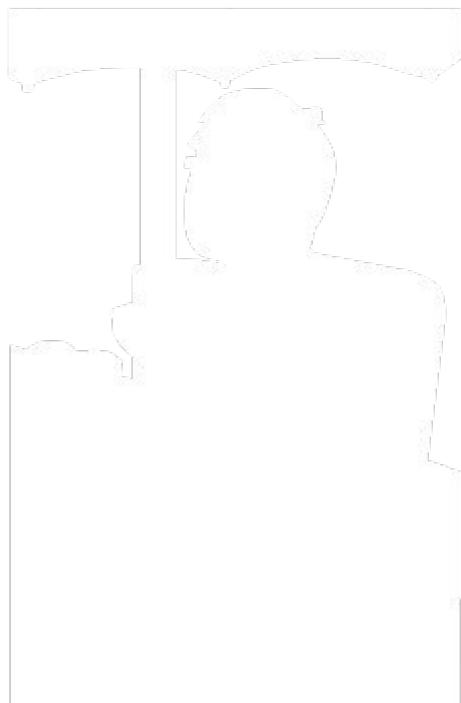
1. Create a list of activities

- 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

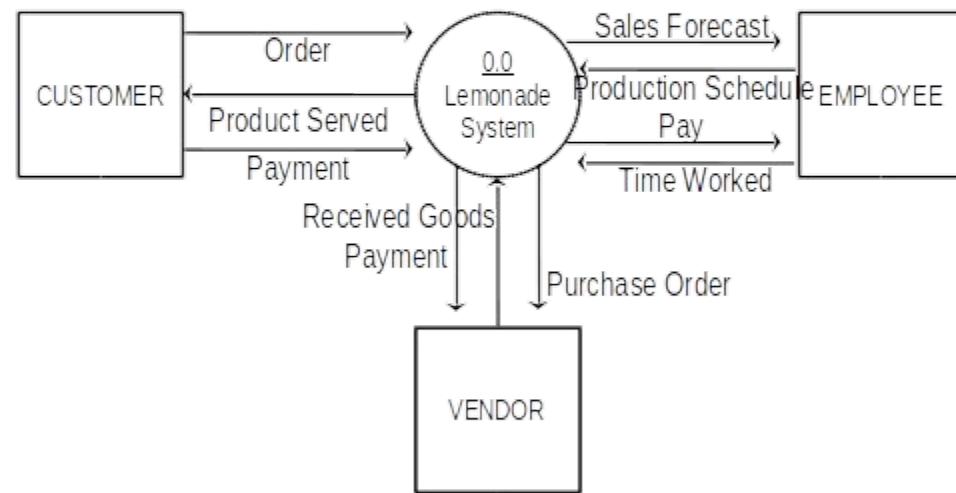
Example

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

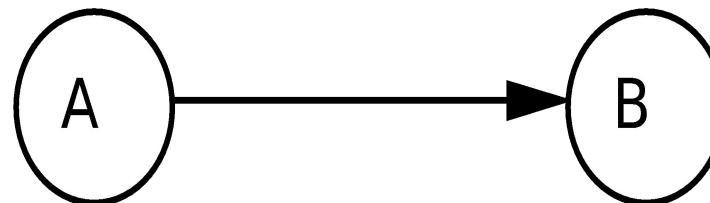
<number>

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

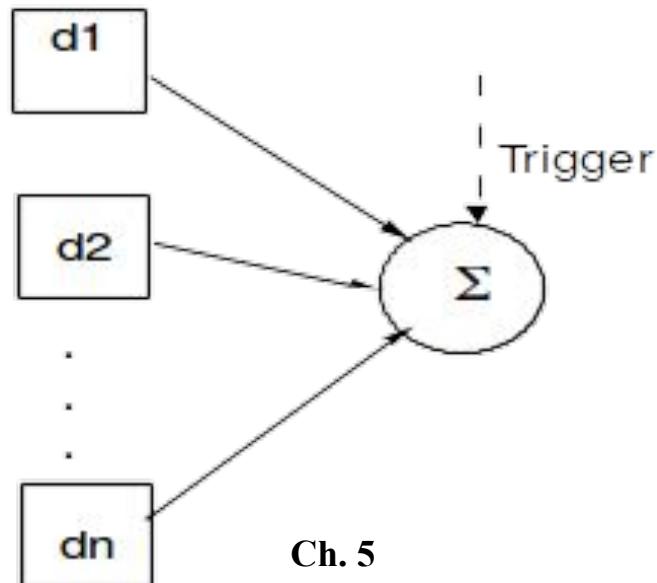


Possible interpretations:

- (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

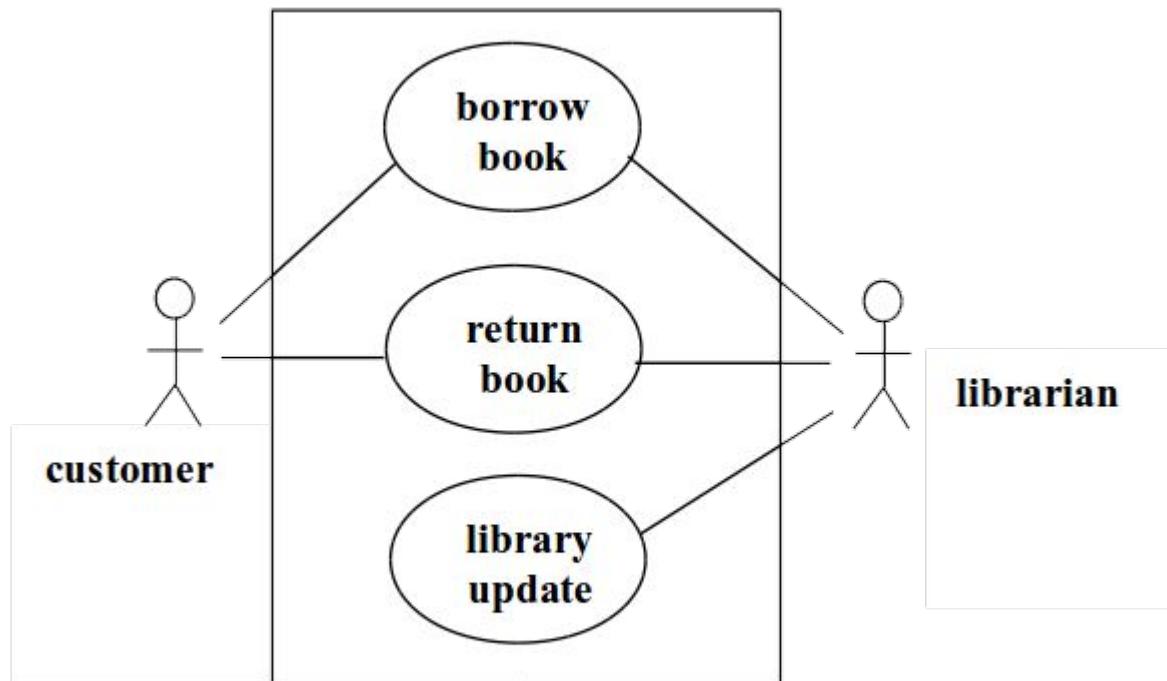
Solutions

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



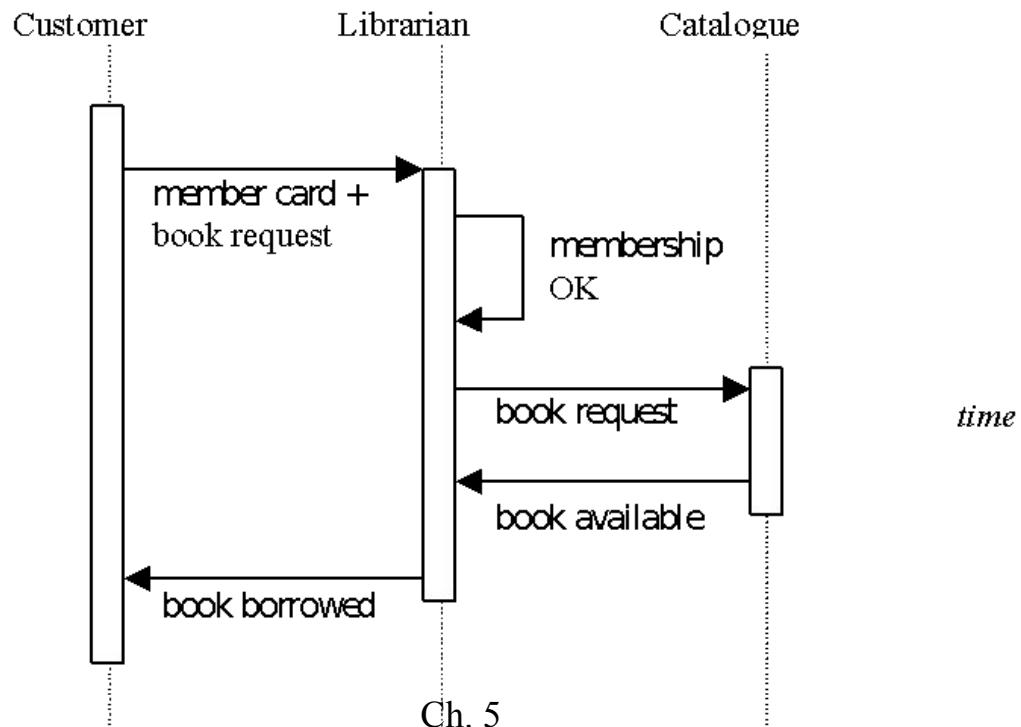
UML use-case diagrams

- Define functions on basis of actors and actions



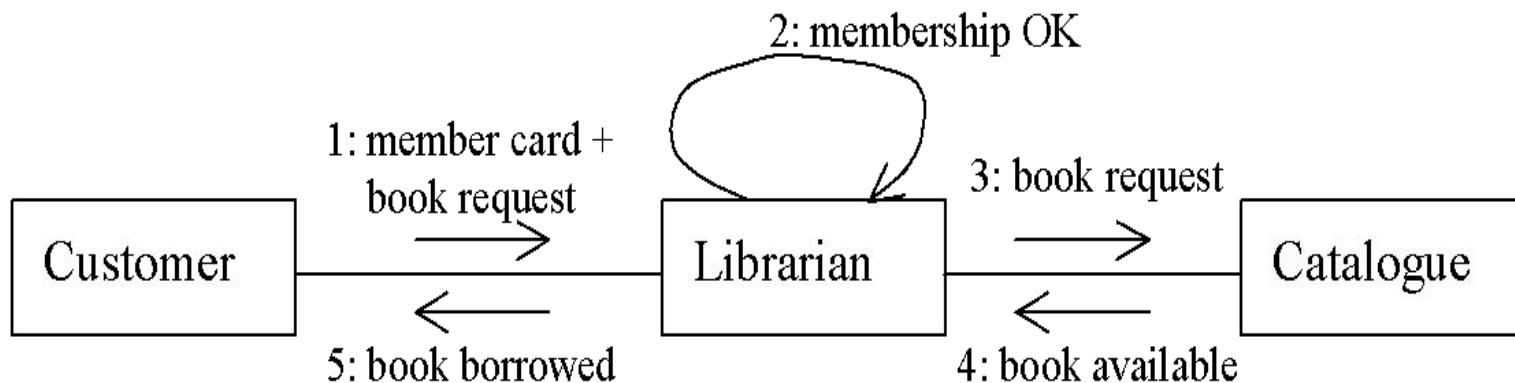
UML sequence diagrams

- 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)

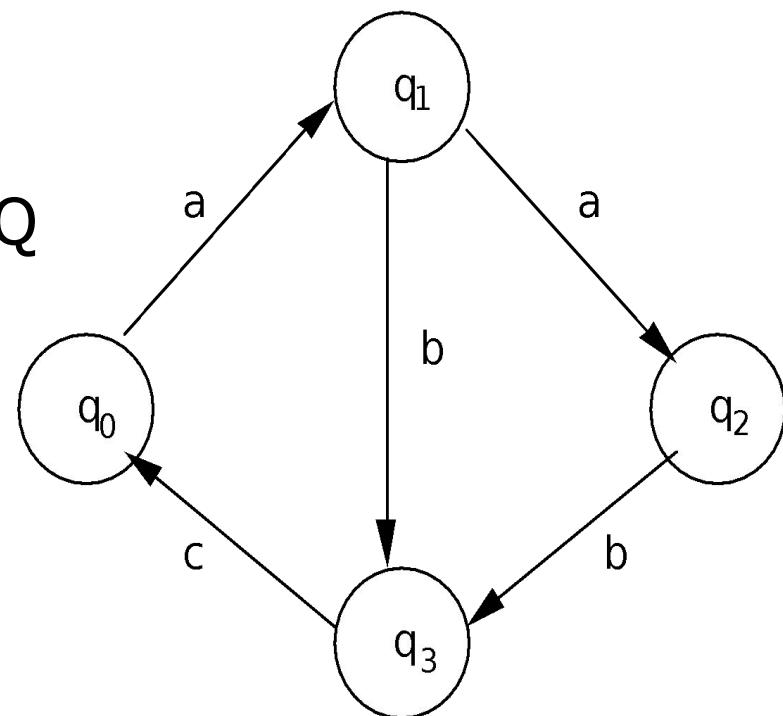
- Can specify control flow aspects
- Defined as

a finite set of states, Q ;

a finite set of inputs, I ;

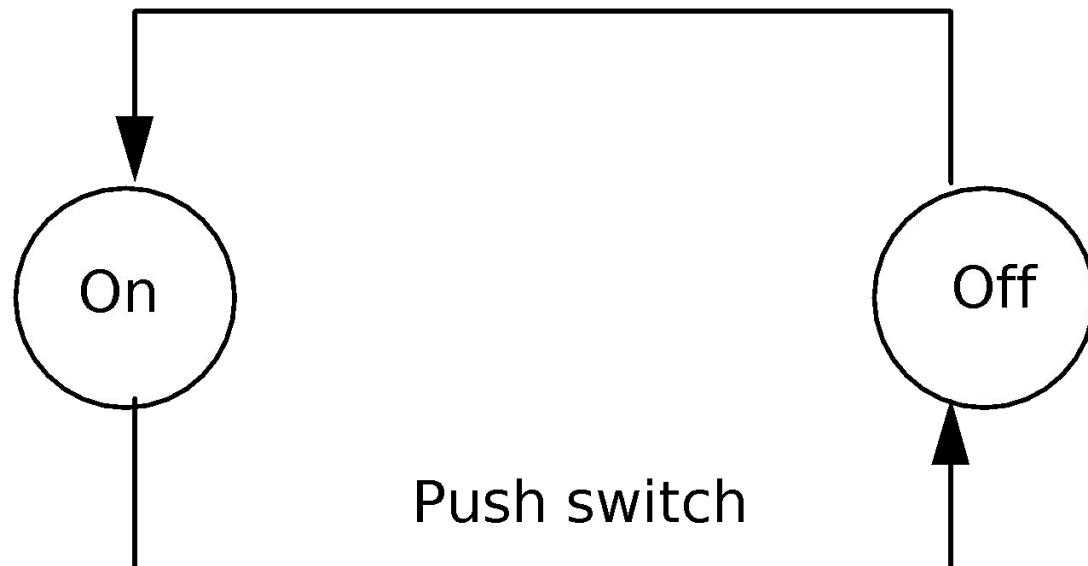
a transition function $d : Q \times I \rightarrow Q$

(d can be a partial function)

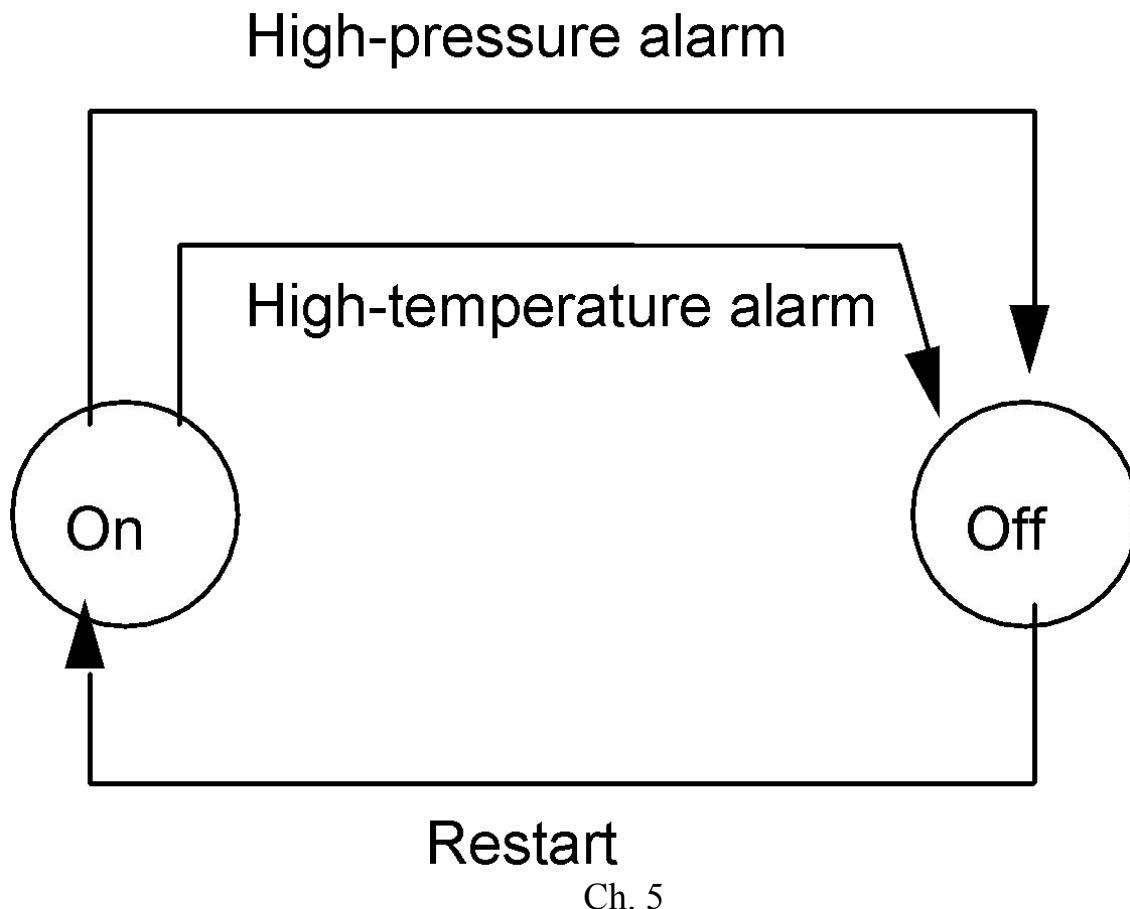


Example: a lamp

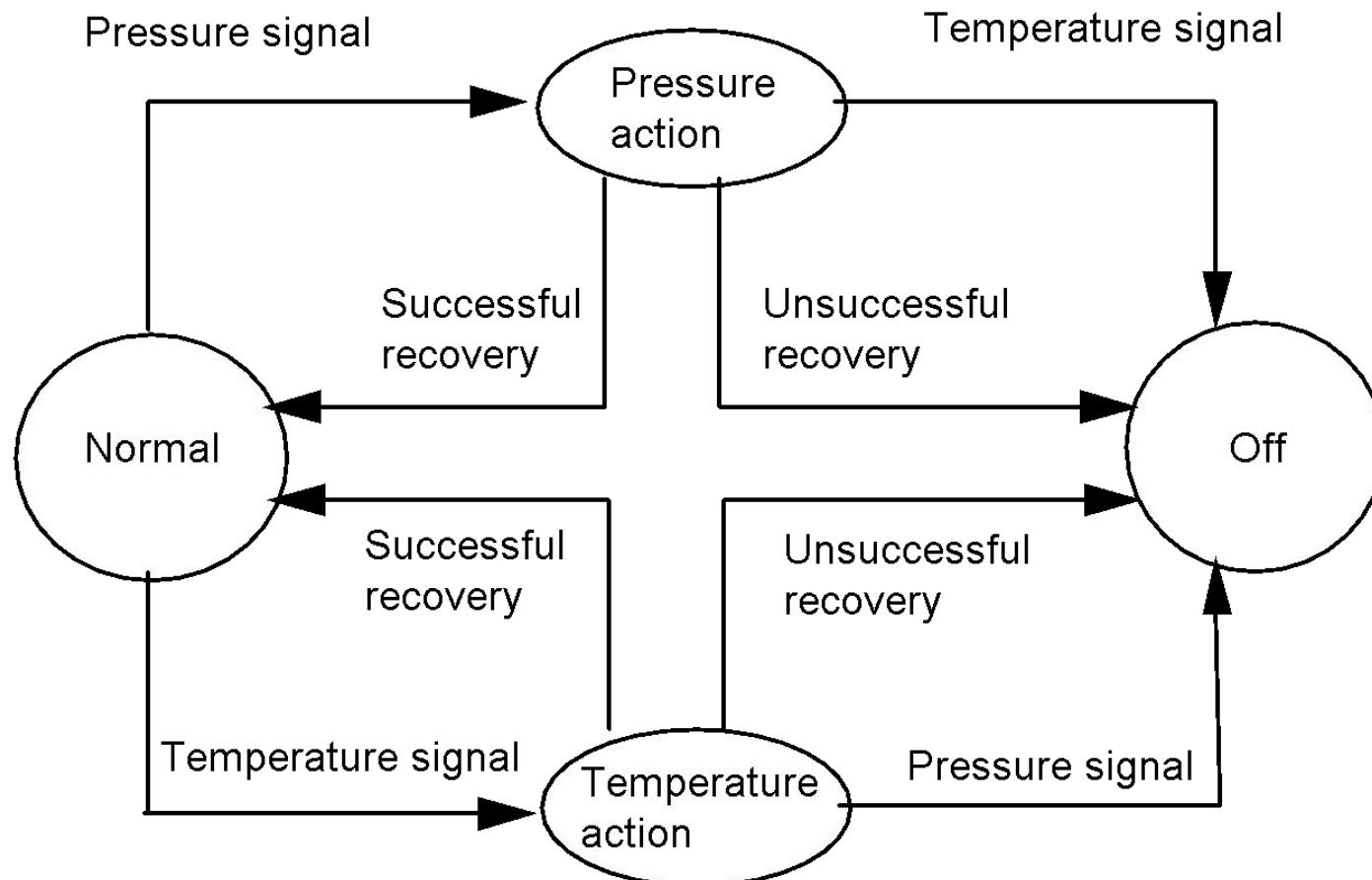
Push switch



Another example: a plant control system



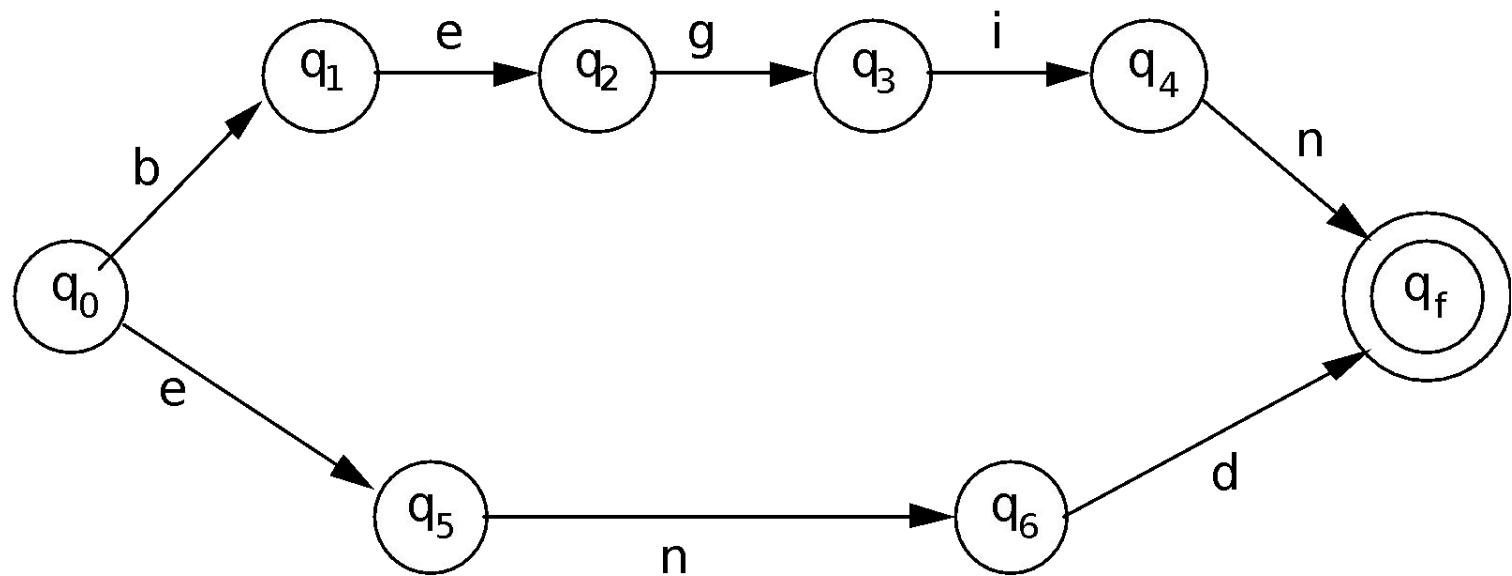
A refinement



Classes of FSMs

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

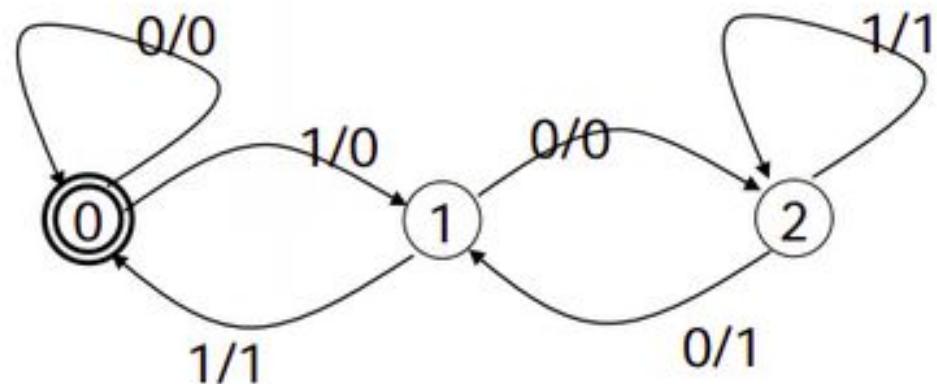
FSMs as recognizers



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) $P \cap T = \emptyset$

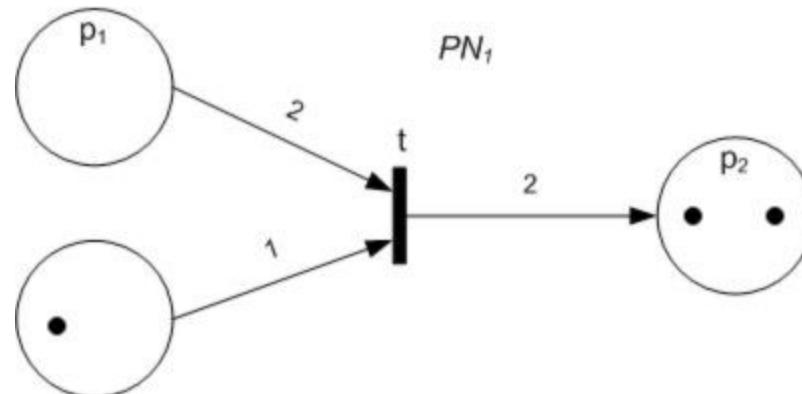
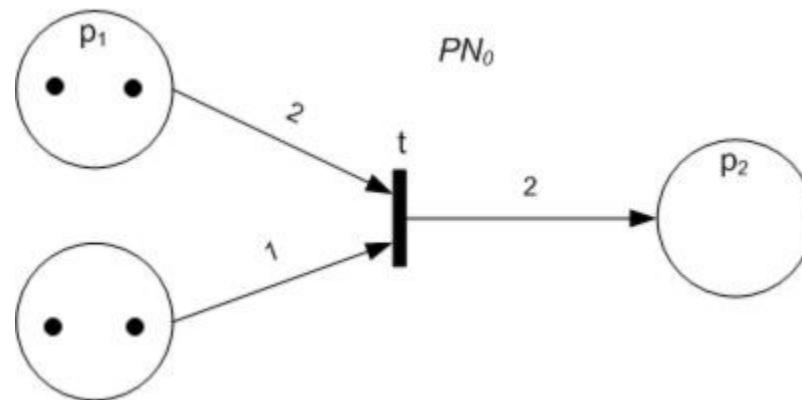
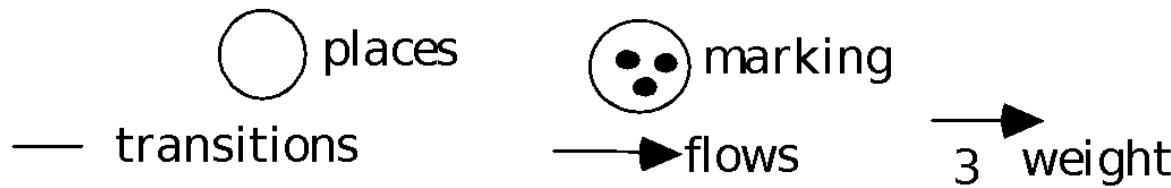
(2) $P \cup T \neq \emptyset$

(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:
$$M'(p) = M(p) - W(<p,t>) + W(<t,p>)$$

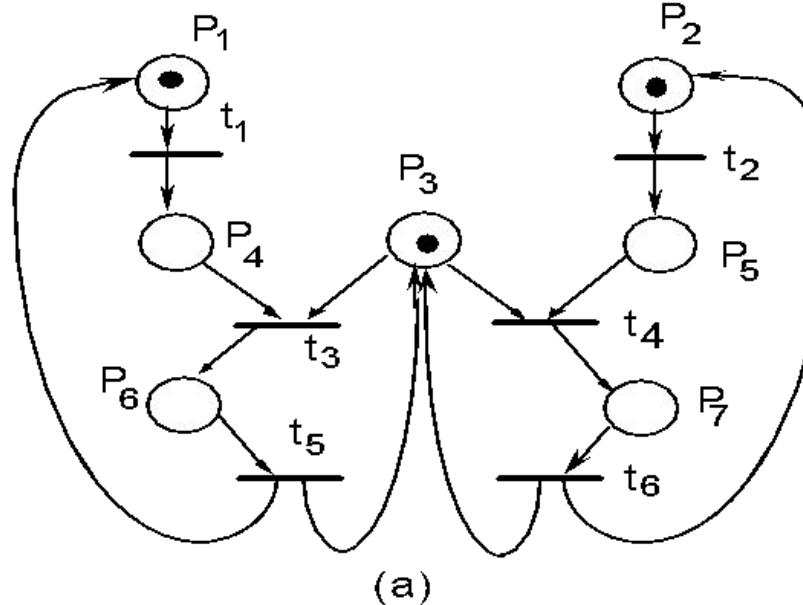
Modeling with Petri nets

- 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

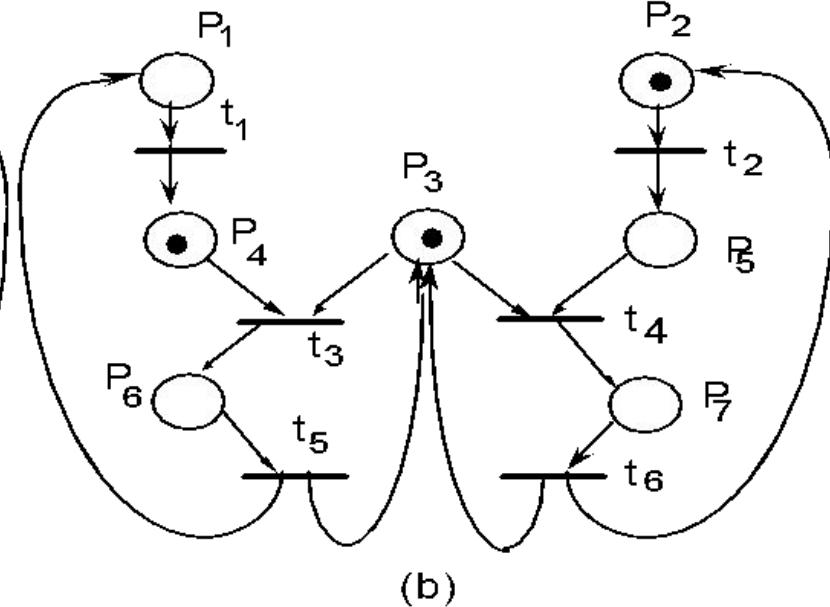
Nondeterminism

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

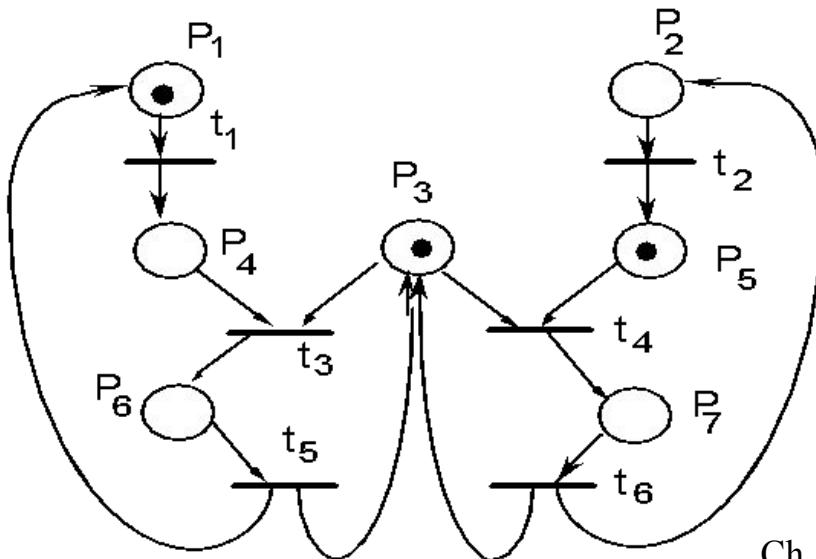
after (a) either (b) or (c) may occur, and then (d)



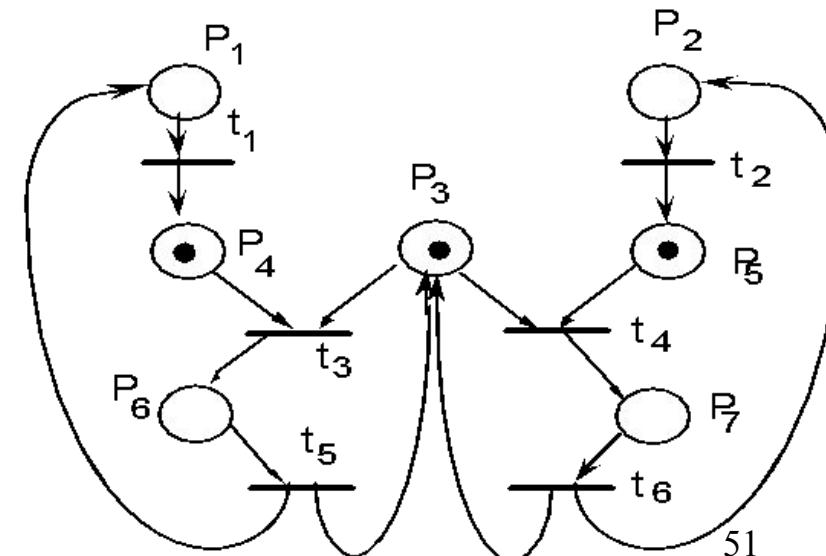
(a)



(b)



(c)



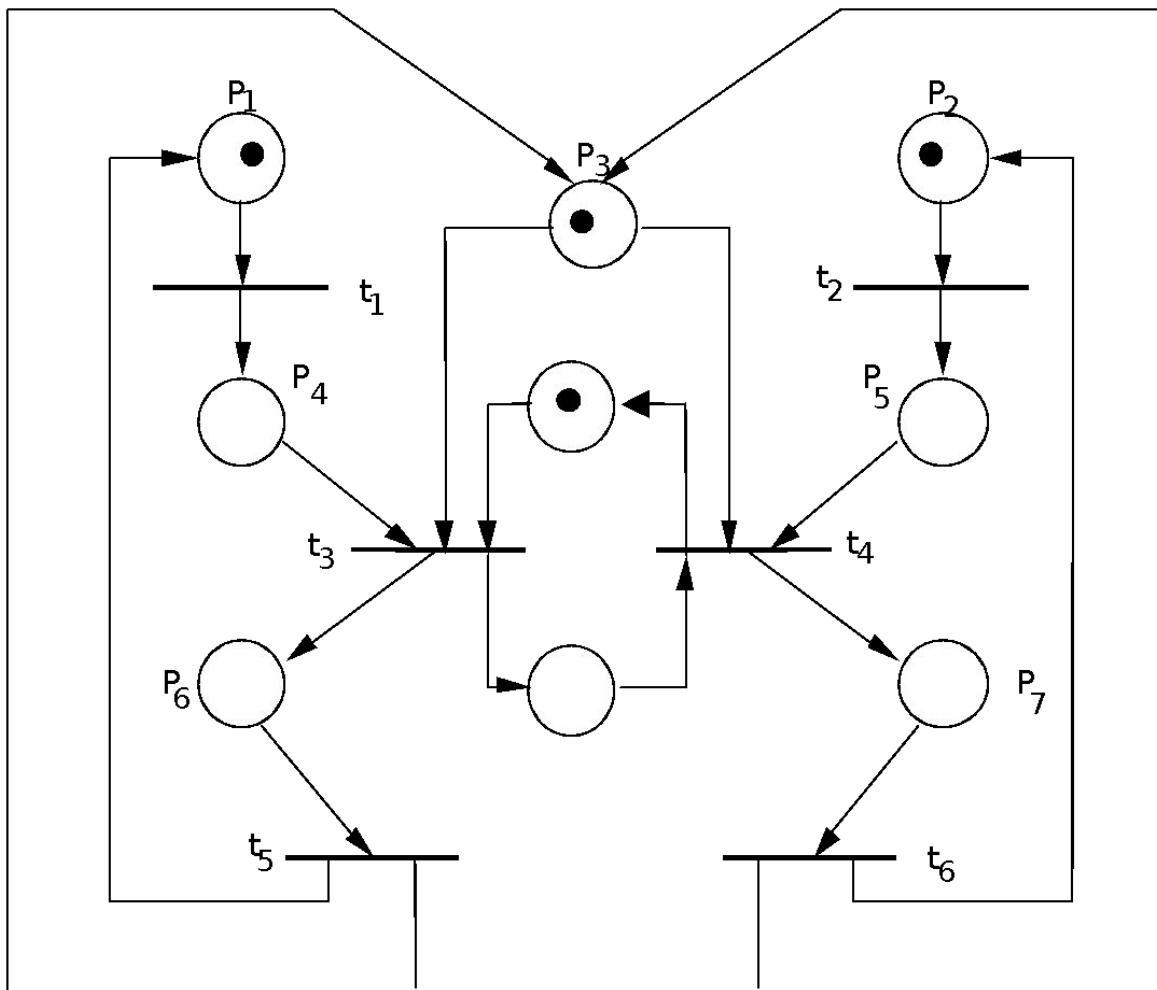
Ch. 5

(d)

Common cases

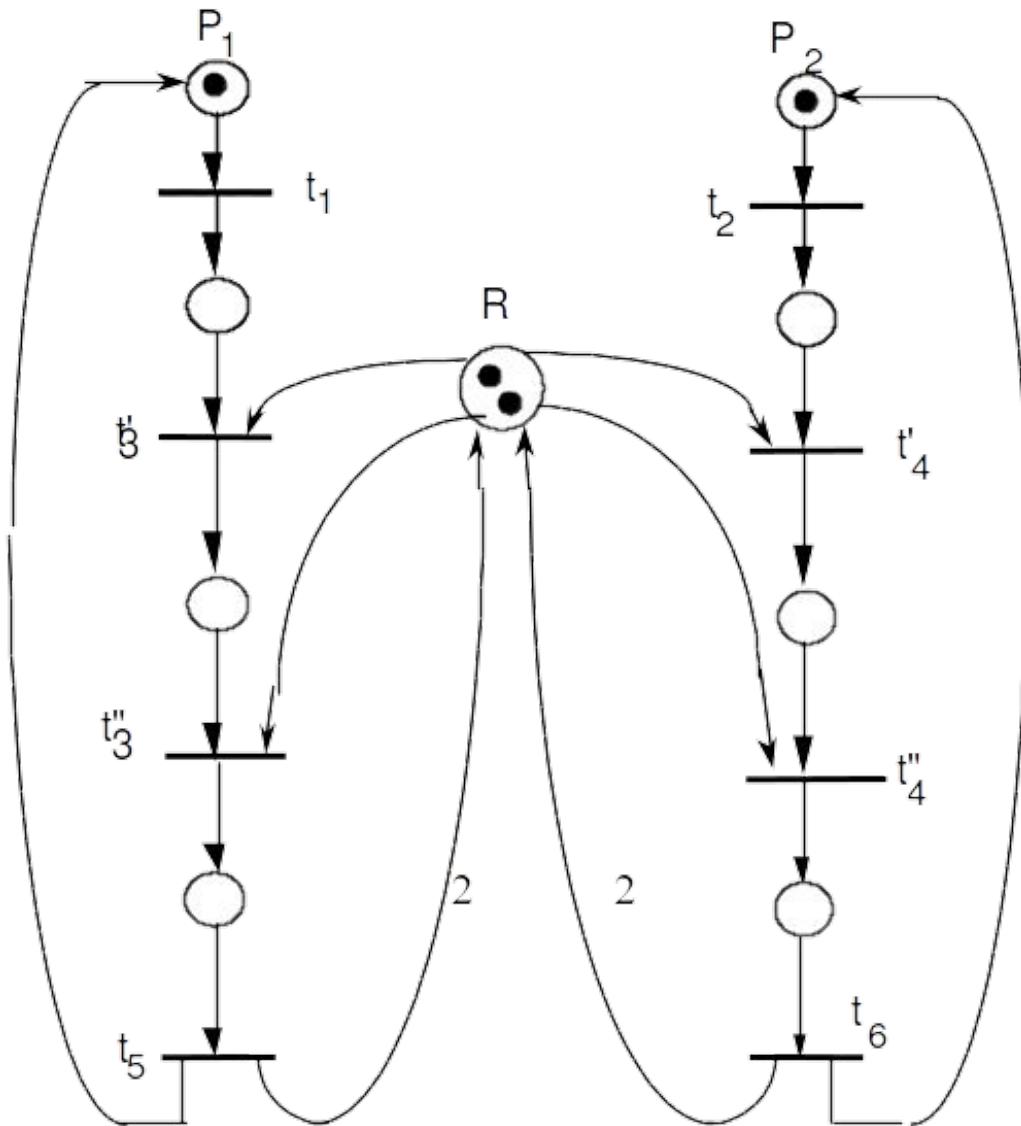
- Concurrency
 - two transitions are enabled to fire in a given state, and the firing of one does not prevent the other from firing
 - see t_1 and t_2 in case (a)
- Conflict
 - two transitions are enabled to fire in a given state, but the firing of one prevents the other from firing
 - see t_3 and t_4 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



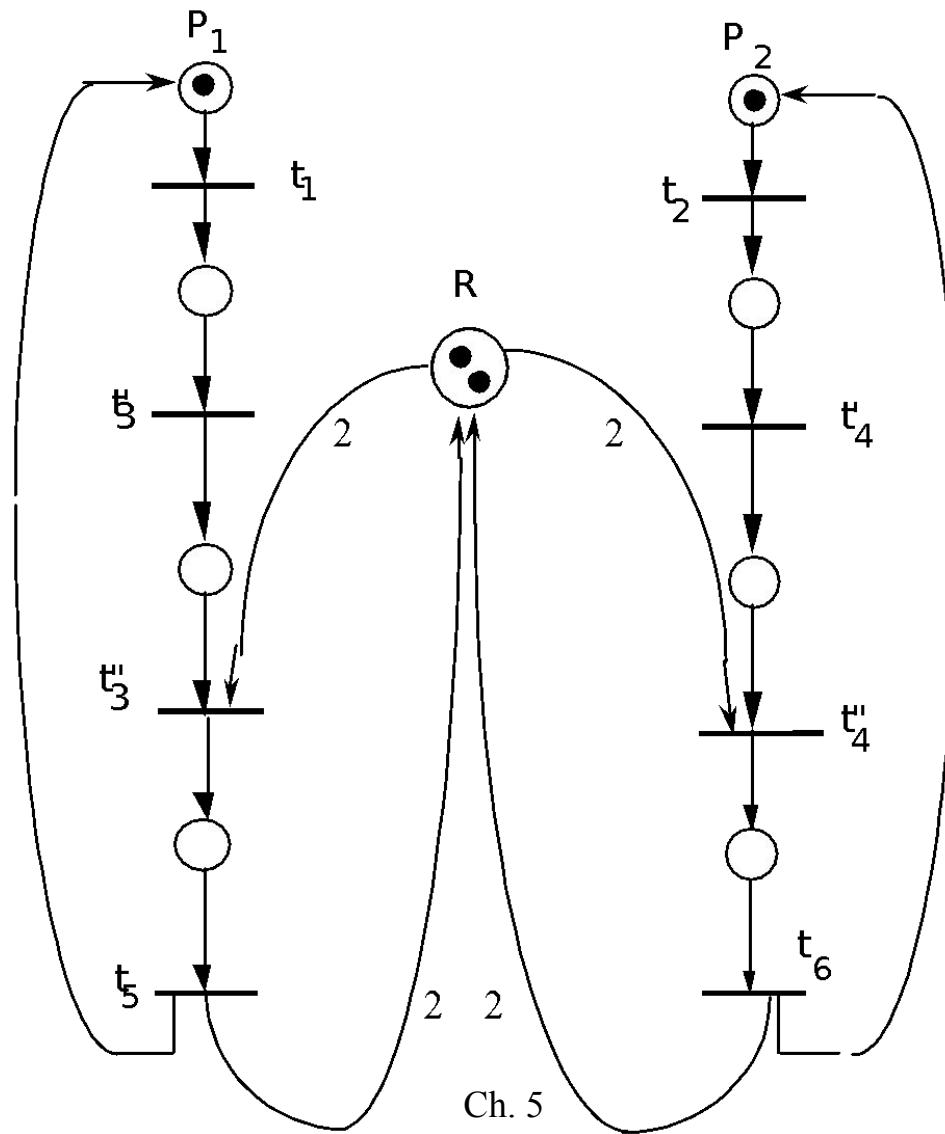
imposes
alternation

Deadlock issue

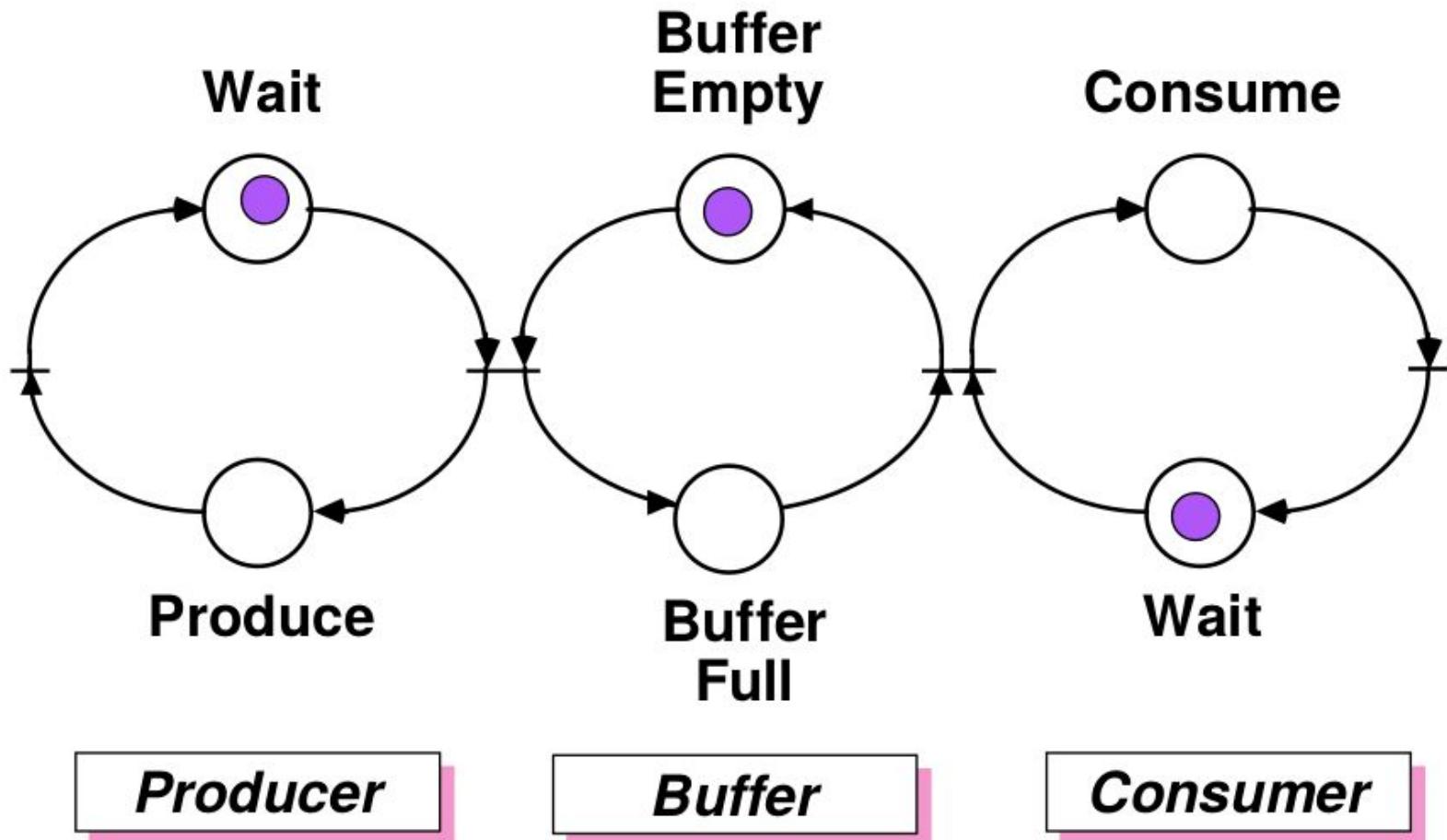


consider
 $< t_1, t'_3, t_2, t'_4 >$

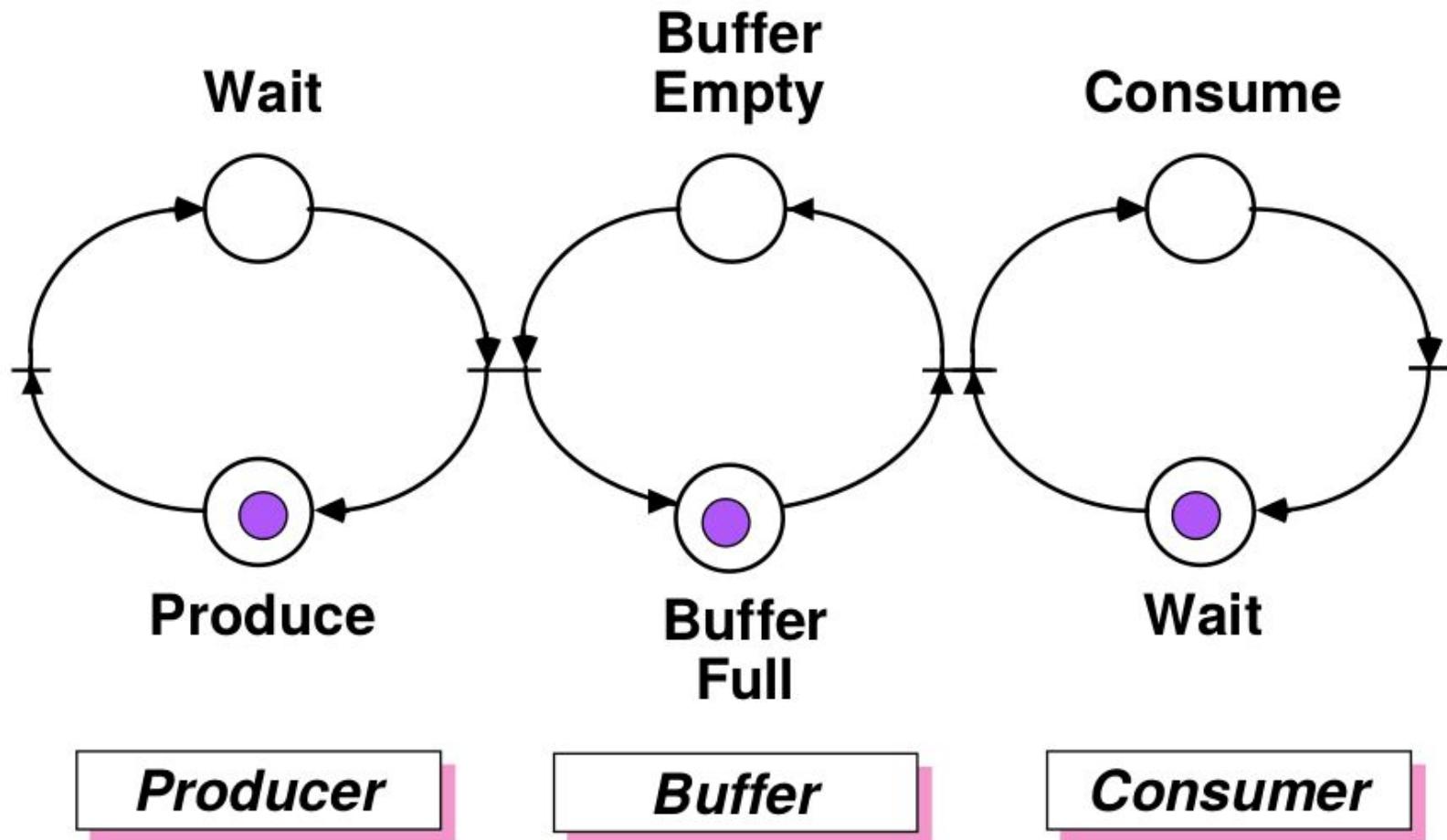
A deadlock-free net



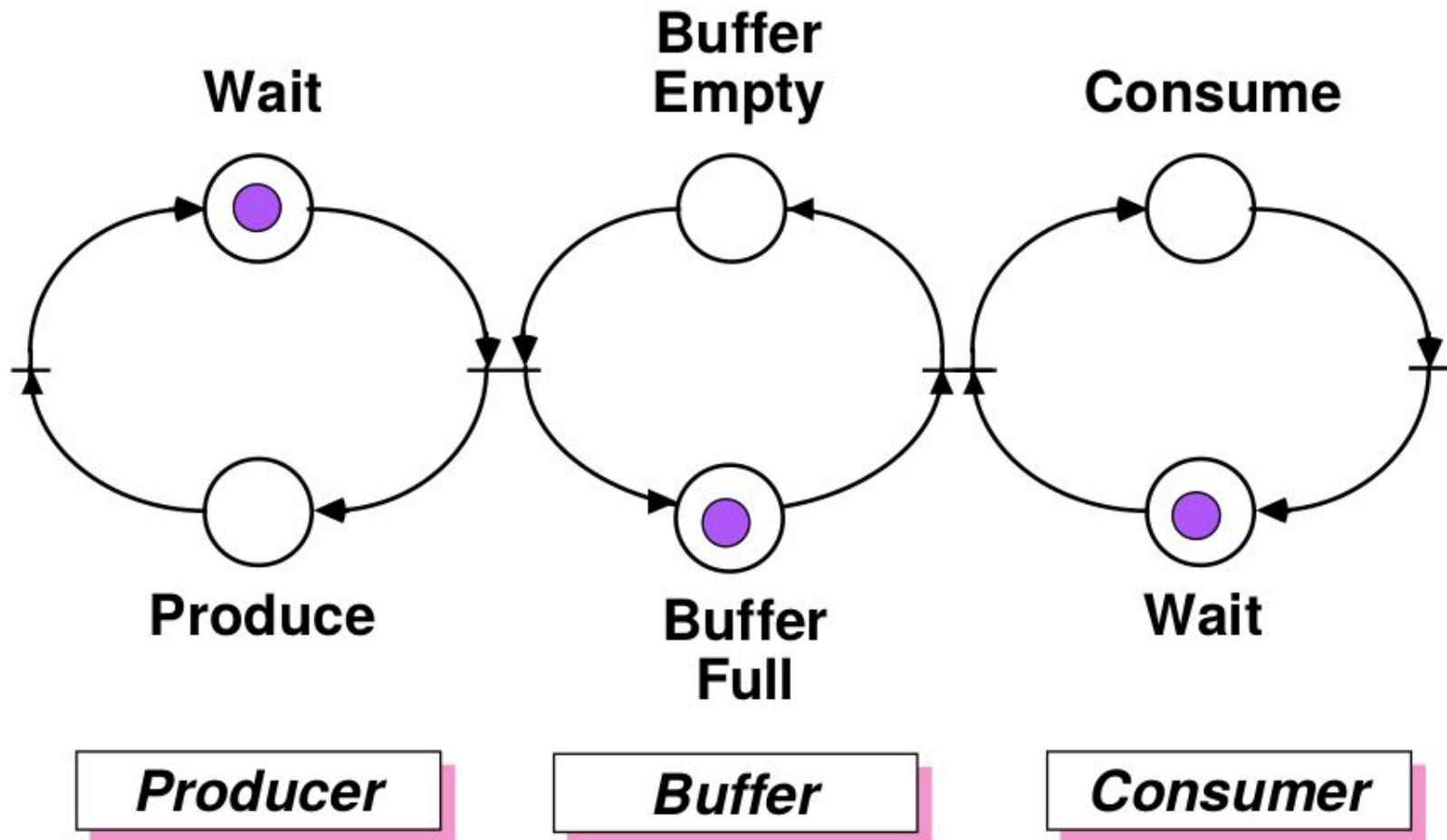
Producer-consumer example (1)



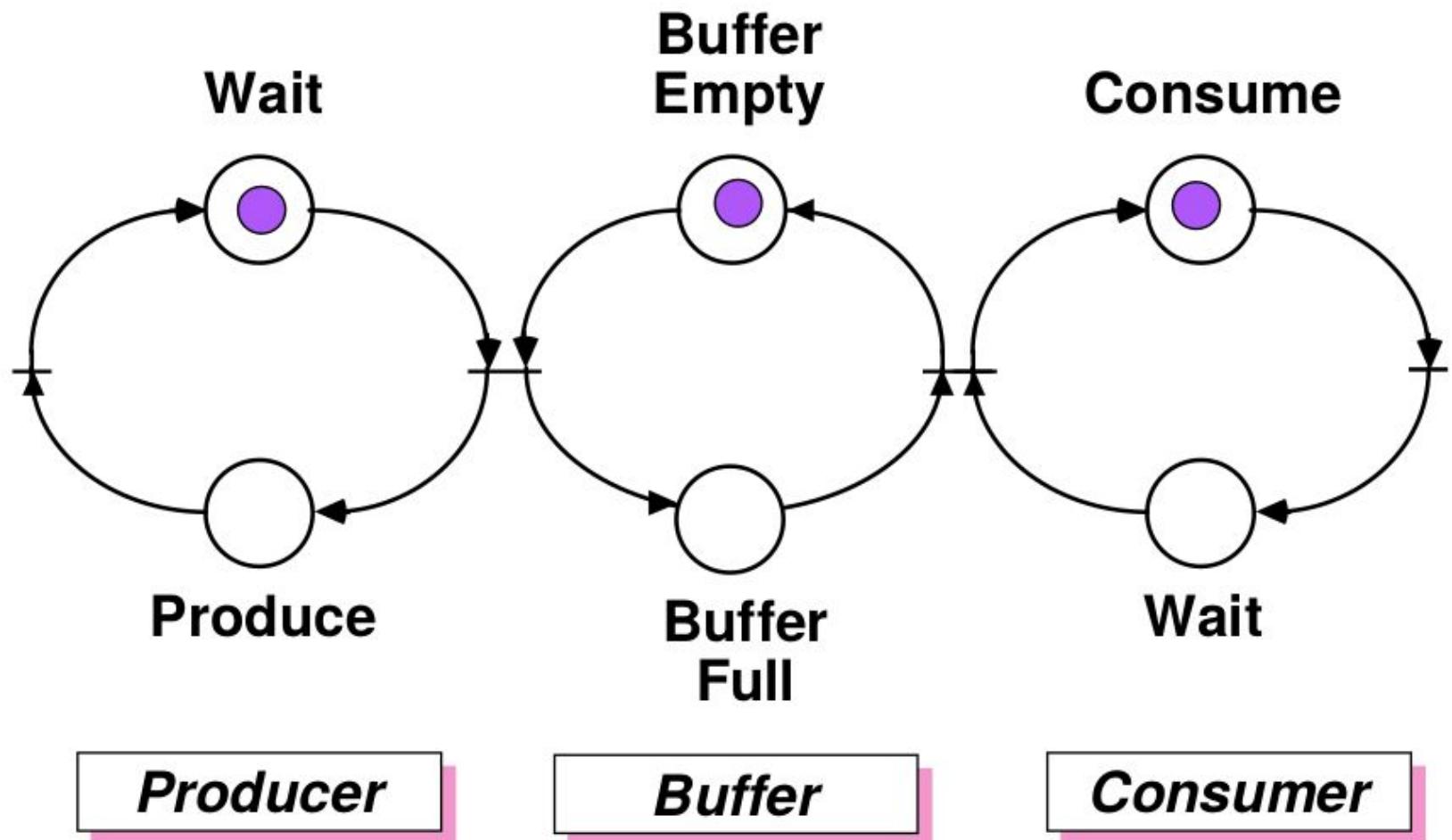
Producer-consumer example (2)



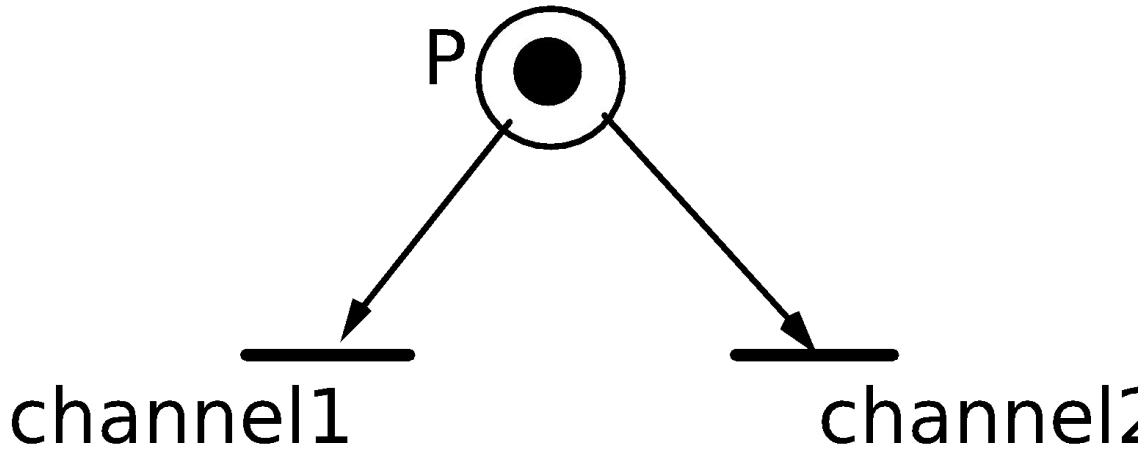
Producer-consumer example (3)



Producer-consumer example (4)



Limitations and extensions



Token represents a message.

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

How?

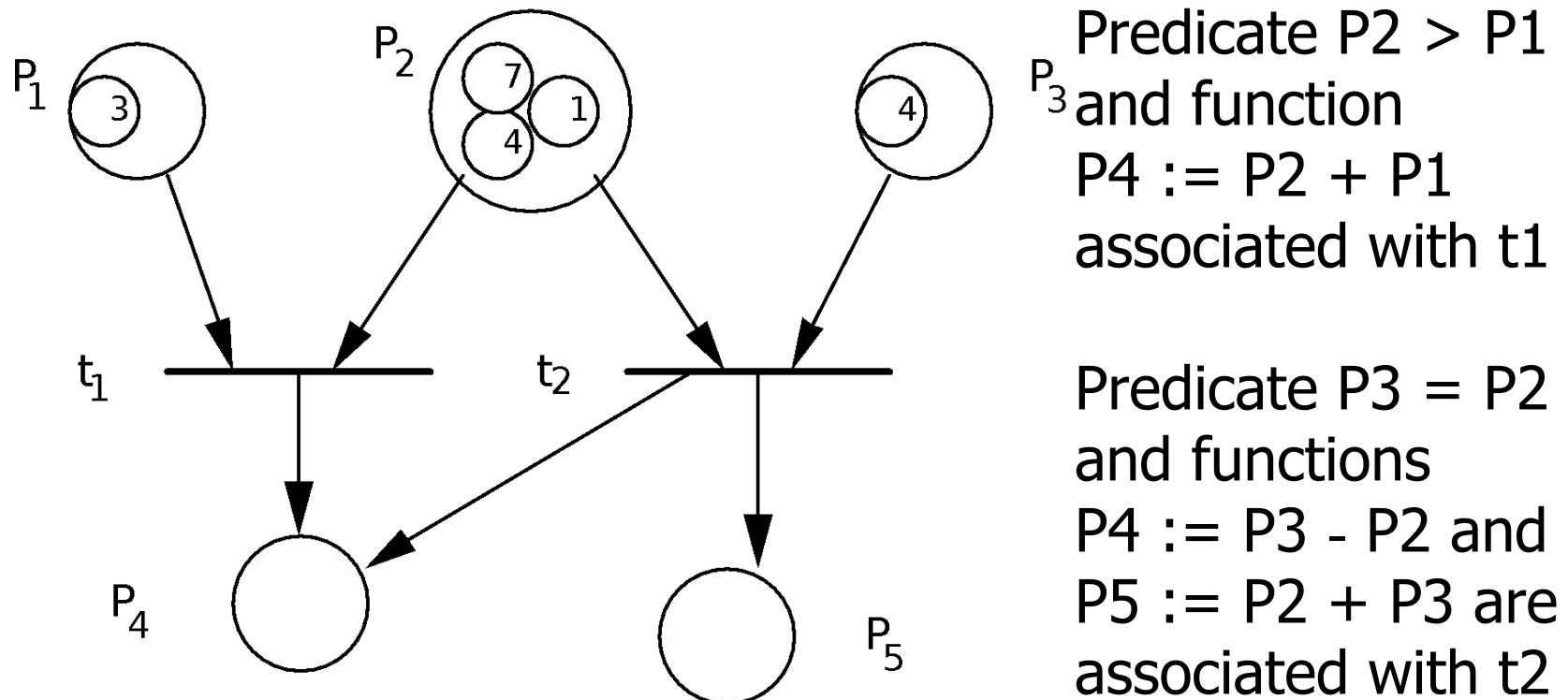
Petri nets cannot specify selection policies.

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 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

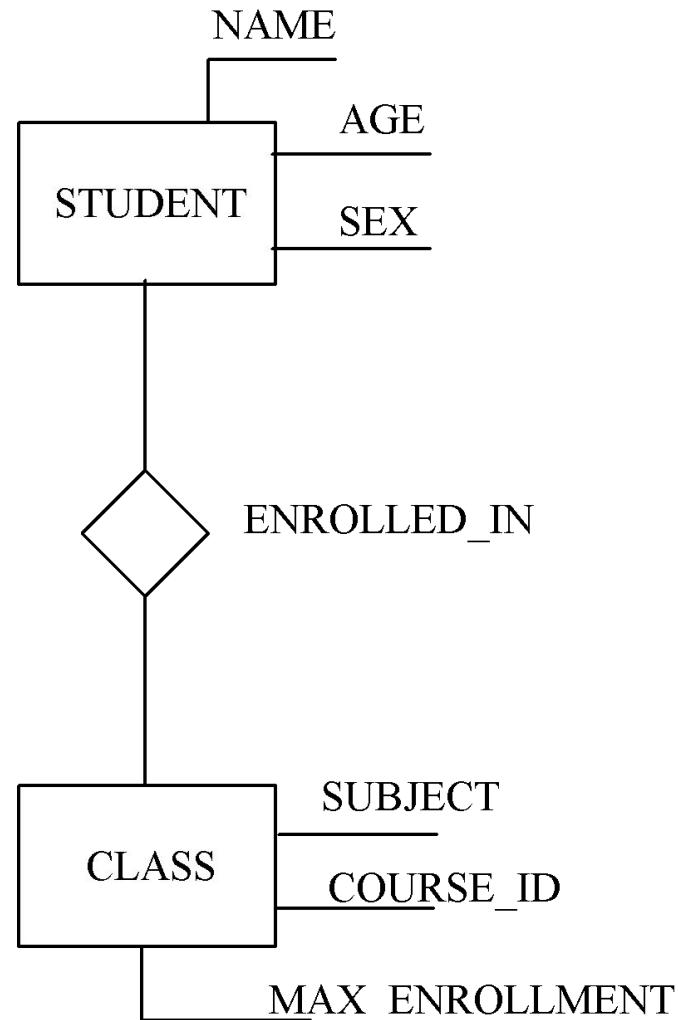
Logic specifications

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

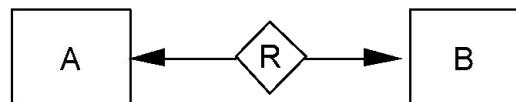
Example



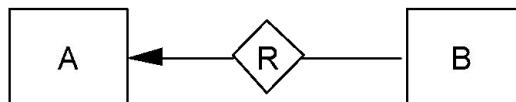
Relations

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

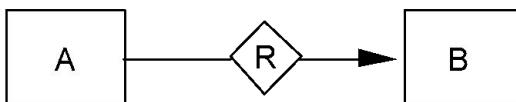
- one to one



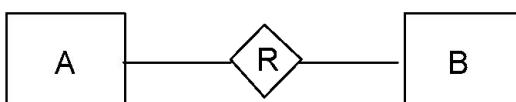
- one to many



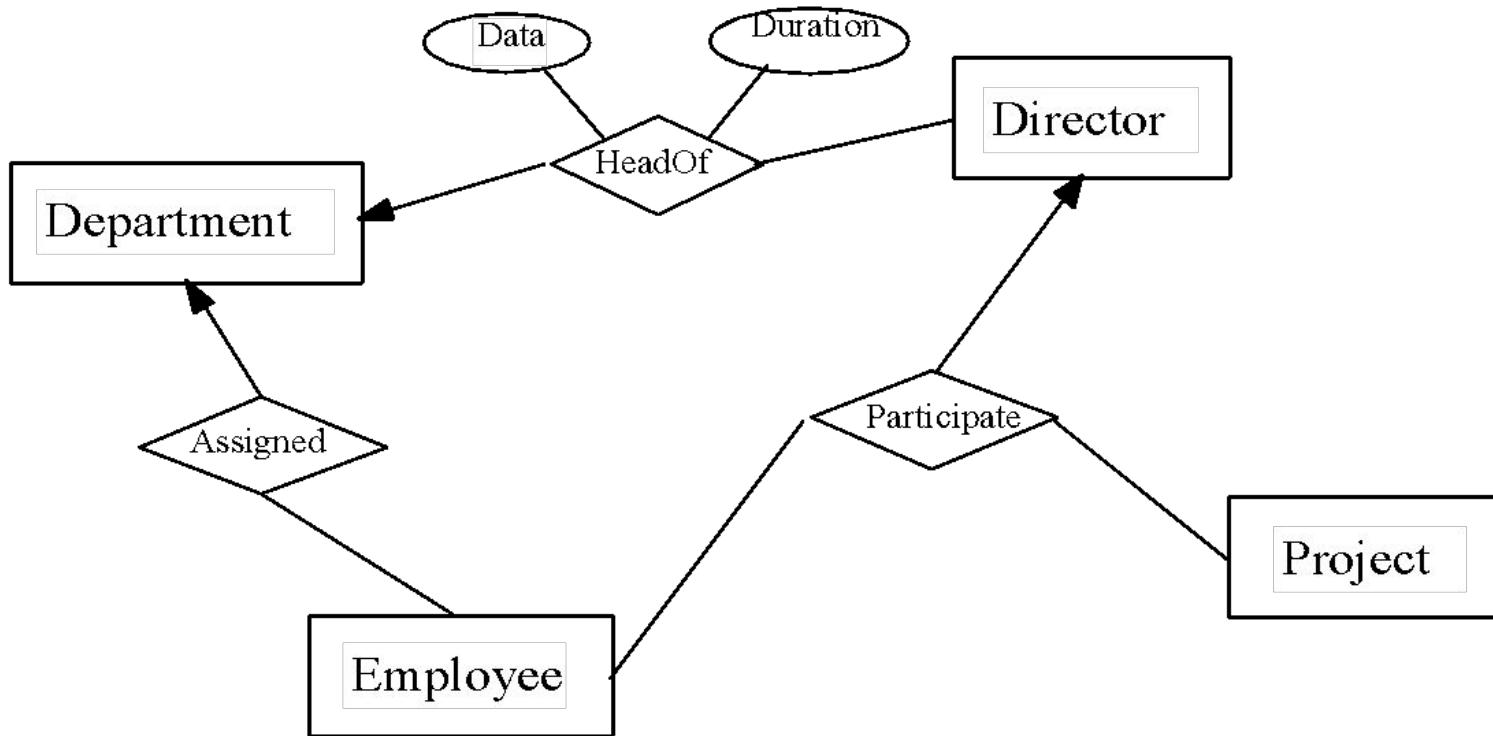
- many to one



- 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

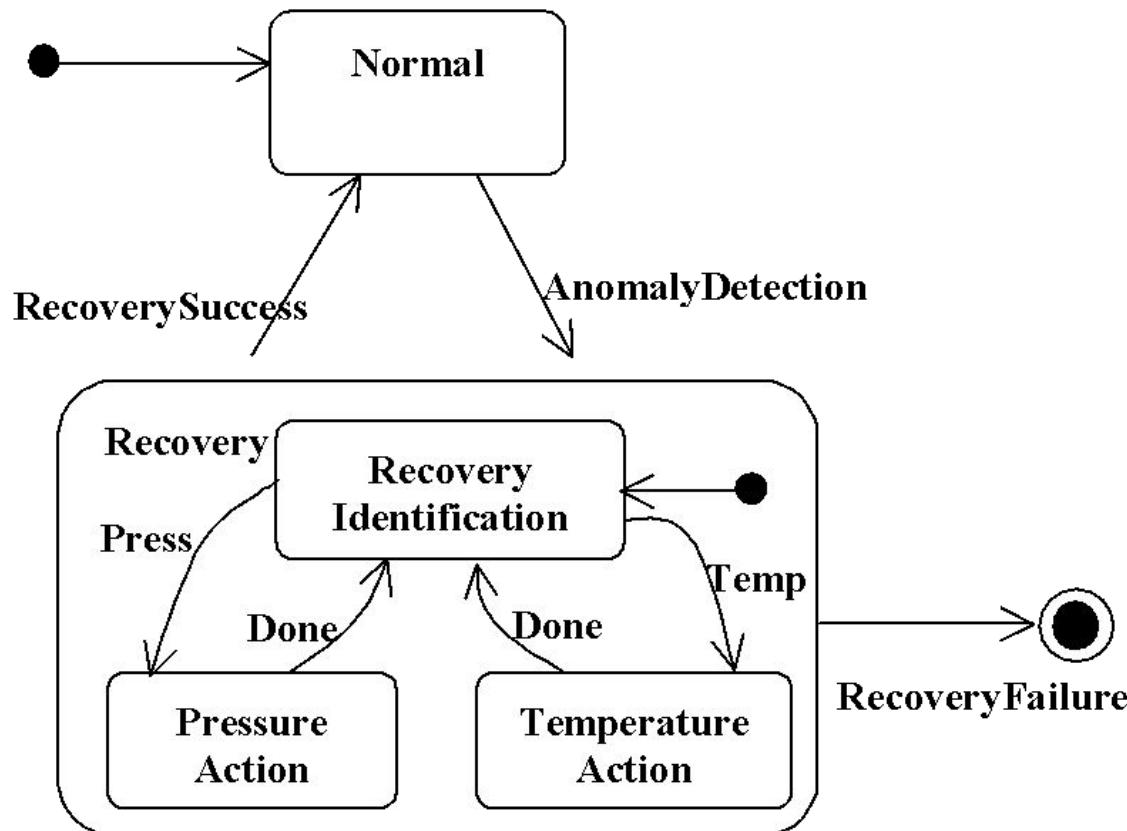
Z

Modularizing finite state machines

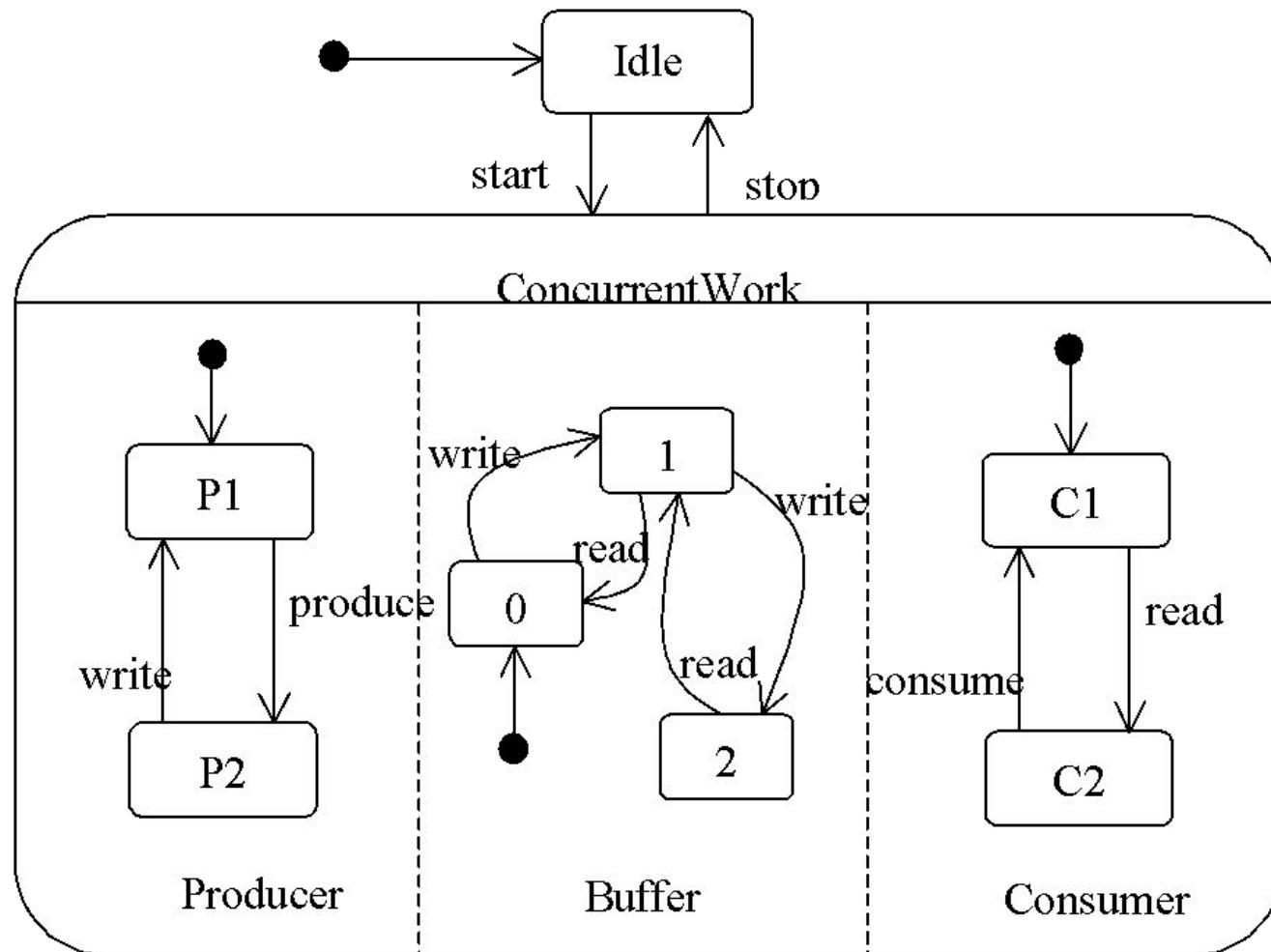
- Statecharts do that
- They have been incorporated in UML
- They provide the notions of
 - superstate
 - state decomposition

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

Operations (2)

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

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

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

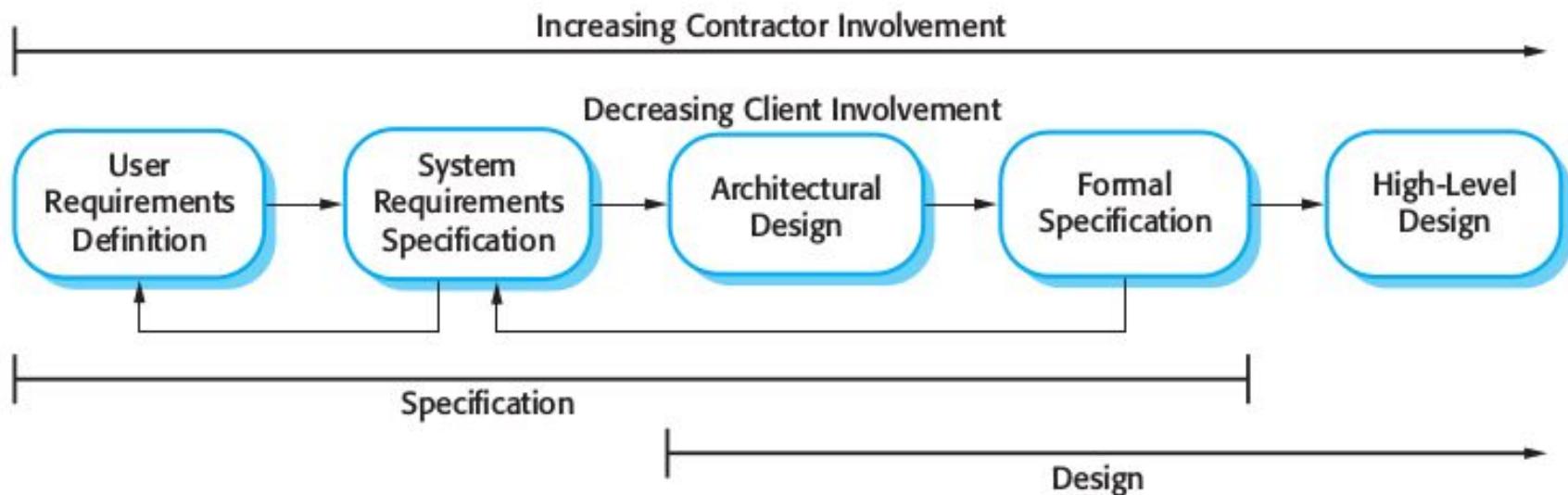
Use of formal methods

- 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

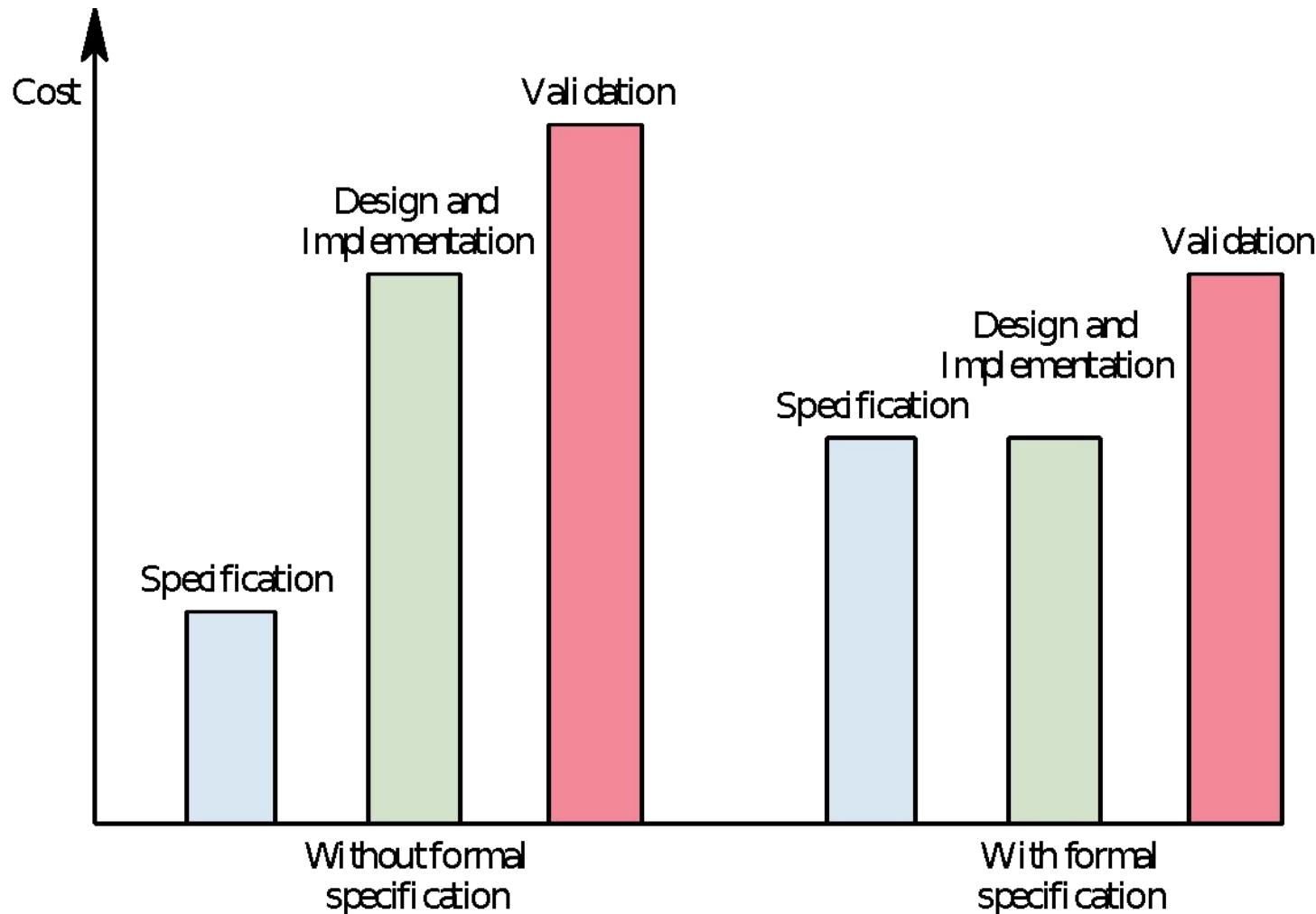
Formal specification languages

	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

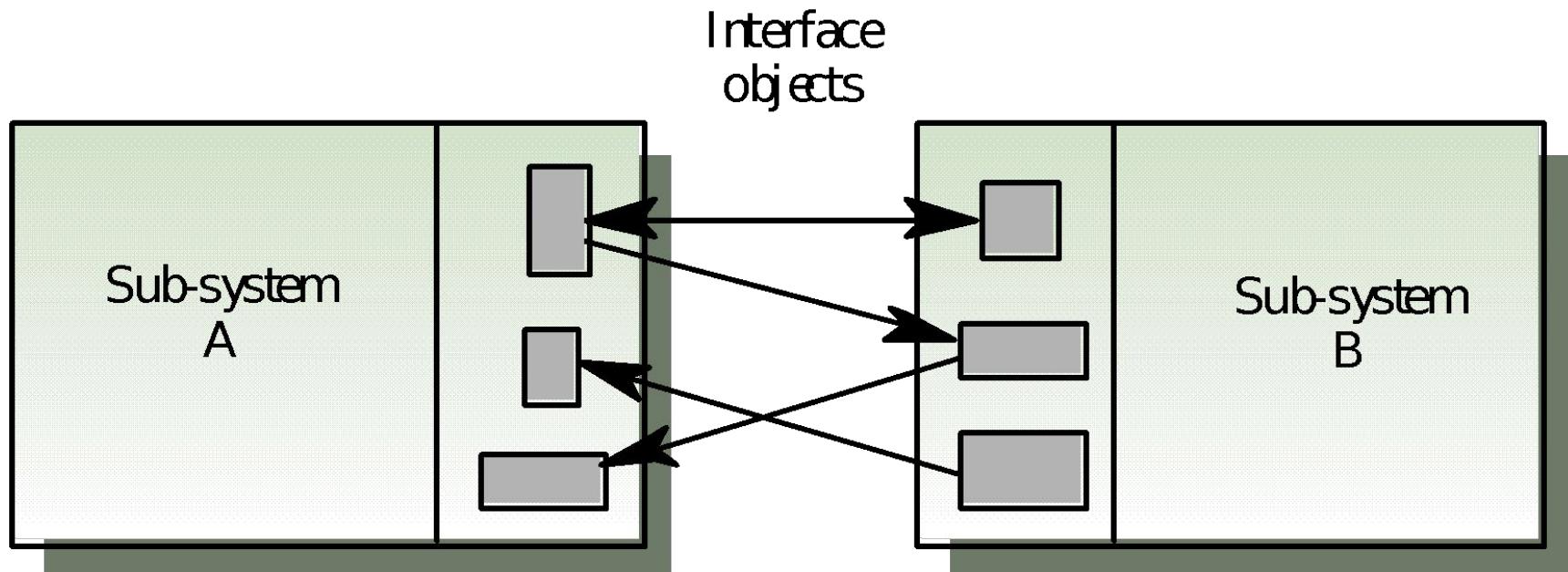
Development costs with formal specification



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

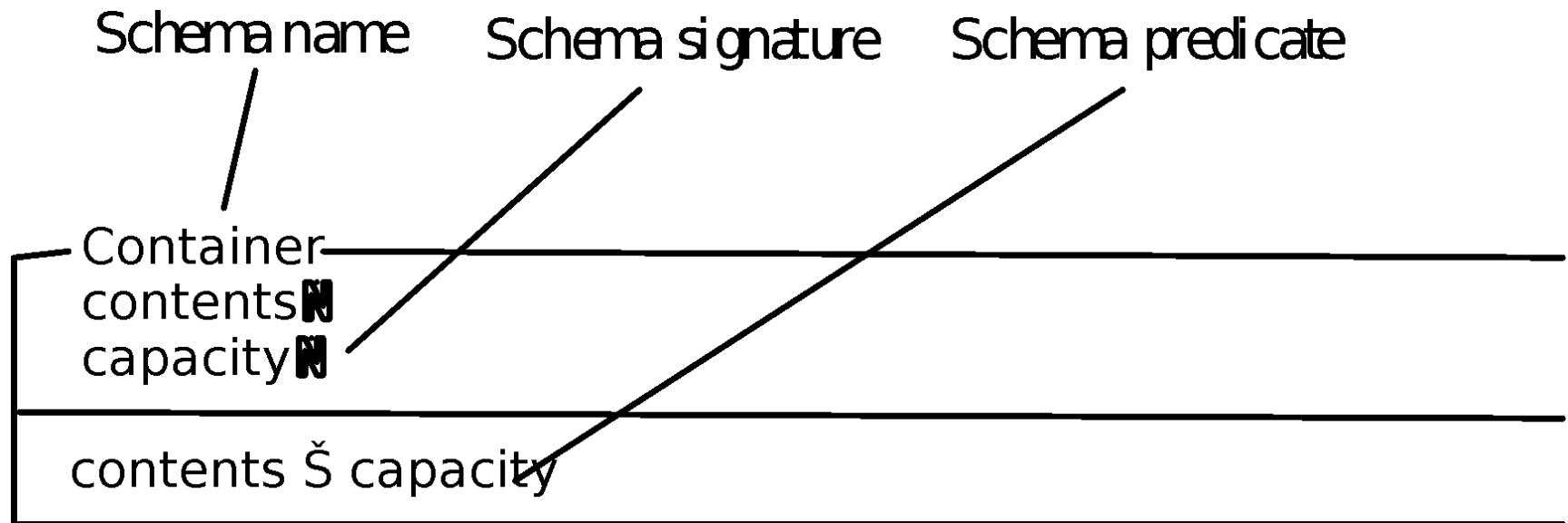
In-space (Put (S, CS1, H1), CS) =

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

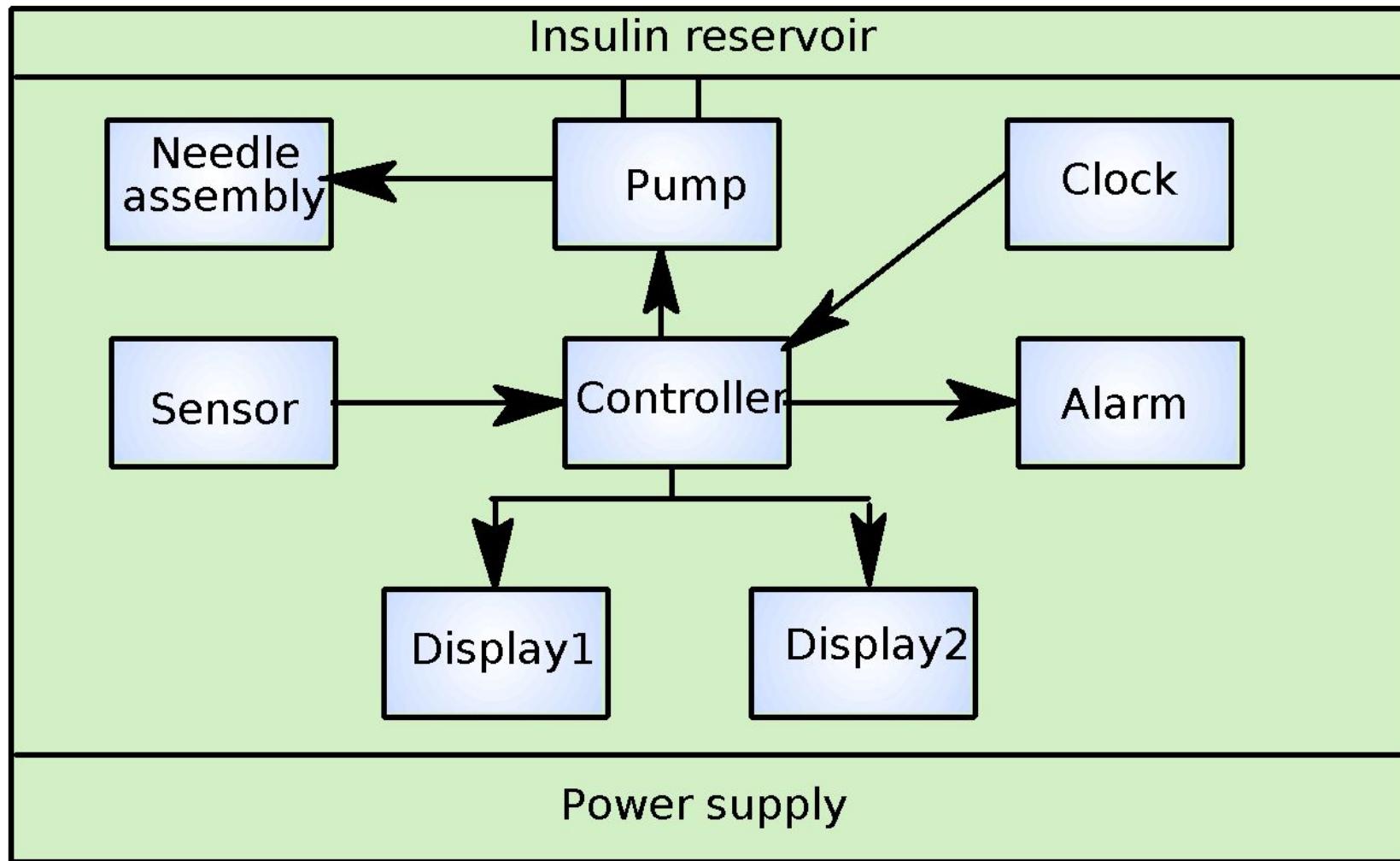
Behavioural specification

- 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.

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.

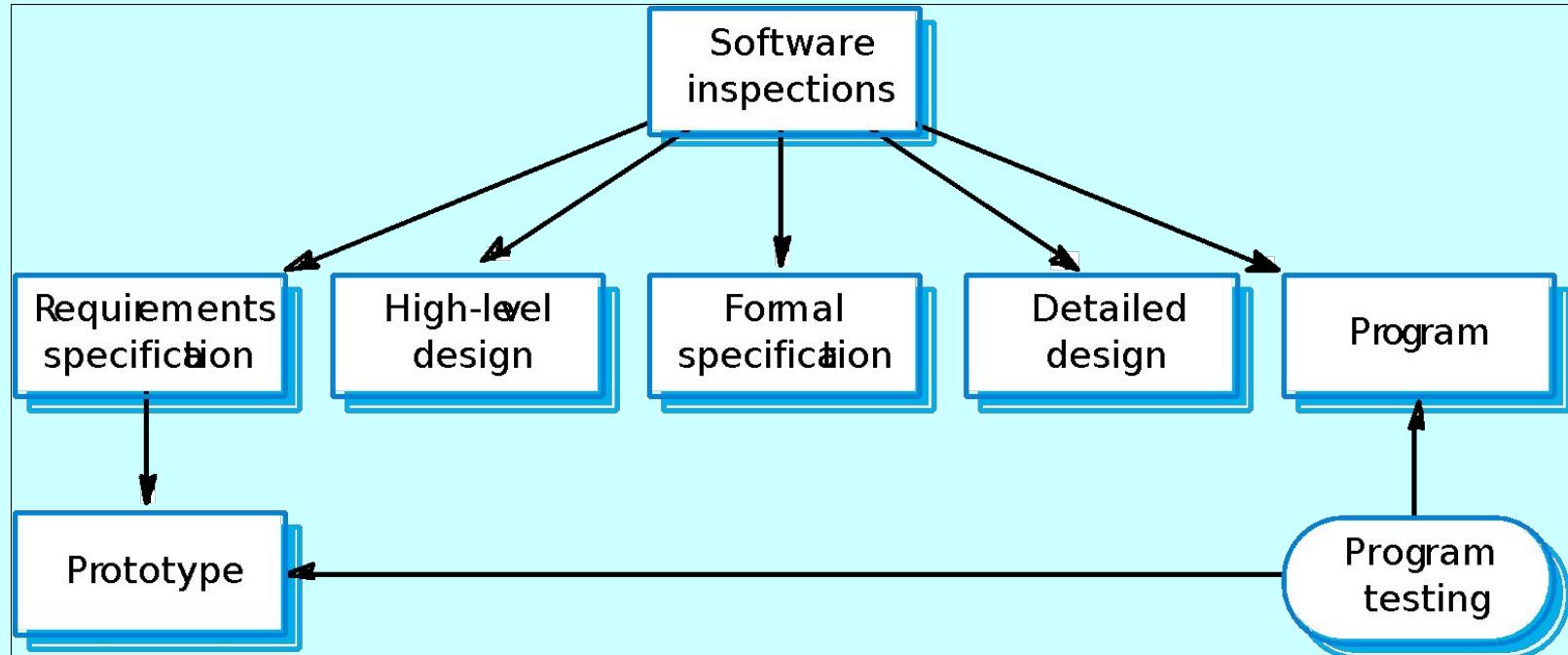
V & V confidence

- 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.

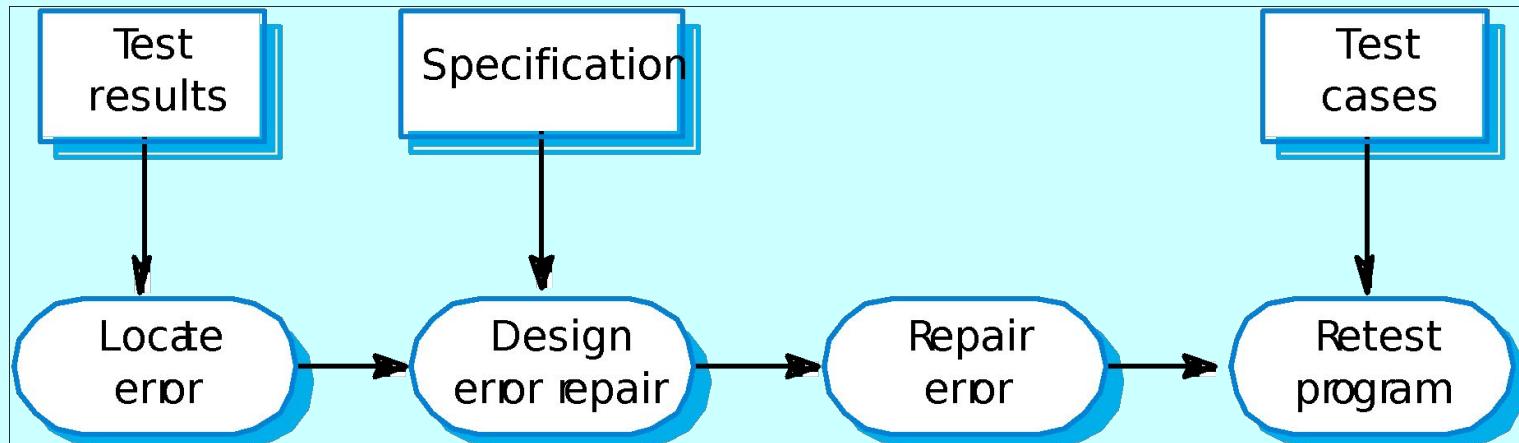
Types of testing

- **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.

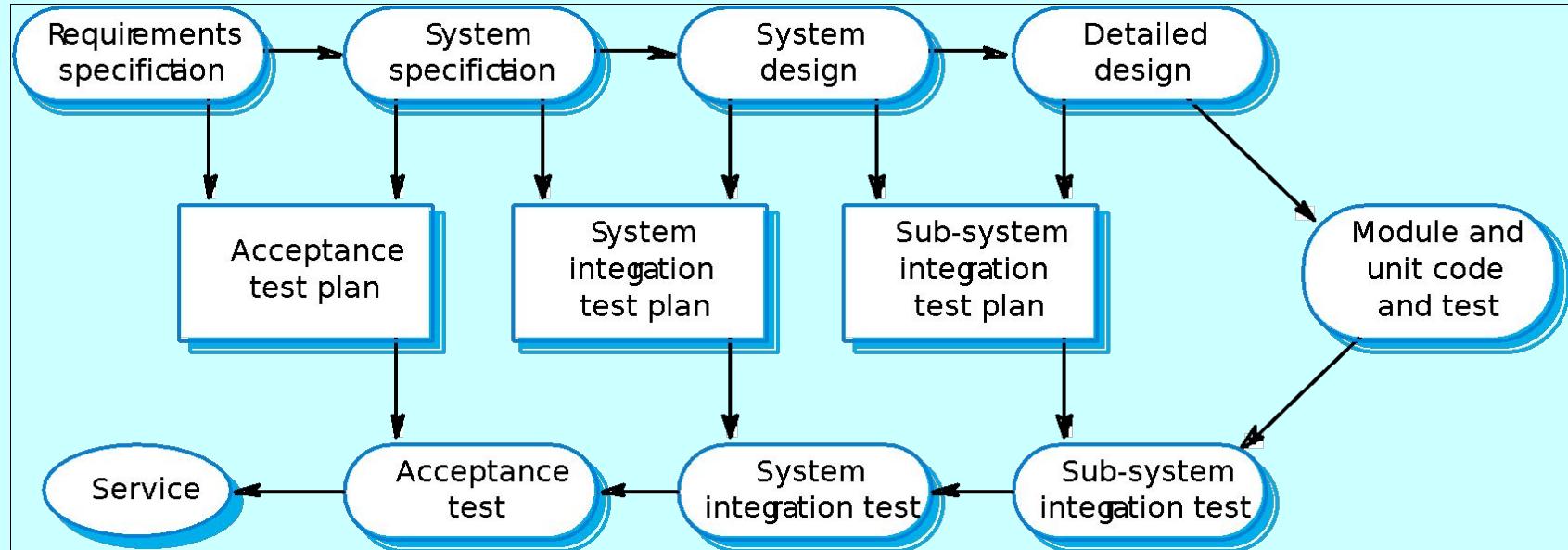
The debugging process



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.

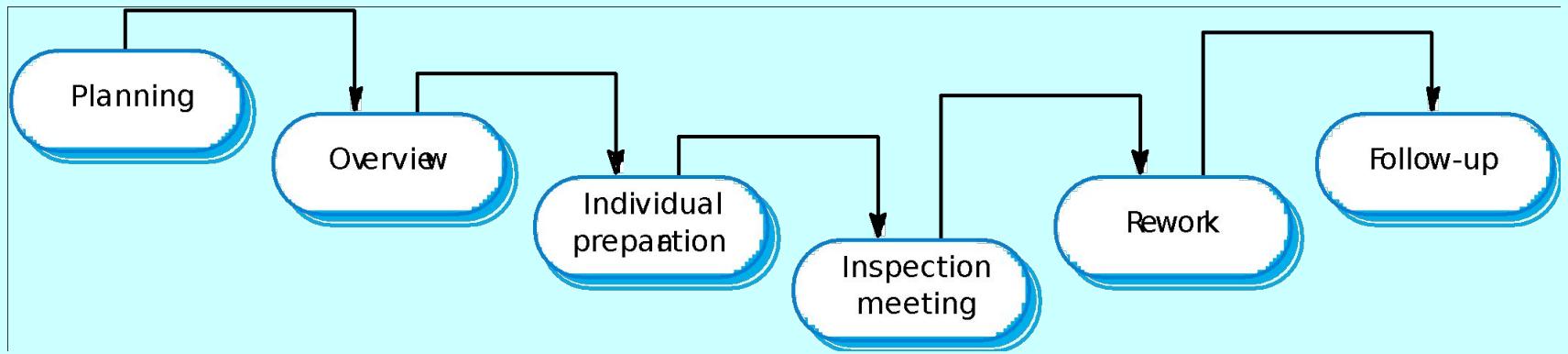
Program inspections

- 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 Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output faults	Variables output twice with no intervening assignment
Interface faults	Parameter type mismatches Parameter number mismatches Non-usage of the results of functions Uncalled functions and procedures
Storage management faults	Unassigned pointers 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.

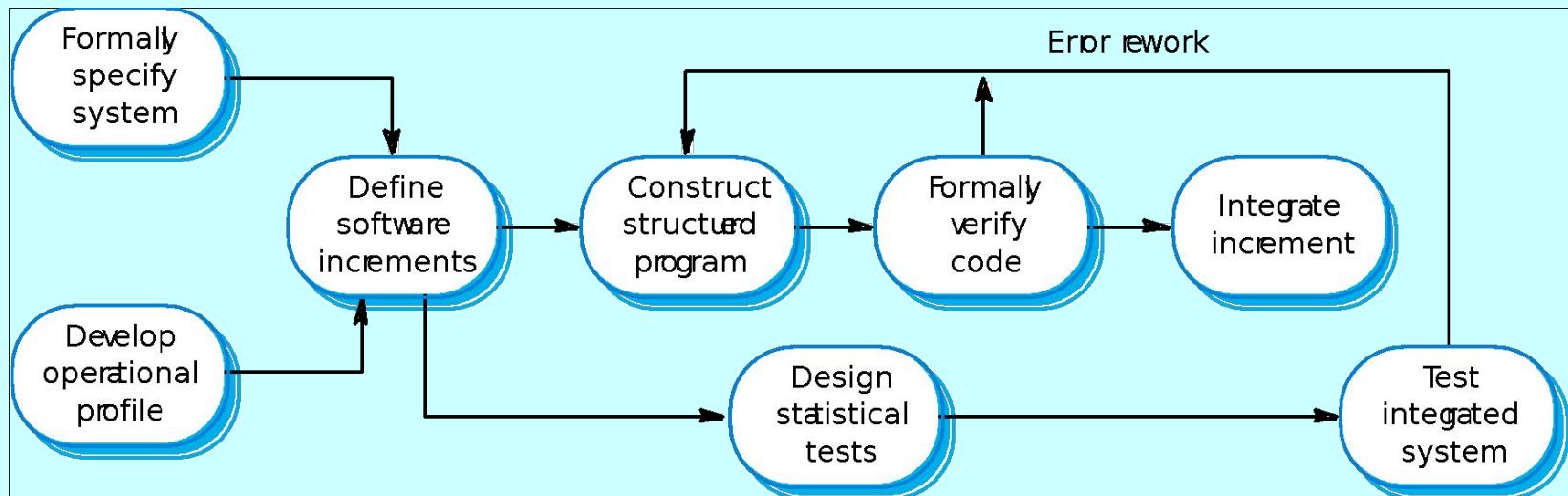
Cleanroom software development

- 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.

Model Checking Using SPIN/PROMELA

Formal Verification :

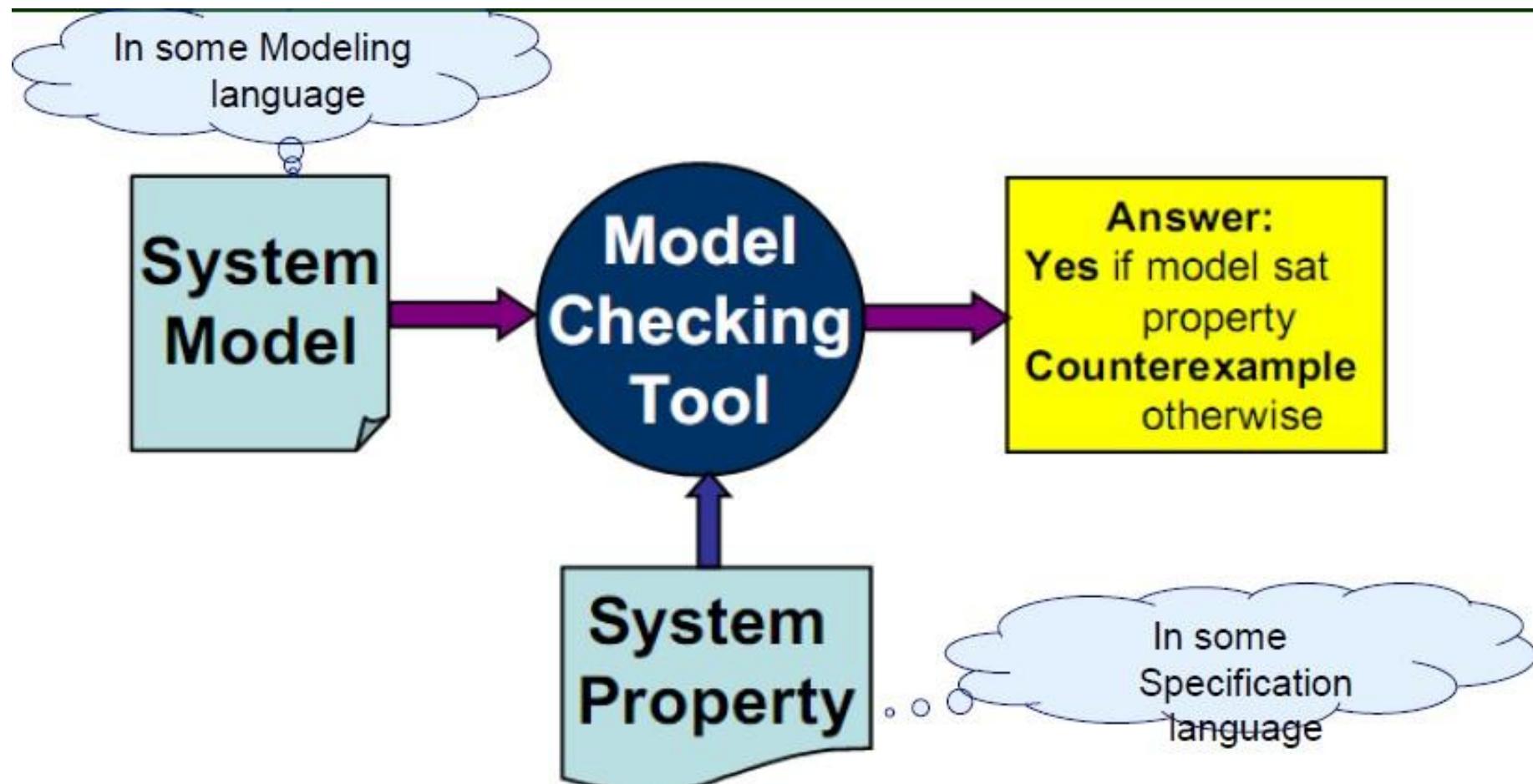
- The process of Formal Verification consists of
 - Requirements
 - Capture Modeling
 - Specification
 - Analysis
 - Documentation
 - n
- In practice, some phases may overlap
The overall process is iterative rather than sequential

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.

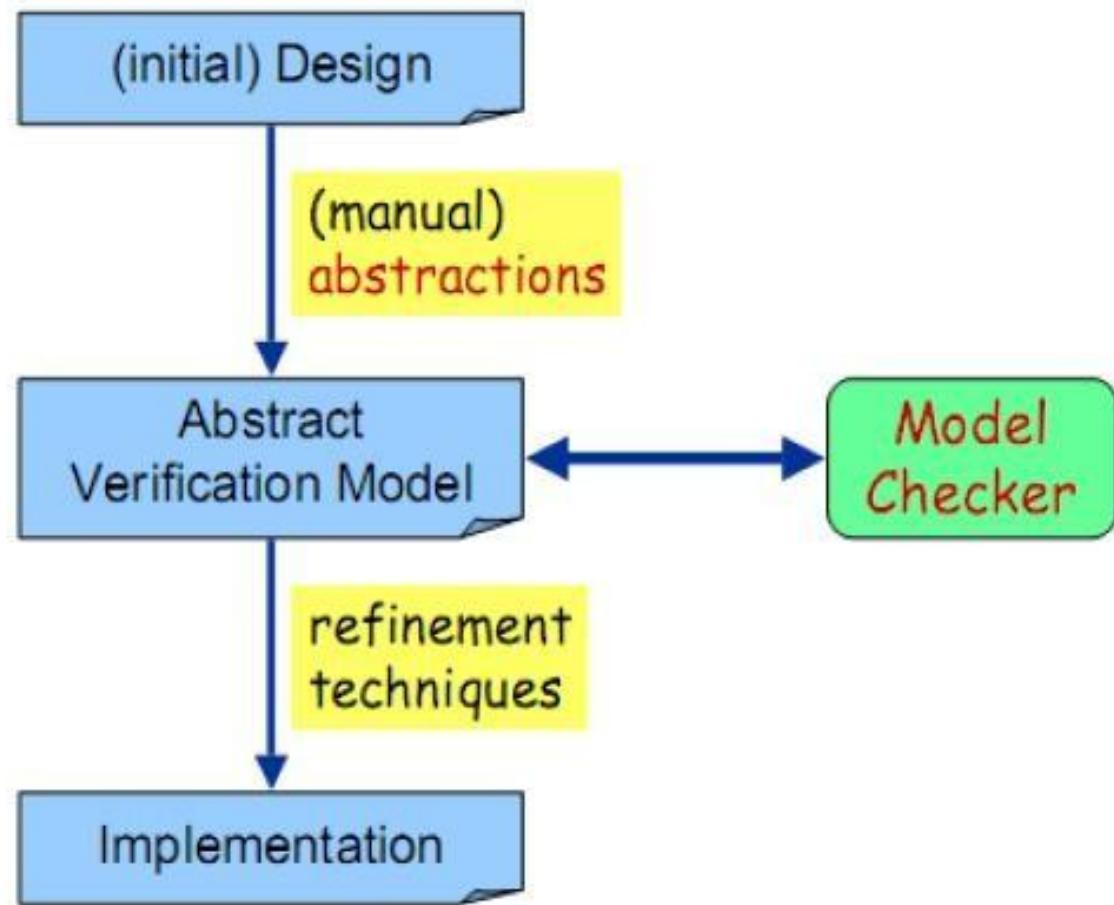
Model Checking

- 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

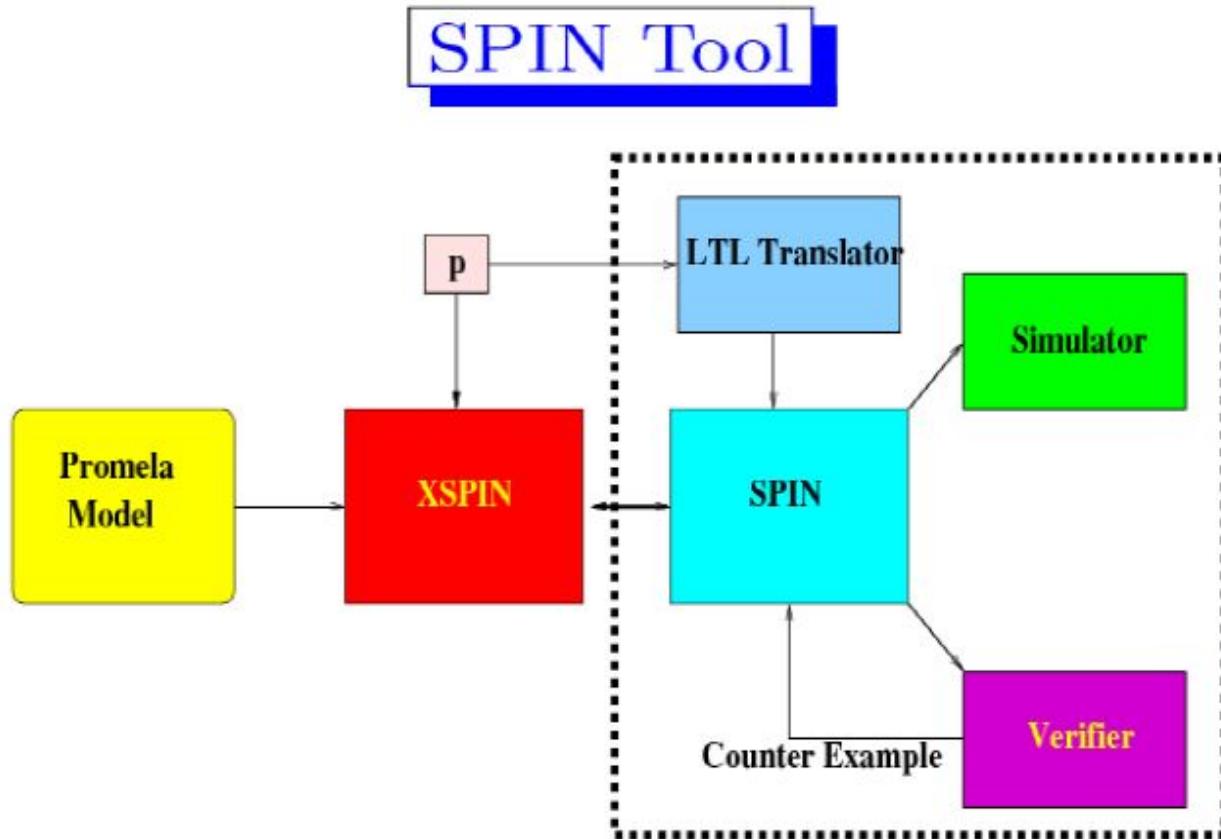
Introduction to

- 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

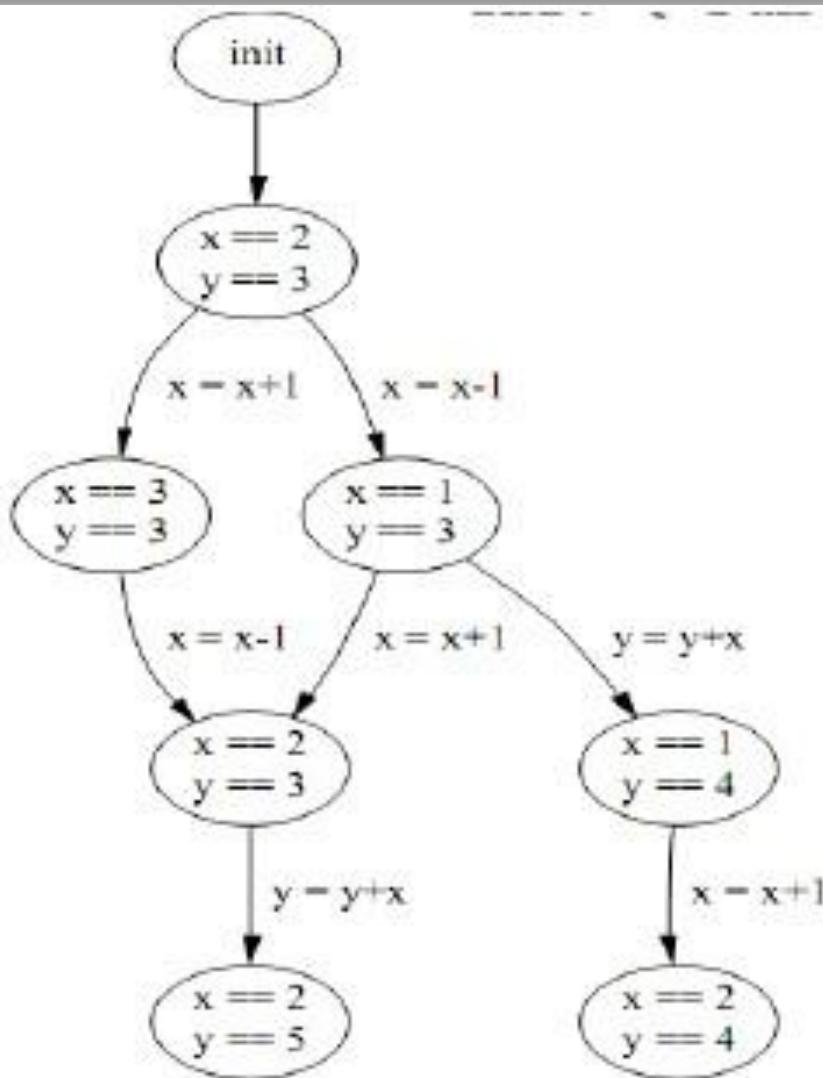
- Two types of statement delimiters
  - ; and ->
  - Use the one that is most appropriate at the given situation Usually ; is used between ordinary statements
  - -> is often used after “guards” in a if OR do statement, pointing at what comes next
  - These can be used interchangeably.

# 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
}
```

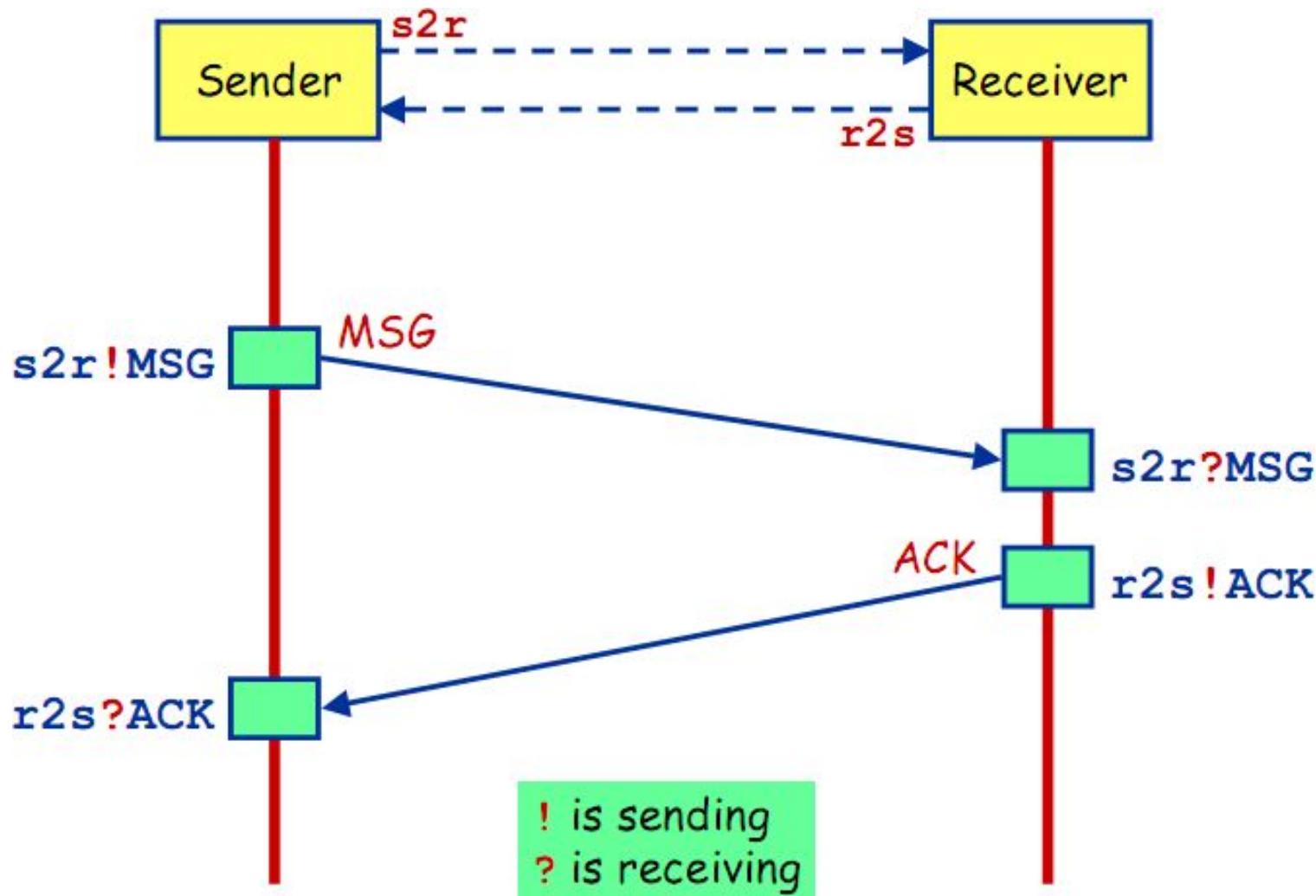
- The timeout condition becomes true when no other statements within the overall system being modeled are executable

# 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
}
```

# 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.

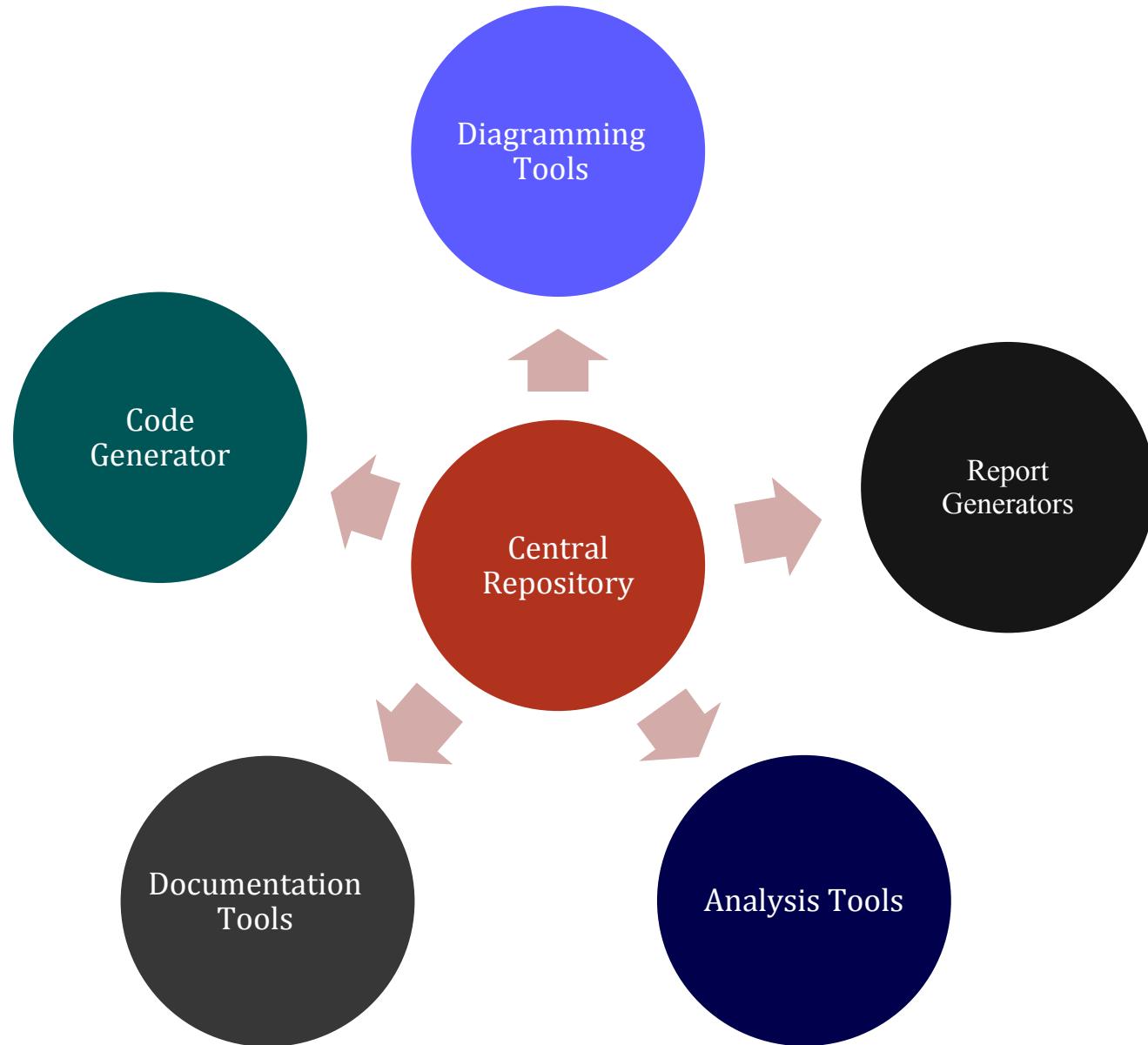
# Categories Of CASE Tools

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

---

# **Design and Implementation**

## Topics covered

---

- ❖ Object-oriented design using the UML
- ❖ Design patterns
- ❖ Implementation

# Design and implementation

---

- ❖ Software design and implementation is the stage in the software engineering process at which an executable software system is developed.
- ❖ Software design and implementation activities are invariably inter-leaved.
  - Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements.
  - Implementation is the process of realizing the design as a program.

# Build or buy

---

- ❖ In a wide range of domains, it is now possible to buy off-the-shelf systems (COTS) that can be adapted and tailored to the users' requirements.
  - For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster to use this approach rather than developing a system in a conventional programming language.
- ❖ When you develop an application in this way, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.

---

# **Object-oriented design using the UML**

# An object-oriented design process

---

- ❖ Structured object-oriented design processes involve developing a number of different system models.
- ❖ They require a lot of effort for development and maintenance of these models and, for small systems, this may not be cost-effective.
- ❖ However, for large systems developed by different groups design models are an important communication mechanism.

# Process stages

---

- ❖ There are a variety of different object-oriented design processes that depend on the organization using the process.
- ❖ Common activities in these processes include:
  - Define the context and modes of use of the system;
  - Design the system architecture;
  - Identify the principal system objects;
  - Develop design models;
  - Specify object interfaces.
- ❖ Process illustrated here using a design for a wilderness weather station.

# System context and interactions

---

- ❖ Understanding the relationships between the software that is being designed and its external environment is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment.
- ❖ Understanding of the context also lets you establish the boundaries of the system. Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems.

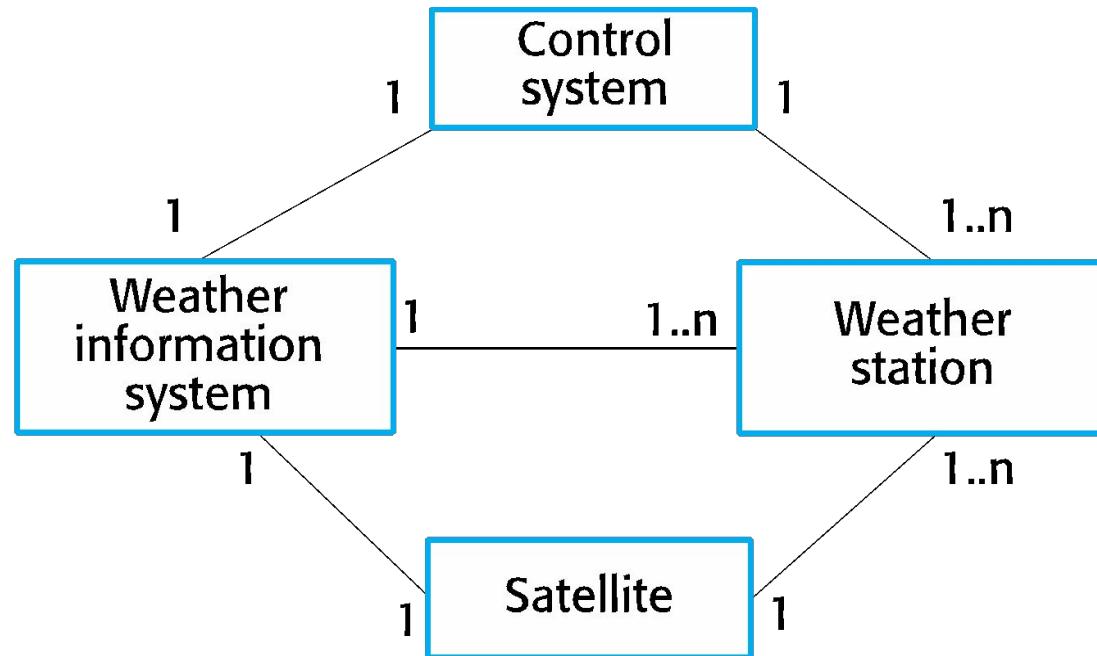
# Context and interaction models

---

- ❖ A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.
- ❖ An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

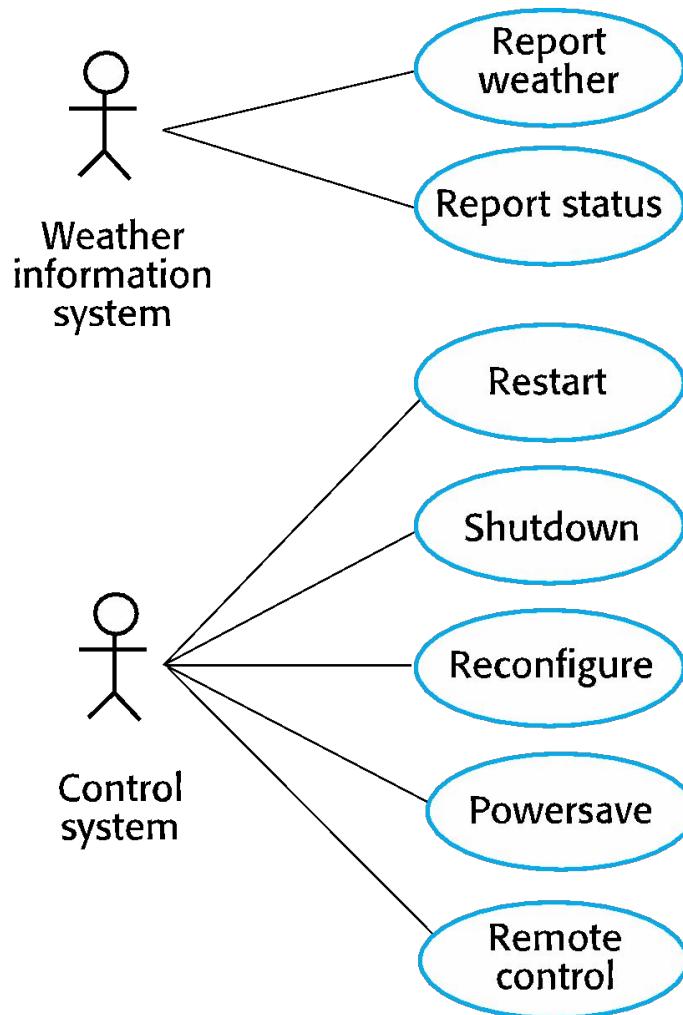
# System context for the weather station

---



# Weather station use cases

---



# Use case description—Report weather

---

| System      | Weather station                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Use case    | Report weather                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Actors      | Weather information system, Weather station                                                                                                                                                                                                                                                                                                                                                                                       |
| Description | The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum, and average wind speeds; the total rainfall; and the wind direction as sampled at five-minute intervals. |
| Stimulus    | The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.                                                                                                                                                                                                                                                                                         |
| Response    | The summarized data is sent to the weather information system.                                                                                                                                                                                                                                                                                                                                                                    |
| Comments    | Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and may be modified in the future.                                                                                                                                                                                                                                                                           |

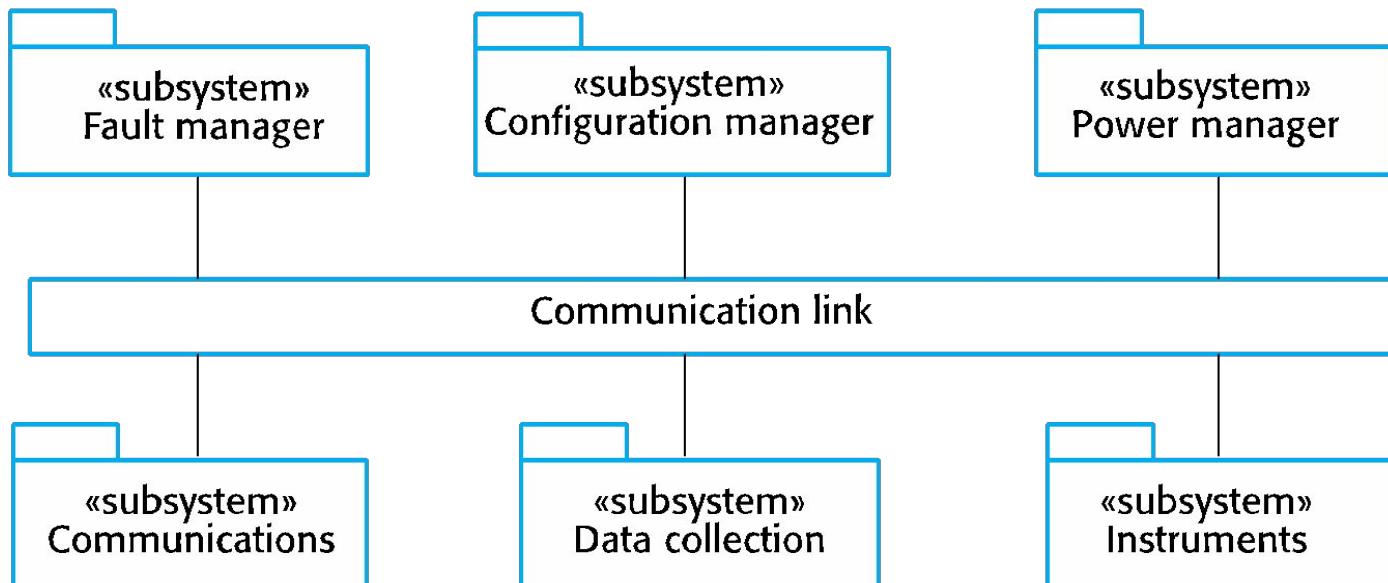
# Architectural design

---

- ❖ Once interactions between the system and its environment have been understood, you use this information for designing the system architecture.
- ❖ You identify the major components that make up the system and their interactions, and then may organize the components using an architectural pattern such as a layered or client-server model.
- ❖ The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure.

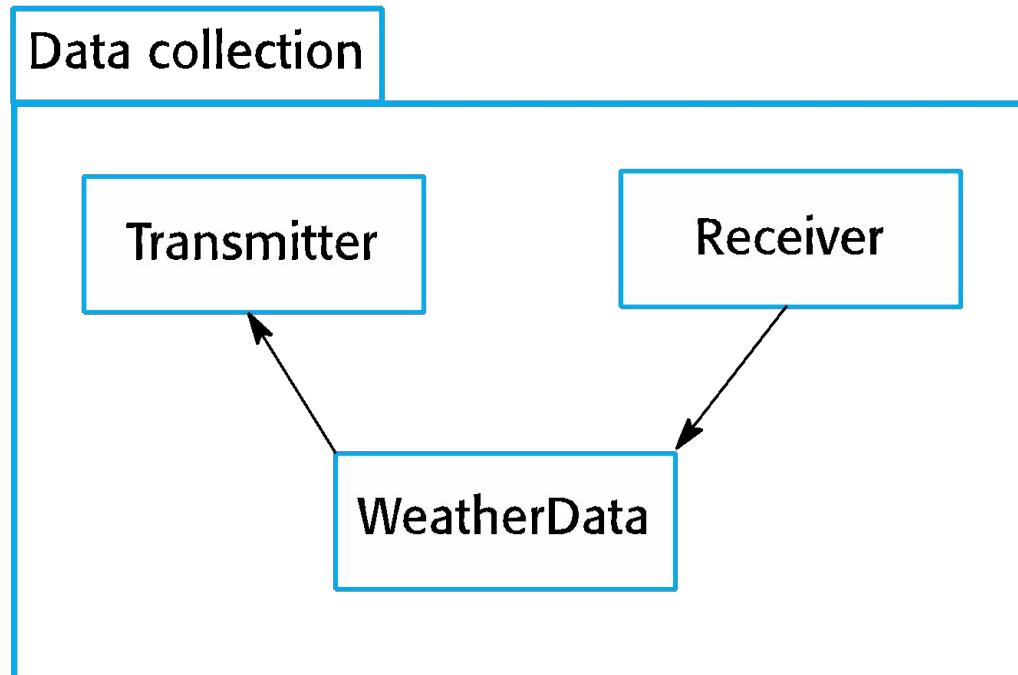
# High-level architecture of the weather station

---



# Architecture of data collection system

---



# Object class identification

---

- ❖ Identifying object classes is often a difficult part of object oriented design.
- ❖ There is no 'magic formula' for object identification. It relies on the skill, experience and domain knowledge of system designers.
- ❖ Object identification is an iterative process. You are unlikely to get it right first time.

# Weather station object classes

---

- ❖ Object class identification in the weather station system may be based on the tangible hardware and data in the system:
  - Ground thermometer, Anemometer, Barometer
    - Application domain objects that are ‘hardware’ objects related to the instruments in the system.
  - Weather station
    - The basic interface of the weather station to its environment. It therefore reflects the interactions identified in the use-case model.
  - Weather data
    - Encapsulates the summarized data from the instruments.

# Design models

---

- ❖ Design models show the objects, object classes and their relationships.
- ❖ There are two kinds of design model:
  - Structural models describe the static structure of the system in terms of object classes and relationships.
  - Dynamic models describe the dynamic interactions between objects.

## Examples of design models

---

- ❖ Subsystem models that show logical groupings of objects into coherent subsystems.
- ❖ Sequence models that show the sequence of object interactions.
- ❖ State machine models that show how individual objects change their state in response to events.
- ❖ Other models include use-case models, aggregation models, generalisation models, etc.

# Interface specification

---

- ❖ Object interfaces have to be specified so that the objects and other components can be designed in parallel.
- ❖ Objects may have several interfaces which are viewpoints on the methods provided.
- ❖ The UML uses class diagrams for interface specification but Java may also be used.

# Weather station interfaces

---

**«interface»**  
**Reporting**

weatherReport (WS-Ident): Wreport  
statusReport (WS-Ident): Sreport

**«interface»**  
**Remote Control**

startInstrument(instrument): iStatus  
stopInstrument (instrument): iStatus  
collectData (instrument): iStatus  
provideData (instrument ): string

---

# **Design patterns**

# Design patterns

---

- ❖ A design pattern is a way of reusing abstract knowledge about a problem and its solution.
- ❖ A pattern is a description of the problem and the essence of its solution.
- ❖ It should be sufficiently abstract to be reused in different settings.
- ❖ Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.

# Patterns

---

- ❖ *Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience.*

# Pattern elements

---

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

# Design Challenges

---

- ❖ Requirements Volatility
- ❖ The Process
- ❖ Technology
- ❖ Tight Deadlines
- ❖ Influences from stakeholders

---

# Implementation

# Implementation

---

- ❖ Software takes shape during the implementation phase.
- ❖ Software design is translated into source code.
- ❖ Programmers are occupied with encoding and designers are involved in developing graphic materials and interfaces.
- ❖ Database designers create the necessary data repositories.

# Implementation issues

---

- ❖ Focus here is not on programming, although this is obviously important, but on other implementation issues that are often not covered in programming texts:
  - **Reuse** Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
  - **Configuration management** During the development process, you have to keep track of the many different versions of each software component in a configuration management system.
  - **Host-target development** Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).

# Reuse

---

- ❖ From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.
  - The only significant reuse of software was the reuse of functions and objects in programming language libraries.
- ❖ Costs and schedule pressure mean that this approach became increasingly unviable, especially for commercial and Internet-based systems.
- ❖ An approach to development based around the reuse of existing software emerged and is now generally used for business and scientific software.

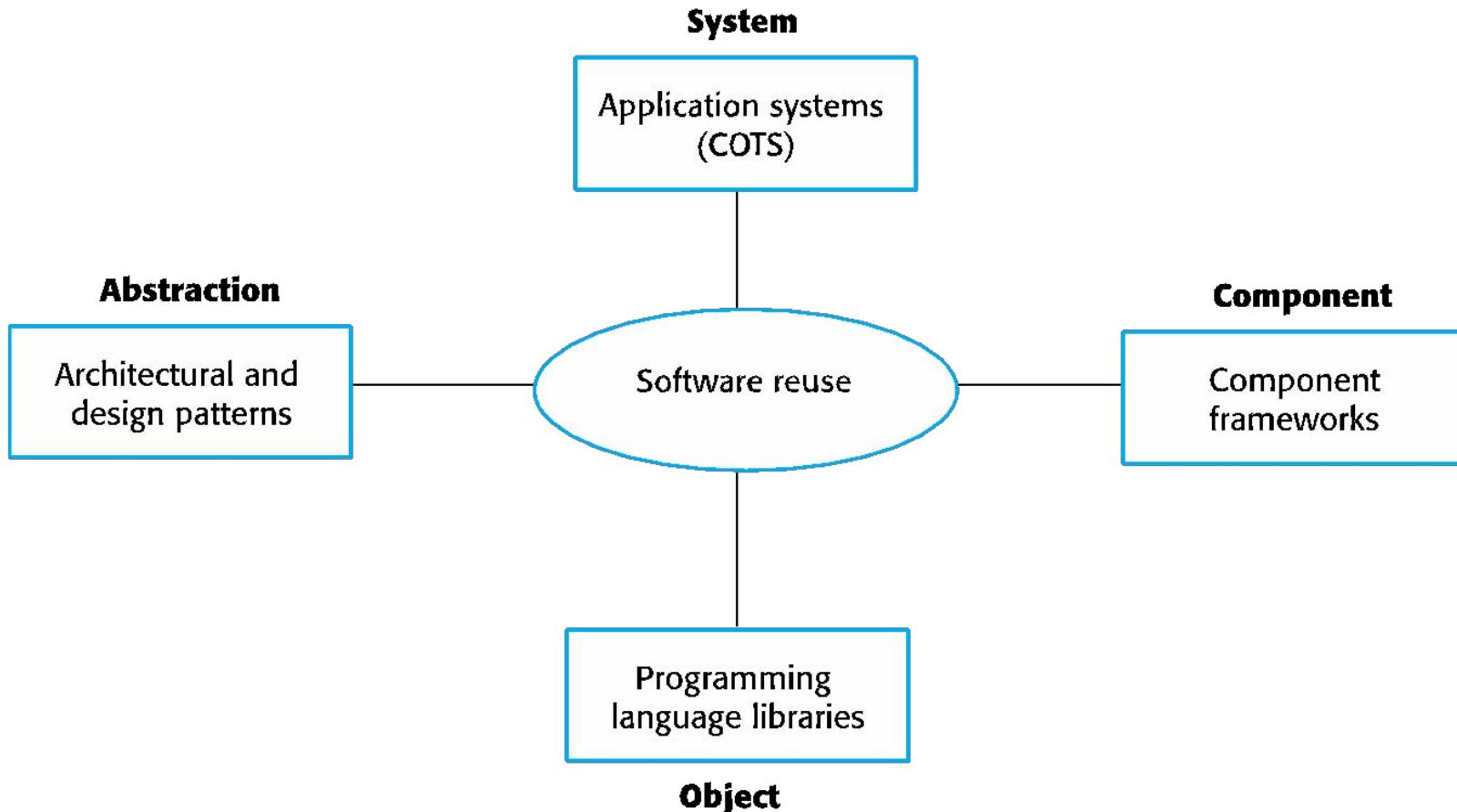
# Reuse levels

---

- ❖ The abstraction level
  - At this level, you don't reuse software directly but use knowledge of successful abstractions in the design of your software.
- ❖ The object level
  - At this level, you directly reuse objects from a library rather than writing the code yourself.
- ❖ The component level
  - Components are collections of objects and object classes that you reuse in application systems.
- ❖ The system level
  - At this level, you reuse entire application systems.

# Software reuse

---



## Reuse costs

---

- ❖ The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.
- ❖ Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
- ❖ The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.
- ❖ The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed.

# Configuration management

---

- ❖ Configuration management is the name given to the general process of managing a changing software system.
- ❖ The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.

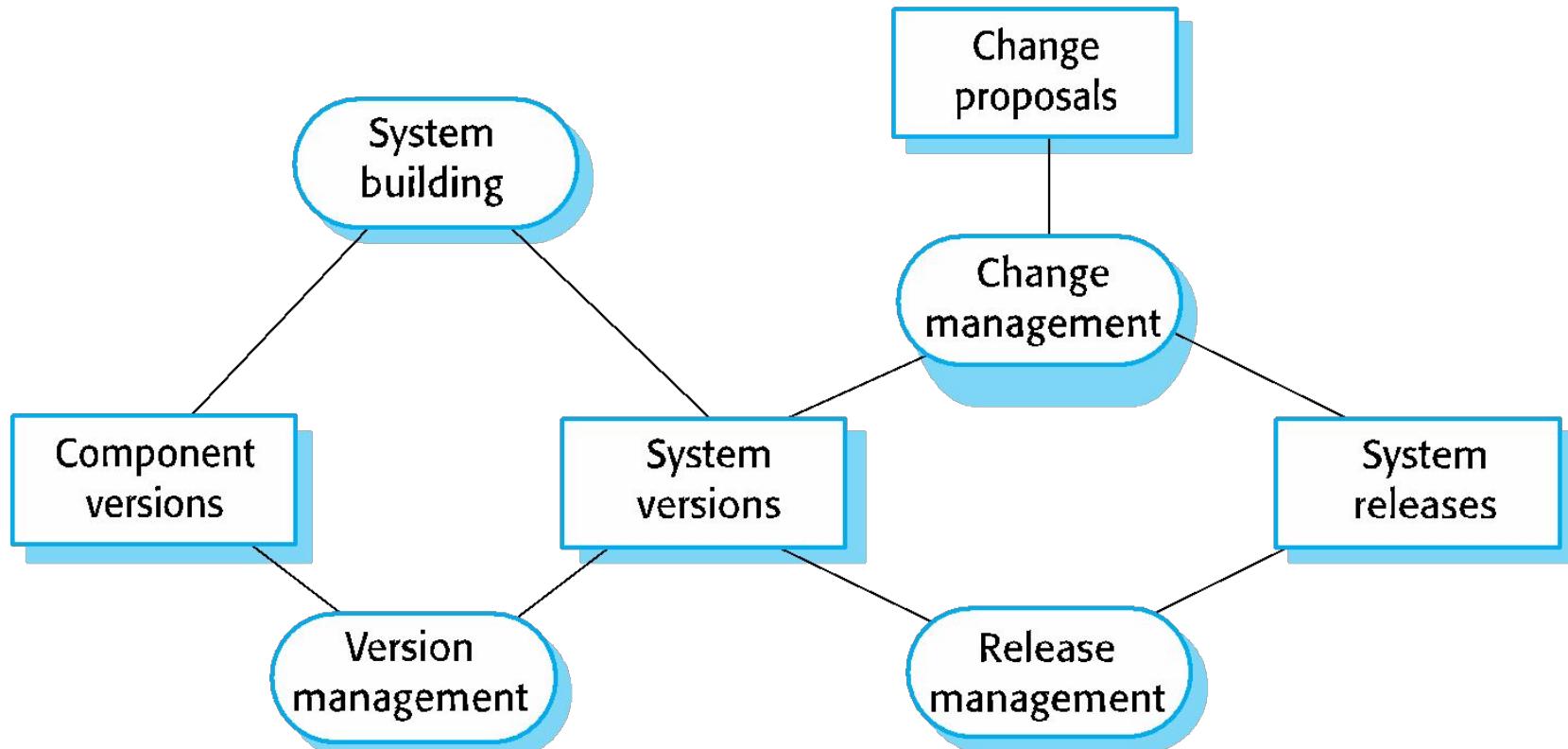
# Configuration management activities

---

- ❖ Version management, where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.
- ❖ System integration, where support is provided to help developers define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.
- ❖ Problem tracking, where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

# Configuration management tool interaction

---



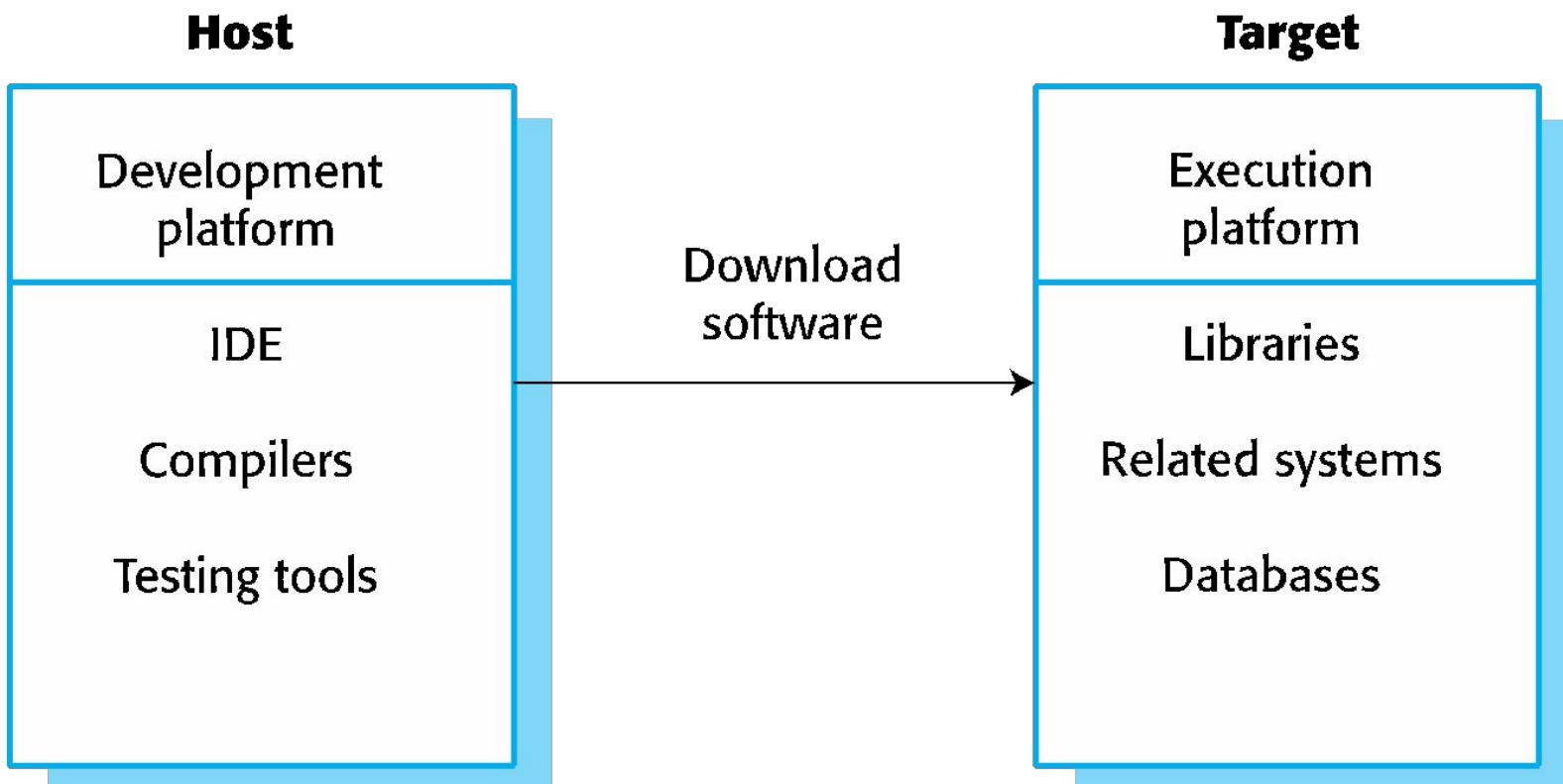
# Host-target development

---

- ❖ Most software is developed on one computer (the host), but runs on a separate machine (the target).
- ❖ More generally, we can talk about a development platform and an execution platform.
  - A platform is more than just hardware.
  - It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.
- ❖ Development platform usually has different installed software than execution platform; these platforms may have different architectures.

# Host-target development

---



# Development platform tools

---

- ❖ An integrated compiler and syntax-directed editing system that allows you to create, edit and compile code.
- ❖ A language debugging system.
- ❖ Graphical editing tools, such as tools to edit UML models.
- ❖ Testing tools, such as Junit that can automatically run a set of tests on a new version of a program.
- ❖ Project support tools that help you organize the code for different development projects.

# Integrated development environments (IDEs)

---

- ❖ Software development tools are often grouped to create an integrated development environment (IDE).
- ❖ An IDE is a set of software tools that supports different aspects of software development, within some common framework and user interface.
- ❖ IDEs are created to support development in a specific programming language such as Java. The language IDE may be developed specially, or may be an instantiation of a general-purpose IDE, with specific language-support tools.

# Component/system deployment factors

---

- ❖ If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.
- ❖ High availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.
- ❖ If there is a high level of communication traffic between components, it usually makes sense to deploy them on the same platform or on platforms that are physically close to one other. This reduces the delay between the time a message is sent by one component and received by another.

---

# **Open source development**

# Open source development

---

- ❖ Open source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process
- ❖ Its roots are in the Free Software Foundation ([www.fsf.org](http://www.fsf.org)), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish.
- ❖ Open source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code.

# Open source systems

---

- ❖ The best-known open source product is, of course, the Linux operating system which is widely used as a server system and, increasingly, as a desktop environment.
- ❖ Other important open source products are Java, the Apache web server and the mySQL database management system.

# Open source issues

---

- ❖ Should the product that is being developed make use of open source components?
- ❖ Should an open source approach be used for the software's development?

# Open source business

---

- ❖ More and more product companies are using an open source approach to development.
- ❖ Their business model is not reliant on selling a software product but on selling support for that product.
- ❖ They believe that involving the open source community will allow software to be developed more cheaply, more quickly and will create a community of users for the software.

# Open source licensing

---

- ❖ A fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code.
  - Legally, the developer of the code (either a company or an individual) still owns the code. They can place restrictions on how it is used by including legally binding conditions in an open source software license.
  - Some open source developers believe that if an open source component is used to develop a new system, then that system should also be open source.
  - Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed source systems.

# License models

---

- ❖ The GNU General Public License (GPL). This is a so-called ‘reciprocal’ license which means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
- ❖ The GNU Lesser General Public License (LGPL) is a variant of the GPL license where you can write components that link to open source code without having to publish the source of these components.
- ❖ The Berkley Standard Distribution (BSD) License. This is a non-reciprocal license, which means you are not obliged to re-publish any changes or modifications made to open source code. You can include the code in proprietary systems that are sold.

# License management

---

- ❖ Establish a system for maintaining information about open-source components that are downloaded and used.
- ❖ Be aware of the different types of licenses and understand how a component is licensed before it is used.
- ❖ Be aware of evolution pathways for components.
- ❖ Educate people about open source.
- ❖ Have auditing systems in place.
- ❖ Participate in the open source community.

# Key points

---

- ❖ Software design and implementation are inter-leaved activities. The level of detail in the design depends on the type of system and whether you are using a plan-driven or agile approach.
- ❖ The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models and document the component interfaces.
- ❖ A range of different models may be produced during an object-oriented design process. These include static models (class models, generalization models, association models) and dynamic models (sequence models, state machine models).
- ❖ Component interfaces must be defined precisely so that other objects can use them.

# Key points

---

- ❖ When developing software, you should always consider the possibility of reusing existing software, either as components, services or complete systems.
- ❖ Configuration management is the process of managing changes to an evolving software system. It is essential when a team of people are cooperating to develop software.
- ❖ Most software development is host-target development. You use an IDE on a host machine to develop the software, which is transferred to a target machine for execution.
- ❖ Open source development involves making the source code of a system publicly available. This means that many people can propose changes and improvements to the software.

# Version Control

# Version Control

1. Software systems are constantly changing during development and use.
2. It is easy to lose track of what changes and component versions have been incorporated into each system version.
3. A version control system allows programmers to keep track of every change.
4. The process of Keeping track of multiple versions of system components is known as version management. Ensures that changes made by different developers will not interfere with each other.

# Multi-version Systems

1. For large systems, there is never just one ‘working’ version of a system.
2. There are always several versions of the system at different stages of development.
3. There may be several teams involved in the development of different system versions.

# Codelines and Baselines

1. A codeline is a sequence of versions of source code with later versions in the sequence derived from earlier versions.
2. Codelines normally apply to components of systems so that there are different versions of each component.
3. A baseline is a definition of a specific system.
4. The baseline therefore specifies the component versions that are included in the system plus a specification of the libraries used, configuration files, etc.

# Baselines

1. Baselines may be specified using a configuration language, which allows you to define what components are included in a version of a particular system.
2. Baselines are important because you often have to recreate a specific version of a complete system.
  - a. For example, a product line may be instantiated so that there are individual system versions for different customers. You may have to recreate the version delivered to a specific customer if, for example, that customer reports bugs in their system that have to be repaired.

# Version Control Systems

1. Version control (VC) systems help us to identify, store and control access to the different versions of system components. There are two types of modern version control system:
  - a. Centralized systems, where there is a single master repository that maintains all versions of the software components that are being developed. Subversion is a widely used example of a centralized VC system.
  - b. Distributed systems, where multiple versions of the component repository exist at the same time. Git is a widely-used example of a distributed VC system.

# Key Features of Version Control Systems

1. Version and release identification
2. Change history recording
3. Support for independent development
4. Project support
5. Storage management

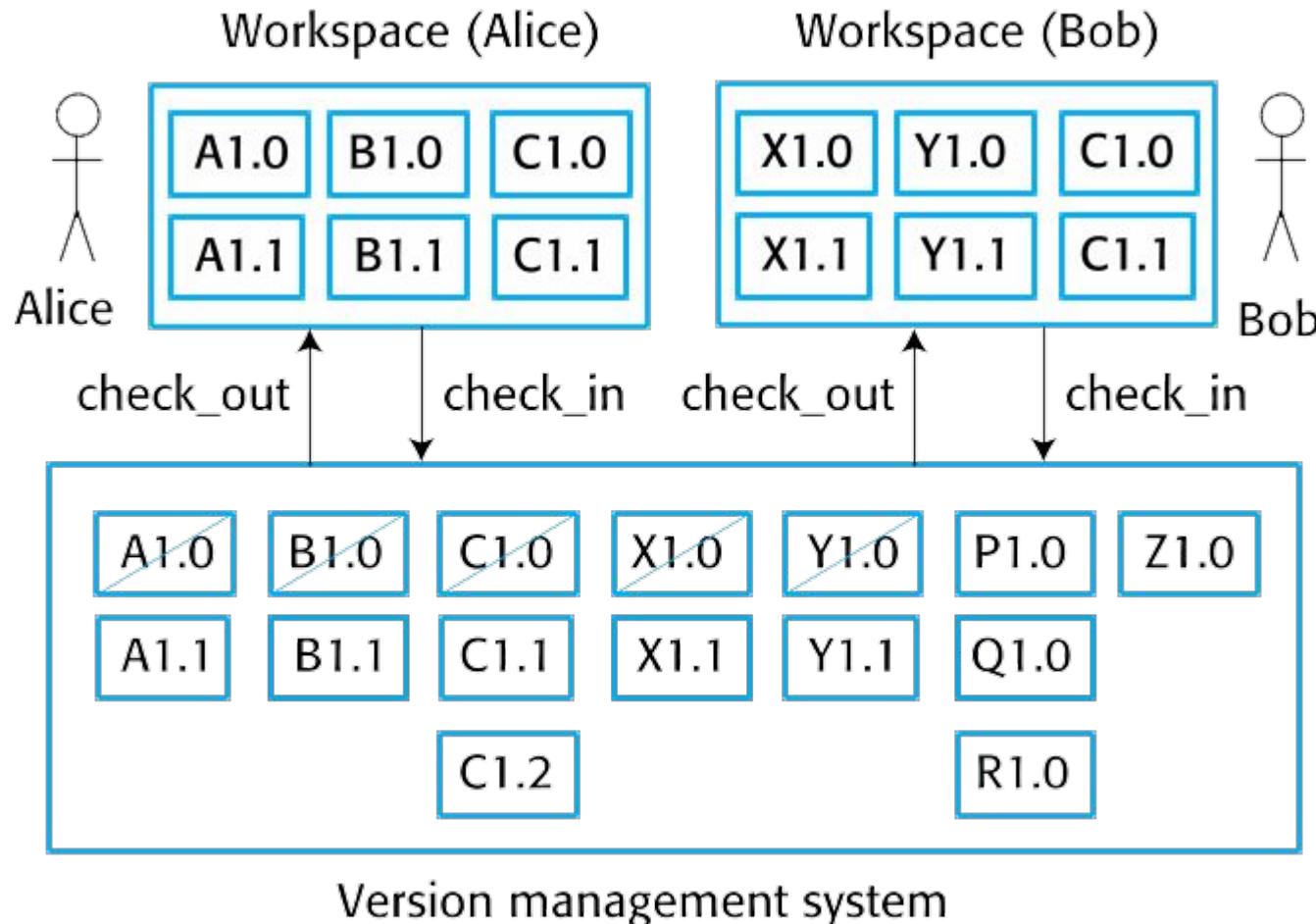
# Public Repository and Private Workspaces

1. To support independent development without interference, version control systems use the concept of a project repository and a private workspace.
2. The project repository maintains the ‘master’ version of all components. It is used to create baselines for system building.
3. When modifying components, developers copy (check-out) these from the repository into their workspace and work on these copies.
4. When they have finished their changes, the changed components are returned (checked-in) to the repository.

# Centralized Version Control

1. Developers check out components or directories of components from the project repository into their private workspace and work on these copies in their private workspace.
2. When their changes are complete, they check-in the components back to the repository.
3. If several people are working on a component at the same time, each check it out from the repository. If a component has been checked out, the VC system warns other users wanting to check out that component that it has been checked out by someone else.

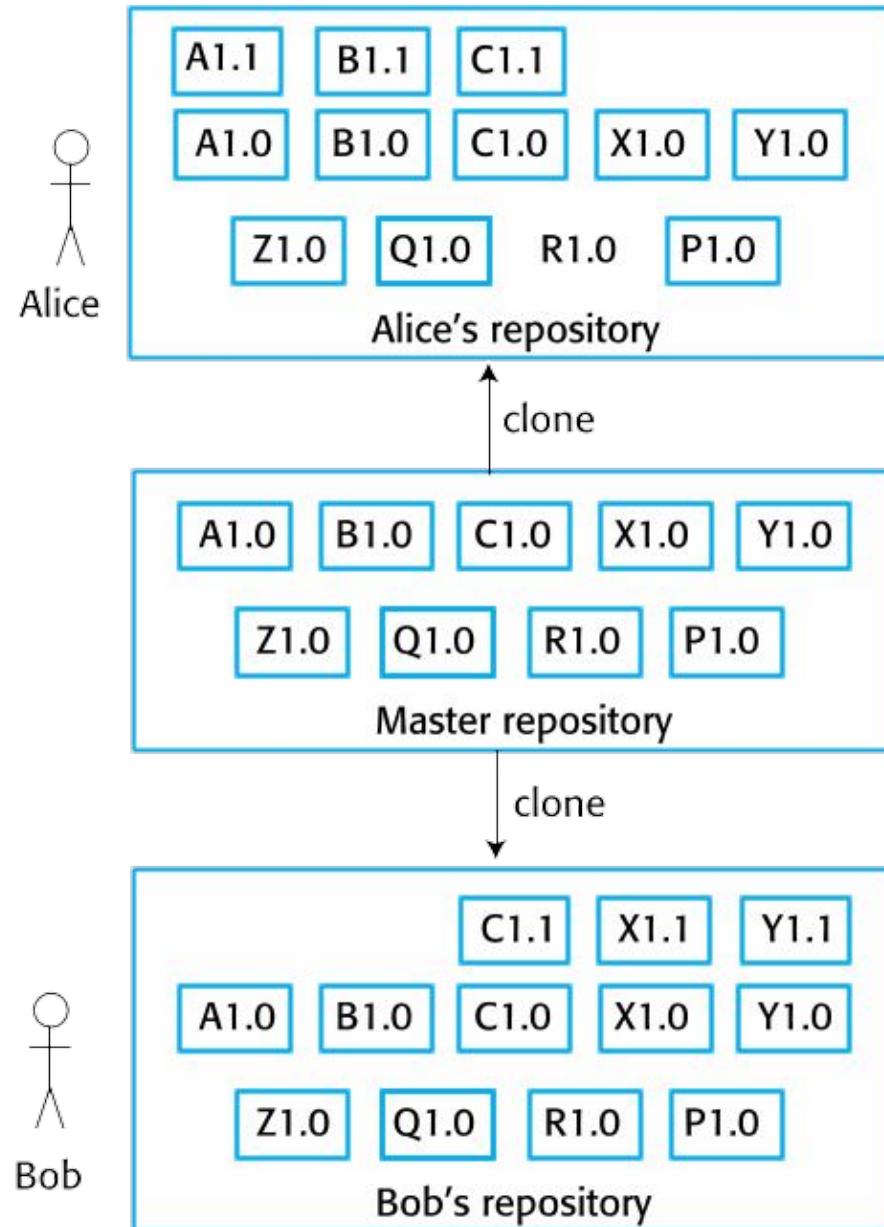
# Centralized Version Control



# Distributed Version Control

1. A ‘master’ repository is created on a server that maintains the code produced by the development team.
2. Instead of checking out the files that they need, a developer creates a clone of the project repository that is downloaded and installed on their computer.
3. Developers work on the files required and maintain the new versions on their private repository on their own computer.
4. When changes are done, they ‘commit’ these changes and update their private server repository. They may then ‘push’ these changes to the project repository.

# Distributed Version Control



# Benefits of Distributed Version Control

1. It provides a backup mechanism for the repository.
  - a. If the repository is corrupted, work can continue and the project repository can be restored from local copies.
2. It allows for off-line working so that developers can commit changes if they do not have a network connection.
3. Project support is the default way of working.
  - a. Developers can compile and test the entire system on their local machines and test the changes that they have made.

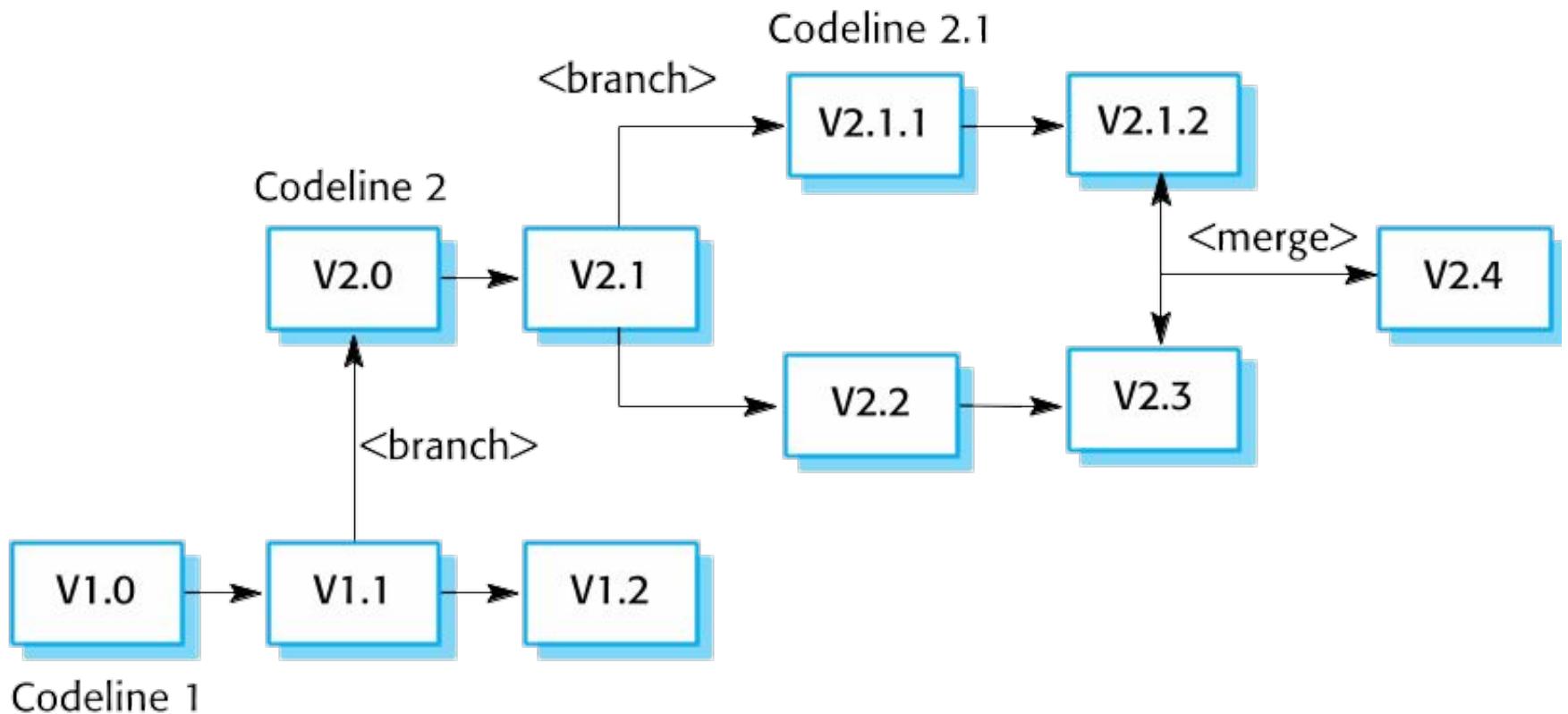
# Open Source Development

1. Distributed version control is essential for open source development.
  - a. Several people may be working simultaneously on the same system without any central coordination.
2. As well as a private repository on their own computer, developers also maintain a public server repository to which they push new versions of components that they have changed.
  - a. It is then up to the open-source system ‘manager’ to decide when to pull these changes into the definitive system.

# Branching and Merging

1. Rather than a linear sequence of versions that reflect changes to the component over time, there may be several independent sequences.
  - a. This is normal in system development, where different developers work independently on different versions of the source code and so change it in different ways.
2. At some stage, it may be necessary to merge codeline branches to create a new version of a component that includes all changes that have been made.

# Branching and Merging



# Storage Management

1. When version control systems were first developed, storage management was one of their most important functions.
2. Disk space was expensive and it was important to minimize the disk space used by the different copies of components.
3. Instead of keeping a complete copy of each version, the system stores a list of differences (deltas) between one version and another.
  - a. By applying these to a master version (usually the most recent version), a target version can be recreated.

# Storage Management in Git

1. As disk storage is now relatively cheap, Git uses an alternative, faster approach.
2. Git does not use deltas but applies a standard compression algorithm to stored files and their associated meta-information.
3. Retrieving a file simply involves decompressing it, with no need to apply a chain of operations.
4. Git also uses the notion of packfiles where several smaller files are combined into an indexed single file.

# System Building

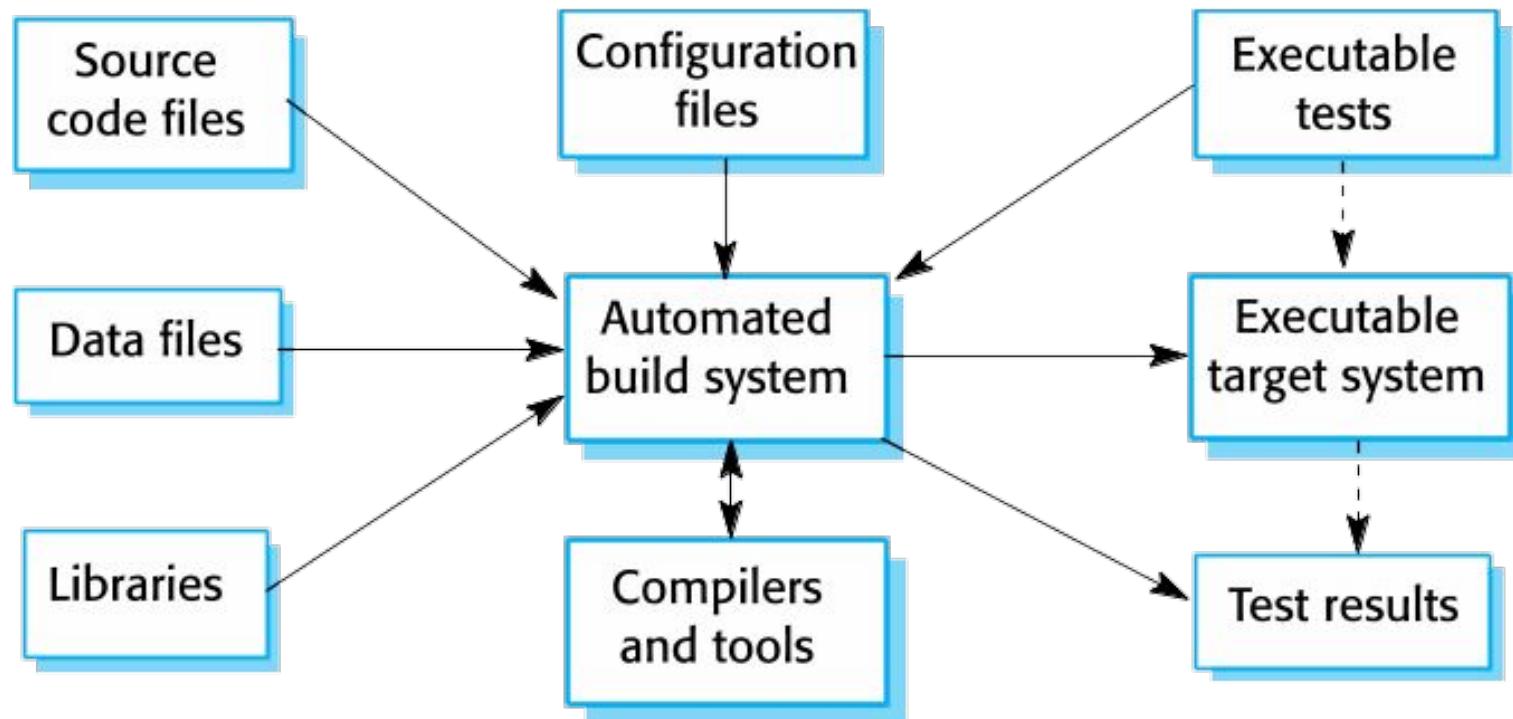
# System Building

1. System building is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, etc.
2. System building tools and version management tools must communicate as the build process involves checking out component versions from the repository managed by the version management system.
3. The configuration description used to identify a baseline is also used by the system building tool.

# Build Platforms

1. The development system, which includes development tools such as compilers, source code editors, etc.
  - a. Developers check out code from the version management system into a private workspace before making changes to the system.
2. The build server, which is used to build definitive, executable versions of the system.
  - a. Developers check-in code to the version management system before it is built. The system build may rely on external libraries that are not included in the version management system.
3. The target environment, which is the platform on which the system executes.

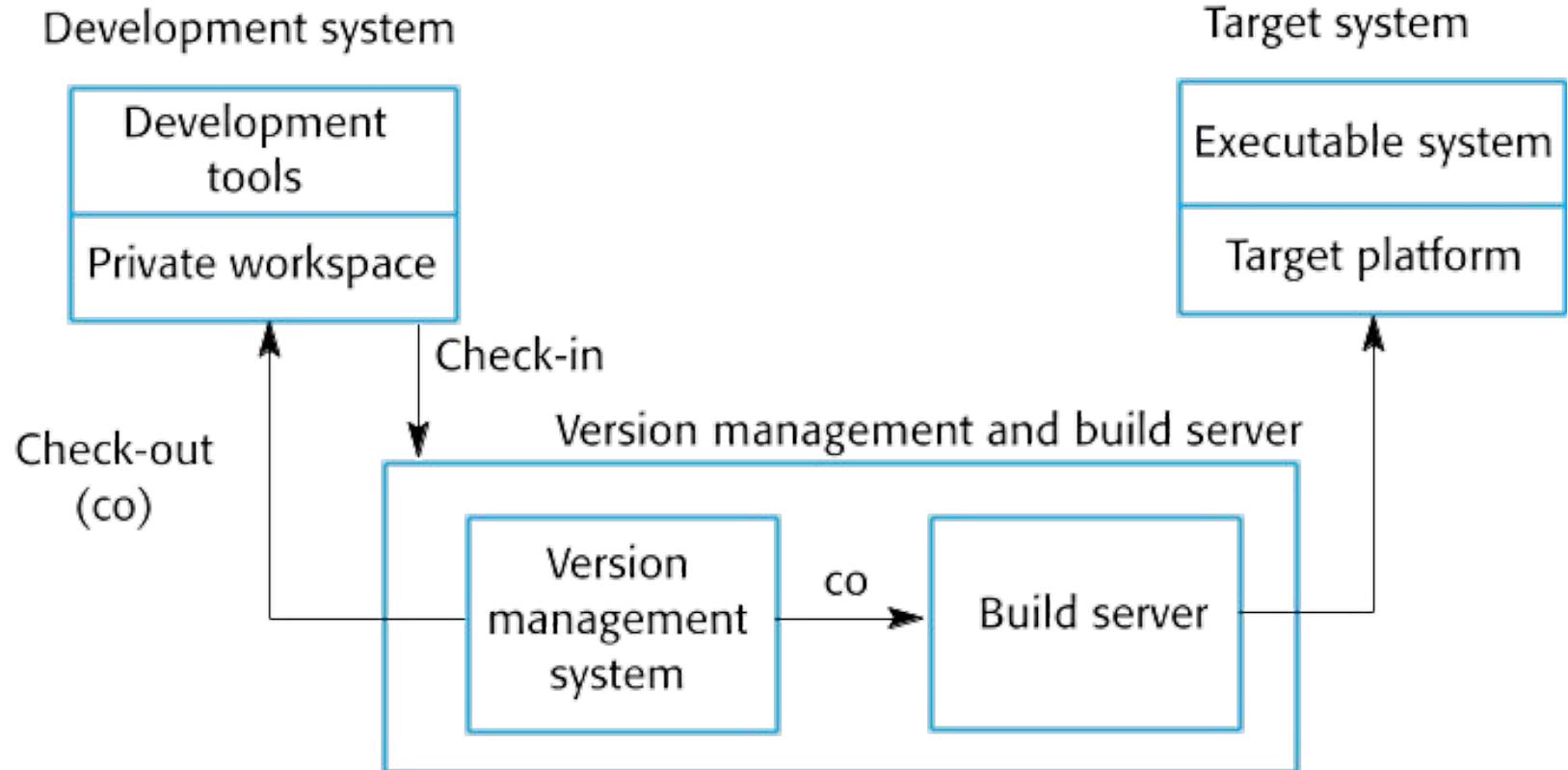
# System Building



# Build System Functionality

1. Build script generation
2. Version management system integration
3. Minimal re-compilation
4. Executable system creation
5. Test automation
6. Reporting
7. Documentation generation

# Development, Build, and Target Platforms



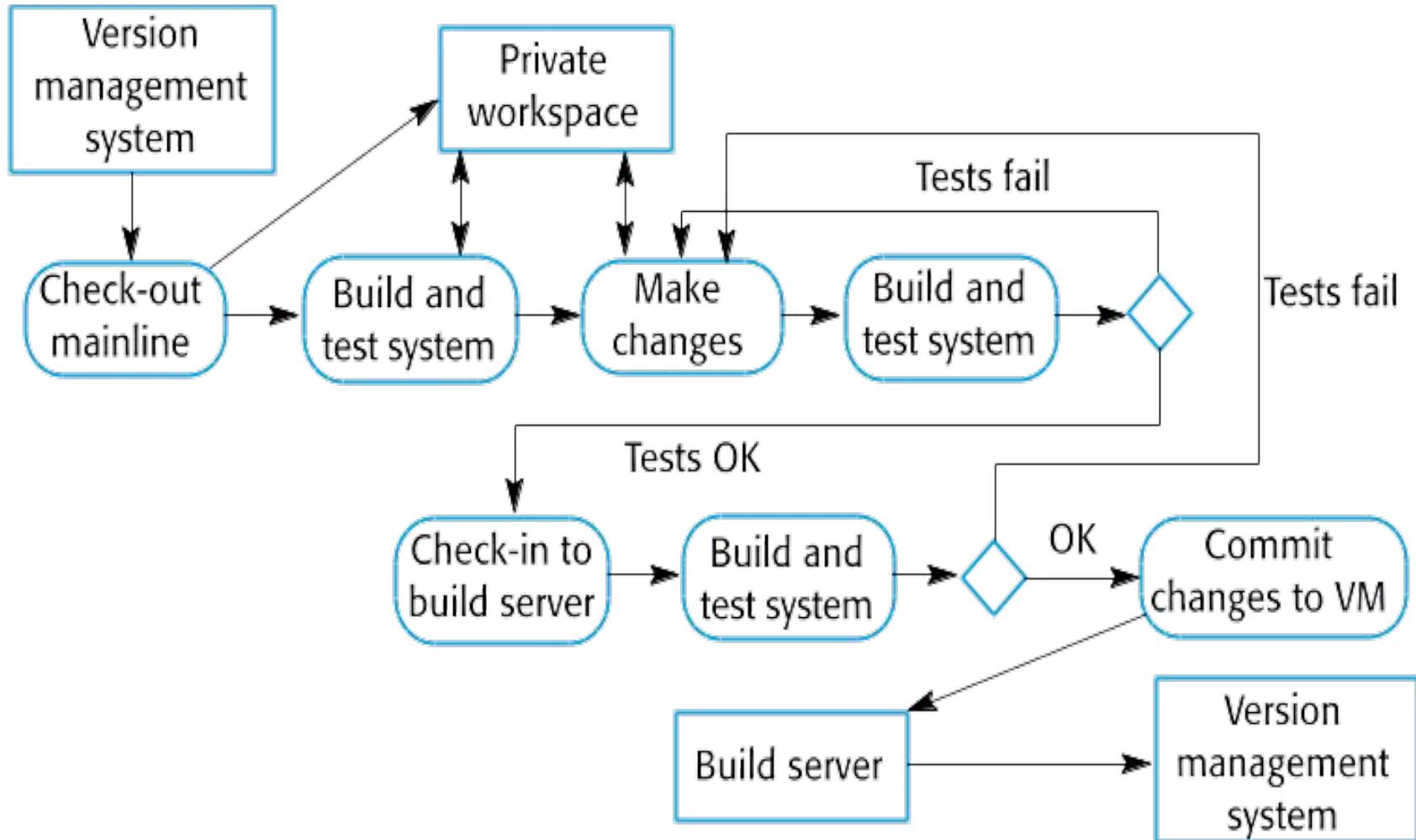
# Agile Building

1. Check out the mainline system from the version management system into the developer's private workspace.
2. Build the system and run automated tests to ensure that the built system passes all tests. If not, the build is broken and you should inform whoever checked in the last baseline system. They are responsible for repairing the problem.
3. Make the changes to the system components.
4. Build the system in the private workspace and rerun system tests. If the tests fail, continue editing.

# Agile Building

1. Once the system has passed its tests, check it into the build system but do not commit it as a new system baseline.
2. Build the system on the build server and run the tests. You need to do this in case others have modified components since you checked out the system. If this is the case, check out the components that have failed and edit these so that tests pass on your private workspace.
3. If the system passes its tests on the build system, then commit the changes you have made as a new baseline in the system mainline.

# Continuous Integration



# Pros and Cons of Continuous Integration

## Pros:

- The advantage of continuous integration is that it allows problems caused by the interactions between different developers to be discovered and repaired as soon as possible.
- The most recent system in the mainline is the definitive working system.

## Cons:

- If the system is very large, it may take a long time to build and test, especially if integration with other application systems is involved.

# Daily Building

1. The development organization sets a delivery time (say 2 p.m.) for system components.
  - a. If developers have new versions of the components that they are writing, they must deliver them by that time.
  - b. A new version of the system is built from these components by compiling and linking them to form a complete system.
  - c. This system is then delivered to the testing team, which carries out a set of predefined system tests
  - d. Faults that are discovered during system testing are documented and returned to the system developers. They repair these faults in a subsequent version of the component.

# Minimizing Recompilation

1. Tools to support system building are usually designed to minimize the amount of compilation that is required.
2. They do this by checking if a compiled version of a component is available. If so, there is no need to recompile that component.
3. A unique signature identifies each source and object code version and is changed when the source code is edited.
4. By comparing the signatures on the source and object code files, it is possible to decide if the source code was used to generate the object code component.

# **File Identification**

## **Modification Timestamps:**

1. The signature on the source code file is the time and date when that file was modified. If the source code file of a component has been modified after the related object code file, then the system assumes that recompilation to create a new object code file is necessary.

## **Source Code Checksums:**

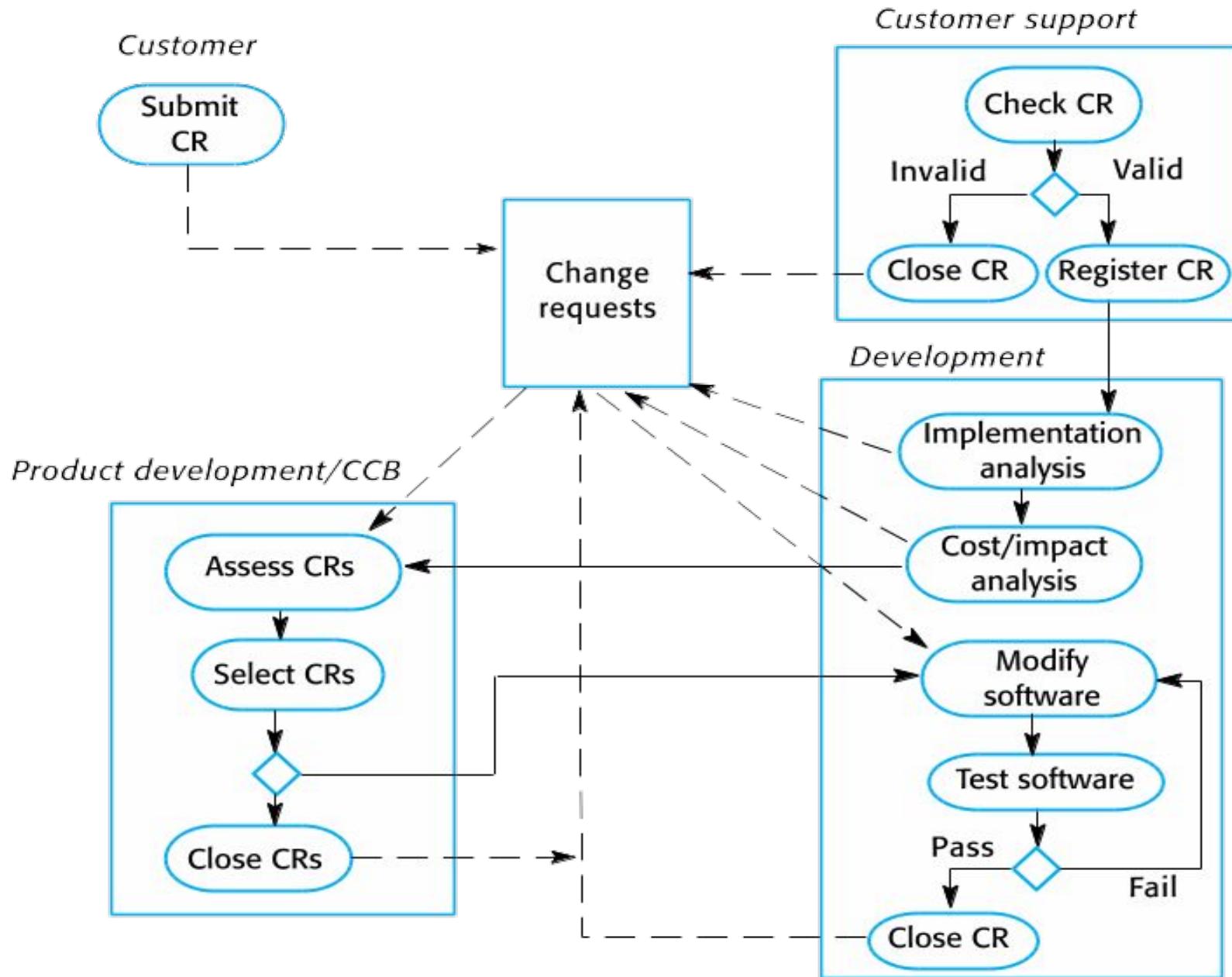
2. The signature on the source code file is a checksum calculated from data in the file. A checksum function calculates a unique number using the source text as input. If you change the source code (even by 1 character), this will generate a different checksum. You can therefore be confident that source code files with different checksums are actually different.

# Change Management

# Change Management

1. Organizational needs and requirements change during the lifetime of a system, bugs have to be repaired and systems have to adapt to changes in their environment.
2. Change management is intended to ensure that system evolution is a managed process and that priority is given to the most urgent and cost-effective changes.
3. The change management process is concerned with analyzing the costs and benefits of proposed changes, approving those changes that are worthwhile and tracking which components in the system have been changed.

# Change Management Process



# Change Management Process

## Change Request Form

**Project:** SICSA/AppProcessing

**Number:** 23/02

**Change requester:** I. Sommerville

**Date:** 20/01/09

**Requested change:** The status of applicants (rejected, accepted, etc.) should be shown visually in the displayed list of applicants.

**Change analyzer:** R. Looek

**Analysis date:** 25/01/09

**Components affected:** ApplicantListDisplay, StatusUpdater

**Associated components:** StudentDatabase

**Change assessment:** Relatively simple to implement by changing the display color according to status. A table must be added to relate status to colors. No changes to associated components are required.

**Change priority:** Medium

**Change implementation:**

**Estimated effort:** 2 hours

**Date to SGA app. team:** 28/01/09

**CCB decision date:** 30/01/09

**Decision:** Accept change. Change to be implemented in Release 1.2

**Change implementor:**

**Date of change:**

**Date submitted to QA:**

**QA decision:**

**Date submitted to CM:**

**Comments:**

# Factors in change analysis

1. The consequences of not making the change
2. The benefits of the change
3. The number of users affected by the change
4. The costs of making the change
5. The product release cycle

# Change management and Agile methods

1. In some agile methods, customers are directly involved in change management.
2. They propose a change to the requirements and work with the team to assess its impact and decide whether the change should take priority over the features planned for the next increment of the system.
3. Changes to improve the software are decided by the programmers working on the system.
4. Refactoring, where the software is continually improved, is not seen as an overhead but as a necessary part of the development process.

# Release Management

# Release Management

1. A system release is a version of a software system that is distributed to customers.
2. For mass market software, it is usually possible to identify two types of release: major releases which deliver significant new functionality, and minor releases, which repair bugs and fix customer problems that have been reported.
3. For custom software or software product lines, releases of the system may have to be produced for each customer and individual customers may be running several different releases of the system at the same time.

# Release Components

1. As well as the executable code of the system, a release may also include:
  - a. configuration files defining how the release should be configured for particular installations;
  - b. data files, such as files of error messages, that are needed for successful system operation;
  - c. an installation program that is used to help install the system on target hardware;
  - d. electronic and paper documentation describing the system;
  - e. packaging and associated publicity that have been designed for that release.

# Factors Influencing System Release Planning

| Factor                          | Description                                                                                                                                                                                                                                                                                                      |
|---------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Competition                     | For mass-market software, a new system release may be necessary because a competing product has introduced new features and market share may be lost if these are not provided to existing customers.                                                                                                            |
| Marketing requirements          | The marketing department of an organization may have made a commitment for releases to be available at a particular date.                                                                                                                                                                                        |
| Platform changes                | You may have to create a new release of a software application when a new version of the operating system platform is released.                                                                                                                                                                                  |
| Technical quality of the system | If serious system faults are reported which affect the way in which many customers use the system, it may be necessary to issue a fault repair release. Minor system faults may be repaired by issuing patches (usually distributed over the Internet) that can be applied to the current release of the system. |

# Release Creation

1. The executable code of the programs and all associated data files must be identified in the version control system.
2. Configuration descriptions may have to be written for different hardware and operating systems.
3. Update instructions may have to be written for customers who need to configure their own systems.
4. Scripts for the installation program may have to be written.
5. Web pages have to be created describing the release, with links to system documentation.
6. When all information is available, an executable master image of the software must be prepared and handed over for distribution to customers or sales outlets.

# Release Tracking

1. In the event of a problem, it may be necessary to reproduce exactly the software that has been delivered to a particular customer.
2. When a system release is produced, it must be documented to ensure that it can be re-created exactly in the future.
3. This is particularly important for customized, long-lifetime embedded systems, such as those that control complex machines.
  - a. Customers may use a single release of these systems for many years and may require specific changes to a particular software system long after its original release date.

# Release Reproduction

1. To document a release, you have to record the specific versions of the source code components that were used to create the executable code.
2. You must keep copies of the source code files, corresponding executables and all data and configuration files.
3. You should also record the versions of the operating system, libraries, compilers and other tools used to build the software.

# Release Planning

1. As well as the technical work involved in creating a release distribution, advertising and publicity material have to be prepared and marketing strategies put in place to convince customers to buy the new release of the system.
2. Release timing:
  - a. If releases are too frequent or require hardware upgrades, customers may not move to the new release, especially if they have to pay for it.
  - b. If system releases are too infrequent, market share may be lost as customers move to alternative systems.

# Software as a Service

1. Delivering software as a service (SaaS) reduces the problems of release management.
2. It simplifies both release management and system installation for customers.
3. The software developer is responsible for replacing the existing release of a system with a new release and this is made available to all customers at the same time.

# Conclusion

- Configuration management is the management of an evolving software system. When maintaining a system, a CM team is put in place to ensure that changes are incorporated into the system in a controlled way and that records are maintained with details of the changes that have been implemented.
- The main configuration management processes are concerned with version management, system building, change management, and release management.
- Version management involves keeping track of the different versions of software components as changes are made to them.

# Conclusion

- System building is the process of assembling system components into an executable program to run on a target computer system.
- Software should be frequently rebuilt and tested immediately after a new version has been built. This makes it easier to detect bugs and problems that have been introduced since the last build.
- Change management involves assessing proposals for changes from system customers and other stakeholders and deciding if it is cost-effective to implement these in a new version of a system.
- System releases include executable code, data files, configuration files and documentation. Release management involves making decisions on system release dates, preparing all information for distribution and documenting each system release.

# The COCOMO model

- > Developed by Barry Boehm in 1970
- > Based on a *cost database* of more than 60 different projects
- > Exists in three stages
  1. **Basic** — Gives a “ballpark” estimate based on product attributes
  2. **Intermediate** — Modifies basic estimate using project and process attributes
  3. **Advanced** — Estimates project phases and parts separately

# Basic COCOMO Formula

- >  $\text{Effort} = C \times PM^S \times M$ 
  - Effort is measured in person-months
  - C is a *complexity factor*
  - PM is a product metric (size or functionality, usually KLOC)
  - exponent S is close to 1, but increasing for large projects
  - M is a multiplier based on process, product and development attributes (~ 1)

# COCOMO Project classes

|                                                                                                                                                                                                                                     |                                       |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------|
| <p><b>Organic mode:</b> small teams, familiar environment, well-understood applications, no difficult non-functional requirements (EASY) 2-50 KLOC GUI</p>                                                                          | $Effort = 2.4 (KDSI)^{1.05} \times M$ |
| <p><b>Semi-detached mode:</b> Project team may have experience mixture, system may have more significant non-functional constraints, organization may have less familiarity with application (HARDER) 50-300 KLOC Database Part</p> | $Effort = 3 (KDSI)^{1.12} \times M$   |
| <p><b>Embedded:</b> Hardware/software systems, tight constraints, unusual for team to have deep application experience (HARD) above 300 KLOC Communication</p>                                                                      | $Effort = 3.6 (KDSI)^{1.2} \times M$  |

KDSI = Kilo Delivered Source Instructions

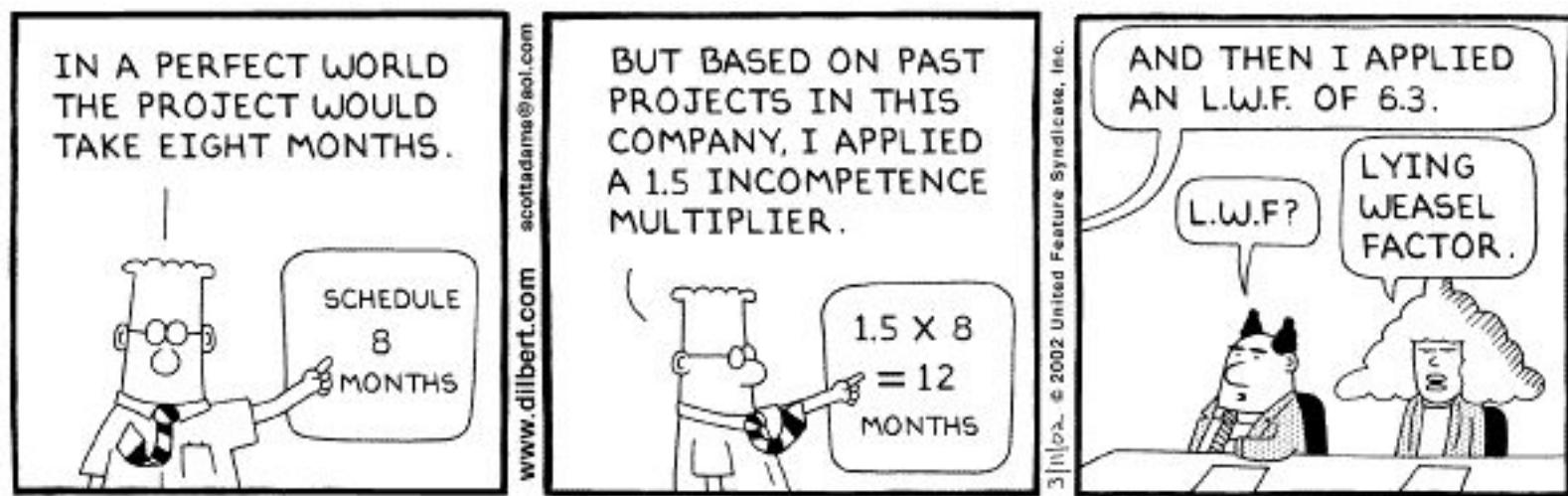
# COCOMO Project classes

|                                                                                                                                                                                                           |                              |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------|
| <p><b>Organic mode:</b> small teams, familiar environment, well-understood applications, no difficult non-functional requirements (EASY)</p>                                                              | $Time = 2.5 (Effort)^{0.38}$ |
| <p><b>Semi-detached mode:</b> Project team may have experience mixture, system may have more significant non-functional constraints, organization may have less familiarity with application (HARDER)</p> | $Time = 2.5 (Effort)^{0.35}$ |
| <p><b>Embedded:</b> Hardware/software systems, tight constraints, unusual for team to have deep application experience (HARD)</p>                                                                         | $Time = 2.5 (Effort)^{0.32}$ |

KDSI = Kilo Delivered Source Instructions

# COCOMO assumptions and problems

- > Time required is a function of total effort *not team size*
- > Not clear how to adapt model to *personnel availability*



Copyright © 2002 United Feature Syndicate, Inc.

# COCOMO assumptions and problems ...

- > *Staff required* can't be computed by dividing the development time by the required schedule
- > The number of people working on a project varies depending on the *phase of the project*
- > The more people who work on the project, the *more total effort* is usually required (!)

# Software Measurement

# What is measurement?

Measurement is the process by which numbers or symbols are assigned to attributes of entities in the world **according to clearly defined rules**.

# Some Questions

1. Can software be measured?
2. Is it software measurement useful?
3. How do you measure software?

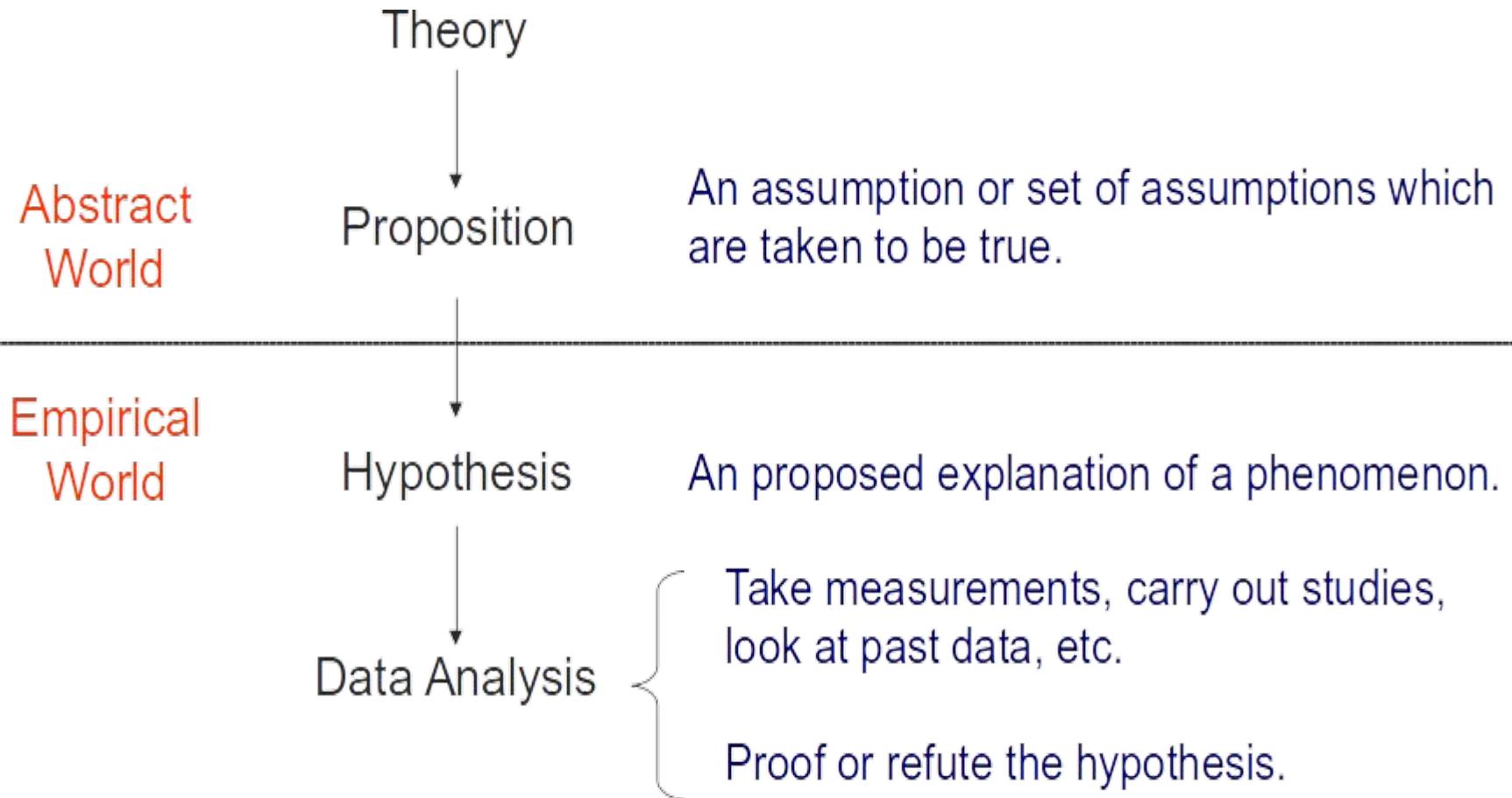
# Importance

1. Measurement is crucial to the progress of all sciences, even Computer Science
2. Scientific progress is made through
  - a. Observations and generalisations...
  - b. ...based on data and measurements
  - c. Derivation of theories and...
  - d. ...confirmation or refutation of these theories
3. Measurement turns an art into a science

# Uses

1. Measurement helps us to understand
  - a. Makes the current activity visible
  - b. Measures establish guidelines
2. Measurement allows us to control
  - a. Predict outcomes and change processes
3. Measurement encourages us to improve
  - a. When we hold our product up to a measuring stick, we can establish quality targets and aim to improve

# Preposition and Hypothesis



# Preposition and Hypothesis: Example

**Preposition:** An increase in student intelligence causes an increase in their academic achievement.

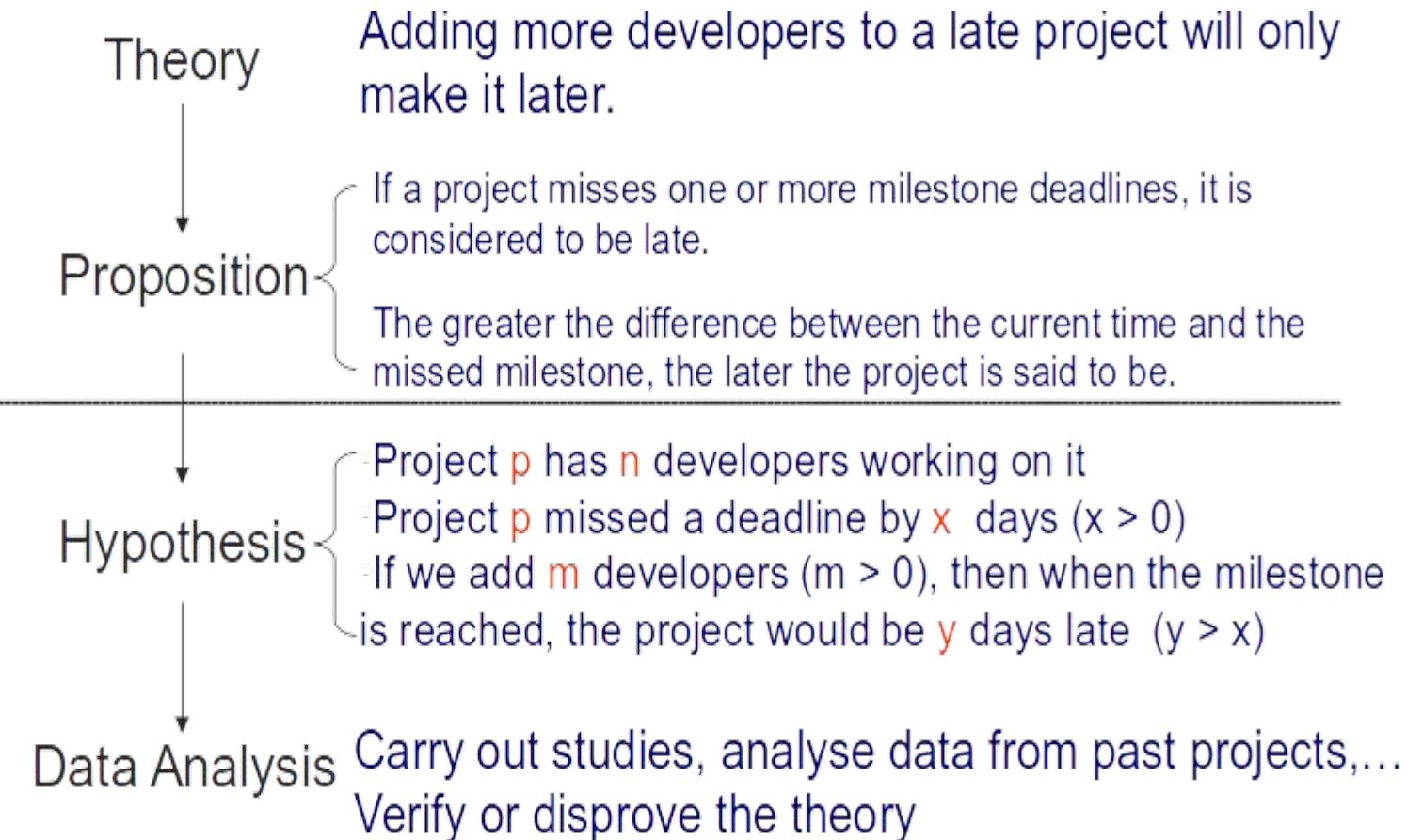
**Hypothesis:** An increase in students' IQ score causes an increase in their grade point average.

(Assuming, IQ scores and grade point average are operational measures of intelligence and academic achievement.)

# Proving/Disproving a Theory

Abstract  
World

Empirical  
World



# Measures, Metrics and Indicators

**Measure** – A "measure" is a number that is derived from taking a measurement. Your height, weight or temperature would all be measures. E.g. Joe's body temperature is 99° fahrenheit.

**Metric** – A "metric" is a calculation between two measures. Typically, the calculation is a form of division. E.g. 2 errors were discovered by customers in 18 months.

**Indicator** – A device, variable or metric can indicate whether a particular state or goal has been achieved. Usually used to draw someone's attention to something. E.g. A half-mast flag indicates that someone has died.

# Software Measurement

- What makes quality software?
  - a. Cheap?
  - b. Reliable?
  - c. Testable?
  - d. Secure?
  - e. Maintainable?

# No Clear-Cut Answer

- It depends on:
  - a. Stakeholders
  - b. Type of system
  - c. Type of users
  - d. ...
- Quality is a multifaceted concept.

# Different Quality Scenarios

- Online banking system
  - a. Security
  - b. Correctness
  - c. Reliability
- Air Traffic Control System
  - a. Robustness
  - b. Real Time Responses
- Educational Game for Children
  - a. User Friendliness

# 3 Ps of Software Measurement

- With regards to software, we can measure:
  - a. Product
  - b. Process
  - c. People

# Measuring the Product

- Product refers to the actual software system, documentation and other deliverables
- We examine the product and measure a number of aspects:
  - a. Size
  - b. Functionality offered
  - c. Cost
  - d. Various Quality Attributes

# Measuring the Process

- Involves analysis of the way a product is developed
- What lifecycle do we use?
- What deliverables are produced?
- How are they analysed?
- How can the process help to produce products faster?
- How can the process help to produce better products?

# Measuring the People

- Involves analysis of the people developing a product
- How fast do they work?
- How much bugs do they produce?
- How many sick-days do they take?
- Very controversial. People do not like being turned into numbers.

# Product Metrics

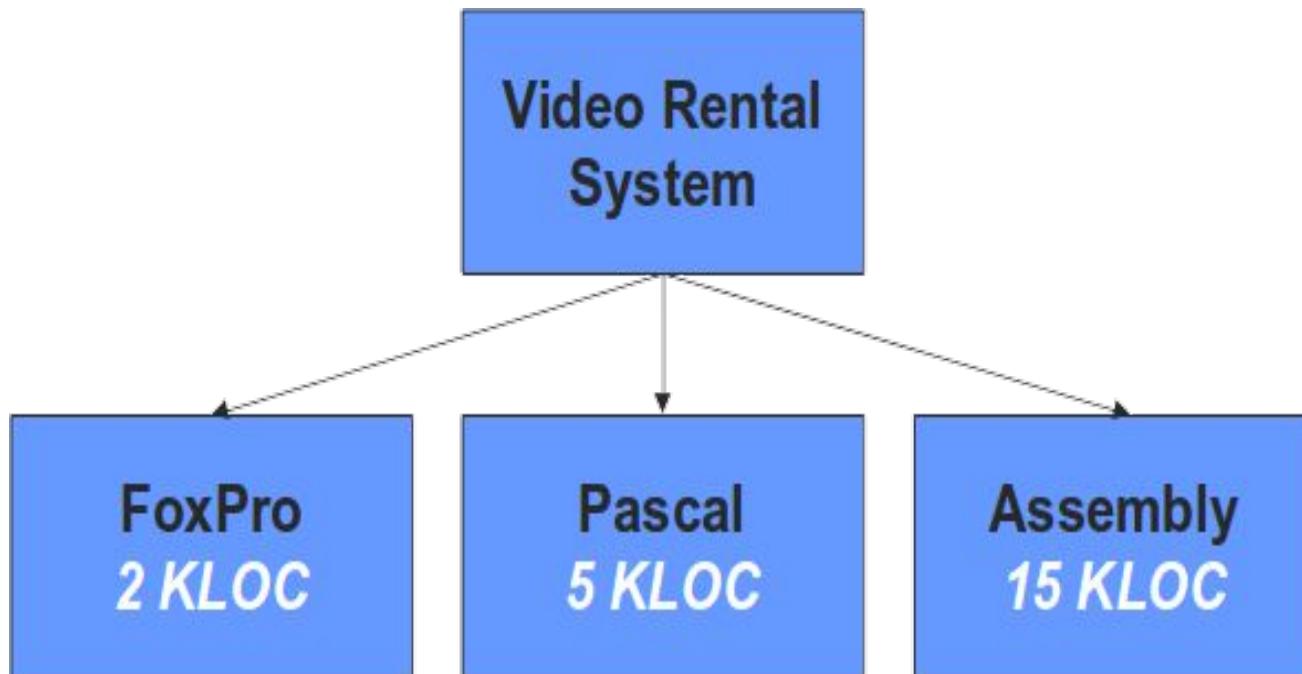
- What can we measure about a product?
  - a. Size metrics
  - b. Defects-based metrics
  - c. Cost-metrics
  - d. Time metrics
  - e. Quality Attribute metrics

# Size Metrics

- Knowing the size of a system was important for comparing different systems together.
- Software measured in lines of code (LOC).
- As systems grew larger KLOC (thousands of lines of code) also used.

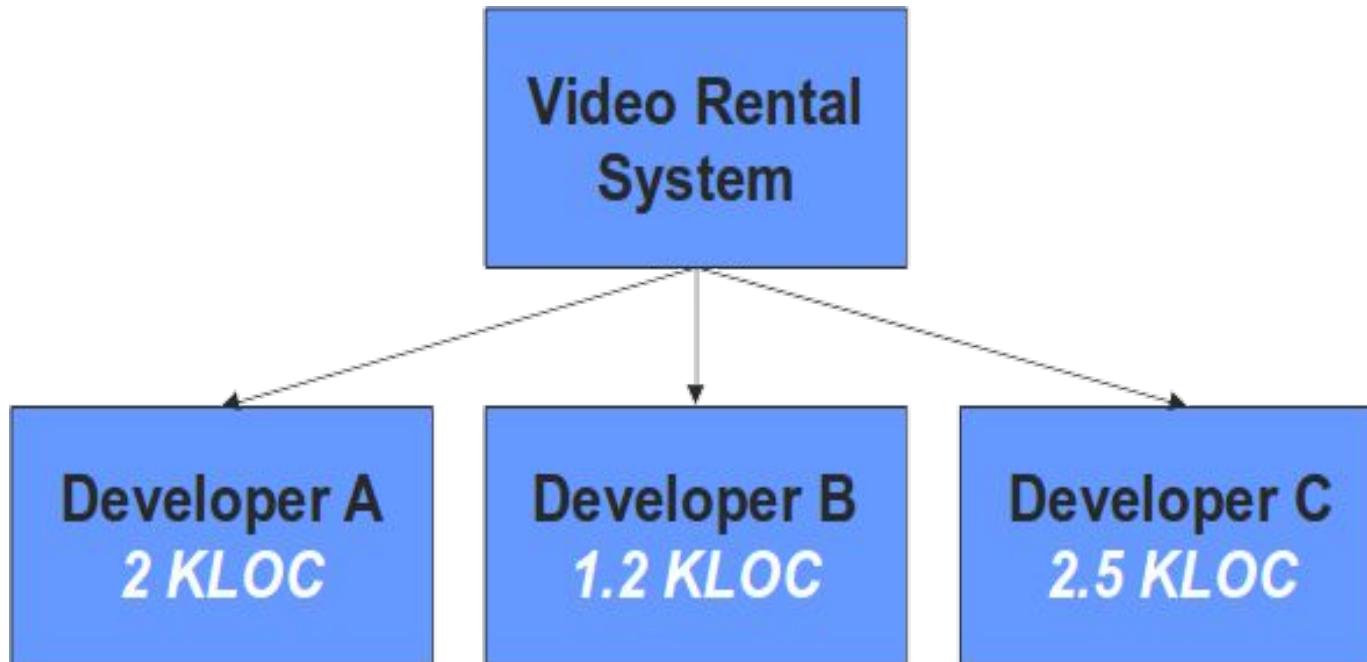
# Problems with LOC

- Same system developed with different programming languages will give different LOC readings.



# Problems with LOC

- Same system developed by different developers using the same language will give different LOC readings.



# Problems with LOC

- To calculate LOC you have to wait until the system is implemented.
- This is not adequate when management requires prediction of cost and effort.
- A different approach is sometimes necessary.

# Function Points

- Instead of measuring size, function points measure the functionality offered by a system.
- Invented by Albrecht at IBM in 1979.
- Still use today.

# Function Points

- A Function can be defined as a collection of executable statements that performs a certain task.
- Function points can be calculated before a system is developed.
- They are language and developer independant.

# Function Points: An Overview

- A function point count is calculated as a weighted total of five major components that comprise an application...
  - a. External Inputs
  - b. External Outputs (e.g. reports)
  - c. Logical Internal Files
  - d. External Interface Files – files accessed by the application but not maintained by it
  - e. External Inquiries – types of online inquiries supported

# Function Points: Calculation

- The simplest way to calculate a function point count is calculated as follows:  
(No. of external inputs x 4) +  
(No. of external outputs x 5) +  
(No. of logical internal files x 10) +  
(No. of external interface files x 7) +  
(No. of external enquiries x 4)

# Function Points: Example

- Consider the following system specs:

Develop a system which allows customers to report bugs in a product. These reports will be stored in a file and developers will receive a daily report with new bugs which they need to solve. Customers will also receive a daily status report for bugs which they submitted. Management can query the system for a summary info of particular months.

# Function Points: Example

- External Inputs: 1
- External Outputs: 2
- Logical Internal Files: 1
- External Interface Files: 0
- External Enquiries: 1
- Total Functionality is  $(1 \times 4) + (2 \times 5) + (1 \times 10) + (0 \times 7)$   
 $+ (1 \times 4) = 28$

# Function Points Extensions

- The original function points were sufficient but various people extended them to make them more expressive for particular domains.
- Examples:
  - a. General System Characteristics (GSC) Extension
  - b. 3D Function Points for real time systems
  - c. Object Points
  - d. Feature Points

# GSC: A Function Point Extension

- Reasoning: Original Function Points do not address certain functionality which systems can offer.
- E.g. Distributed functionality, performance optimisation, etc.
- The GSC extension involves answering 14 questions about the system and modifying the original function point count accordingly.

# GSC: A Function Point Extension

- 1. Data communications
- 2. Distributed Functions
- 3. Performance
- 4. Heavily used configuration
- 5. Transaction rate
- 6. Online Data Entry
- 7. End-user Efficiency
- 8. On-line update
- 9. Complex Processing
- 10. Reusability
- 11. Installation ease
- 12. Operational Ease
- 13. Multiple sites
- 14. Facilitation of Change

# GSC: A Function Point Extension

- The analyst/software engineer assigns a value between 0 and 5 to each question
- 0 = not applicable and 5 = essential
- The Value-Adjustment Factor (VAF) is then calculated as:

$$VAF = 0.65 + 0.01 \sum_{i=1}^{14} Ci$$

You then adjust the original function point count as follows:

$$FP = FC \times VAF$$

# GSC: A Function Point Extension

Consider the bug-reporting system for which we already looked at and suppose the analyst involved answers the GSC questions as follows...

|    |                            |   |     |                        |   |
|----|----------------------------|---|-----|------------------------|---|
| 1. | Data communications        | 5 | 8.  | On-line update         | 3 |
| 2. | Distributed Functions      | 0 | 9.  | Complex Processing     | 1 |
| 3. | Performance                | 1 | 10. | Reusability            | 0 |
| 4. | Heavily used configuration | 0 | 11. | Installation ease      | 2 |
| 5. | Transaction rate           | 1 | 12. | Operational Ease       | 3 |
| 6. | Online Data Entry          | 5 | 13. | Multiple sites         | 4 |
| 7. | End-user Efficiency        | 0 | 14. | Facilitation of Change | 0 |

**Total GSC Score = 25**

# GSC: A Function Point Extension

- As you may remember, when we calculated the function point count for this system, we got a result of 28.
- If we apply the GSC extension, this count will be modified as follows.

$$VAF = 0.65 + (0.01 \times 25) = 0.9$$

$$FC = 28 \times 0.9 = 25.2$$

- Note that the GSC extension can increase or decrease the original count.
- In larger systems, the GSC extension will have a much more significant influence on the Function Point Count.

# Defect Density

- A rate-metric which describes how many defects occur for each size/functionality unit of a system.
- Can be based on LOC or Function Points.

$$\frac{\# \text{defects}}{\text{system\_size}}$$

# Failure Rate

- Rate of defects over time
- May be represented by the  $\lambda$  (lambda) symbol

$$\lambda = \frac{R(t_1) - R(t_2)}{(t_2 - t_1) \times R(t_1)}$$

where,

$t_1$  and  $t_2$  are the beginning and ending of a specified interval of time

$R(t)$  is the reliability function, i.e. probability of no failure before time  $t$

## Example of Failure Rate

- Calculate the failure rate of system X based on a time interval of 60 days of testing. The probability of failure at time day 0 was calculated to be 0.85 and the probability of failure on day 60 was calculated to be 0.2.

$$\lambda = \frac{R(t_1) - R(t_2)}{(t_2 - t_1) \times R(t_1)}$$

$$\lambda = \frac{0.85 - 0.2}{60 \times 0.85}$$

$$= \frac{0.65}{51}$$

$$= 0.013 \text{ Failures per day}$$

# Mean Time Between Failure (MTBF)

- MTBF is useful in safety-critical applications (e.g. avionics, air traffic control, weapons, etc).

$$MTBF = \frac{1}{\lambda}$$

- Consider our previous example where we calculated the failure rate ( $\lambda$ ) of a system to be 0.013. Calculate the MTBF for that system.

$$MTBF = \frac{1}{\lambda}$$

$$= 76.9 \text{ days}$$

# McCabe's Cyclomatic Complexity Metric

1. Complexity is an important attribute to measure.
2. Measuring Complexity helps us to
  - a. Predict testing effort
  - b. Predict defects
  - c. Predict maintenance costs
3. Cyclomatic Complexity Metric was designed by McCabe in 1976.
4. Aimed at indicating a program's testability and understandability.
5. It is based on graph theory.
6. Measures the number of linearly independent paths comprising the program.

# McCabe's Cyclomatic Complexity Metric

The formula of cyclomatic complexity is:

$$M = V(G) = e - n + 2p$$

where

$V(G)$  = cyclomatic number of Graph G

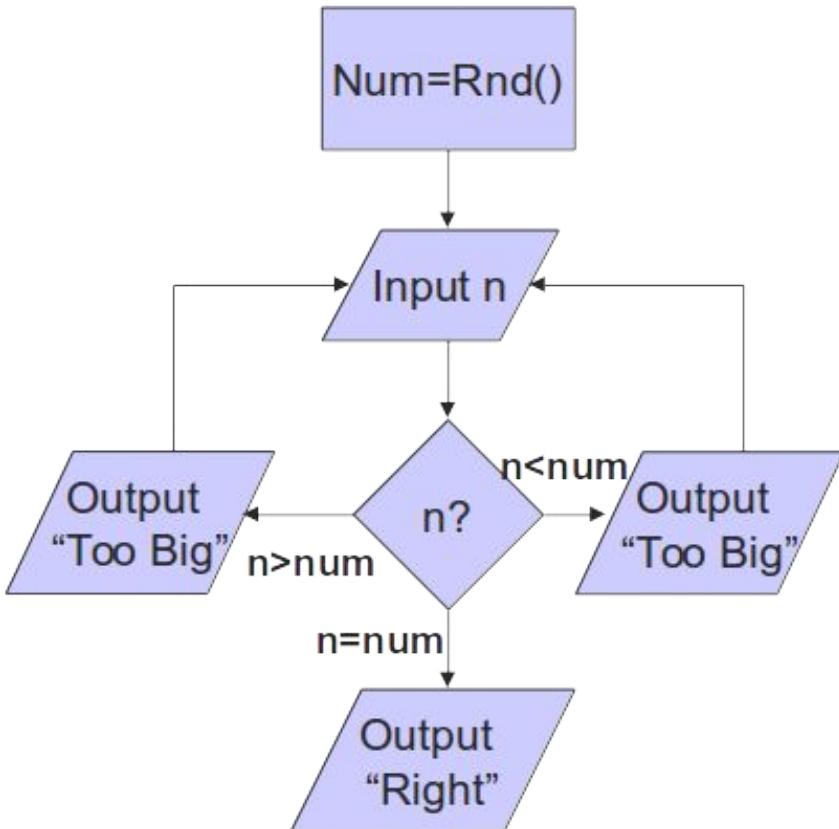
e = number of edges

n = number of nodes

p = number of connected components

# Cyclomatic Complexity: Example

Consider the following flowchart...



Calculating cyclomatic complexity

$$e = 7, n=6, p=1$$

$$M = 7 - 6 + (2 \times 1) = \mathbf{3}$$

# McCabe's Cyclomatic Complexity

1. Note that the number delivered by the cyclomatic complexity is equal to the number of different paths which the program can take.
2. Cyclomatic Complexity is additive. i.e.  $M(G_1 \text{ and } G_2) = M(G_1) + M(G_2)$ .
3. To have good testability and maintainability, McCabe recommends that no module have a value greater than 10.
4. This metric is widely used and accepted in industry.

# Halstead's Software Science

1. Halstead (1979) distinguished software science from computer science.
2. Premise: Any programming task consists of selecting and arranging a finite number of program “tokens”
3. Tokens are basic syntactic units distinguishable by a compiler.
4. Computer Program: A collection of tokens that can be classified as either operators or operands.

# Halstead's Software Science

- Primitives:
  - $n_1 = \# \text{ of distinct operators appearing in a program}$
  - $n_2 = \# \text{ of distinct operands appearing in a program}$
  - $N_1 = \text{total } \# \text{ of operator occurrences}$
  - $N_2 = \text{total } \# \text{ of operand occurrences}$
- Based on these primitive measures, Halstead defined a series of equations.

# Halstead's Software Science

Vocabulary (n):  $n = n_1 + n_2$

Length (N):  $N = N_1 + N_2$

Volume (V):  $V = N \log_2(n)$  #bits required to  
represent a program

Level (L):  $L = V^* / V$  Measure of abstraction and  
therefore complexity

Difficulty (D):  $D = 1/L$

Effort (E):  $E = V/L$

Faults (B):  $B = V/S^*$

Where:

$$V^* = 2 + n_2 \times \log_2(2 + n_2)$$

$S^*$  = average number of decisions between errors  
(3000 according to Halstead)

# Other Useful Product Metrics

- Cost per function point.
- Defects generated per function point.
- Percentage of fixes.

# Process Metrics

- Why measure the process?
- The process creates the product.
- If we can improve the process, we indirectly improve the product.
- Through measurement, we can understand, control and improve the process.
- This will lead us to engineering quality into the process rather than simply taking product quality measurements when the product is done.
- We will look briefly at a number of process metrics.

# Defect Density during Testing

- Defect rate during formal testing is usually positively correlated with the defect rate experienced in the field.
- Higher defect rates found during testing is an indicator that higher defect rates will be experienced in the field.
- **Exception:** In the case of exceptional testing effort or more effective testing methods being employed.
- It is useful to monitor defect density metrics of subsequent releases of the same product.

# Defect Arrival Pattern During Testing

- Overall defect density during testing is a summary indicator.
- However, the pattern of defect arrivals gives more information.
- Even with the same overall defect rate during test, arrival patterns can be different.

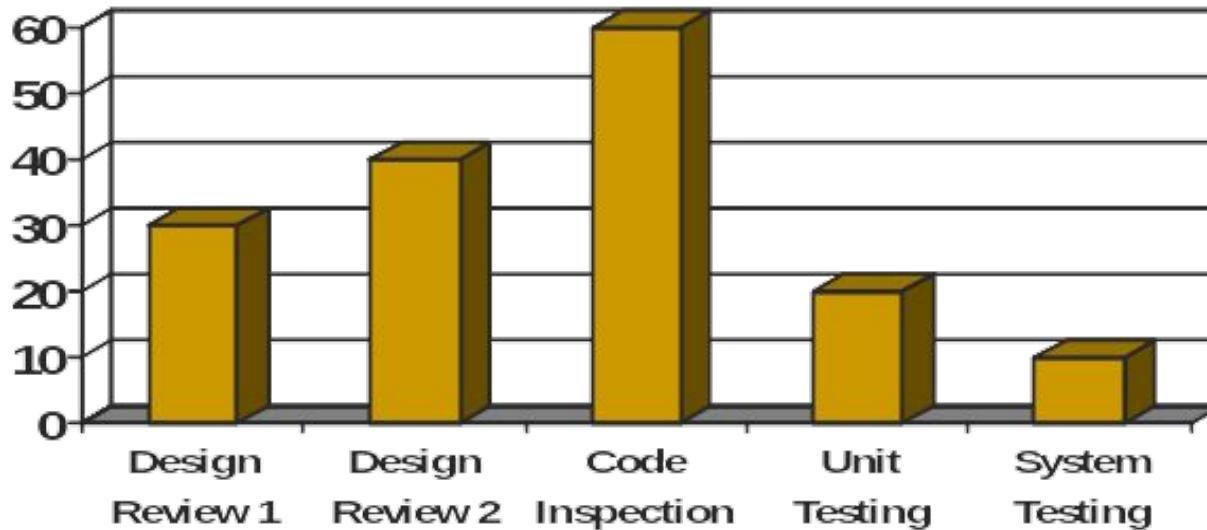
# Interpreting Defect Arrival Patterns

- Always look for defect arrivals stabilising at a very low level.
- If they do not stabilise at a low rate, risking the product will be very risky.
- Also keep track of defect backlog over time. It is useless detecting defects if they are not fixed and the system re-tested.

# Phase-Based Defect Removal Pattern

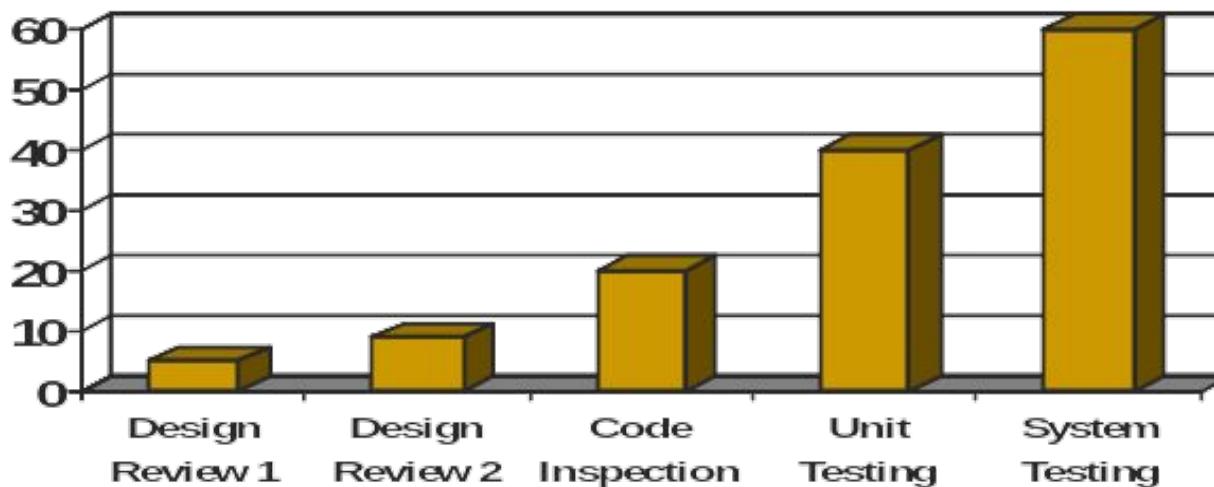
- An extension of the defect density metric.
- Tracks defects at all phases of the life cycle.
- The earlier defects are found, the cheaper they are to fix.
- This metric helps you monitor when your defects are being found.

# Phase-Based Defect Removal Pattern: Example



## Project A

Most defects found before testing  
Ideal situation



## Project B

Most defects found **during** testing  
More expensive to fix  
Should be corrected

# Other Useful Process Metrics

- Fix response time
  - Average time to fix a defect
- Percent delinquent fixes
  - Fixes which exceed the recommended fix time according to their severity level
- Fix quality
  - Percentage of fixes which turn out to be defective

# People Metrics (Why Measure People?)

- People metrics are of interest to management for:
- Financial purposes (e.g. Putting Joe on project A will cost me 500 Rs per function point)
- Project management purposes (e.g. Michele needs to produce 5 function points per day in order to be on time)
- HR problem identification (e.g. On average, developers produce 5 defects per hour. James produces 10. Why?)

# Warning on People Measurement

- People do not like being measured.
- In many cases, you will not be able to look at numbers and draw conclusions.
- For example, at face value, Clyde may take a longer time to finish his work when compared to colleagues. However, further inspection might reveal that his code is bug free whilst that of his colleagues needs a lot of reworking.
- Beware when using people metrics. Only use them as indicators for potential problems.
- You should never take disciplinary action against personnel simply based on people metrics.

# Some People Metrics

- For individual developers or teams:
  - a. Cost per Function Point
  - b. Mean Time required to develop a Function Point
  - c. Defects produced per hour
  - d. Defects produced per function point

# OO Design Metrics

- Why measure OO Designs?
  - a. OO has become a very popular paradigm.
  - b. Measuring the Quality of a design helps us identify problems early on in the life cycle.
  - c. A set of OO Design metrics were proposed by Chidamer and Kemerer (MIT) in 1994.
  - d. We will look at some important metrics in OO designs.

# Weighted Methods Per Class (WMC)

- Consider the class C with methods m<sub>1</sub>, m<sub>2</sub>, ... m<sub>n</sub>.
- Let c<sub>1</sub>, c<sub>2</sub> ... c<sub>n</sub> be the complexity of these methods.

$$WMC = \sum_{i=1}^n c_i$$

# Weighted Methods Per Class (WMC)

- Refers to the complexity of an object.
- The number of methods involved in an object is an indicator of how much time and effort is required to develop.
- Complex classes also make their child classes complex.
- Objects with large number of methods are likely to be more application-specific and less reusable.
- Affects:
  - a. Understandability, Maintainability, Reusability

# Depth of Inheritance Tree (DIT)

- The Depth of Inheritance of a class is its depth in the inheritance tree.
- If multiple inheritance is involved, the DIT of a class is the maximum distance between the class and the root node.
- The root class has a DIT of 0.

# Depth of Inheritance Tree (DIT)

- The deeper a class is in the hierarchy, the greater the number of methods likely to inherit from parent classes – **more complex**.
- **Deeper trees:** More Reuse.
- **Deeper trees:** Greater Design Complexity.
- DIT can analyse efficiency, reuse, understandability and testability.

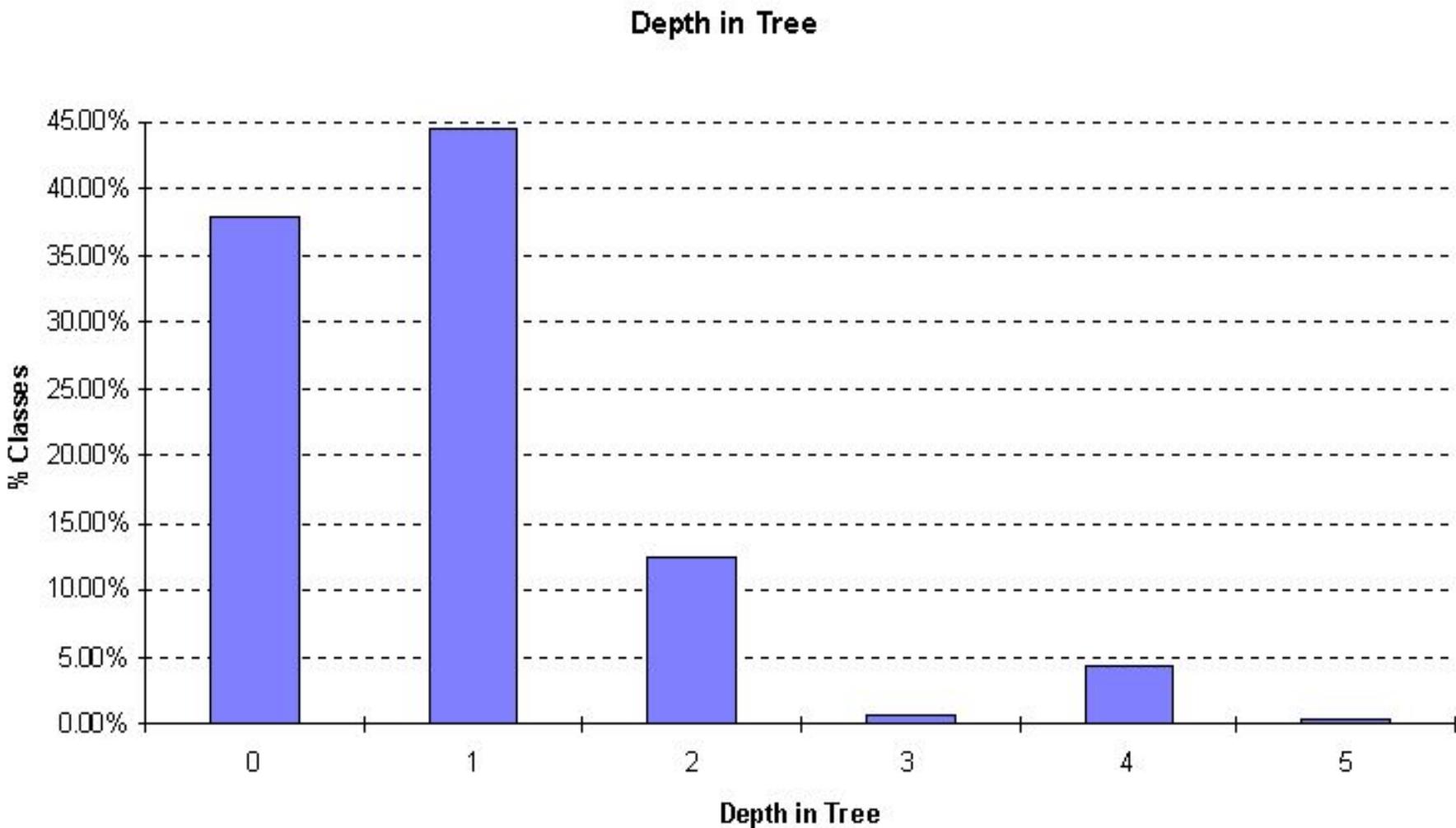
# Number of Children (NOC)

- A simple metric.
- Counts the number of immediate subclasses of a particular class.
- It is a measure of how many subclasses are going to inherit attributes and methods of a particular class.

# Number of Children (NOC)

- Generally it is better to have depth than breadth in the class hierarchy
- Promotes reuse of methods through inheritance
- Classes higher up in the hierarch should have more subclasses
- Classes lower down should have less
- The NOC metric gives an indication of the potential influence of a class on the overall design
- Attributes: Efficiency, Reusability, Testability

# Number of Children (NOC)



## Other Metrics

- Coupling between objects (CBO)
- Response for a class (RFC)
- Lack of cohesion in methods (LCOM)

# Software Testing

**23 October 2020**

# **Software Testing**

- **Software Testing, when done correctly, can increase overall software quality by testing that the product conforms to its requirements.**
- **Testing begins with the software engineer in early stages, but later specialists may be involved in testing process.**
- **After generating source code, the software must be tested to uncover and correct as many errors as possible before delivering to customer.**
- **It's important to test the software because a customer, after running it many times might encounter an error.**

# Testing Objectives

## PREVENT DEFECTS

Efficient testing helps preventing defects and that helps in providing an error-free application.

## FIND FAILURE AND DEFECTS

To find failures and defects. Defects should be identified as early in the test cycle as possible..

## SHARE INFORMATION TO STAKEHOLDERS

To provide enough information to stakeholders to allow them to make informed decisions, especially regarding the level of quality of the test object

## REDUCE RISK

To reduce the level of risk of inadequate software quality (e.g., previously undetected failures occurring in operation)



## EVALUATE WORK PRODUCTS

To evaluate work products such as requirements, user stories, design, and code

## VERIFY REQUIREMENT

To verify whether all specified requirements have been fulfilled

## VALIDATE TEST OBJECT

To validate whether the test object is complete and works as the users and other stakeholders expect

## BUILD CONFIDENCE

To build confidence in the level of quality of the test object

# **Software Testing Fundamentals**

- During software engineering activities, an engineer attempts to build software, whereas during testing activities, the intent is to demolish the software.
- Hence testing is psychologically destructive rather constructive.
- A successful test is one that uncovers an undiscovered error.
- All the tests that are carried out should be traceable to customer requirements, sometimes exhaustive testing is impossible.

## **Categorised As:**

### **1) Functional Testing:**

- This is a type of black-box testing that is based on the specifications of the software that is to be tested.
- The application is tested by providing input and then the results are examined that need to conform to the functionality it was intended for.

### **2) Non-Functional Testing:**

- This section is based upon testing an application from its non-functional attributes.
- Non-functional testing involves testing a software from the requirements which are nonfunctional in nature but important such as performance, security, user interface, etc.

## **Differences:-**

### **Functional Testing**

1. It tests ‘What’ the product does. It checks the operations and actions of an Application.
2. Functional testing is done based on the business requirement.
3. It tests as per the customer requirements.
4. It is testing the functionality of the software.
5. It is carried out manually.

## **Differences:-**

### **Non Functional Testing**

1. It checks the behaviour of an Application.
2. It is based on the customer expectation and Performance requirement.
3. It tests as per customer expectations.
4. It is testing the performance of the functionality of the software.
5. It is more feasible to test using automated tools.

# **Taxonomy of Software Testing**

## **Functional Testing**

- Unit Testing
- Integration Testing
- Regression Testing
- Smoke Testing
- Alpha Testing
- Beta Testing
- System Testing

## **Non-Functional Testing**

- Stress Testing
- Performance Testing
- Usability Testing
- Security Testing
- Portability Testing

# **Functional Testing**

## **1. Unit Testing**

It focuses on smallest unit of software design. In this we test an individual unit or group of interrelated units. It is often done by programmer by using sample input and observing its corresponding outputs.

### **Example:**

- a) In a program we are checking our loop, method or function is working fine.
- b) Misunderstood or incorrect, arithmetic precedence.
- c) Incorrect initialization

## **2. Integration Testing**

Integration testing is a software testing where individual units/components are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units.

Integration testing is of four types:

- (i) Big-Bang
- (ii) Bottom up
- (iii) Top down
- (iv) Mixed or Sandwiched

## 2. Integration Testing

- (i) Big-Bang: All the modules of the system are simply put together and tested. This approach is practicable only for very small systems. If once an error is found during the integration testing, it is very difficult to localize the error as the error may potentially belong to any of the modules being integrated.
- (ii) Bottom up: In bottom-up testing, each module at lower levels is tested with higher modules until all modules are tested. Uses test drivers to drive and pass appropriate data to the lower level modules.

## 2. Integration Testing

- (iii) Top down: Testing takes place from top to bottom. First high-level modules are tested and then low-level modules and finally integrating the low-level modules to a high level to ensure the system is working as intended.
- (iv) Mixed: Follows a combination of top down and bottom-up testing approaches. In top-down approach, testing can start only after the top-level module have been coded and unit tested. In bottom-up approach, testing can start only after the bottom level modules are ready. This sandwich or mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. Disadvantage: Expensive and can not be used for smaller systems.

### **3. Regression Testing**

Every time new module is added leads to changes in program. This type of testing make sure that whole component works properly even after adding components to the complete program.

#### **Example:**

In school record suppose we have module staff, students and finance combining these modules and checking if on integration these module works fine is regression testing.

## **4. Smoke Testing**

This test is done to make sure that software under testing is ready or stable for further testing.

It is called smoke test as testing initial pass is done to check if it did not catch the fire or smoked in the initial switch on.

## **5. Alpha Testing**

It is a type of acceptance testing which is done before the product is released to customers. It is typically done by QA people.

### **Example:**

When software testing is performed internally within the organization.

## **6. Beta Testing**

The beta test is conducted at one or more customer sites by the end-user of the software. This version is released for the limited number of users for testing in real time environment.

### **Example:**

When software testing is performed for the limited number of people.

## **7. System Testing**

In this software is tested such that it works fine for different operating system. It is covered under the black box testing technique. In this we just focus on required input and output without focusing on internal working.

In this we have security testing, recovery testing, stress testing and performance testing

### **Example:**

This include functional as well as non functional testing.

# Non-Functional Testing

## 8. Stress Testing

Stress testing includes testing the behaviour of a software under abnormal conditions. For example, it may include taking away some resources or applying a load beyond the actual load limit.

The aim of stress testing is to test the software by applying the load to the system and taking over the resources used by the software to identify the breaking point.

### **Example:**

- (a) Test cases that require maximum memory or other resources are executed.
- (b) Test cases that may cause thrashing in a virtual operating system.
- (c) Test cases that may cause excessive disk requirement.

## **9. Performance Testing**

It is designed to test the run-time performance of software within the context of an integrated system. It is used to test speed and effectiveness of program.

It is also called load testing. In it we check, what is the performance of the system in the given load.

### **Example:**

Checking number of processor cycles.

## **10. Usability Testing**

Usability testing is a black-box technique which is used to identify any error(s) and improvements in the software by observing the users through their usage and operation.

Usability can be defined in terms of five factors, i.e. efficiency of use, learn-ability, memory-ability, errors/safety, and satisfaction.

### **Example:**

Ensuring a good and user-friendly GUI that can be easily handled.

## 11. Security Testing

Security testing involves testing a software in order to identify any flaws and gaps from security and vulnerability point of view.

It should ensure:

- Confidentiality
- Integrity
- Authentication
- Availability
- Authorization
- Non-repudiation

## **12. Portability Testing**

Portability testing includes testing a software with the aim to ensure its reusability and that it can be moved from another software as well.

This includes testing of a software w.r.t its usage over different environments. Like Computer hardware, operating systems, and browser.

## **Other Testing Categories:**

### **1. Non-execution based:**

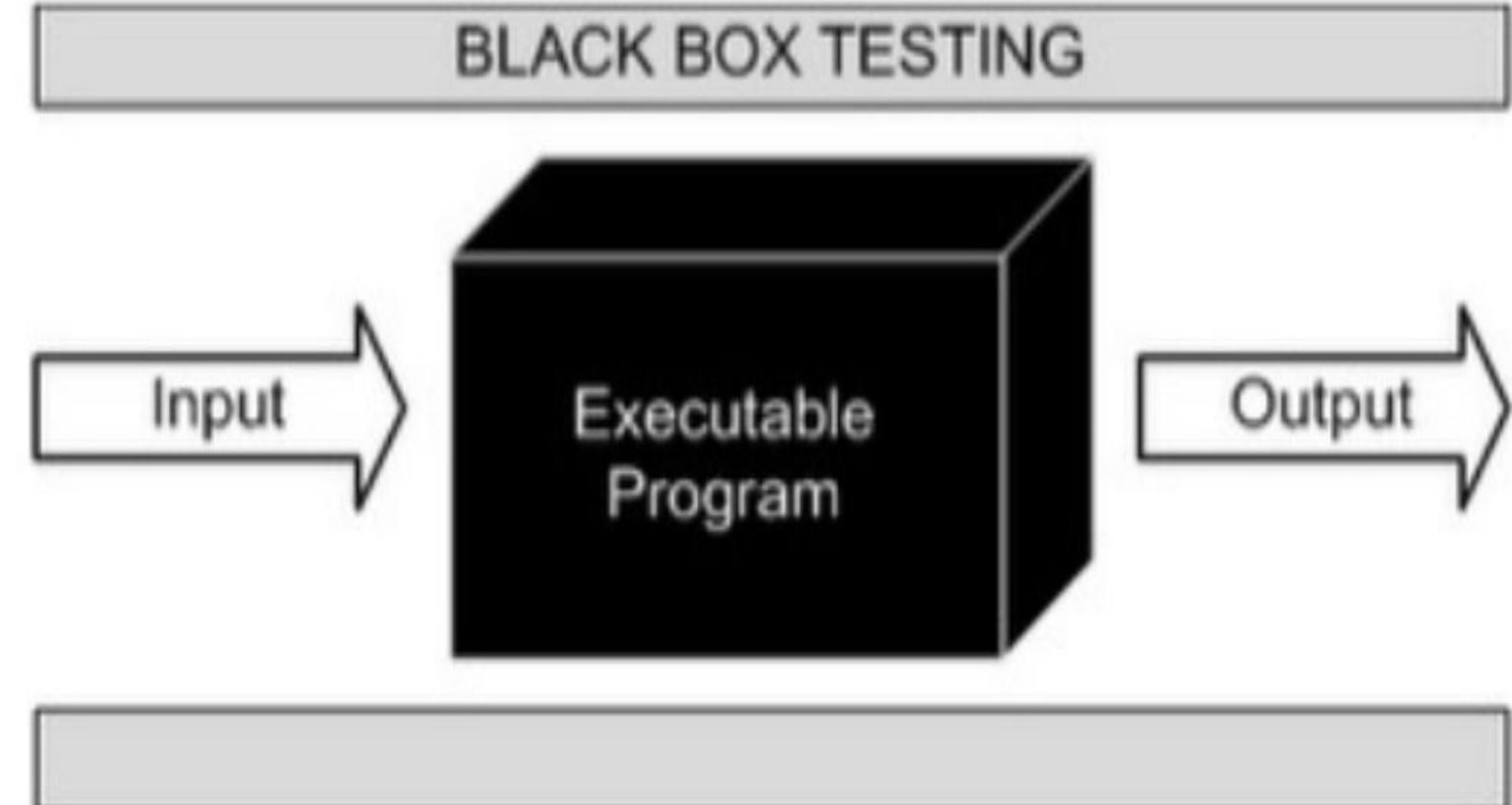
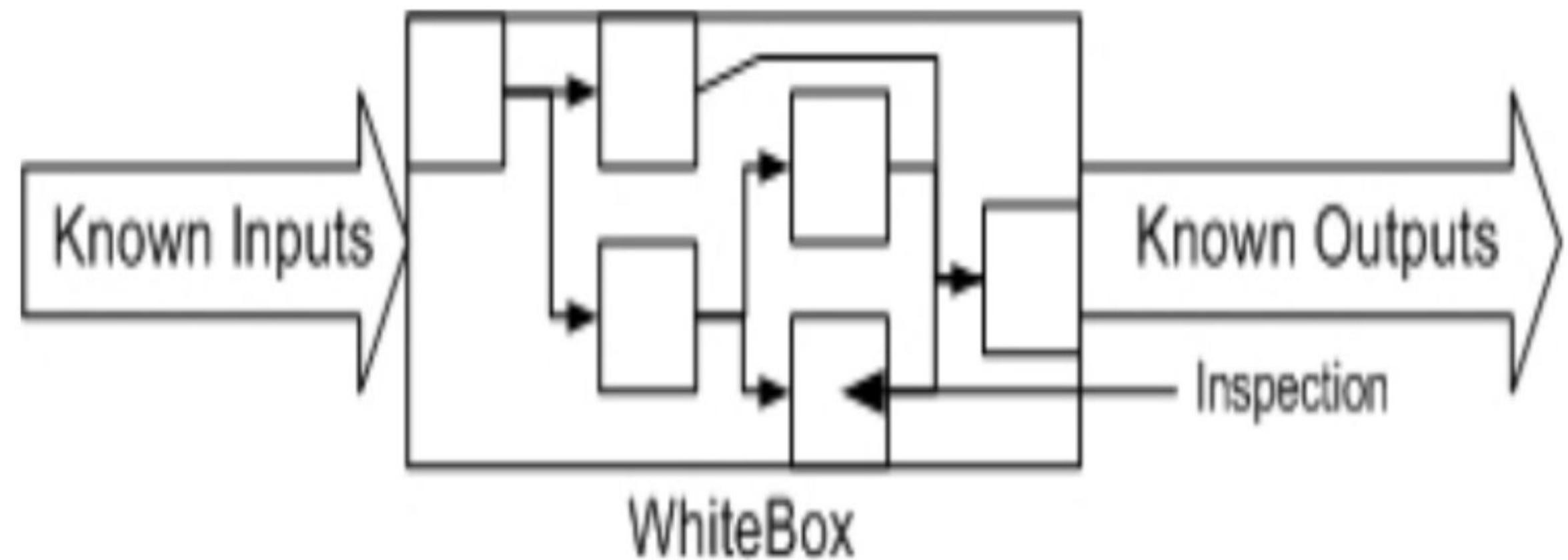
Non-execution based testing means the module is always reviewed by a team. It relies on fault detection strategy.

- The fault-detecting power of these non-execution based testing techniques leads to rapid, thorough, and early fault detection.
- Non-execution based testing is also known by the name static testing or you can say by static program analysis.

## **2. Execution based:**

Here, modules are run against test cases. Following are the two types of execution based testing:

1. Black-box testing: It is the sort of testing in which the code itself is ignored; the exclusive information handle in designing test cases are the specification document.
2. White-box testing: It is the sort of testing in which the code itself is tested, without regard of the specifications.



## Points to Remember:

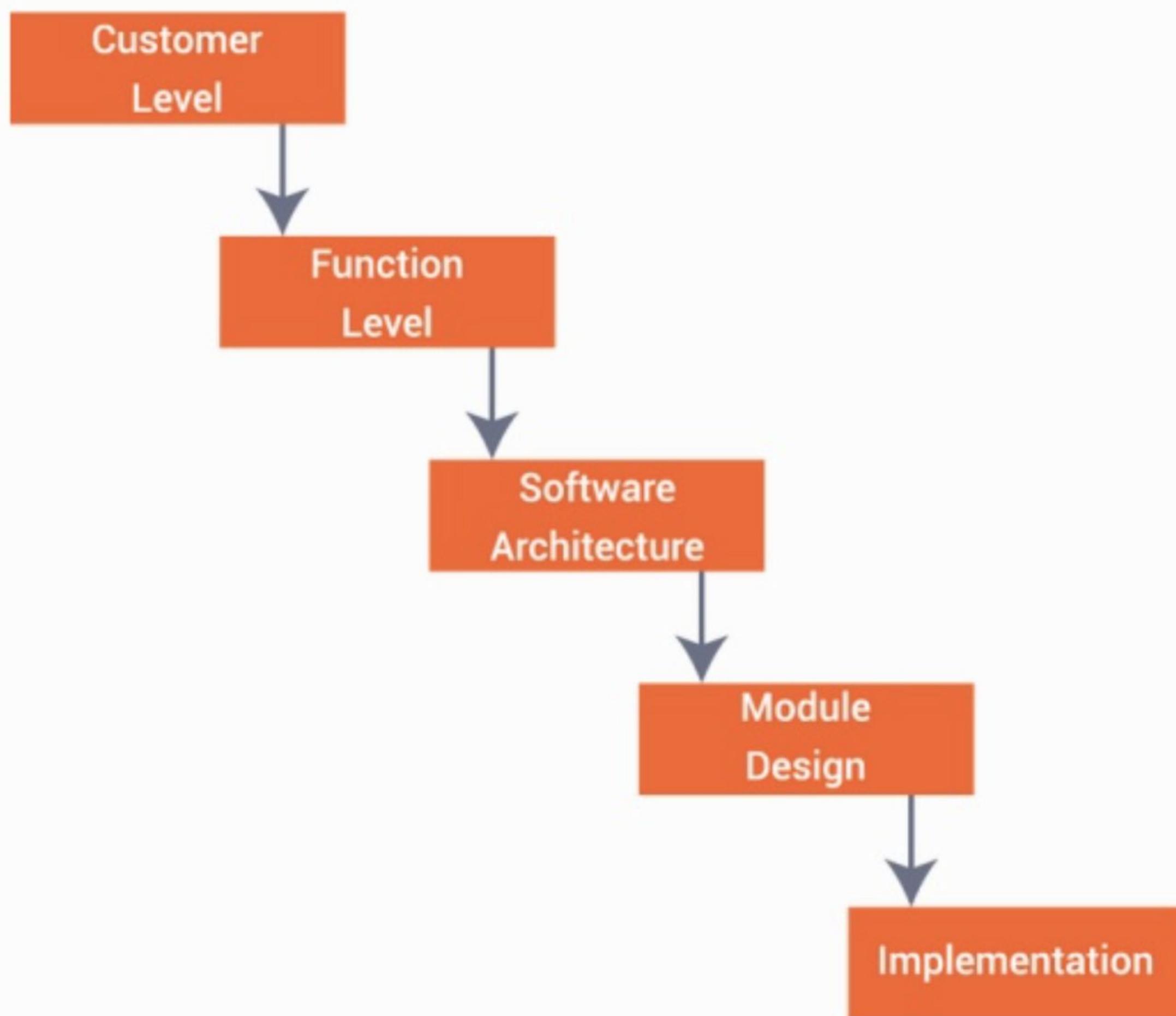
- Non-execution based testing is also called as Static Testing.
- Execution based testing is known as Dynamic Testing.

# Static Analysis v/s Dynamic Analysis

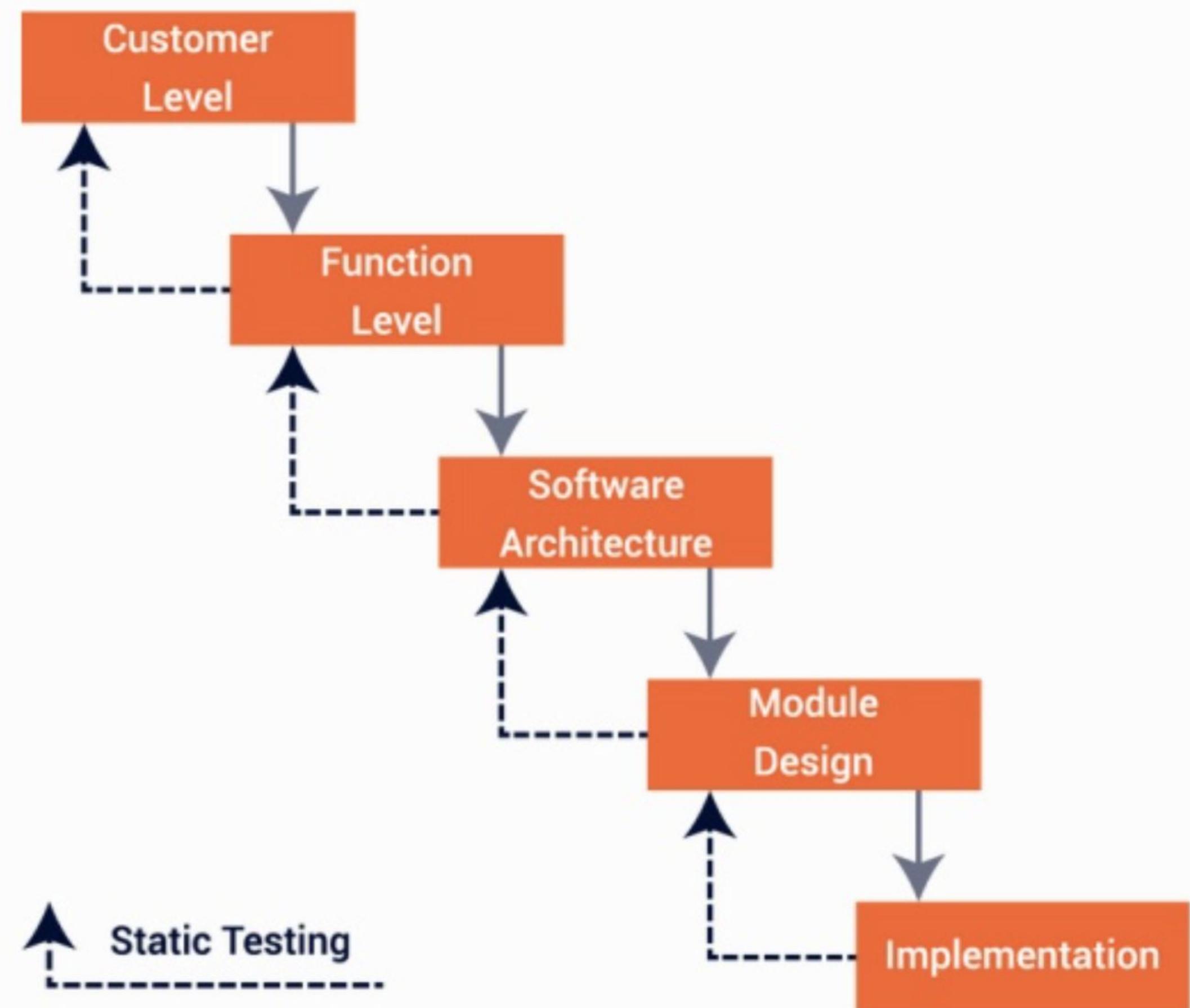
## Static Analysis

- Analyse code & generated code
- Typical use cases:
  - . Syntax checking
  - . Code smell detection
  - . Coding standard compliance
- Typical Defects found:
  - . Syntax violation
  - . Unreachable code
  - . Overly complicated constructs

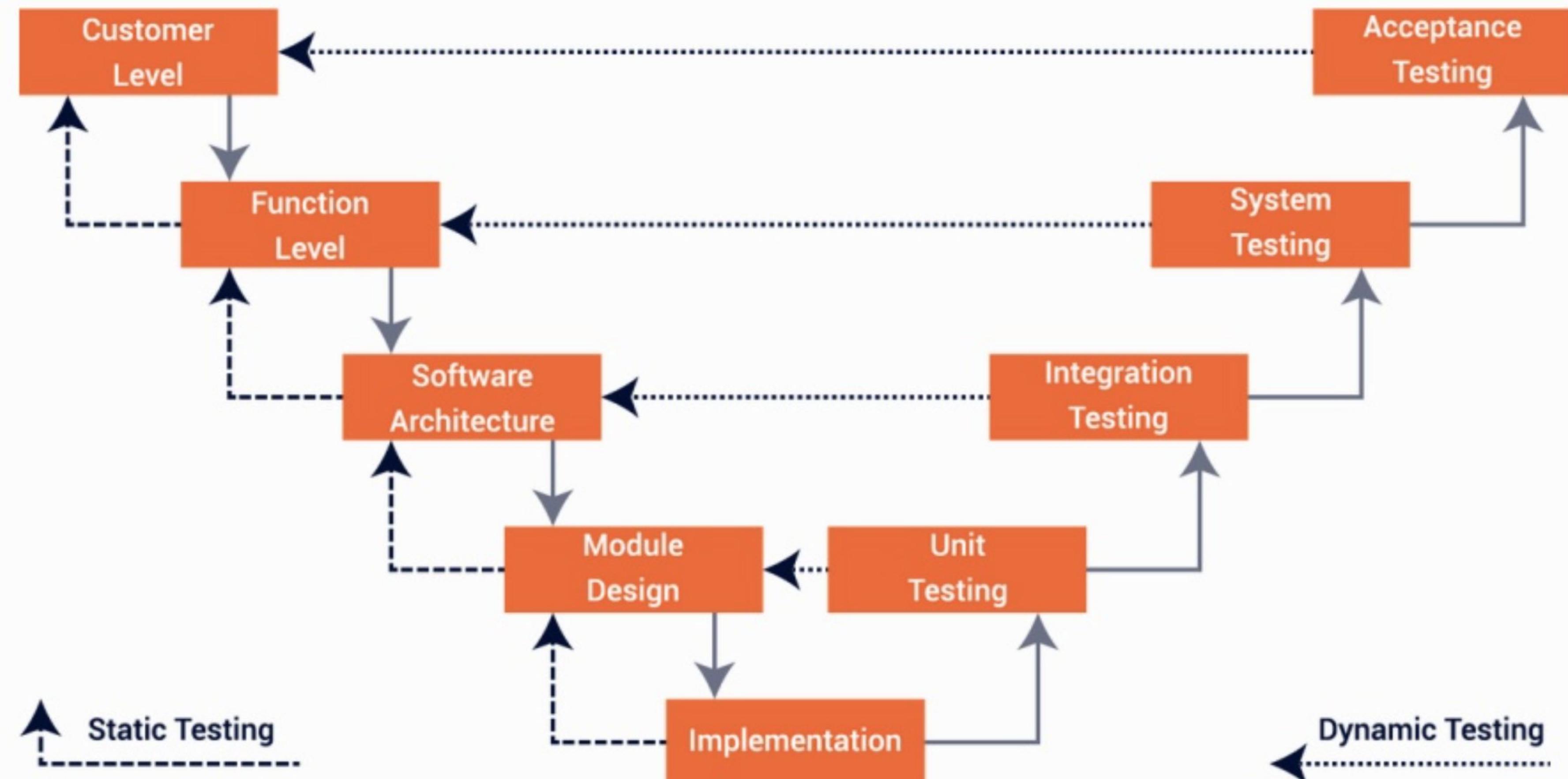
# Development Phases



# Static Testing



# Dynamic Testing



# **SOFTWARE TEST CASE GENERATION TECHNIQUES**

- Manual Software Testing:**

**Trying various usage and input combinations, comparing the results to the expected behaviour and recording their observations.**

- Automation Testing:**

**Using an automation tool to execute test case suite. It can enter test data, compare expected and actual results and generate detailed test reports but demands money and resources.**

## **About Test Case:**

- A test case is a set of inputs given to the software or application to get the pre-specified output.
- The total cost, time and effort required for overall testing depends on total number of test cases and size of the application.

# Scope of Automation in Testing

- Automation is the process of evaluating the SUT(System under Test) against the specification with help of a tool.

Depending on the nature of testing there are two main branches under automation:

1. Functional testing with automation.
2. Performance testing with automation.



## **Functional Testing with help of Automation:**

- The main area where the functional testing tools are used is for regression test case execution.
- Mostly in the agile scrum methodology where frequent releases are happening, it is almost impossible to execute all the regression test cases manually with the short span of time.
- Automation gives a high ROI (Return of Investment) in this area since it is one time effort for generating the scripts.
- It is advisable to keep at least 60 to 70% of regression cases to be automated.

## **Performance Testing with help of Automation:**

- Performance testing is the process of evaluating the application performance which is being a critical requirement now a days.
- Performance testing is almost impossible by manual means.
- Automation of load and stress testing: load testing confirms that the application can withstand the maximum specified load and stress testing is used for finding out the maximum load the application can accommodate.

## **Testing Terms:-**

- **Test Case:** A test case is a set of inputs, execution conditions, and a pass/fail criterion.
- **Test Case Specification:** It is a requirement to be specified by one or more actual test cases.
- **Test Suite:** It is a set of test cases. Typically a method for functional testing is concerned with creating a test suite.
- **Test Execution:** It refers to the activity of executing test cases and evaluating their results.
- **Adequacy Criterion:** It is expressed in the form of rule for deriving a set of test obligations from another artifact, such as a program or specification.

# **Selecting A Test Tool**

**Broad categories for classifying the criteria are:**

- 1. Meeting Requirements**
- 2. Technology expectations**
- 3. Training/skills**
- 4. Management aspects**

## 1. Meeting requirements

- A) Huge delay is involved in selecting and implanting test tools.
- B) Test tools may not provide backward or forward compatibility with product-under-test (PUT).
- C) A number of test tools cannot distinguish between a product failure and a test failure. This increases analysis time and manual testing.

## 2. Technology Expectations

- A) Test tools may not allow test developers to extend/modify the functionality of the framework.
- B) A number of test tools require their libraries to be linked with product binaries.
- C) Test tools are not 100% cross-platform.

### **3. Training Skills**

- A) Organisation-level training is needed to deploy the Test tools.**
- B) Test tools expect the users to learn new language/scripts.**
- C) This increases skill requirements for automation and increases the need for a learning curve inside the organization.**

### **4. Management Aspects**

- A) It is imp to note the system requirements and the cost involved in upgrading the software and hardware needs to be included with the cost of the tools.**
- B) Migration from one test tool to another may be difficult and requires a lot of effort.**
- C) Changing tools are only permitted if returns on investment (ROI) are justified.**

## **Steps to select and deploy a test tool:**

- 1. Identify your test suite requirements.**
- 2. Make to take care of experiences of other organisations.**
- 3. Collect the experiences of other Org, which used similar tools.**
- 4. Create list of questions to be asked to the vendors on cost/effort.**
- 5. Identify list of tools meeting above requirements and give priority to one which is available with the source code.**
- 6. Evaluate and shortlist one/set of tools and train developers on the tool.**
- 7. Deploy the tool across test teams after training all potential users of the tool.**

## **Test Case/data**

A test case is a pair consisting of test data to be input to the program and the expected output. The test data is a set of values, one for each input variable.

A test set is a collection of zero or more test cases.

Sample test case for sort:

Test data: <"A" 12 -29 32 >

Expected output: -29 12 32

# Testing and Verification

Program verification aims at proving the correctness of programs by showing that it contains no errors.

This is very different from testing that aims at uncovering errors in a program.

Program verification and testing are best considered as complementary techniques.

In practice, program verification is often avoided, and the focus is on testing.

## Testing and Verification

Testing is not a perfect technique in that a program might contain errors despite the success of a set of tests.

Verification promises to verify that a program is free from errors. However, the person/tool who verified a program might have made a mistake in the verification process; there might be an incorrect assumption on the input conditions; incorrect assumptions might be made regarding the components that interface with the program, and so on.

# Test Generation

Any form of test generation uses a source document.

In the most informal of test methods, the source document resides in the mind of the tester who generates tests based on a knowledge of the requirements.

In several commercial environments, the process is a bit more formal.

The tests are generated using a mix of formal and informal methods either directly from the requirements document serving as the source.

In more advanced test processes, requirements serve as a source for the development of formal models.

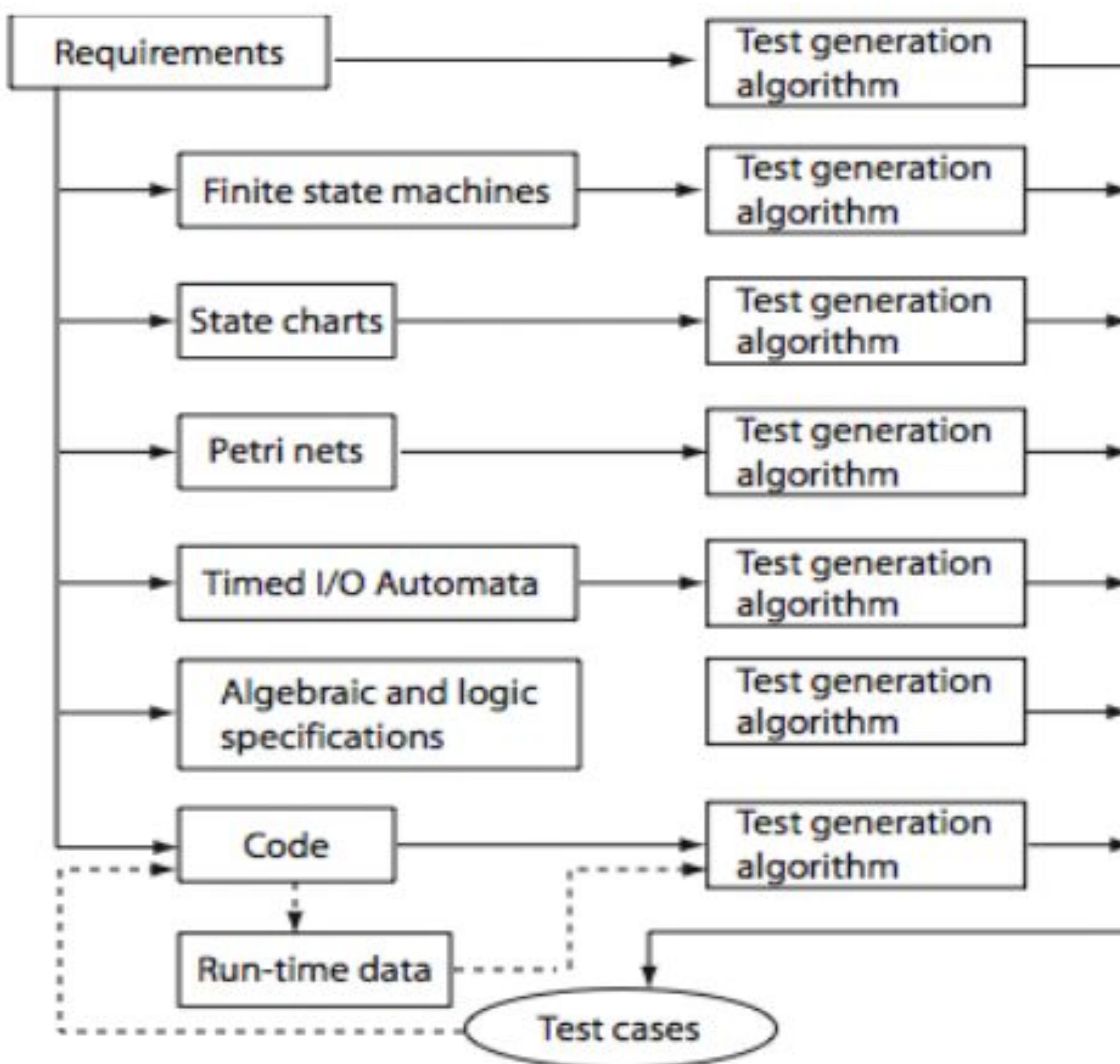
# **Test Generation Strategies**

**Model based:** require that a subset of the requirements be modeled using a formal notation (usually graphical). Models: Finite State Machines, Timed automata, Petri net, etc.

**Specification based:** require that a subset of the requirements be modeled using a formal mathematical notation. Examples: B, Z, and Larch.

**Code based:** generate tests directly from the code.

# Test Generation



# Sources of Test Generation

| Artifact                                                 | Technique                    | Example                                                                                                                                                                                                |
|----------------------------------------------------------|------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Requirements (informal)                                  | Black-box                    | Ad-hoc testing<br>Boundary value analysis<br>Category partition<br>Classification trees<br>Cause-effect graphs<br>Equivalence partitioning<br>Partition testing<br>Predicate testing<br>Random testing |
| Code                                                     | White-box                    | Adequacy assessment<br>Coverage testing<br>Data-flow testing<br>Domain testing<br>Mutation testing<br>Path testing<br>Structural testing<br>Test minimization using coverage                           |
| Requirements and code                                    | Black-box and White-box      |                                                                                                                                                                                                        |
| Formal model:<br>Graphical or mathematical specification | Model-based<br>Specification | Statechart testing<br>FSM testing<br>Pairwise testing<br>Syntax testing                                                                                                                                |
| Component interface                                      | Interface testing            | Interface mutation<br>Pairwise testing                                                                                                                                                                 |

# Predicates in Software Testing

Where do predicates arise?

Predicates arise from requirements in a variety of applications.

Here is an example from Paradkar, Tai, and Vouk, “Specification based testing using cause-effect graphs, Annals of Software Engineering,” V 4, pp 133-157, 1997.

A boiler needs to be shut down when the following conditions hold:

# Predicates in Software Testing

1. The water level in the boiler is below X lbs. (a)
2. The water level in the boiler is above Y lbs. (b)
3. A water pump has failed. (c)
4. A pump monitor has failed. (d)
5. Steam meter has failed. (e)

The boiler is to be shut down when a or b is true or the boiler is in degraded mode (when either c or d is true) and the steam meter fails. We combine these five conditions to form a compound condition (predicate) for boiler shutdown.

# Predicates in Software Testing

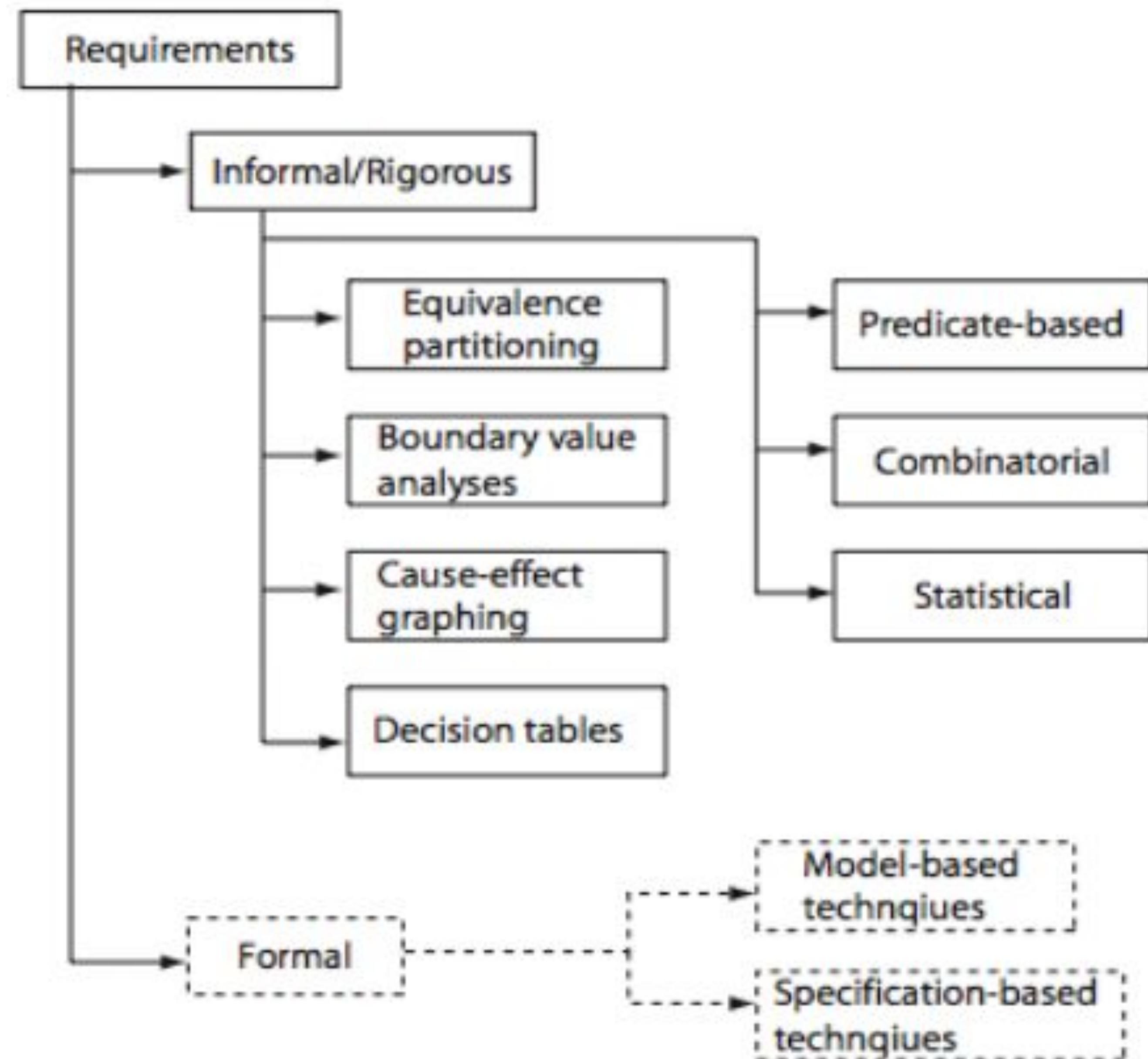
Denoting the five conditions above as a through e, we obtain the following Boolean expression E that when true must force a boiler shutdown:

$$E = a + b + (c + d)e$$

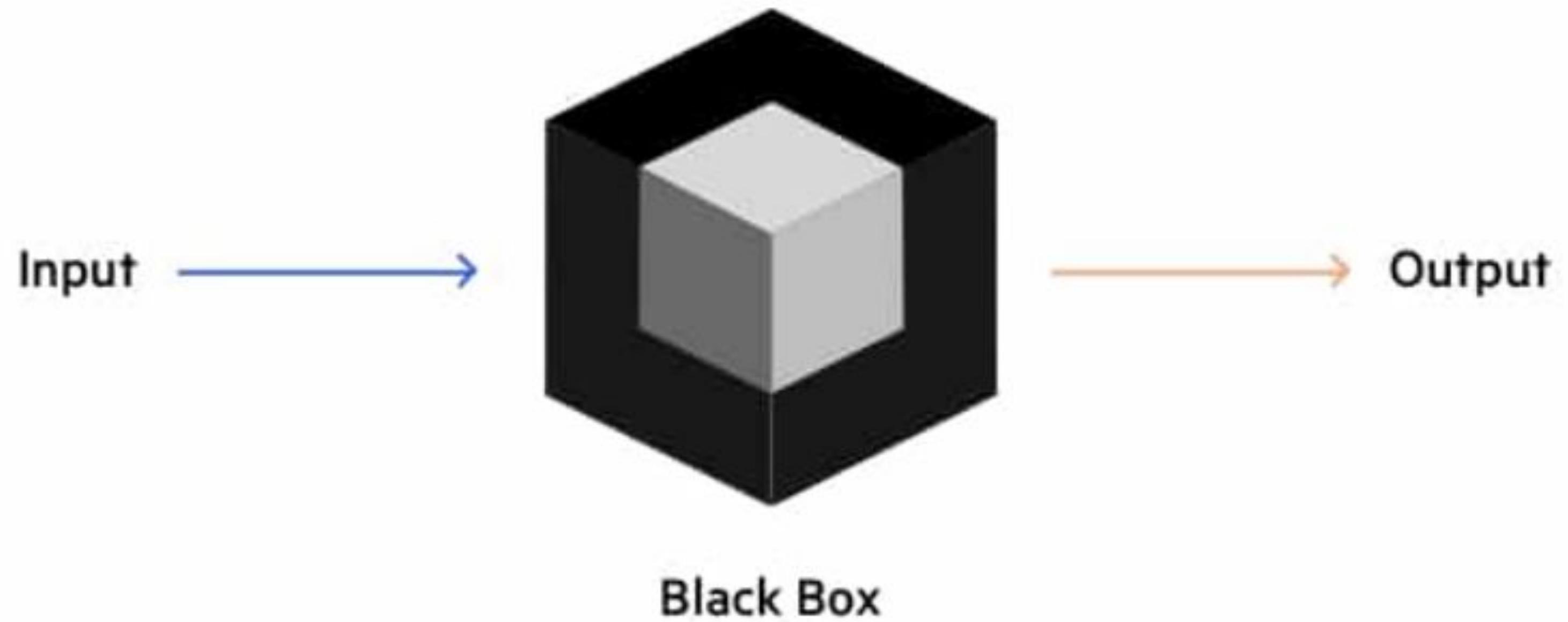
where the + sign indicates “OR” and a multiplication indicates “AND.”

The goal of predicate-based test generation is to generate tests from a predicate p that guarantee the detection of any error that belongs to a class of errors in the coding of p.

# Test Generation Techniques



# Black Box Testing



- Black box testing is testing technique having no knowledge of the internal functionality/structure of the system.
- It focuses on testing the function of program or application against its specification.

Contd...

- Determines whether combinations of inputs and operations produce expected results.
- For this technique, the tester would only know the “legal” inputs and what the expected outputs should be, but not how the program actually arrives at that output.
- Synonyms for Black Box are Behavioral, Functional, Opaque Box, Closed Box, etc.

# **Focus of Black Box Testing**

- In this technique, we do not use the code to determine a test suite; rather knowing the problem that we are trying to solve, we come up with four types of test data:
  - Easy to compute data
  - Typical data
  - Boundary / Extreme data
  - Bogus data

# **Black Box Testing Techniques**

Equivalence Partitioning

Boundary Value Analysis

Error Guessing

Cause Effect Graphing

# 1. Equivalence Partitioning

- An equivalence class is subset of data that is representative of a larger class.
- It is technique for testing equivalence classes rather than undertaking exhaustive testing of each value of the larger class.
- Example:
  - A program which edits credit limits within a given range (\$10,000 - \$15,000) would have three equivalence classes
    1. <\$10,000(invalid)
    2. Between \$10,000 & \$15,000(valid)
    3. >\$15,000(invalid)

## 2. Boundary Value Analysis

- This technique consist of building test cases and data that focus on the input and the output boundaries of a given function.
- Example:
  - In the same credit limit example, Boundary Analysis would test:
    1. Low Boundary +/- one(\$9,999 & \$10,001)
    2. On the Boundary ( \$10,000 & \$15,000)
    3. Upper Boundary -/+ one(\$14,999 & \$15,001)

### **3. Error Guessing**

- Error Guessing is a Software Testing technique on guessing the error which can prevail in the code.
- Test cases can be developed based upon the intuition and experience of the tester.
- Example:
  - Where one of the inputs is the date, a tester may try February 29, 2001.

## 4. Cause Effect Graphing

- It is a technique for developing test cases for programs from the high level specifications. (A high level specifications states desired characteristics of the system.)
- These characteristics can be used to derive test data.
- Example: A program that has specified responses to eight characteristic stimuli (called causes) given some input has 256 “types” of input.
  - . Poor approach is to generate 256 test cases
  - . Better would be to analyse program’s effects on the various types of inputs depending on programs specifications.

## **Advantages of Black Box Testing**

- More effective on larger units of code than Glass Box Testing.
- Tester need no knowledge of implementation.
- Tester and programmer are independent of each other.
- Tests are done from a user's point of view.
- Will help to expose any ambiguities or inconsistencies.

## **Disadvantages of Black Box Testing**

- Only a small number of inputs can actually be tested.
- Without clear and concise specifications, test cases are hard to design.
- Tester and programmer are independent of each other.
- May leave many program paths untested.
- Can't direct toward specific segment of code, may become complex.

# Glass Box Testing



- Glass Box Testing allows one to peek inside the “box”
- It assumes that the path of logic in a unit or program is known

Contd...

- Glass Box Testing consists of testing paths, branch by branch, to produce predictable results
- Focuses specifically on using internal knowledge of the software to guide the selection of the test data.
- Synonyms for Glass Box are Structural, White Box, Clear Box, etc.

## **Focus of Glass Box Testing**

- Statement Coverage
  - Execute all statement at least ones
- Decision Coverage
  - Execute each decision direction at least once
- Condition Coverage
  - Execute each decision with all possible outcomes
- Multiple Condition Coverage
  - Invoke each point of entry at least once

## 1. Statement Coverage

- Necessary but not sufficient, doesn't address all outcomes of decisions

- . For example

```
Begin()
If(func1())
 Open file1()
Else
 Shutdown();
```

- Here if the first If statement is true then shutdown will never occur

## **2. Decision Coverage**

- Validates the branch statements in software
- Outcomes and drawbacks of statement coverage
- Each decision is tested for a true and false value
- Each branch direction must be traversed at least once
- Branch like If-Else, While, for, do..while are to be evaluated for both true and false
- Test cases will be arrived with the help of a Decision table

## An Example

- Procedure liability(age, sex, married, premium)

Begin()

Premium=500;

if((age<25) and (sex=male) and (not married))

IF1

then premium=premium+1500;

Elseif((married) or (sex=female))

IF2

then premium=premium-200;

if((age>45) and (age<65))

IF3

then premium=premium-100;

End()

- Here variables are age,sex and married

Decisions are IF1, IF2 and IF3

# Decision Table

| Decision_Coverage | Age       | Sex    | Married | Test Case  |
|-------------------|-----------|--------|---------|------------|
| IF1               | <25       | Male   | FALSE   | (1) 23 M F |
| IF1               | <25       | Female | FALSE   | (2) 23 F F |
| IF2               | *         | Female | *       | (2)        |
| IF2               | >=25      | Male   | FALSE   | (3) 50 M F |
| IF3               | <=45      | Female | *       | (2)        |
| IF3               | >45 & <65 | *      | *       | (3)        |

### **3. Condition Coverage**

- Validates logic relations and conditions
- All the conditions should be executed at least once for both false and true
- Test cases will be arrived from using same Decision table by putting conditions in place of decisions.

# Test Conditions

| Condition_Coverage | Age  | Sex    | Married | Test Case  |
|--------------------|------|--------|---------|------------|
| IF1                | <25  | Female | FALSE   | (1) 23 F F |
| IF1                | >=25 | Male   | True    | (2) 30 M T |
| IF2                | *    | Male   | True    | (2)        |
| IF2                | *    | Female | FALSE   | (1)        |
| IF3                | <=45 | *      | *       | (1)        |
| IF3                | >45  | *      | *       | (3) 70 F F |
| IF3                | <65  | *      | *       | (2)        |
| IF3                | >=65 | *      | *       | (3)        |

### **3. Multiple Condition Coverage**

- Checks whether every possible combinations of boolean sub expression occurs at least once
- Test cases required for this method can be arrived at by using truth table of the conditions.
- Large number of test cases may be required for full multiple condition coverage

# Decision Table

| Multiple_Condition_Coverage | Age       | Sex    | Married | Test Case  |
|-----------------------------|-----------|--------|---------|------------|
| IF1                         | <25       | Male   | TRUE    | (1) 23 M T |
| IF1                         | <25       | Male   | FALSE   | (2) 23 M F |
| IF1                         | <25       | Female | TRUE    | (3) 23 F T |
| IF1                         | <25       | Female | FALSE   | (4) 23 F F |
| IF1                         | ≥25       | Male   | TRUE    | (5) 30 M T |
| IF1                         | ≥25       | Male   | FALSE   | (6) 70 M F |
| IF1                         | ≥25       | Female | TRUE    | (7) 50 F T |
| IF1                         | ≥25       | Female | FALSE   | (8) 30 F F |
| IF2                         | *         | Male   | TRUE    | (5)        |
| IF2                         | *         | Male   | FALSE   | (6)        |
| IF2                         | *         | Female | TRUE    | (7)        |
| IF2                         | *         | Female | FALSE   | (8)        |
| IF3                         | ≤45 & ≥65 | *      | *       | Impossible |
| IF3                         | ≤45 & <65 | *      | *       | (8)        |
| IF3                         | >45 & ≥65 | *      | *       | (6)        |
| IF3                         | >45 & <65 | *      | *       | (7)        |

# Grey Box Testing

- Grey box testing is a combination of white box testing and black box testing. The aim of this testing is to search for the defects if any due to improper structure or improper usage of applications.
- It is performed with limited information about the internal functionality of the system.
- Grey Box testers have access to the detailed design documents along with information about requirements.
- Grey Box tests are generated based on the state-based models, UML Diagrams or architecture diagrams of the target system.

# Grey Box Testing

- Pros:
  - Grey box testing provides combined benefits of both white box and black box testing
  - Grey-box tester handles can design complex test scenario more intelligently
  - Maintains the boundary between independent testers and developers
- Cons:
  - In grey-box testing, complete white box testing cannot be done due to inaccessible source code/binaries.

# **Comparison Between White Box, Black Box and Grey Box Testing**

- 1) **White Box Testing:** Detailed investigation of internal logic and structure
- 2) **Black Box Testing:** Testing without having any knowledge of the internal working of the application.
- 3) **Grey Box Testing:** White box + Black box = Grey box, a technique to test the application with limited knowledge of the internal working.

| <b>S.<br/>No.</b> | <b>Black Box Testing</b>                                                               | <b>Grey Box Testing</b>                                                                                                            | <b>White Box Testing</b>                            |
|-------------------|----------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------|
| 1.                | Analyses fundamental aspects only i.e. no proved edge of internal working              | Partial knowledge of internal working                                                                                              | Full knowledge of internal working                  |
| 2.                | Granularity is low                                                                     | Granularity is medium                                                                                                              | Granularity is high                                 |
| 3.                | Performed by end users and also by tester and developers (user acceptance testing)     | Performed by end users and also by tester and developers (user acceptance testing)                                                 | It is performed by developers and testers           |
| 4.                | Testing is based on external exceptions – internal behaviour of the program is ignored | Test design is based on high level database diagrams, data flow diagrams, internal states, knowledge of algorithm and architecture | Internal are fully known                            |
| 5.                | It is least exhaustive and time consuming                                              | It is somewhere in between                                                                                                         | Potentially most exhaustive and time consuming      |
| 6.                | It can test only by trial and error method                                             | Data domains and internal boundaries can be tested and over flow, if known                                                         | Test better: data domains and internal boundaries   |
| 7.                | Not suited for algorithm testing                                                       | Not suited for algorithm testing                                                                                                   | It is suited for algorithm testing (suited for all) |

## **Concluding Remarks:**

We can define software testing as an activity aimed at evaluating an attribute, or capability of a program to determine, that it meets its required specification.

Software testing can provide an independent view of the software to allow the business to appreciate and understand the risk of software implementation.

# **Model-based and Specification-based Testing**

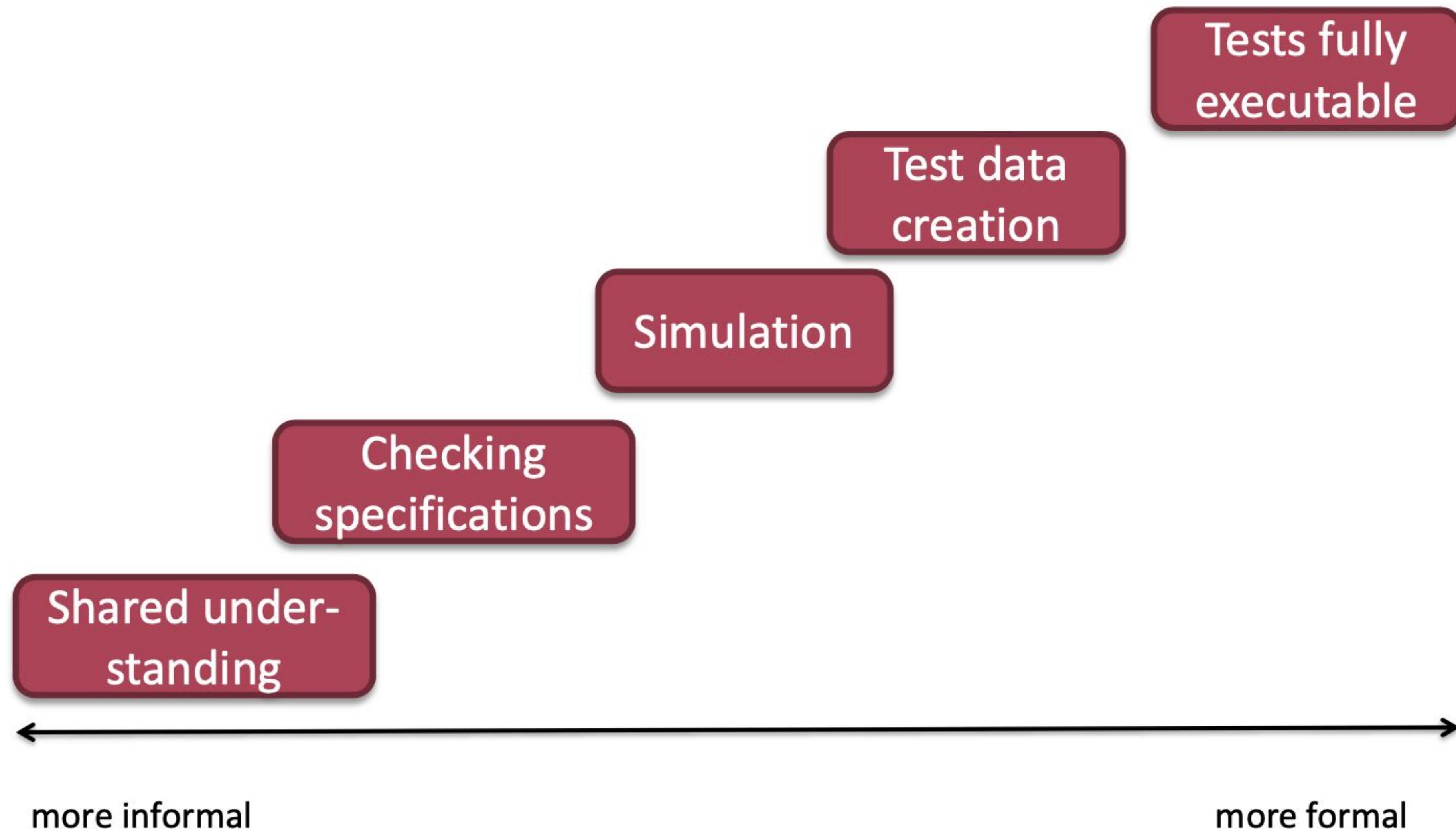
**5 Nov**

# **What is Model-based testing (MBT)?**

“Testing based on or involving models”

- Not just test generation
- Not just automatic execution

# Landscape of MBT Goals



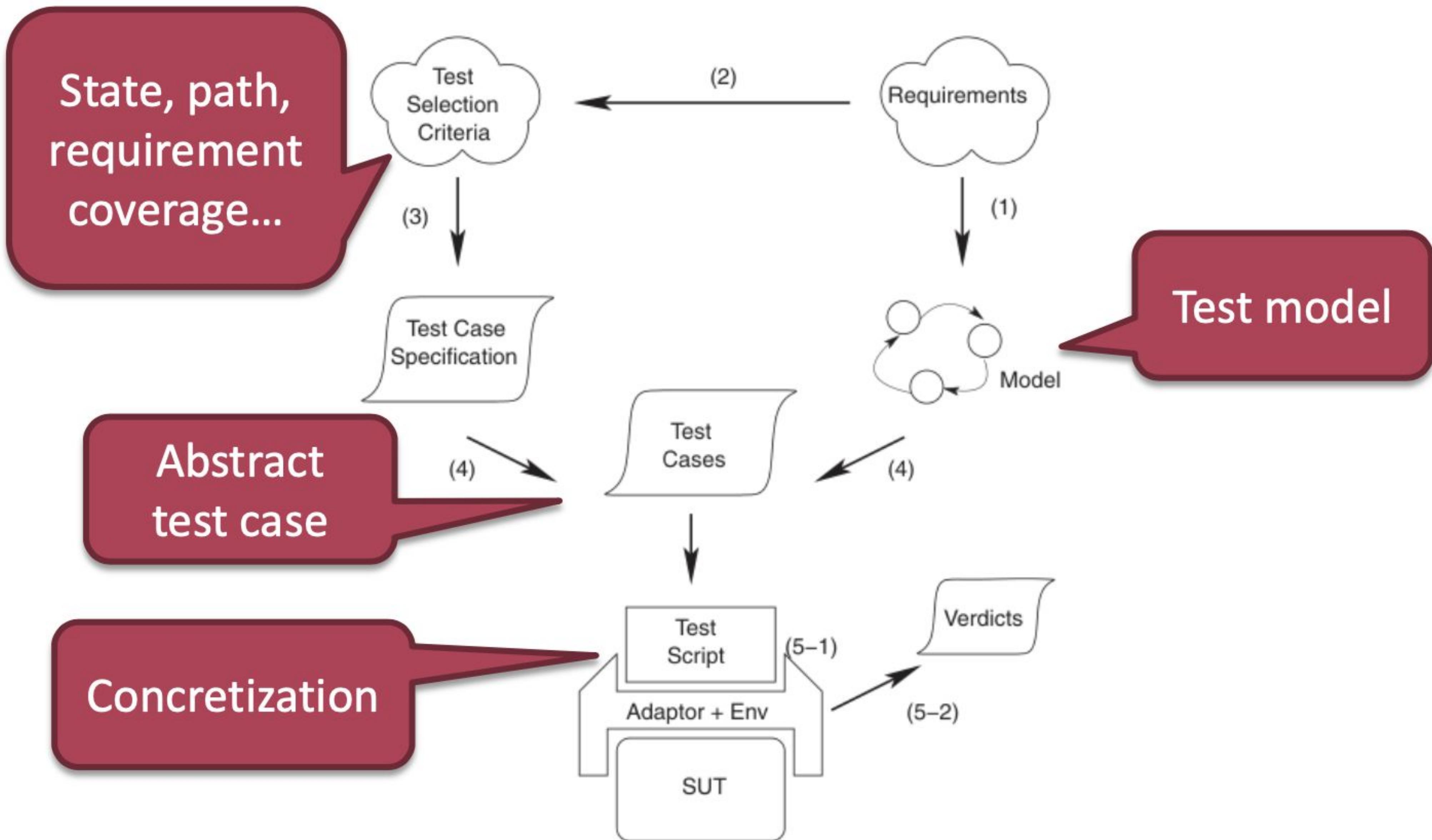
# **Benefits of using models**

- Close communication with stakeholders
  - . Understanding of domain and requirements
- Early testing: modeling/simulation/generation
- Higher abstraction level (manage complexity)

## Typical MBT

- MBT encompasses the processes and techniques for
  - automatic derivation of abstract test cases from abstract models
  - generation of concrete tests from abstract tests,
  - manual or automated execution of the resulting concrete test cases

# Typical MBT



## MBT Example

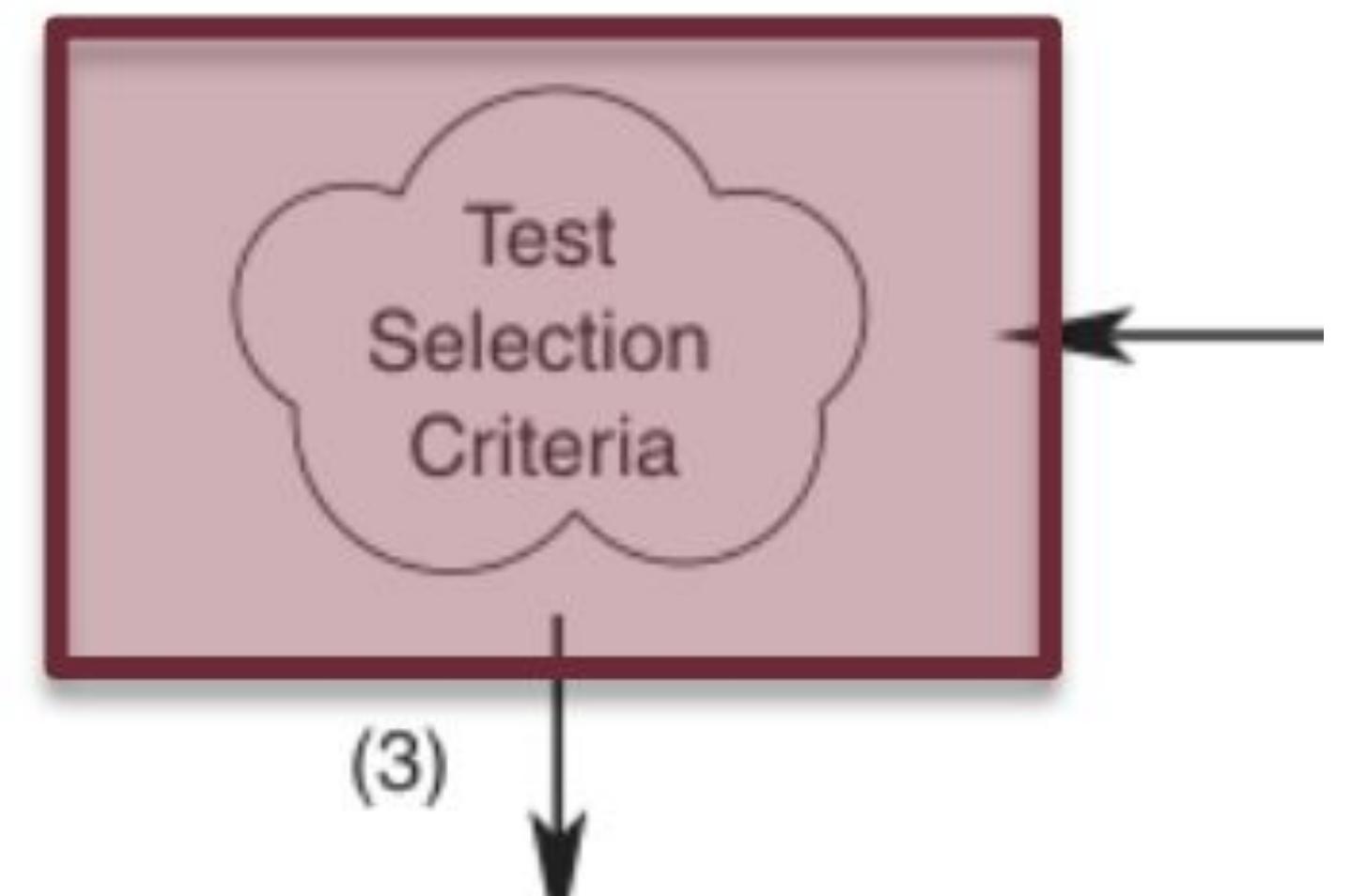
- Create test model using FSMs
- Use GraphWalker to generate test sequences
- Write adaptation to connect to Java code

# 1. Focus of the model

- System
  - . System and intended to be
  - . Conformance of model-SUT
- Usage
  - . Model environment/users
  - . Input to the system
- Test
  - . Model one or more test case
  - . E.g. sequences + evaluation

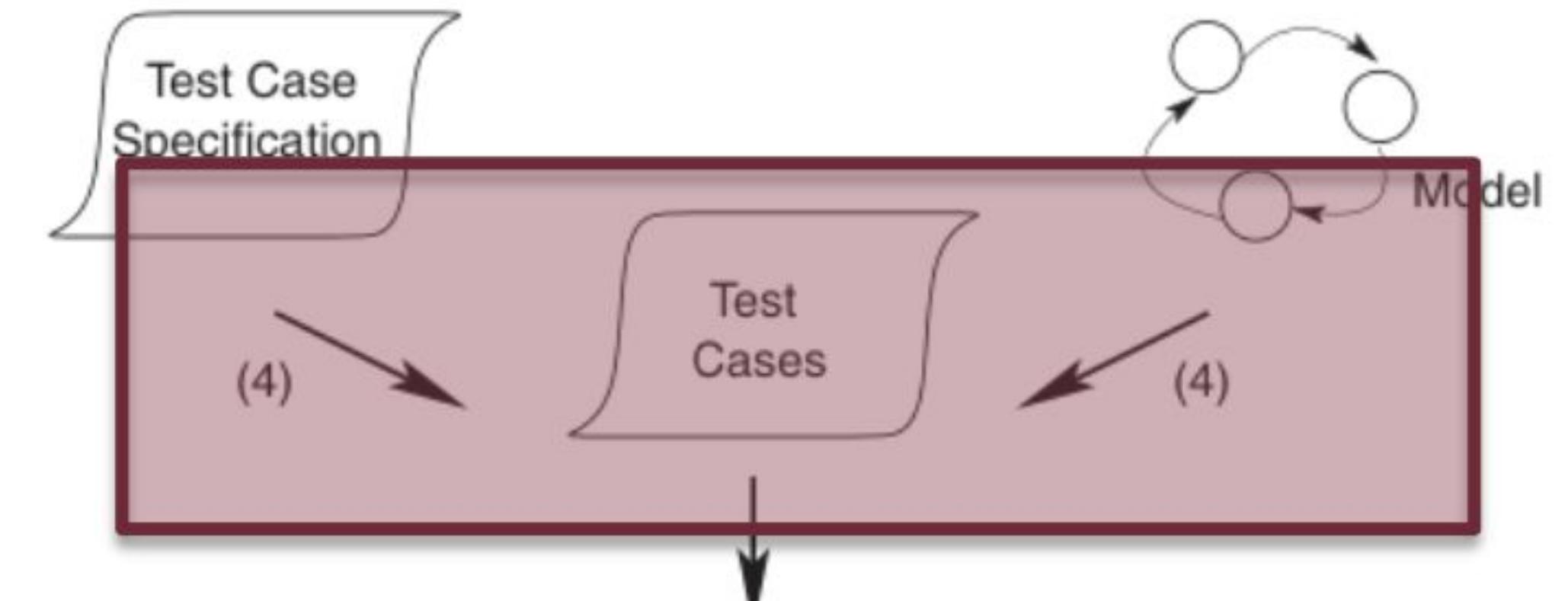
## 2. Typical test selection criteria

- Coverage Based
  - Requirements linked to the model
  - MBT model elements (state, transition, decision...)
  - Data related
- Random / stochastic
- Scenario and pattern based (Use case...)
- Project driven (risk, effort, resources...)



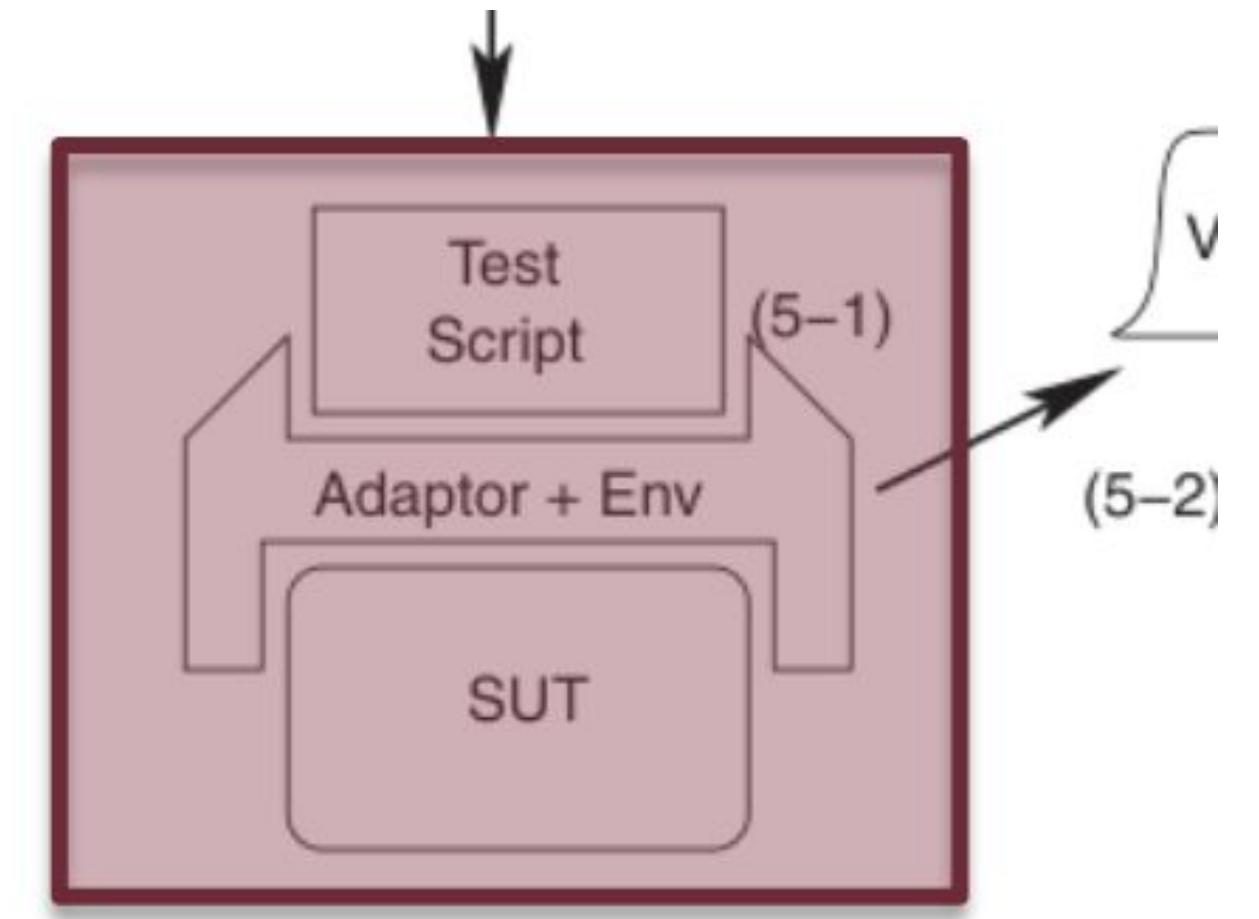
### 3. Test generation methods

- Direct Graph Algorithms
- FSM Testing
  - Homing and synchronising sequences
- LTS Testing
  - Equivalence and preorder relations
- Using model checkers
  - Equipment coverage
  - State coverage

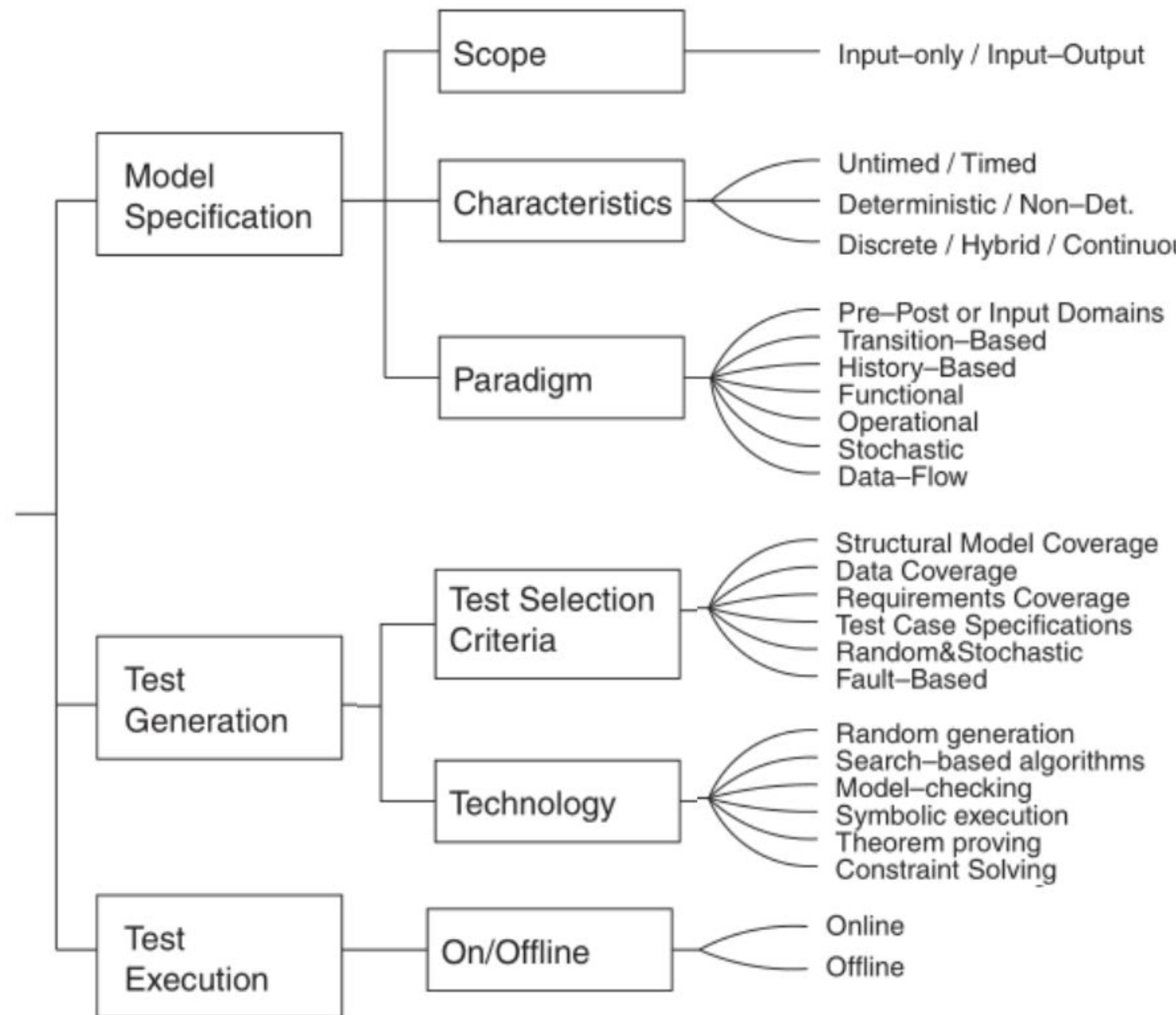


## 4. Abstract and Concrete Test Cases

- Abstract test case
  - Logical predicate instead of values
  - High level events and actions
- Concrete test case
  - Concrete input data
  - Detailed test procedure
- Adaptation Layer
  - Code blocks for each model-level event and action
  - Wrapper around the System-under-testing



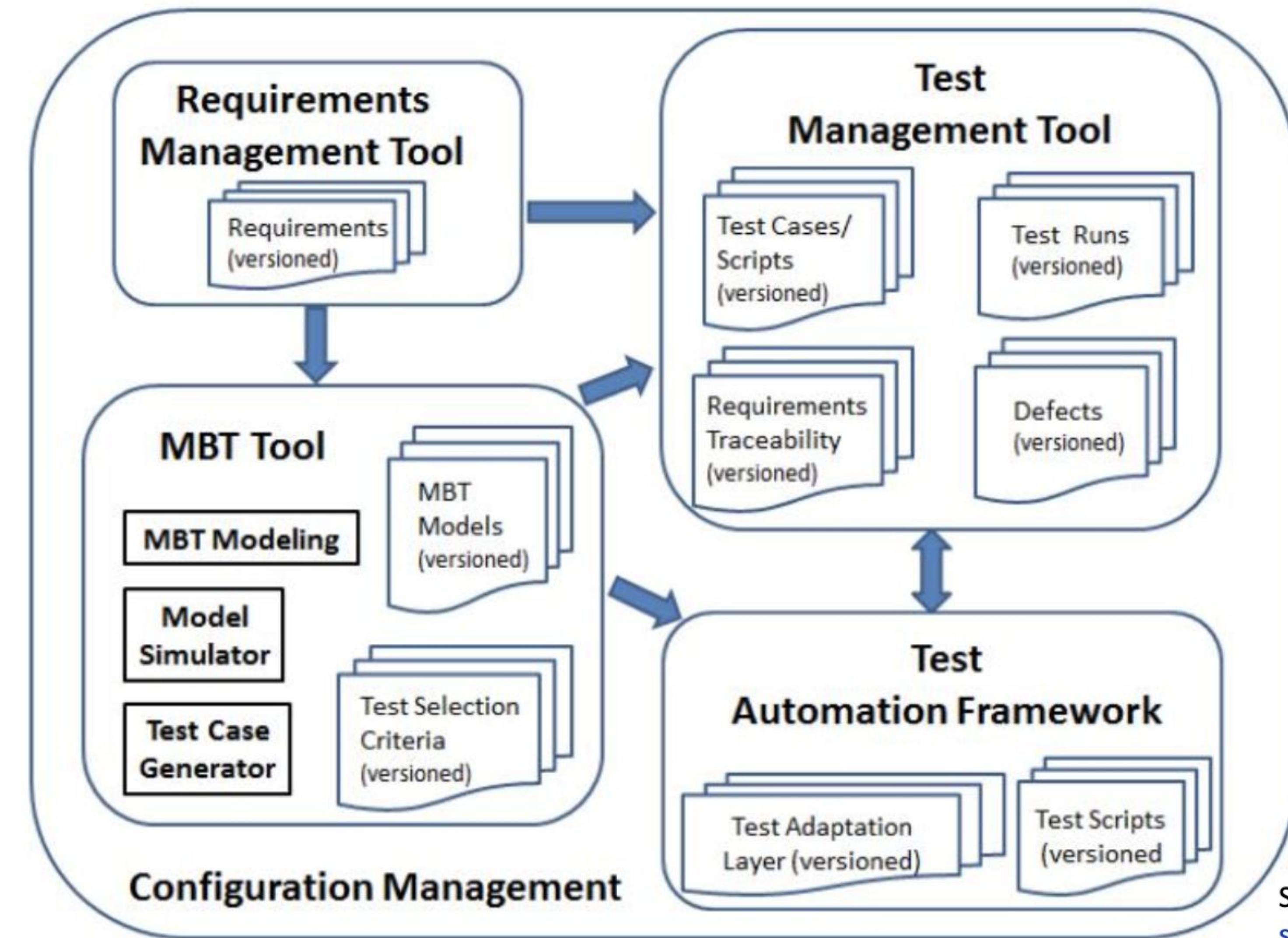
# Taxonomy of MBT approaches



## Typical Use Cases

- **Fast & easy**
  - . Simple modelling
  - . Using open tools
- **Full fledged**
  - . Complex, Commercial tool
  - . Full lifecycle support
- **Advanced**
  - . Custom modeling languages/tools

# MBT tool chain

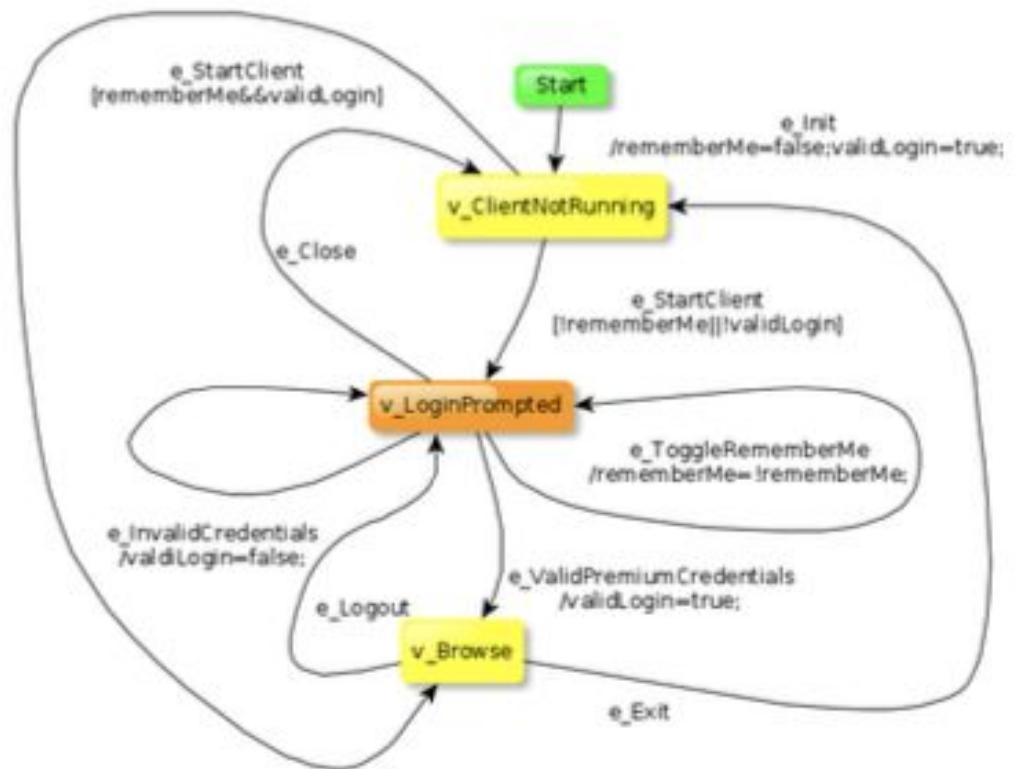


Source: [ISTQB syllabus](#)

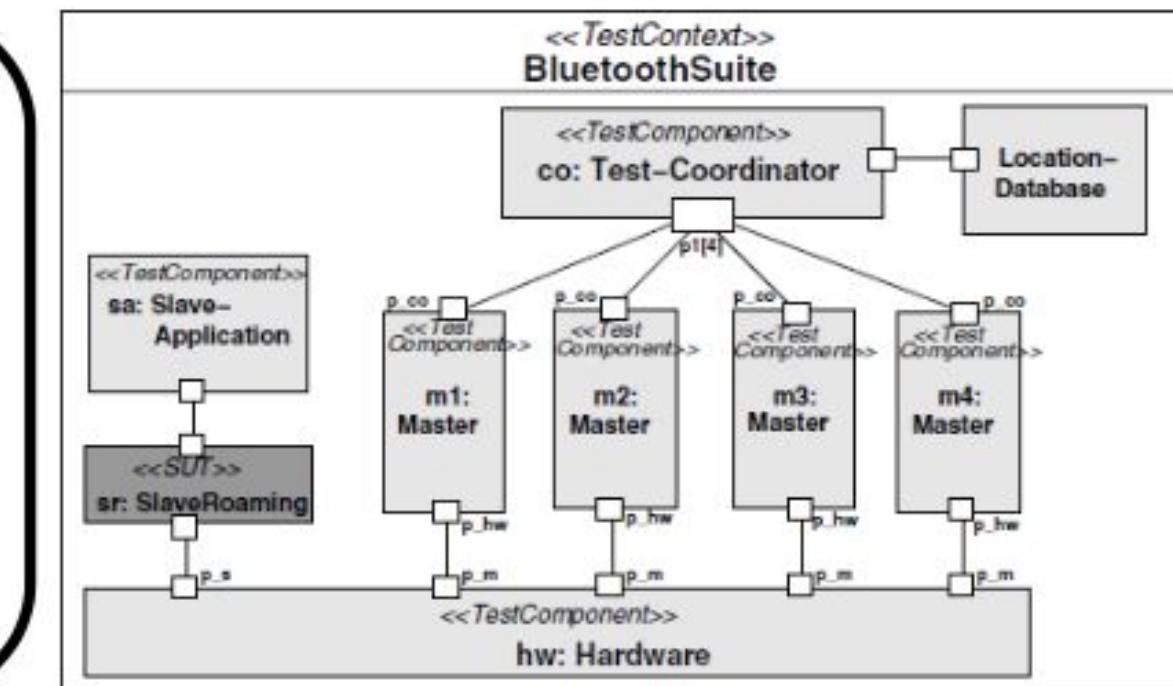
## Some Popular Tools

- GraphWalker (open source): FSM model
- Conformiq (industrial MBT tool): State machine model
- SpecExplorer (industrial MBT tool): C# model program
- CertifyIt: UML + OCL models
- MoMuT::UML (academic): UML state machines

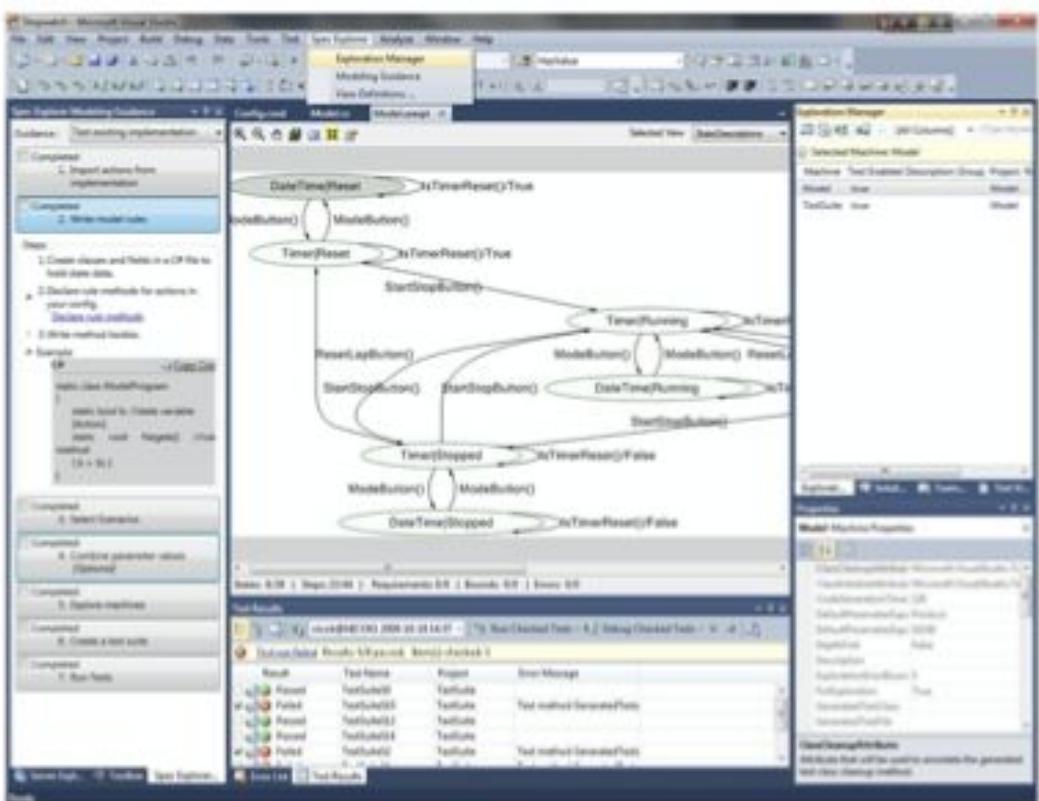
# Summary MBT



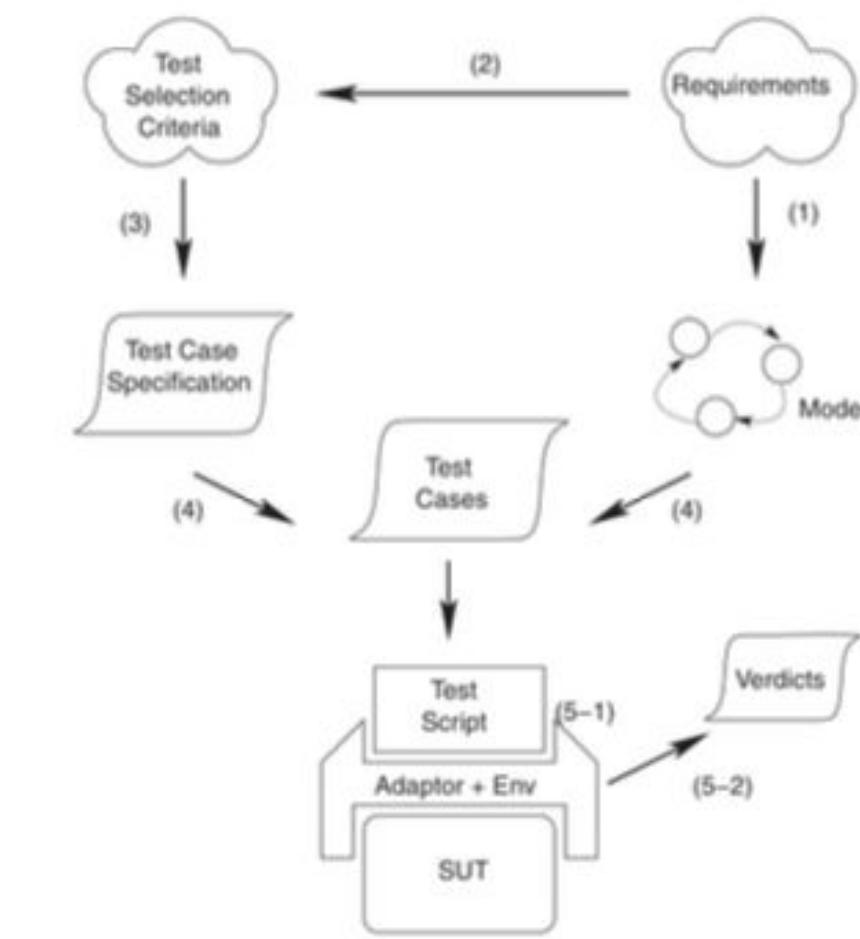
Many models,  
test goals and tools



MBT = using models in testing



Scaling from  
brainstorming to  
fully automatic  
test case generation



# Specifications Based Testing

- Adopted by software developers to ensure the quality of the software.
- Execution of software with the intention of finding failures
- Derives test cases directly from the specification
- Identify test conditions, test cases, and test data.

## **Advantages**

- Since test cases are generated without referring to the source code, the test case generation processes can be done independently.
- This enhances test case reusability and saves resources.
- Generating test cases from specifications, serve as an additional check on ambiguity or inconsistency.

# Cause-Effect Graphs

- In software testing, a cause–effect graph is a directed graph that maps a set of causes to a set of effects.
- The causes may be thought of as the input to the program, and the effects may be thought of as the output.
- Usually the graph shows the nodes representing the causes on one say, left side and the nodes representing the effects on the other say, right side.
- There may be intermediate nodes in between that combine inputs using logical operators such as AND and OR.

## CATEGORY-PARTITION METHOD (CPM)

- CPM categorises input domain of the software under characteristics of inputs and environment conditions.
- It combines different choices from different categories to form test frames
- CPM is a sound method as:
  - **Its step-by-step approach to test case generation.**
  - **Its supported by an automated tool to reduce manual effort.**
- Testing method, applicable to specs written in informal language.

## CHOICE RELATION FRAMEWORK

- It captures possible relations between choices in different categories
- Automatic deduction of certain choice relations is possible.
- Benefits:
  - **The effort spent on manually specifying choice relations is reduced.**
  - **The correctness of these relations is increased.**
- Choices with higher priorities are first used for test frame generation

## CLASSIFICATION-TREE METHOD (CTM)

- CTM introduces a hierarchical structure, called *classification tree*, to organise the relations among *classifications* and *classes*.
- Automatic deduction of certain choice relations is possible.
- Benefits:
  - Capture some constraints that can be clumsy to express in CPM.
  - Depict in a graphical form that greatly facilitates visualisation and comprehension
- CPM was better as:
  - Some test cases generated from CTM may still be infeasible
  - In CPM generated test set was affordable for testing under the limited resources

## LOGICAL EXPRESSION BASED TESTING

- Generating test cases from logical expressions is fundamental concept
- These predicate based methods can also be applied to logical expressions derived from specifications based testing techniques:
  - . **Foster's Method:** test cases *sensitive* enough to distinguish the correct behaviour of program from its incorrect implementations.
  - . **Boolean OperatoR(BOR):** BOR method can generate a test set that detects any *Boolean operator fault*
  - . **Modified condition/decision coverage (MC/DC) criterion:** generate test cases from logical predicates of program source.

Thank You

# Cause and Effect Graphs



- Testing would be a lot easier:
  - if we could automatically generate test cases from requirements.
- Work done at IBM:
  - Can requirements specifications be systematically used to design functional test cases?

# Cause and Effect Graphs



- Examine the requirements:
  - restate them as logical relation between inputs and outputs.
  - The result is a Boolean graph representing the relationships
    - called a **cause-effect graph**.

# Cause and Effect Graphs



- Convert the graph to a decision table:
  - each column of the decision table corresponds to a test case for functional testing.

# **Steps to create cause-effect graph**

- Study the functional requirements.
- Mark and number all causes and effects.
- Numbered causes and effects:
  - become nodes of the graph.

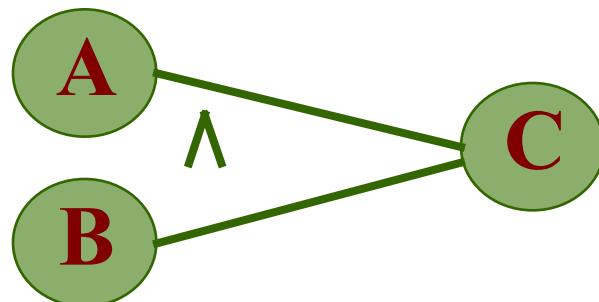
# **Steps to create cause-effect graph**

- Draw causes on the LHS
- Draw effects on the RHS
- Draw logical relationship between causes and effects
  - as edges in the graph.
- Extra nodes can be added
  - to simplify the graph

# Drawing Cause-Effect Graphs

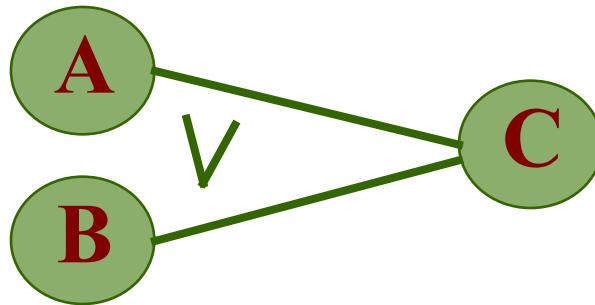


If A then B

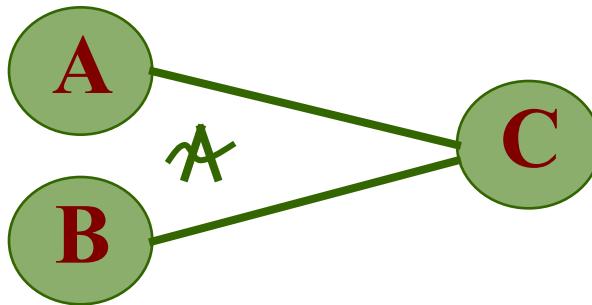


If (A and B) then C

# Drawing Cause-Effect Graphs

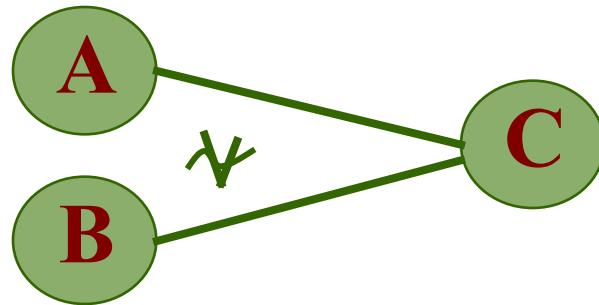


If (A or B)then C



If (not(A and B))then C

# Drawing Cause-Effect Graphs



If (not (A or B)) then C



If (not A) then B

# Cause-effect graph Example



- A water level monitoring system
  - used by an agency involved in flood control.
  - **Input:** level(a,b)
    - a is the height of water in dam in meters
    - b is the rainfall in the last 24 hours in cms

# Cause effect graph- Example



- Processing
  - The function calculates whether the level is safe, high, or low.
- Output
  - message on screen
    - level=safe
    - level=high
    - invalid syntax

# Cause effect graph- Example



- We can separate the requirements into 5 clauses:
  - 1 first five letters of the command is “level”
  - 2 command contains exactly two parameters
    - separated by comma and enclosed in parentheses

# Cause effect graph- Example



- Parameters A and B are real numbers:
  - 3● such that the water level is calculated to be low
  - 4● or safe.
- The parameters A and B are real numbers:
  - 5● such that the water level is calculated to be high.

# Cause effect graph- Example



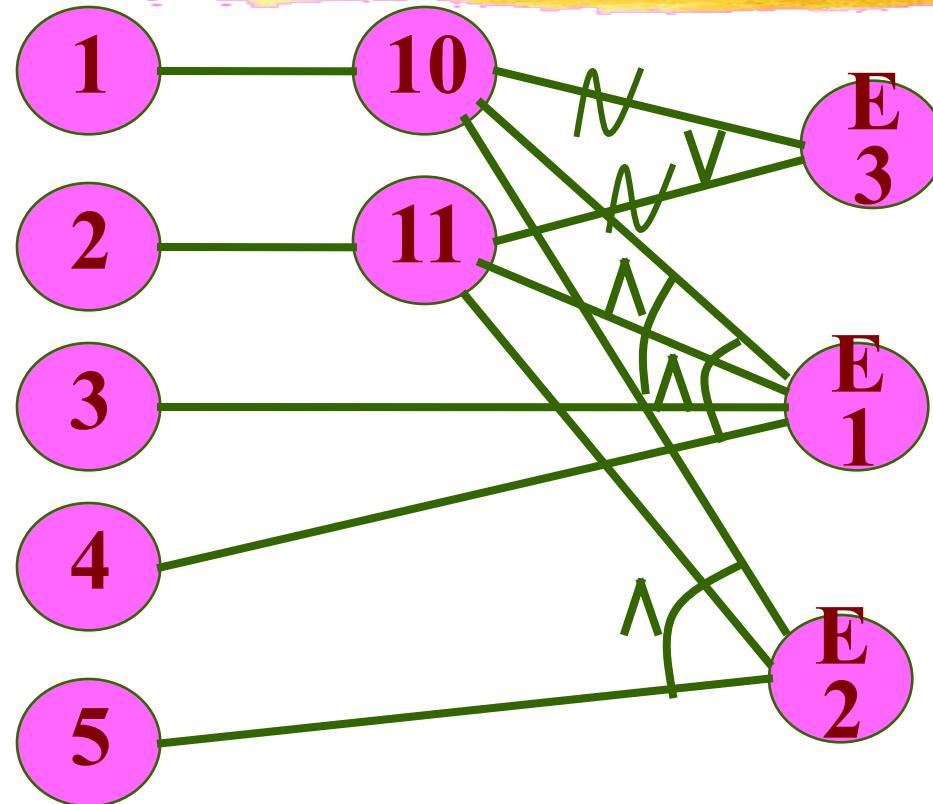
- 10• Command is syntactically valid
- 11• Operands are syntactically valid.

# Cause effect graph- Example



- Three effects
  - level = safe E1
  - level = high E2
  - invalid syntax E3

# Cause effect graph- Example



# Cause effect graph- Example



- Put a row in the decision table for each cause or effect:
  - in the example, there are five rows for causes and three for effects.

# Cause effect graph- Example



- The columns of the decision table correspond to test cases.
- Define the columns by examining each effect:
  - list each combination of causes that can lead to that effect.

# Cause effect graph- Decision table

|          | Test 1 | Test 2 | Test 3 | Test 4 | Test 5 |                |
|----------|--------|--------|--------|--------|--------|----------------|
| Cause 1  | I      | I      | I      | S      | I      |                |
| Cause 2  | I      | I      | I      | X      | S      | I = Invoked    |
| Cause 3  | I      | S      | S      | X      | X      | x = don't care |
| Cause 4  | S      | I      | S      | X      | X      | s = suppressed |
| Cause 5  | S      | S      | I      | X      | X      |                |
| Effect 1 | P      | P      | A      | A      | A      | P = present    |
| Effect 2 | A      | A      | P      | A      | A      | A = absent     |
| Effect 3 | A      | A      | A      | P      | P      |                |

# Cause effect graph- Example



- Theoretically we could have generated 32 test cases.
  - Using cause effect graphing technique reduces that number to 5.

# Cause effect graph



- Not practical for systems which:
  - include timing aspects
  - feedback from processes is used for some other processes.

# **Object-Oriented Testing and Testing vs. Correctness Proof**

**Nov 24**

## Learning objectives

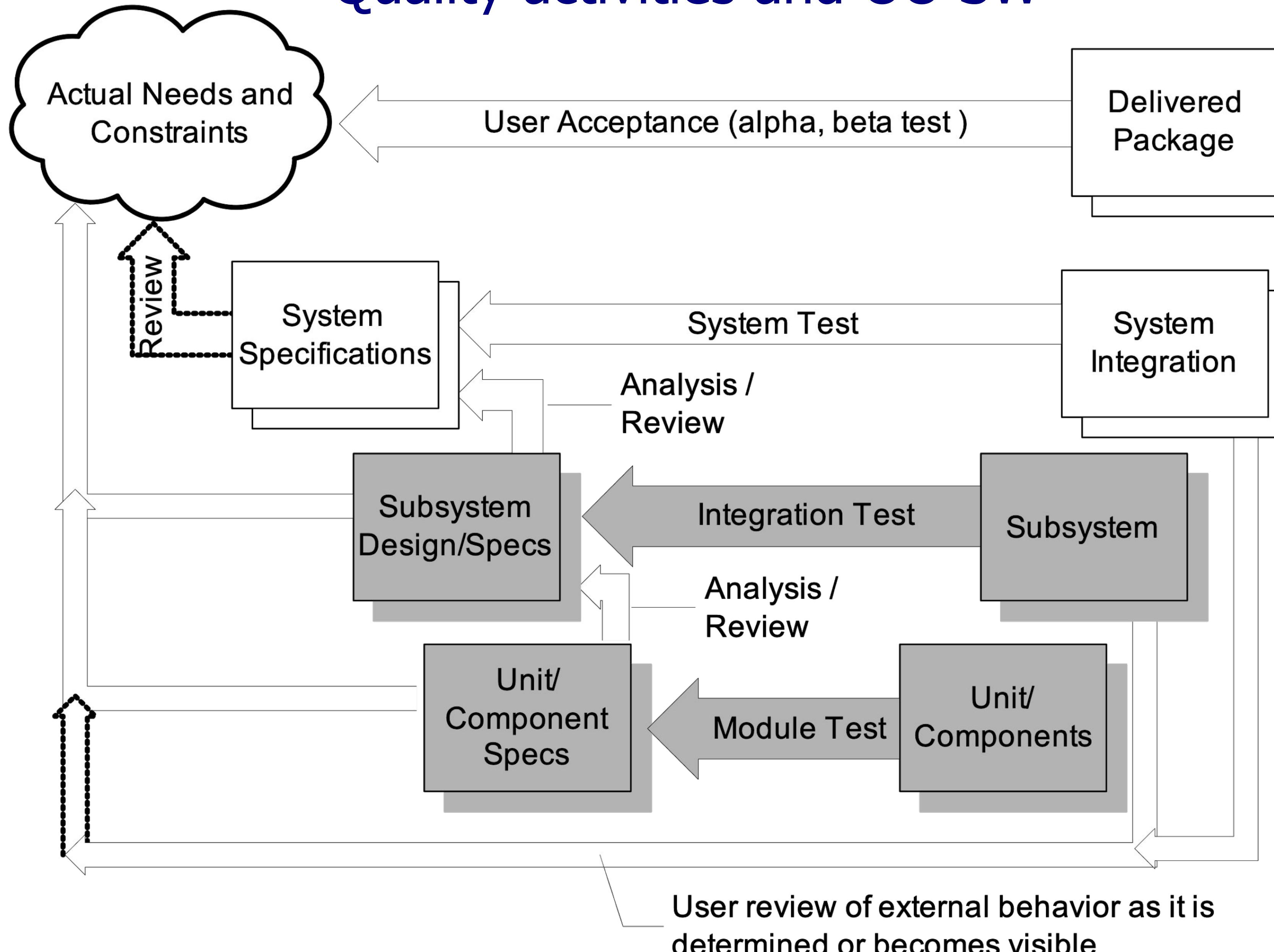
- Understand how object orientation impacts software testing
  - What characteristics matter? Why?
  - What adaptations are needed?
    - Understand basic techniques to cope with each key characteristic
- Understand staging of unit and integration testing for OO software (intra-class and inter-class testing)

# Characteristics of OO Software

Typical OO software characteristics that impact testing

- State dependent behavior
- Encapsulation
- Inheritance
- Polymorphism and dynamic binding
- Abstract and generic classes
- Exception handling

# Quality activities and OO SW



# Orthogonal approach: Stages

## Intra-Class Testing

|                          |            |
|--------------------------|------------|
| Super/subclass relations | Functional |
| State machine testing    |            |
| Augmented state machine  | Structural |
| Data flow model          |            |
| Exceptions               |            |
| Polymorphic binding      |            |

## Inter-Class Testing

|                            |            |
|----------------------------|------------|
| Hierarchy of clusters      | Functional |
| Functional cluster testing |            |
| Data flow model            | Structural |
| Exceptions                 |            |
| Polymorphic binding        |            |

System and Acceptance Testing (unchanged)

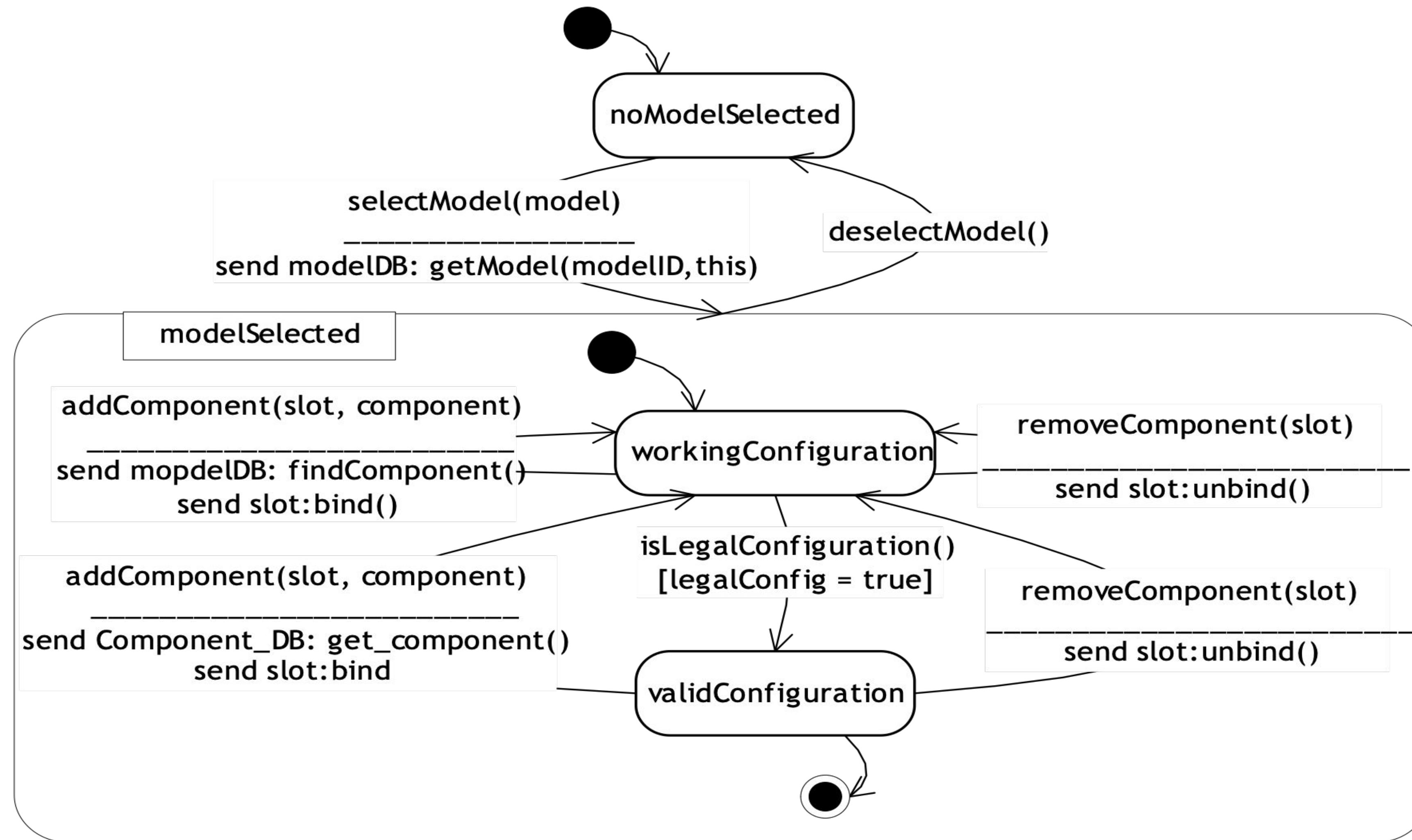
## 1. Intraclass State Machine Testing

- Basic idea:
  - The state of an object is modified by operations
  - Methods can be modeled as state transitions
  - Test cases are sequences of method calls that traverse the state machine model
- State machine model can be derived from specification (functional testing), code (structural testing), or both

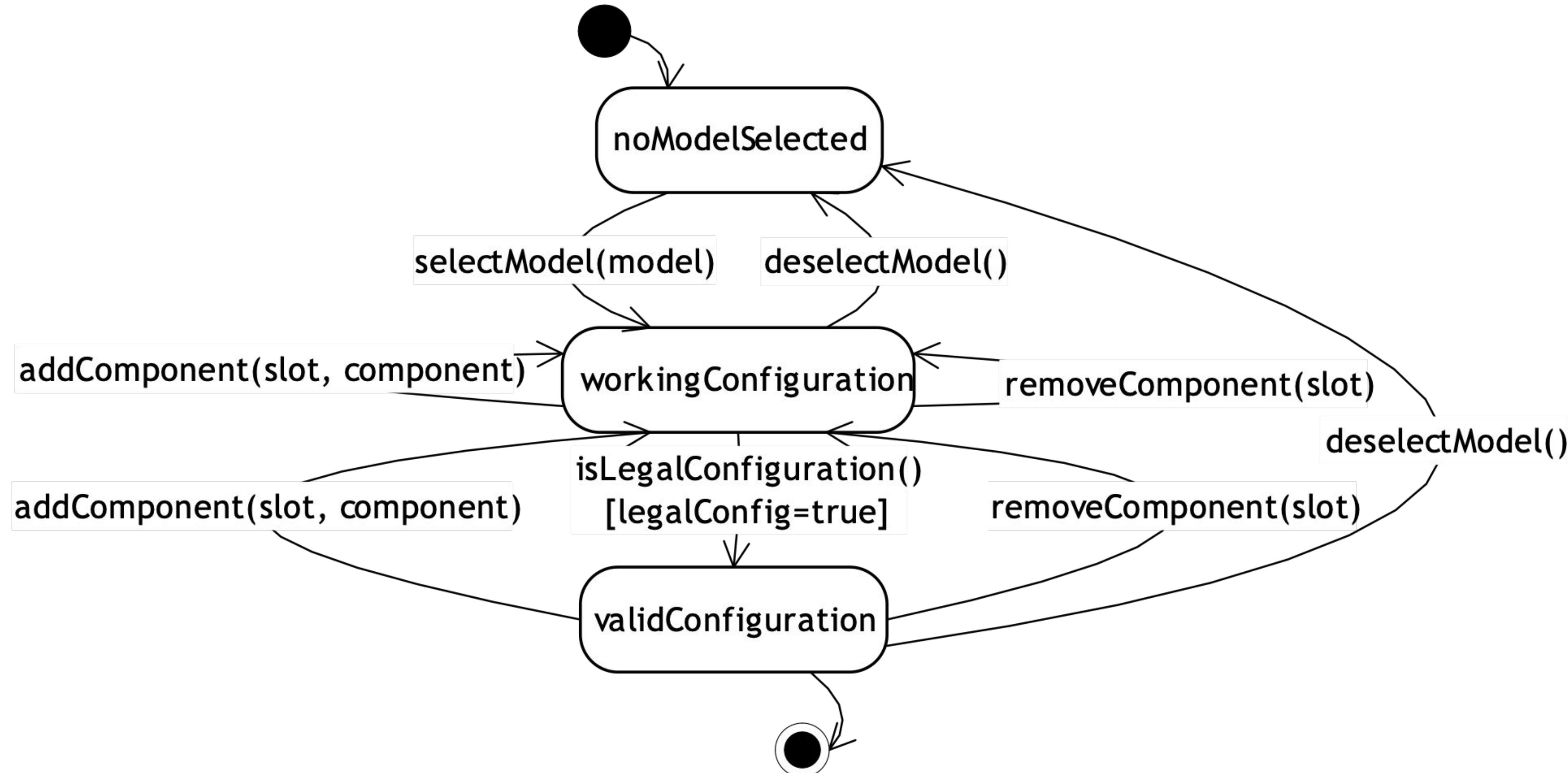
### Testing with State Diagrams

- A statechart (called a “state diagram” in UML) may be produced as part of a specification or design

# Statecharts specification



# From Statecharts to FSMs

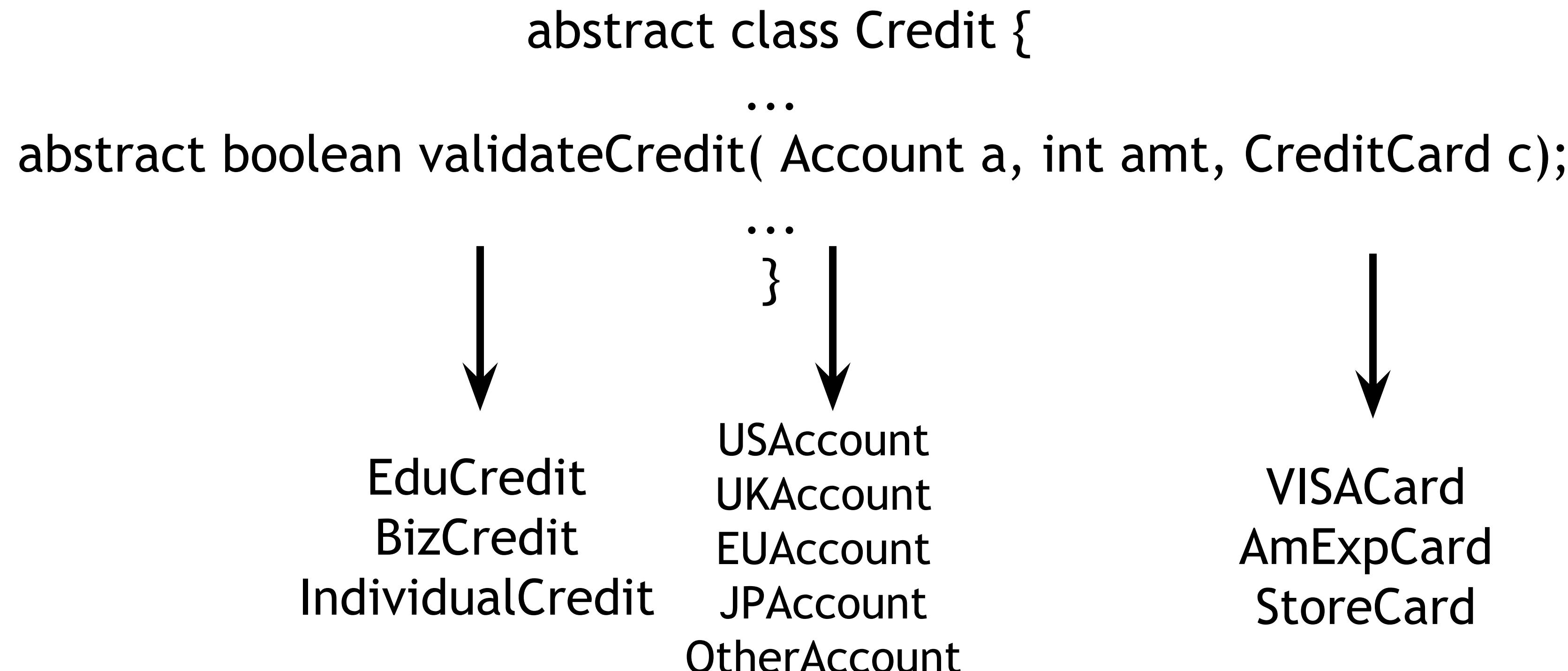


## Statechart based criteria

- In some cases, “flattening” a Statechart to a finite-state machine may cause “state explosion”
  - Particularly for super-states with “history”
- Alternative: Use the statechart directly
- Simple transition coverage:
  - execute all transitions of the original Statechart
- Incomplete transition coverage:
  - useful for complex statecharts and strong time constraints (combinatorial number of transitions)

## 2. Polymorphism and dynamic binding

Combinatorial explosion problem:



The combinatorial problem:  $3 \times 5 \times 3 = 45$  possible combinations of dynamic bindings (just for this one method!)

# The combinatorial approach

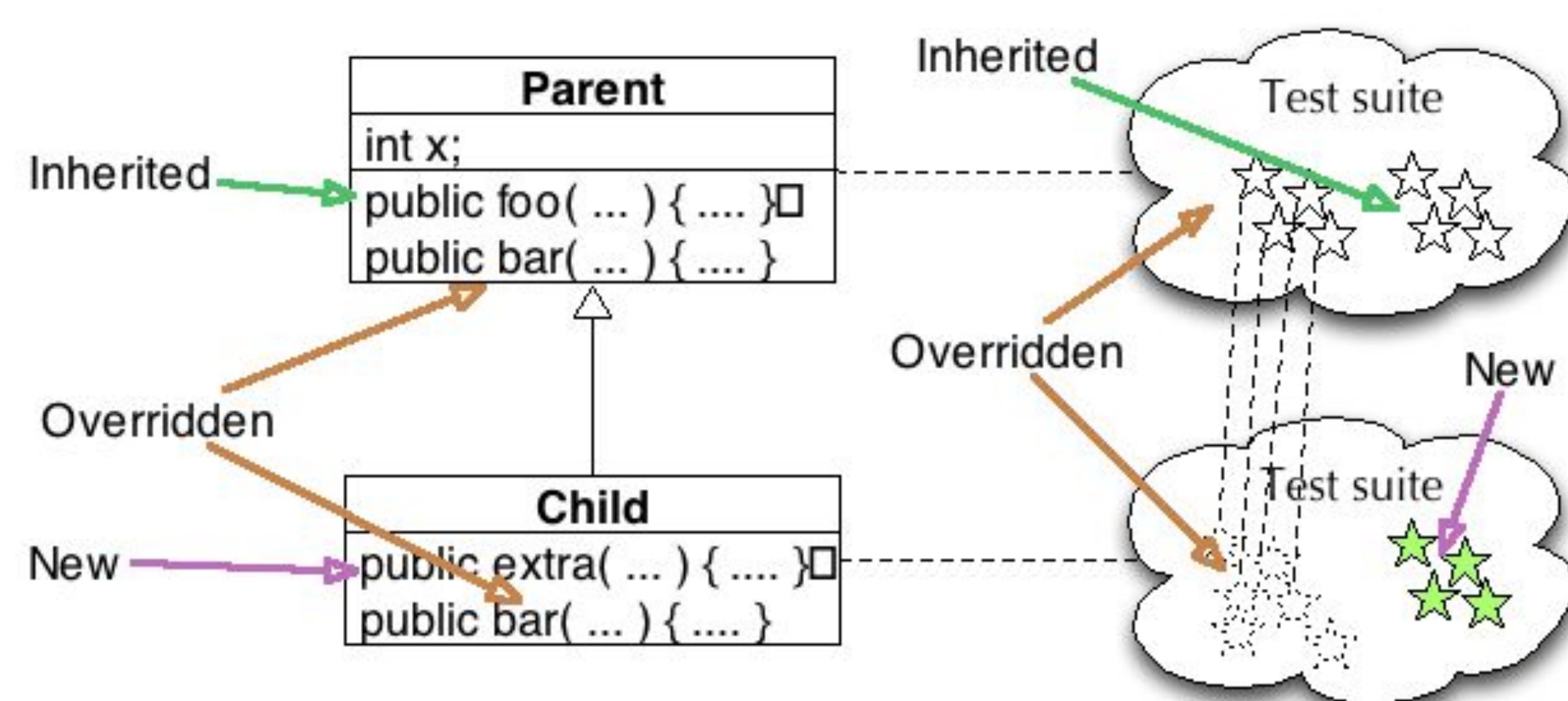
Identify a set of combinations that cover all pairwise combinations of dynamic bindings

| Account      | Credit           | creditCard   |
|--------------|------------------|--------------|
| USAccount    | EduCredit        | VISACard     |
| USAccount    | BizCredit        | AmExpCard    |
| USAccount    | individualCredit | ChipmunkCard |
| UKAccount    | EduCredit        | AmExpCard    |
| UKAccount    | BizCredit        | VISACard     |
| UKAccount    | individualCredit | ChipmunkCard |
| EUAccount    | EduCredit        | ChipmunkCard |
| EUAccount    | BizCredit        | AmExpCard    |
| EUAccount    | individualCredit | VISACard     |
| JPAccount    | EduCredit        | VISACard     |
| JPAccount    | BizCredit        | ChipmunkCard |
| JPAccount    | individualCredit | AmExpCard    |
| OtherAccount | EduCredit        | ChipmunkCard |
| OtherAccount | BizCredit        | VISACard     |
| OtherAccount | individualCredit | AmExpCard    |

### 3. Inheritance

- When testing a subclass ...
  - We would like to re-test only what has not been thoroughly tested in the parent class
    - for example, no need to test hashCode and getClass methods inherited from class Object in Java
  - But we should test any method whose behavior may have changed
    - even accidentally!

#### Testing Inheritance



## 4. Testing generic classes

A Generic class

```
class PriorityQueue<Elem Implements Comparable>
{ . . . }
```

*is designed to be instantiated with many different parameter types*

```
PriorityQueue<Customers>
```

```
PriorityQueue<Tasks>
```

A generic class is typically designed to behave consistently some set of permitted parameter types.

Testing can be broken into two parts

- Showing that some instantiation is correct
- showing that all permitted instantiations behave consistently

## 5. Exception handling

exceptions  
create implicit  
control flows  
and may be  
handled by  
different  
handlers

```
void addCustomer(Customer theCust) {
 customers.add(theCust);
}
public static Account
newAccount(...) throws InvalidRegionException {
 Account thisAccount = null;
 String regionAbbrev = Regions.regionOfCountry(
 mailAddress.getCountry());
 if (regionAbbrev == Regions.US) {
 thisAccount = new USAccount();
 } else if (regionAbbrev == Regions.UK) {

 } else if (regionAbbrev == Regions.Invalid) {
 throw new
 InvalidRegionException(mailAddress.getCountry());
 }
 ...
}
```

# Testing exception handling

- Impractical to treat exceptions like normal flow
  - too many flows: every array subscript reference, every memory allocation, every cast, ...
  - multiplied by matching them to every handler that could appear immediately above them on the call stack.
  - many actually impossible
- So we separate testing exceptions
  - and ignore program error exceptions (test to prevent them, not to handle them)
- What we do test: Each exception handler, and each explicit throw or re-throw of an exception

## Summary

- Several features of object-oriented languages and programs impact testing
  - from encapsulation and state-dependent structure to generics and exceptions
  - but only at unit and subsystem levels
  - and fundamental principles are still applicable
- Basic approach is orthogonal
  - Techniques for each major issue (e.g., exception handling, generics, inheritance, ...) can be applied incrementally and independently

# Testing vs. Correctness Proof

- Testing Can Never Consider All Possible Operating Conditions
- Approaches Focus on Identifying Test Cases and Scenarios of Access for Likely Behaviour
  - If Bridge OK at 1 Ton, OK < 1 Ton
  - What is an Analogy in Software?
    - If a System Works with 100,000 Data Items, it may be Reasonable to Assume it Works for < 100,000 Items
  - Realistic Example

```
bottom := table'first; top := table'last;
while bottom < top loop
 if (bottom + top) rem 2 ≠ 0 then
 middle := (bottom + top - 1) / 2;
 else
 middle := (bottom + top) / 2; ←
 end if;
 if key ≤ table (middle) then
 top := middle;
 else
 bottom := middle + 1;
 end if;
end loop;
```

if we omit this  
the routine  
works if the else  
is never hit!  
(i.e. if size of table  
is a power of 2)

# Testing vs. Correctness Proof

## ○ Realistic Example

```
bottom := table'first; top := table'last;
while bottom < top loop
 if (bottom + top) rem 2 ≠ 0 then
 middle := (bottom + top - 1) / 2;
 else
 middle := (bottom + top) / 2; ←
 end if;
 if key ≤ table (middle) then
 top := middle;
 else
 bottom := middle + 1;
 end if;
end loop;
```

if we omit this  
the routine  
works if the else  
is never hit!  
(i.e. if size of table  
is a power of 2)

# Four Goals of Testing

- Goal 1: Testing Must be Based on Sound and Systematic Techniques
  - Test Different Execution Paths
  - Provides Understanding of Product Reliability
  - May Require the Insertion of Testing Code  
(e.g., Timing, Conditional Compilation, etc.)
  
- Goal 2: Testing Must Help Locate Errors
  - Test Cases Must be Organised to Assist in the Isolation of Errors
  - This Facilitates Debugging
  - This Requires Well Thought Out Testing Paradigm as Software is Developed

# Four Goals of Testing

- Goal 3: Testing Must be Repeatable
  - Same Input Produces the Same Output
  - Execution Environment Influences Repeatability
    - if  $x=0$  then return true else return false;
    - If  $x$  not Initialised, Behaviour Can't be Predicted
    - In C, the Memory Location is Access for a Value (Which Could have Data from a Prior Execution)
  
- Goal 4: Testing Must be Accurate
  - Depends on the Precision of SW Specification
    - Test Cases Created from Scenarios of Usage
  - Mathematical Formulas for Expressing SW can Greatly Assist in Testing Process

# Analysis and Testing

- Analysis Can
  - Address Verification of Highly Desired Software Properties
  - Performed by People or Instruments (Software)
  - Apply Formal Methods or Intuition/Experience
  - Applied from Specification to Deployment
- We'll Consider Both Informal and Formal Techniques
- Informally
  - Code (Design) Walkthroughs
  - Code (Design) Inspections
- Formally
  - Proving Correctness
  - Pre- and Post- Conditions of Design/Code/Program

## (a) Informal Analysis –Walkthroughs

- Organized Activity with Group Participants
- Participants Mimic “Computer” – Paper Execution
  - Test Cases are Selected in Advance
  - Execution by Hand – Simulate Test
  - Record State (Results) on Paper, Whiteboard, Computer File, etc.
- Key Personnel
  - Designer (Design walkthrough)
  - Developer (Code walkthrough)
  - Attendees (Technical – Designers/Developers)
  - Moderator (Control Meeting)
  - Secretary
    - Take Notes for Changes
    - Subsequent Walkthroughs Verified Against Notes

## (a) Informal Analysis –Walkthroughs

- Recommended Guidelines
  - Small Number of People (Three to Five)
  - Participants Receive Written Documentation From the Designer a Few Days Before the Meeting
  - Predefined Duration of Meeting (A Few Hours)
  - Focus On the ***Discovery*** of Errors, Not on Fixing Them
  - Participants: Designer, Developer, Moderator, and a Secretary
  - Foster Cooperation

## TRACK Files

- Utilization of Track files to Track Program State in the Event of Error
- File Located on each User's Machine and can be Tracked and Forwarded when Errors Occur
  - Key User's Identified
  - File sent by User or Retrieved by Administrator
  - File Overwrites Periodically
- Software Developers can Leverage Extensive Source Code Infrastructure for Tracking Errors
  - Framework Extensible
  - Developers can Include their Own Tracking

## (b) Formal Verification – Towards Correctness

- Axiomatic Semantics is a Field of Computer Science that Seeks to Prove the Correctness of Software
- Techniques Utilizes
  - Pre Condition: True Before Code Executes
  - Post Condition: True After Code Executes
  - Proof Rules: Programming Language Statements
- Axiomatic Semantics Can be Applied at:
  - Program, Module, Method, and Code Segments
- Notationally: If Claim 1 and Claim 2 have been Proven, then you Can Deduce Claim 3
- Other Possibilities are:
  - $\{\text{true} \text{ and } a = 0 \text{ and } b = 0\}$
  - $\{\text{true}\}$  – pre-condition

```
{true} – pre-condition
begin
 read (a); read (b);
 x := a + b;
 write (x);
end
{output = input1 + input2} – post-condition
```

# Proving Correctness

- Partial Correctness
  - Validity of {Pre} Program {Post} Guarantees that if Pre holds Before Execution of Program, *and if program terminates*, then Post will be Achieved
- Total Correctness
  - Pre Guarantees Program's Termination *and* the Truth of Post

*These problems are undecidable!!!*

## Correctness at Module Level

- Prove Correctness of Individual Operations and Use this to Prove Correctness of Overall Module

```
module TABLE;
exports
 type Table_Type (max_size: NATURAL): ?;
 no more than max_size entries should be
 stored in a table; user modules must guarantee this
 procedure Insert (Table: in out TableType ; ELEMENT: in ElementType);
 procedure Delete (Table: in out TableType; ELEMENT: in ElementType);
 function Size (Table: in Table_Type) return NATURAL;
 provides the current size of a table
 ...
end TABLE
```

# Module Correctness

- Suppose that the Following have been Proven:

{true}

Delete (Table, Element);

{Element  $\notin$  Table};

{Size (Table) < max\_size}

Insert (Table, Element)

{Element  $\in$  Table};

- Using these Two, we can Prove Properties of Module Table

- Consider the Sequence:

Insert (T, x);

Delete (T, x);

- After the Delete, we Guarantee X is not Present in T

# Relative Correctness: A Bridge between Testing and Proving

Nafi Diallo, Wided Ghardallou, and Ali Mili

- Relative correctness is the property of a program to be more-correct than another with respect to a specification
- Whereas, traditionally we deploy proof methods on correct programs to prove their correctness
- we deploy testing methods on incorrect programs to detect and remove their faults,
- Relative correctness enables us to bridge this gap by showing that we can deploy static analytical methods to an incorrect program
- To prove that while it may be incorrect.
- Relative correctness has other broad implications for testing and proving.

## Summary

- Verification can span many aspects of the SDLC process from the specification through deployment
- Testing ideas/concepts from different perspectives
  - Testing in the large vs. testing in the Small
  - White box testing vs. black box Testing
  - Condition criteria
  - Informal vs. formal Techniques
  - Debugging/walkthroughs/inspections
- Objective of all is to yield programs that work correctly from execution standpoint (no Errors) and from a user Standpoint (behaves as Expected).