

Software Engineering

Dr: Abhishek Vaish

Overview of the Unit 1

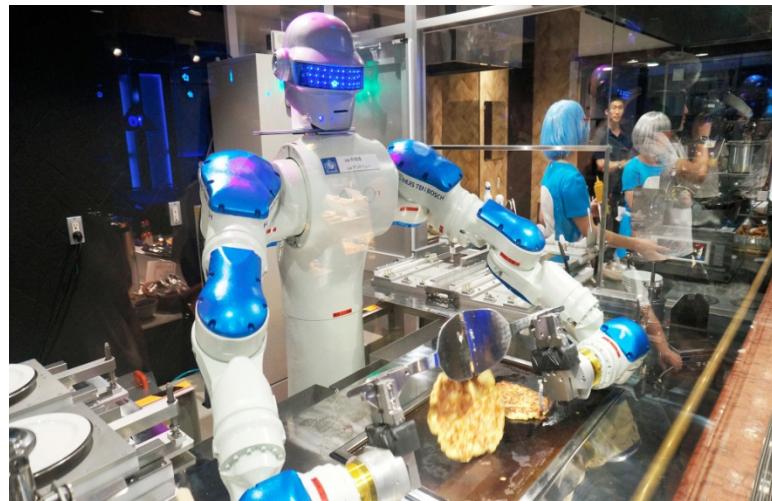
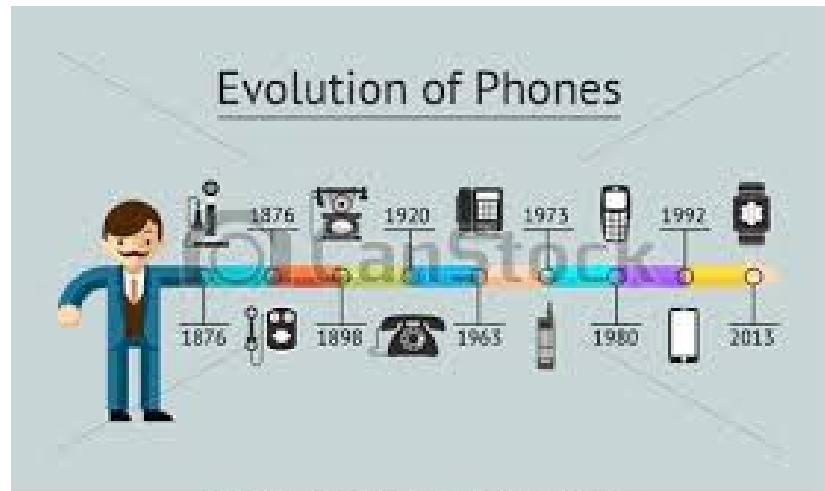
- Industry Revolution 4.0 and its impact in SW engineering
- Concept about SW
- SE activities
- Issues of professional Responsibility
- Key challenges facing SW engineering
- Software Engineering Method.

Introduction – SW as an integral part of every walk of life.

- Industry 4.0 and its impact in Software Engineering.
 - Ubiquitous computing has become a reality.
 - Convergence of Physical world and cyber world in the form of Cyber Physical system.
 - SE has gained more importance and has become core competence for developing and maintaining smart interconnected system.

Industry 1.0 to 4.0

- **1760** – The First Industrial Revolution begins around 1760 in the Textile Industry in Great Britain.
- **1870** – Second Industrial Revolution begins. rapid expansion of New Technologies such as the telephone, railroads, and electrical power.
- 1960s-70– Third revolution was related with Digitalization and PCs.
- Fourth is about Cyber Physical system.



The other side of the coin- Challenges

Software glitches from 2017

Various airports – Check-in chaos, It affected seven airports in seven different countries

American Airlines – No pilots for the holidays, scheduling platform that would have affected around 15,000 flights if not caught.

A lucky Christmas: thousands of winning lottery tickets were printed on Christmas day – totaling 19.6 million dollars if they were all validated.

The most recent one Amazon sold Digital camera worth Rs: 9 lakhs (approximately) to Rs: 6k-7k

What is Software?

*The product that software professionals **build** and then **support** over the long term.*

*Software encompasses: (1) **instructions** (computer programs) that when executed provide desired features, function, and performance; (2) **data structures** that enable the programs to adequately store and manipulate information and (3) **documentation** that describes the operation and use of the programs.*

"Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and work efficiently on real machine"

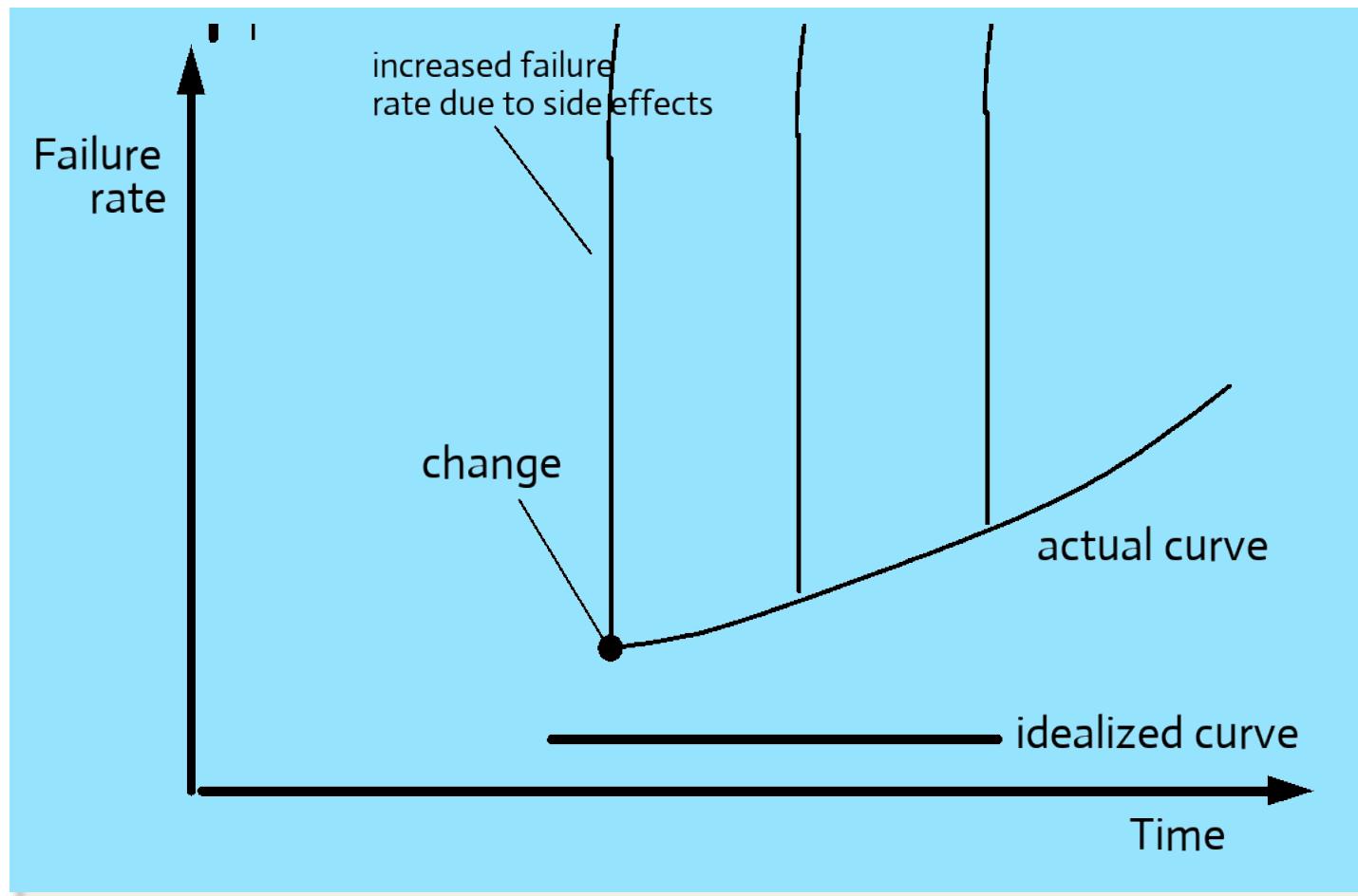
Features of Software?

- Its characteristics that make it different from other things human being build.

Features of such logical system:

- Software is developed or **engineered**, it is not manufactured in the classical sense which has quality problem.
- Software **doesn't "wear out."** but it deteriorates (due to change). Hardware has bathtub curve of failure rate (high failure rate in the beginning, then drop to steady state, then cumulative effects of dust, vibration, abuse occurs).
- Although the industry is moving toward component-based construction (e.g. standard screws and off-the-shelf integrated circuits), most software continues to be **custom-built**. Modern reusable components encapsulate data and processing into software parts to be reused by different programs. E.g. graphical user interface, window, pull-down menus in library etc. ↴

Wear vs. Deterioration



Software Engineering Definition

The seminal definition:

*[Software engineering is] the establishment and use of **sound engineering principles** in order to obtain **economically** software that is **reliable** and **works efficiently** on real machines.*

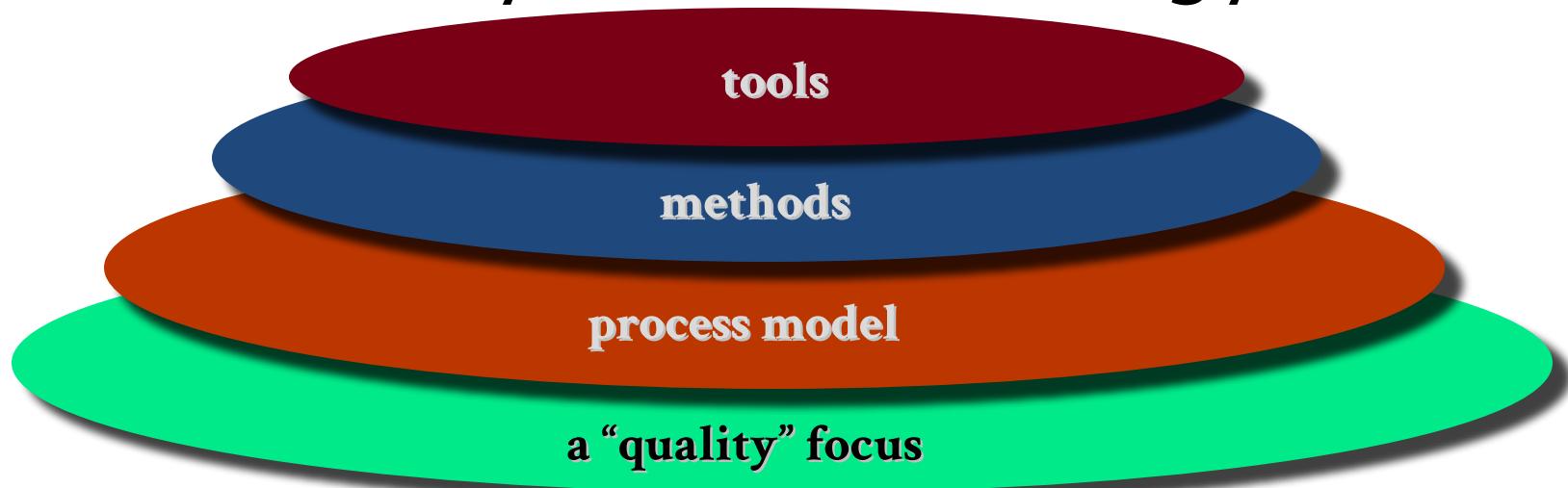
The IEEE definition:

*Software Engineering: (1) The application of a **systematic, disciplined, quantifiable approach** to the **development, operation, and maintenance** of software; that is, the application of engineering to software. (2) The study of approaches as in (1).*

Importance of Software Engineering

- **Large software** – It is easier to build a wall than to a house or building, likewise, as the size of software become large engineering has to step to give it a scientific process.
- **Scalability-** If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.
- **Cost-** As hardware industry has shown its skills and huge manufacturing has lower down the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.
- **Dynamic Nature-** The always growing and adapting nature of software hugely depends upon the environment in which user works. If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.
- **Quality Management-** Better process of software development provides better and quality software product.

A Layered Technology



- Any engineering approach must rest on organizational commitment to **quality** which fosters a continuous process improvement culture.
- **Process** layer as the foundation defines a framework with activities for effective delivery of software engineering technology. Establish the context where products (model, data, report, and forms) are produced, milestone are established, quality is ensured and change is managed.
- **Method** provides technical how-to's for building software. It encompasses many tasks including communication, requirement analysis, design modeling, program construction, testing and support.
- **Tools** provide automated or semi-automated support for the process and methods.

Essential attributes of good software

Product characteristic	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and security	Software dependability includes a range of characteristics including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilisation, etc.
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable and compatible with other systems that they use.

Part 2

Software Process Model:

A software process model is an abstract representation of a process that presents a description of a process from some particular perspective.

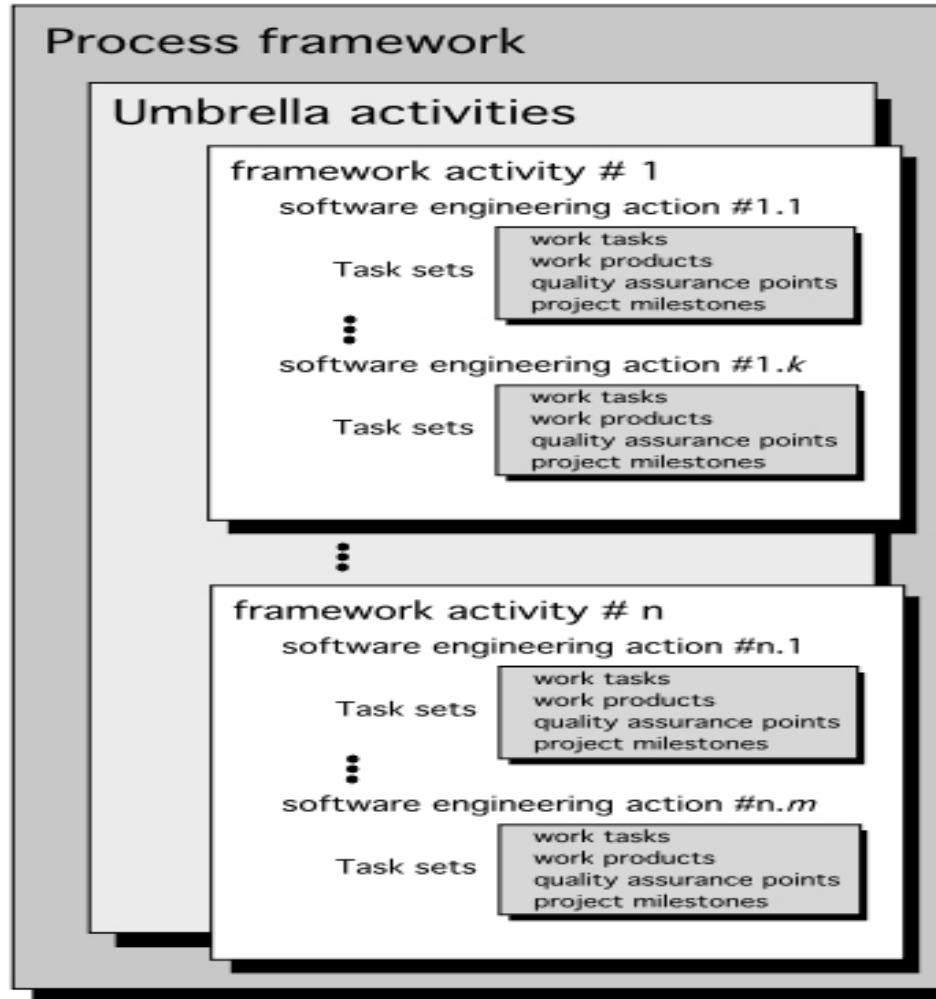
Software Evolution

Lehman has given laws for software evolution. He divided the software into three different categories:

- **S-type (static-type)** – This is a software, which works strictly according to defined specifications and solutions. The solution and the method to achieve it, both are immediately understood before coding. The s-type software is least subjected to changes hence this is the simplest of all. For example, calculator program for mathematical computation.
- **P-type (practical-type)** – This is a software with a collection of procedures. This is defined by exactly what procedures can do. In this software, the specifications can be described but the solution is not obvious instantly. For example, gaming software.
- **E-type (embedded-type)** – This software works closely as the requirement of real-world environment. This software has a high degree of evolution as there are various changes in laws, taxes etc. in the real world situations. For example, Online trading software.

A Generic Process Model

Software process



Adapting a Process Model

The process should be **agile and adaptable** to problems. Process adopted for one project might be significantly different than a process adopted from another project. (to the problem, the project, the team, organizational culture). Among the differences are:

- the **overall flow** of activities, actions, and tasks and the interdependencies among them
- the **degree** to which actions and tasks are defined within each framework activity
- the degree to which work products are identified and required
- the manner which quality assurance activities are applied
- the manner in which project tracking and control activities are applied
- the overall degree of detail and rigor with which the process is described
- the degree to which the customer and other stakeholders are involved with the project
- the level of autonomy given to the software team
- the degree to which team organization and roles are prescribed

Prescriptive and Agile Process Models

- The **prescriptive process** models stress detailed definition, identification, and application of process activates and tasks. Intent is to **improve system quality, make projects more manageable, make delivery dates and costs more predictable**, and guide teams of software engineers as they perform the work required to build a system.
- Unfortunately, there have been times when these objectives were not achieved. If prescriptive models are applied dogmatically and without adaptation, they can increase the level of bureaucracy.
- Agile process models** emphasize project “agility” and follow a set of principles that lead to a more informal approach to software process. It **emphasizes maneuverability and adaptability**. It is particularly useful when Web applications are engineered.

Five Activities of a Generic Process framework

- **Communication:** communicate with customer to understand objectives and gather requirements
- **Planning:** creates a “map” defines the work by describing the tasks, risks and resources, work products and work schedule.
- **Modeling:** Create a “sketch”, what it looks like architecturally, how the constituent parts fit together and other characteristics.
- **Construction:** code generation and the testing.
- **Deployment:** Delivered to the customer who evaluates the products and provides feedback based on the evaluation.
- These five framework activities can be used to **all software development regardless of the application domain**, size of the project, complexity of the efforts etc, though the details will be different in each case.
- For many software projects, these framework activities are applied **iteratively** as a project progresses. Each iteration produces a software increment that provides a subset of overall software features and functionality.

Umbrella Activities

Complement the five process framework activities and help team **manage and control** progress, quality, change, and risk.

- **Software project tracking and control:** assess progress against the plan and take actions to maintain the schedule.
- **Risk management:** assesses risks that may affect the outcome and quality.
- **Software quality assurance:** defines and conduct activities to ensure quality.
- **Technical reviews:** assesses work products to uncover and remove errors before going to the next activity.
- **Measurement:** define and collects process, project, and product measures to ensure stakeholder's needs are met.
- **Software configuration management:** manage the effects of change throughout the software process.
- **Reusability management:** defines criteria for work product reuse and establishes mechanism to achieve reusable components.
- **Work product preparation and production:** create work products such as models, documents, logs, forms and lists.

Part 3

The Essence of Practice

- How does the practice of software engineering fit in the process activities mentioned above? Namely, communication, planning, modeling, construction and deployment.
- George Polya outlines the essence of problem solving, suggests:
 1. *Understand the problem* (communication and analysis).
 2. *Plan a solution* (modeling and software design).
 3. *Carry out the plan* (code generation).
 4. *Examine the result for accuracy* (testing and quality assurance).

Understand the Problem

- *Who has a stake in the solution to the problem?* That is, who are the stakeholders?
- *What are the unknowns?* What data, functions, and features are required to properly solve the problem?
- *Can the problem be compartmentalized?* Is it possible to represent smaller problems that may be easier to understand?
- *Can the problem be represented graphically?* Can an analysis model be created?

Plan a Solution

- *Have you seen similar problems before?* Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- *Has a similar problem been solved?* If so, are elements of the solution reusable?
- *Can subproblems be defined?* If so, are solutions readily apparent for the subproblems?
- *Can you represent a solution in a manner that leads to effective implementation?* Can a design model be created?

Carry Out the Plan

- *Does the solutions conform to the plan?* Is source code traceable to the design model?
- *Is each component part of the solution provably correct?* Has the design and code been reviewed, or better, have correctness proofs been applied to algorithm?

Examine the Result

- *Is it possible to test each component part of the solution?* Has a reasonable testing strategy been implemented?
- *Does the solution produce results that conform to the data, functions, and features that are required?* Has the software been validated against all stakeholder requirements?

Help you establish mind-set for solid software engineering practice (David Hooker 96).

- 1: *The Reason It All Exists: provide values to users*
- 2: *KISS (Keep It Simple, Stupid! As simple as possible)*
- 3: *Maintain the Vision (otherwise, incompatible design)*
- 4: *What You Produce, Others Will Consume* (code with concern for those that must maintain and extend the system)
- 5: *Be Open to the Future* (never design yourself into a corner as specification and hardware changes)
- 6: *Plan Ahead for Reuse*
- 7: *Think! Place clear complete thought before action produces better results.*

Hooker's General Principles for Software Engineering Practice: important underlying law

Software Myths

Erroneous beliefs about software and the process that is used to build it.

- Affect managers, customers (and other non-technical stakeholders) and practitioners
- Are believable because they often have elements of truth,
but ...
- Invariably lead to bad decisions,
therefore ...
- Insist on reality as you navigate your way through software engineering

Software Myths Examples

- **Myth 1:** Once we write the program and get it to work, our job is done.
- Reality: the sooner you begin writing code, the longer it will take you to get done. 60% to 80% of all efforts are spent after software is delivered to the customer for the first time.
- **Myth 2:** Until I get the program running, I have no way of assessing its quality.
- Reality: technical review are a quality filter that can be used to find certain classes of software defects from the inception of a project.
- **Myth 3:** software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.
- Reality: it is not about creating documents. It is about creating a quality product. Better quality leads to a reduced rework. Reduced work results in faster delivery times.
- Many people recognize the fallacy of the myths. Regrettably, **habitual attitudes and methods** foster poor management and technical practices, even when reality dictates a better approach.

How It all Starts

- *SafeHome:*
 - Every software project is precipitated by some business need—
 - the need to correct a defect in an existing application;
 - the need to adapt a ‘legacy system’ to a changing business environment;
 - the need to extend the functions and features of an existing application, or
 - the need to create a new product, service, or system.

Part 4

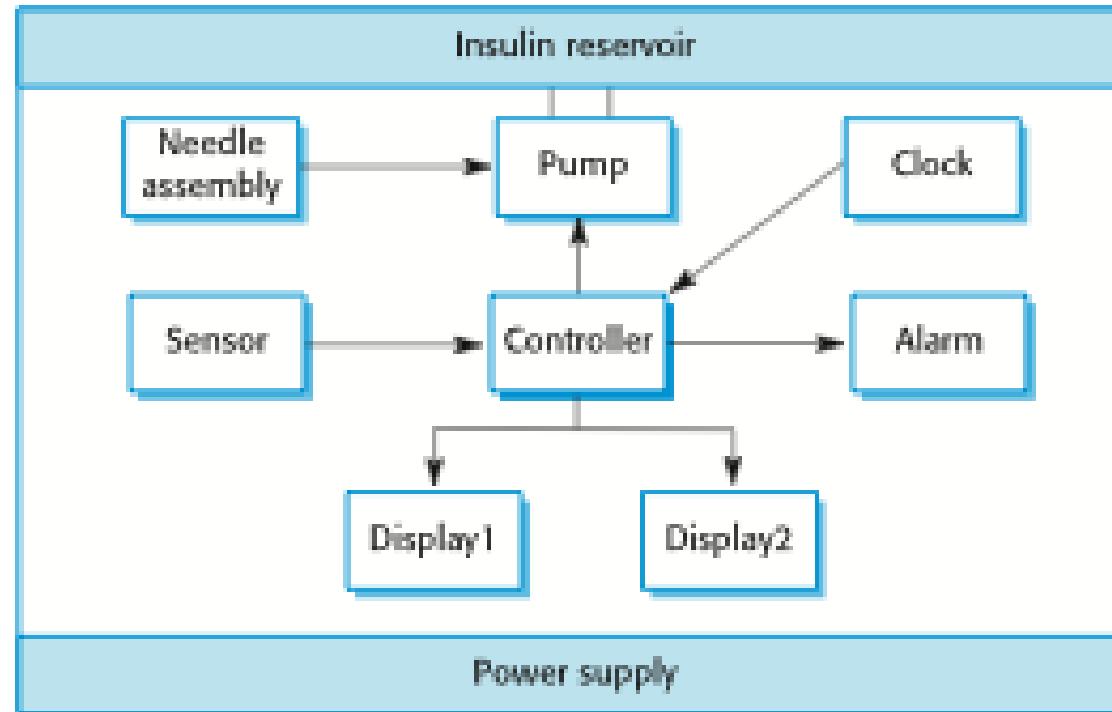
Case studies on Monday for Section for Section B

- A personal insulin pump
 - An embedded system in an insulin pump used by diabetics to maintain blood glucose control.
- A mental health case patient management system
 - A system used to maintain records of people receiving care for mental health problems.
- A wilderness weather station
 - A data collection system that collects data about weather conditions in remote areas.

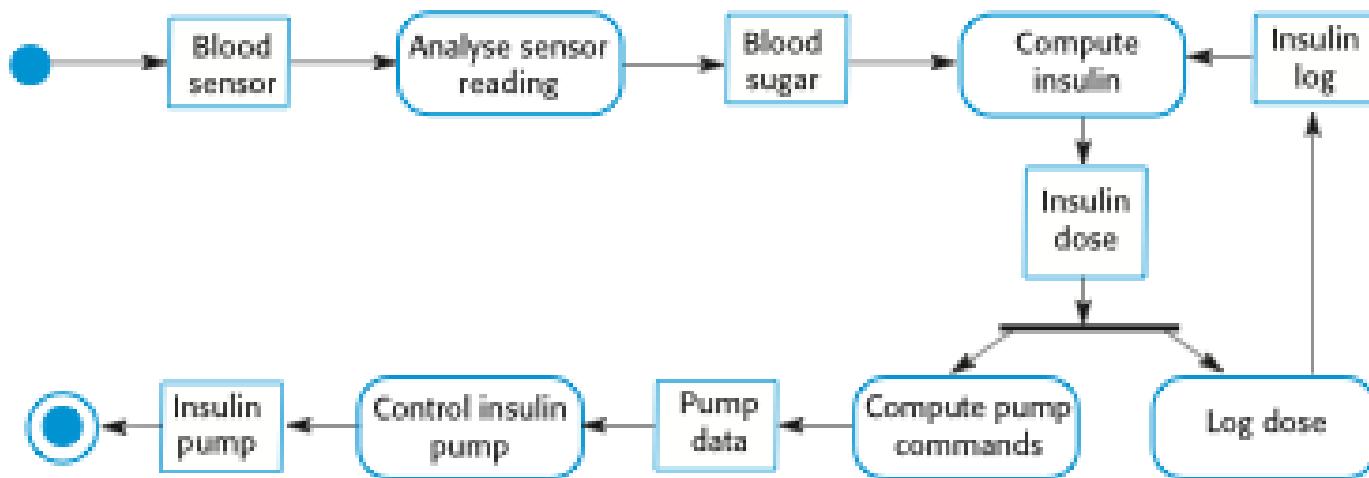
Insulin pump control system

- Collects data from a blood sugar sensor and calculates the amount of insulin required to be injected.
- Calculation based on the rate of change of blood sugar levels.
- Sends signals to a micro-pump to deliver the correct dose of insulin.
- Safety-critical system as low blood sugars can lead to brain malfunctioning, coma and death; high-blood sugar levels have long-term consequences such as eye and kidney damage.

Insulin pump hardware architecture



Activity model of the insulin pump



Essential high-level requirements

- The system shall be available to deliver insulin when required.
- The system shall perform reliably and deliver the correct amount of insulin to counteract the current level of blood sugar.
- The system must therefore be designed and implemented to ensure that the system always meets these requirements.

A patient information system for mental health care

- A patient information system to support mental health care is a medical information system that maintains information about patients suffering from mental health problems and the treatments that they have received.
- Most mental health patients do not require dedicated hospital treatment but need to attend specialist clinics regularly where they can meet a doctor who has detailed knowledge of their problems.
- To make it easier for patients to attend, these clinics are not just run in hospitals. They may also be held in local medical practices or community centres.

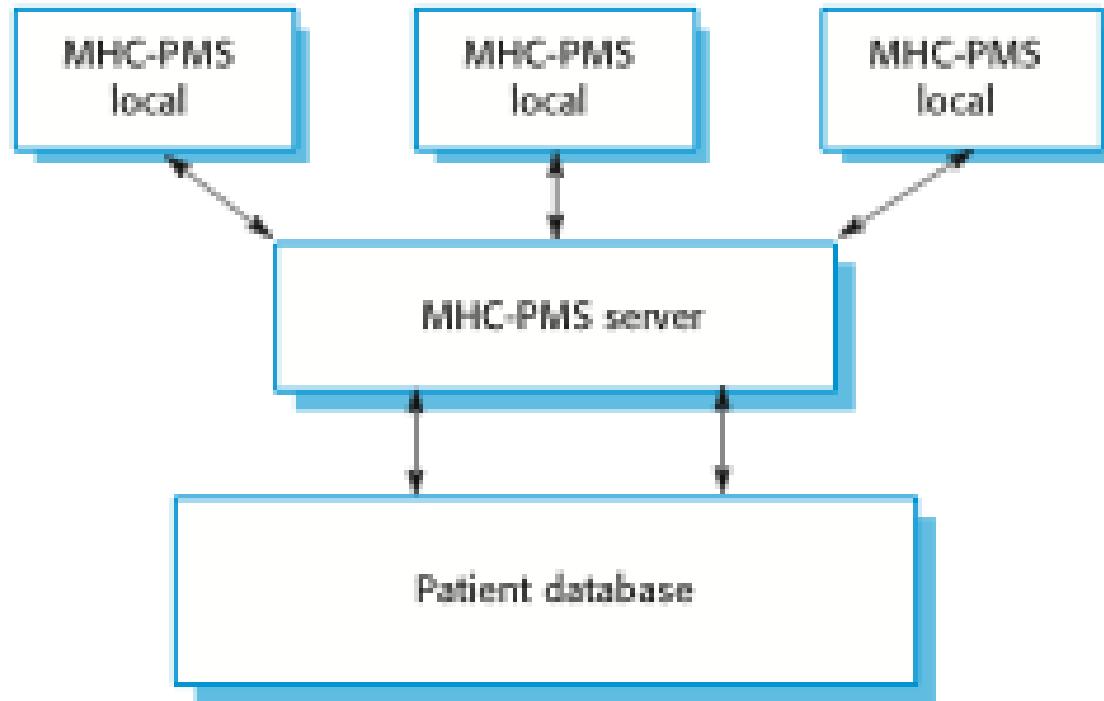
MHC-PMS

- The MHC-PMS (Mental Health Care-Patient Management System) is an information system that is intended for use in clinics.
- It makes use of a centralized database of patient information but has also been designed to run on a PC, so that it may be accessed and used from sites that do not have secure network connectivity.
- When the local systems have secure network access, they use patient information in the database but they can download and use local copies of patient records when they are disconnected.

MHC-PMS goals

- To generate management information that allows health service managers to assess performance against local and government targets.
- To provide medical staff with timely information to support the treatment of patients.

The organization of the MHC-PMS



MHC-PMS key features

- Individual care management
 - Clinicians can create records for patients, edit the information in the system, view patient history, etc. The system supports data summaries so that doctors can quickly learn about the key problems and treatments that have been prescribed.
- Patient monitoring
 - The system monitors the records of patients that are involved in treatment and issues warnings if possible problems are detected.
- Administrative reporting
 - The system generates monthly management reports showing the number of patients treated at each clinic, the number of patients who have entered and left the care system, number of patients sectioned, the drugs prescribed and their costs, etc.

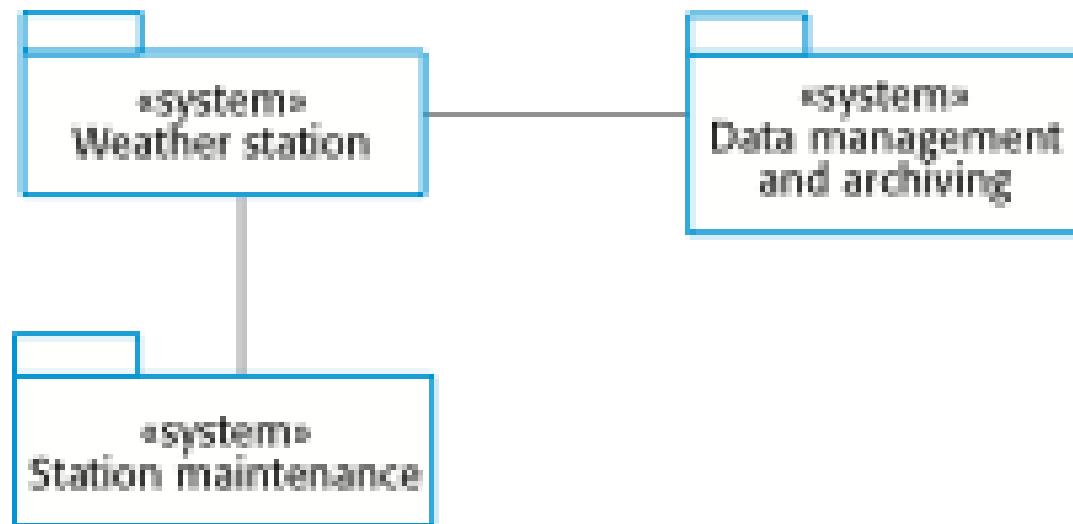
MHC-PMS concerns

- Privacy
 - It is essential that patient information is confidential and is never disclosed to anyone apart from authorised medical staff and the patient themselves.
- Safety
 - Some mental illnesses cause patients to become suicidal or a danger to other people. Wherever possible, the system should warn medical staff about potentially suicidal or dangerous patients.
 - The system must be available when needed otherwise safety may be compromised and it may be impossible to prescribe the correct medication to patients.

Wilderness weather station

- The government of a country with large areas of wilderness decides to deploy several hundred weather stations in remote areas.
- Weather stations collect data from a set of instruments that measure temperature and pressure, sunshine, rainfall, wind speed and wind direction.
 - The weather station includes a number of instruments that measure weather parameters such as the wind speed and direction, the ground and air temperatures, the barometric pressure and the rainfall over a 24-hour period. Each of these instruments is controlled by a software system that takes parameter readings periodically and manages the data collected from the instruments.
 -

The weather station's environment



Weather information system

- The weather station system
 - This is responsible for collecting weather data, carrying out some initial data processing and transmitting it to the data management system.
- The data management and archiving system
 - This system collects the data from all of the wilderness weather stations, carries out data processing and analysis and archives the data.
- The station maintenance system
 - This system can communicate by satellite with all wilderness weather stations to monitor the health of these systems and provide reports of problems.

Additional software functionality

- Monitor the instruments, power and communication hardware and report faults to the management system.
- Manage the system power, ensuring that batteries are charged whenever the environmental conditions permit but also that generators are shut down in potentially damaging weather conditions, such as high wind.
- Support dynamic reconfiguration where parts of the software are replaced with new versions and where backup instruments are switched into the system in the event of system failure.

Unit 2

Definition of Software Process

- A **framework** for the activities, actions, and tasks that are required to build high-quality software.
- SP defines the approach that is taken as software is engineered.
- Is not equal to software engineering, which also encompasses **technologies** that populate the process— technical methods and automated tools.

Software Process

What is well engineered software?

If the software system does what the user wants, and can be made to continue to do what the user wants, it is well engineered.

- Any well engineered software system should have the following attributes:
- Be easy to maintain
- Be reliable
- Be efficient
- Provides an appropriate user interface

The development of software must make trade-offs between these attributes.

Distribution of software effort

The typical life-span for a typical software product is 1 to 3 years in development and 5 to 15 years in use. The distribution of effort between development and maintenance has been variously reported depending on the type of software as 40/60, 30/70 and 10/90.

Maintenance:

- **Corrective:** Even with the best quality of software, it is likely that customer will uncover defect in software. Corrective maintenance changes the software to correct the defects.
- **Adaptive:** Over time, the original environment (CPU, OS, business rules, external product character etc.) for which the software was developed may change. Adaptive maintenance results in modification to the software to accommodate the change to its environment.
- **Perfective:** As the software is used, the customer / user will recognize additional function that will provide benefit. Perfective maintenance extends the software beyond its original functional requirements.

Software Process

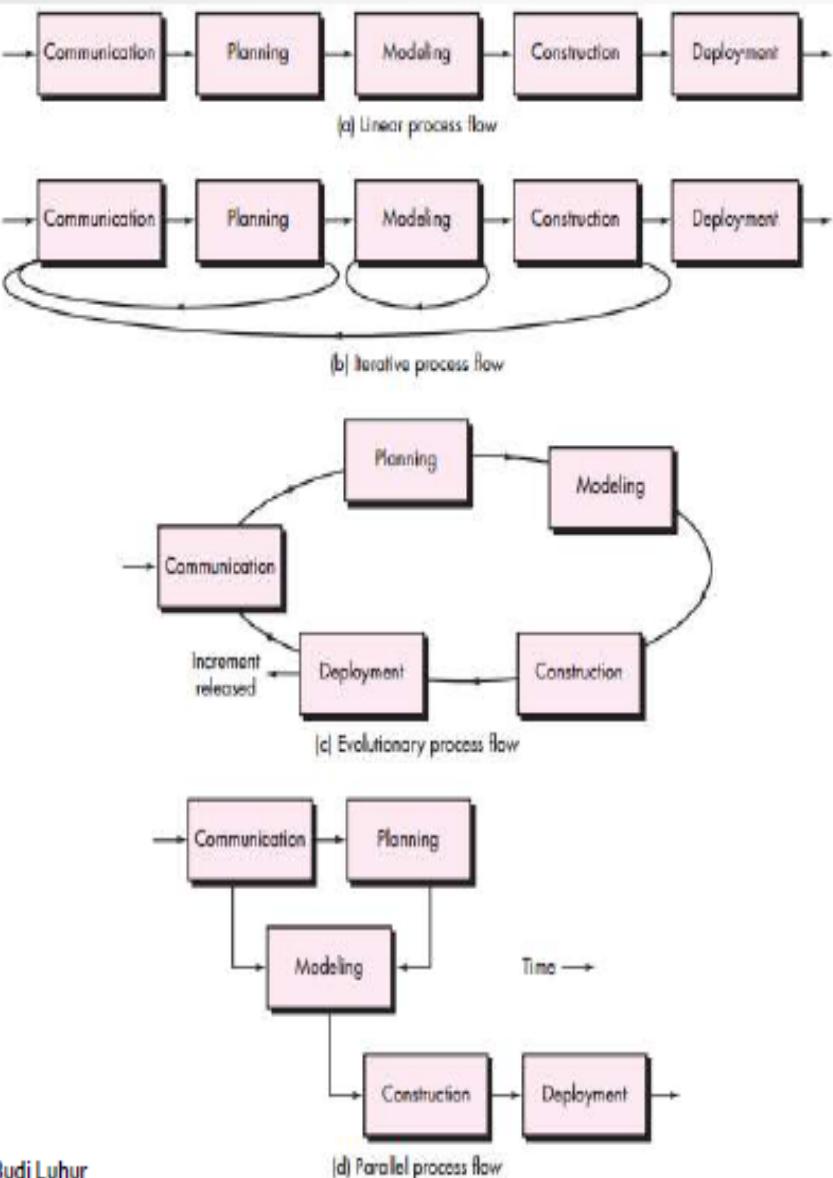
- A structured **set of activities** required to develop a software system.
- Many different software processes but all involve:
 - **Specification** – defining what the system should do;
 - **Design and implementation** – defining the organization of the system and implementing the system;
 - **Validation** – checking that it does what the customer wants;
 - **Evolution** – changing the system in response to changing customer needs.
- A **software process model** is an abstract representation of a process. It presents a description of a process from some particular perspective

Types of Process Models

- Universal
 - Describe the basic process steps and provide general guidance on their role and order (e.g., Waterfall and Spiral Model). Permit global understanding and provide a framework for policies.
- Worldly
 - Guide the sequence of tasks, define task prerequisites and results, specify who does what when, models anticipate results, measure and key checkpoints. Guide daily work.
- Atomic
 - Precise data definitions, algorithmic specifications, information flows and user procedures. Atomic process definitions are often embodied in process standards and conventions. Provide atomic detail for training and task mechanization.

Software process

Process Flow



Process framework

Umbrella activities

framework activity # 1

software engineering action #1.1

Task sets

work tasks
work products
quality assurance points
project milestones

software engineering action #1.k

Task sets

work tasks
work products
quality assurance points
project milestones

framework activity # n

software engineering action #n.1

Task sets

work tasks
work products
quality assurance points
project milestones

software engineering action #n.m

Task sets

work tasks
work products
quality assurance points
project milestones

Task Set

A **task set** defines the actual work to be done to accomplish the objectives of a software engineering action.

For a **small**, relatively simple project, the task set for requirements gathering might look like this:

1. Make a list of stakeholders for the project.
2. Invite all stakeholders to an informal meeting.
3. Ask each stakeholder to make a list of features and functions required.
4. Discuss requirements and build a final list.
5. Prioritize requirements.
6. Note areas of uncertainty

Task Set

For a larger, more complex software project:

1. Make a list of stakeholders for the project.
2. Interview each stakeholder separately to determine overall wants and needs.
3. Build a preliminary list of functions and features based on stakeholder input.
4. Schedule a series of facilitated application specification meetings.
5. Conduct meetings.
6. Produce informal user scenarios as part of each meeting.
7. Refine user scenarios based on stakeholder feedback.
8. Build a revised list of stakeholder requirements.
9. Use quality function deployment techniques to prioritize requirements.
10. Package requirements so that they can be delivered incrementally.
11. Note constraints and restrictions that will be placed on the system.
12. Discuss methods for validating the system.

Software Development Life cycle (SDLC)

A life cycle model prescribes the different activities that need to be carried out to develop a software product and sequencing of these activities.

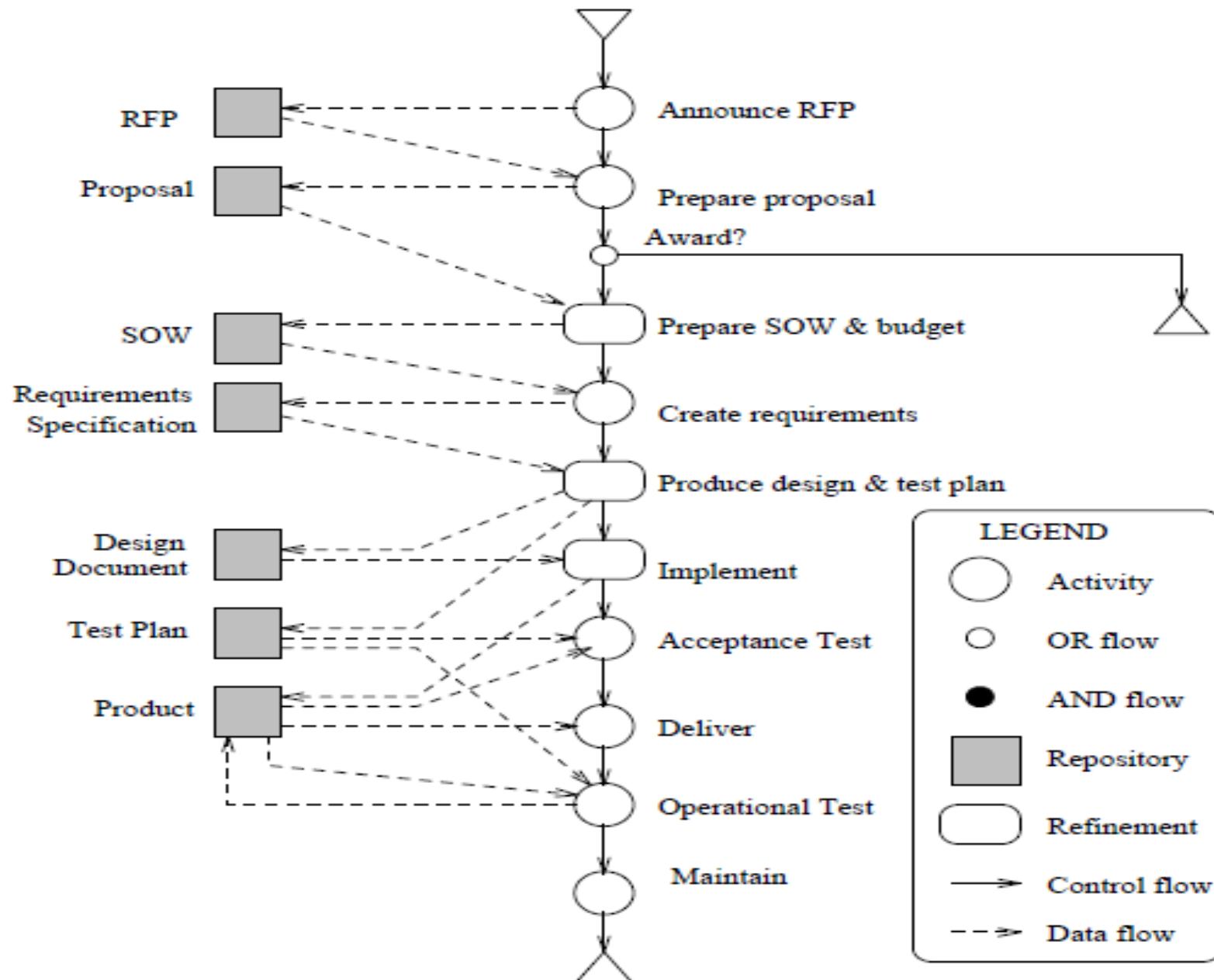
Also referred to as Systems Development Life cycle.

Every software product starts with a request for the product by the customer.- **Production conception.**

The software life cycle can be considered as the business process for software development and therefore is often referred to as a Software process. (SLCM).

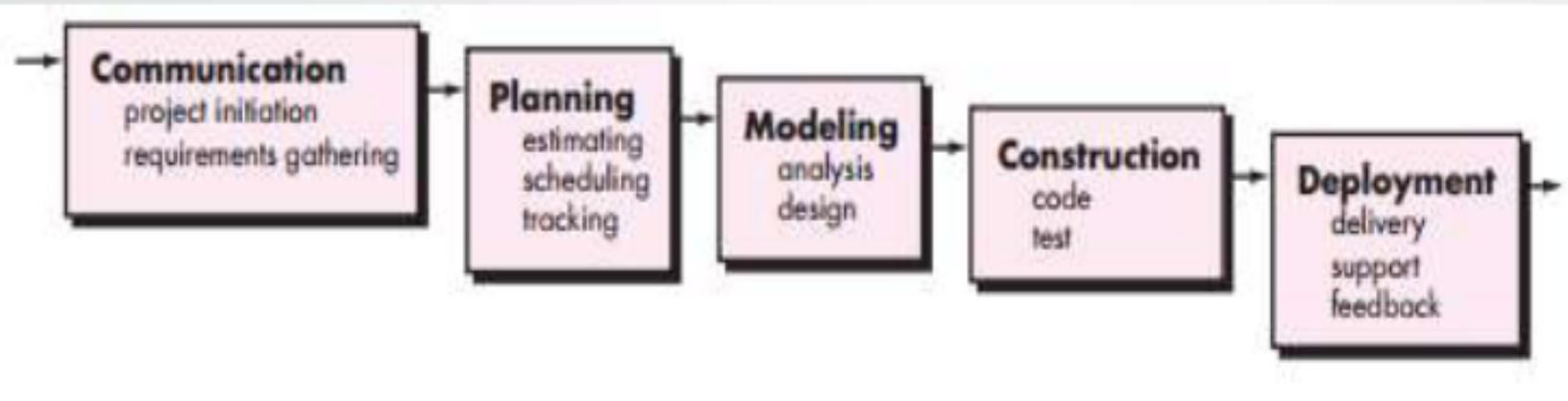
Process models – More detailed and precise life cycle activities.

- Why use a life cycle model?
 - Encourages development of software in a systematic and disciplined manner
 - S.D organisations have realized that adherence to a suitable well-defined life cycle model helps to produce good quality products and that too without time and cost overruns.
-
- Why document a life cycle model?
 - A documented life cycle model, besides preventing misinterpretations that occur when the life cycle model is not adequately documented, also helps to identify inconsistencies, redundancies, and omissions in the development process

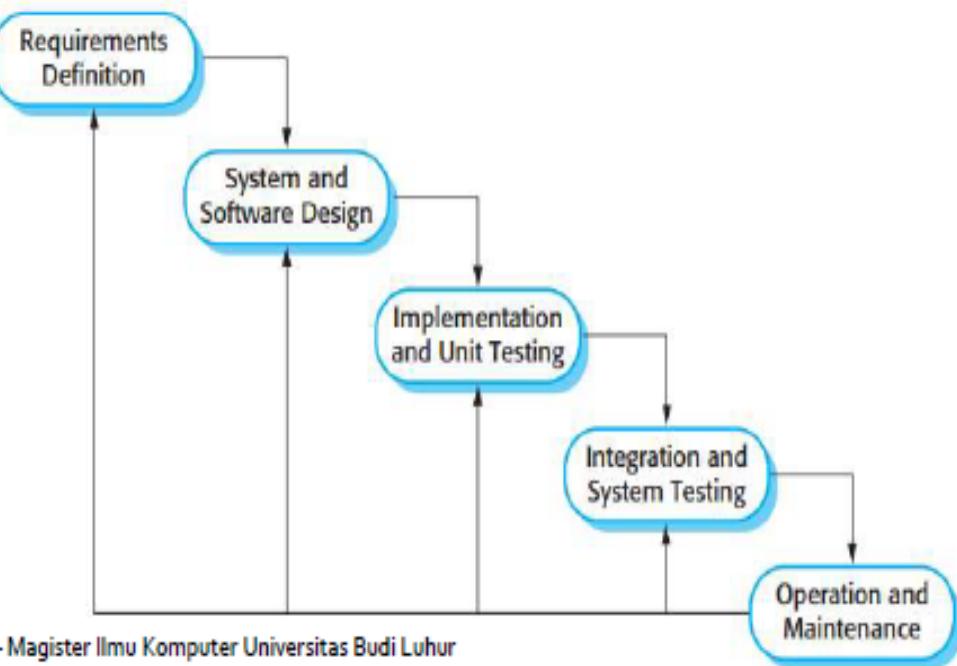


- Different stages in a life cycle model: After Product conception. The stages are: (**Life cycle phase**)
 - Feasibility study stage
 - Requirements analysis and specification.
 - Design
 - Coding
 - Testing and
 - Maintenance.
- A SDLC is a series of identifiable stages that a software product undergoes during its lifetime.
- A SDLC is a descriptive and diagrammatic representation of the software life cycle.
- A life cycle model maps the different activities performed on a software product from its beginning to retirement into a set of life cycle phases.

The Waterfall Model



[Pressman, 2010]

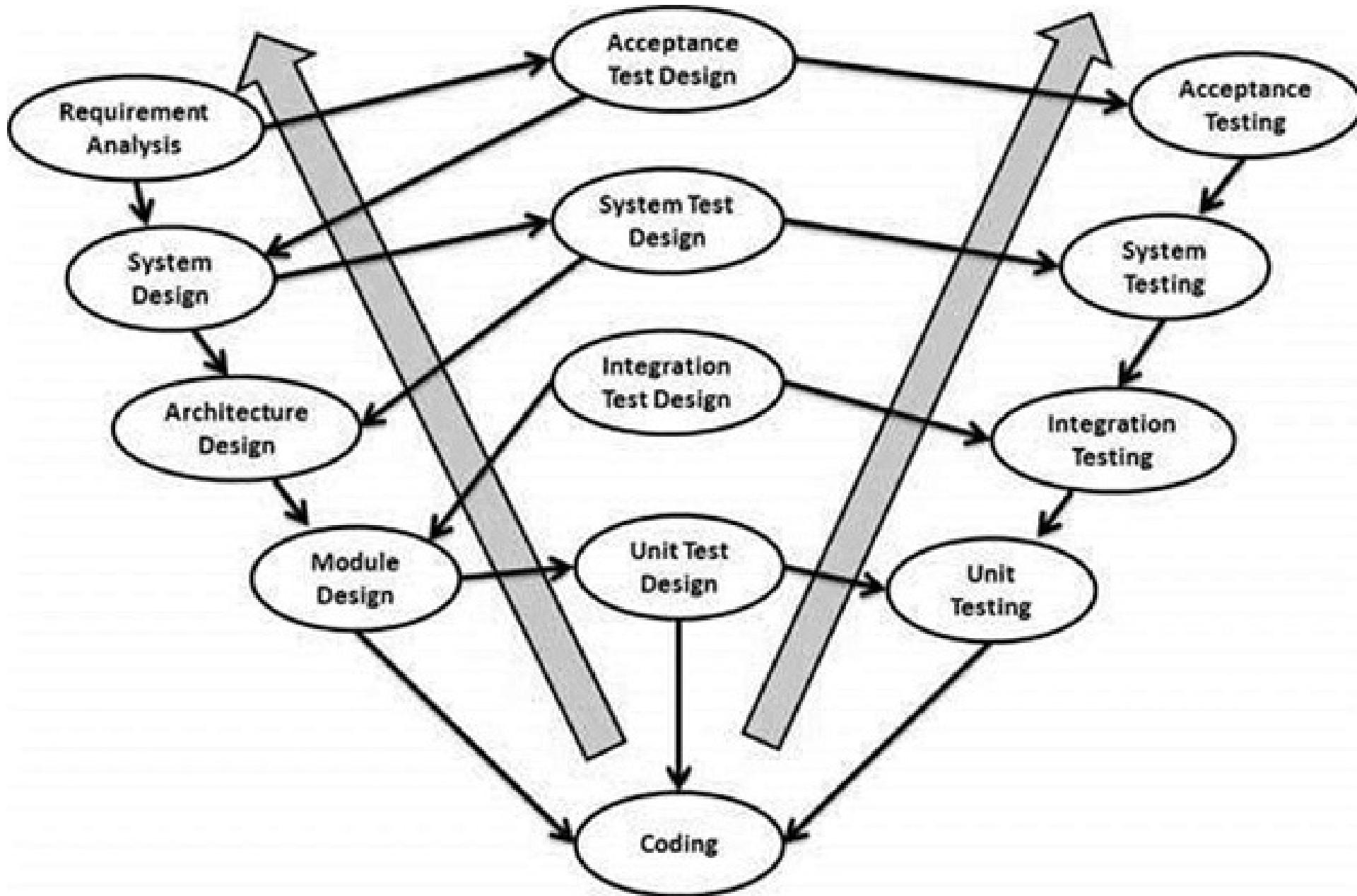


[Sommerville, 2011]

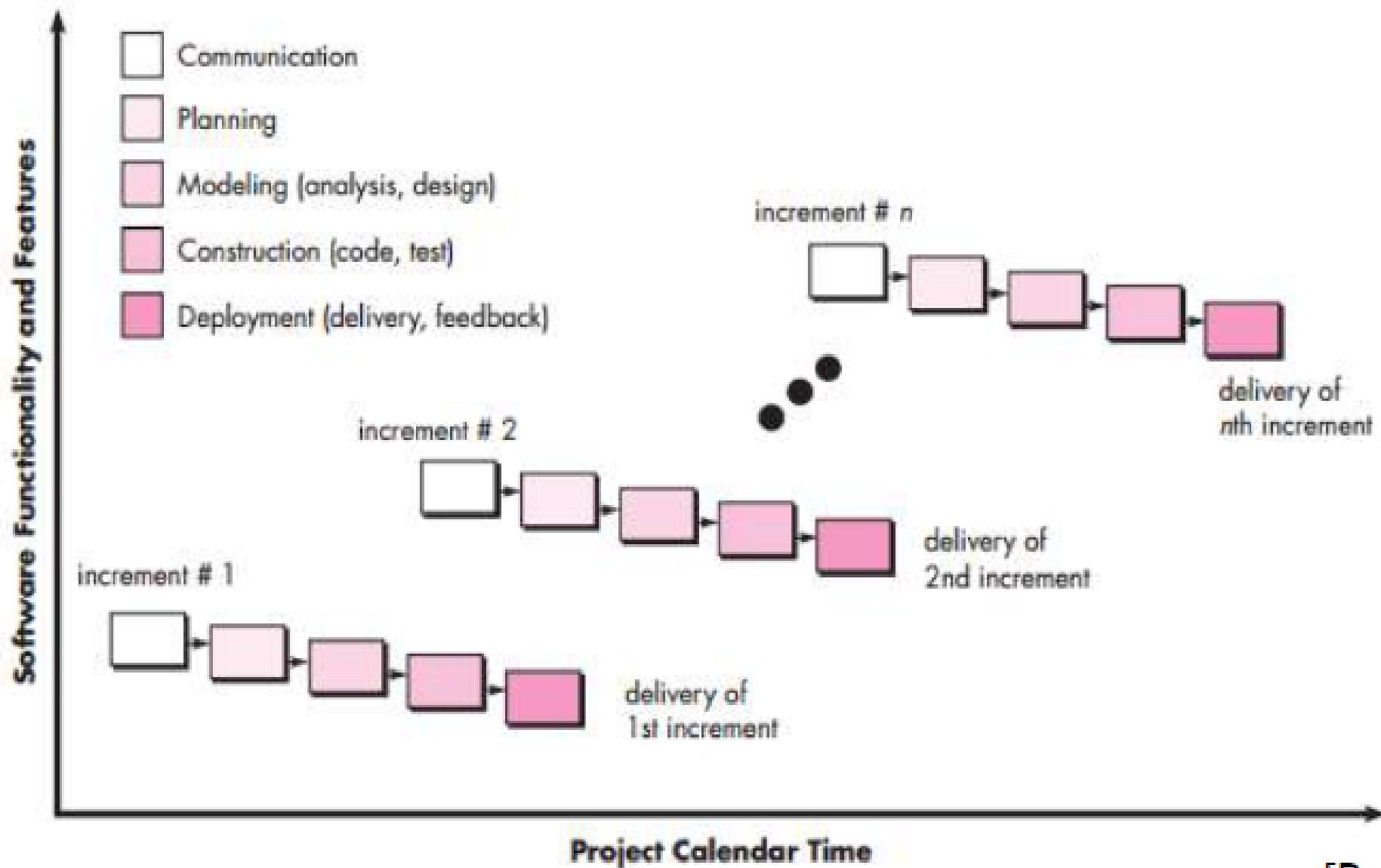
Waterfall Model Problems

- Inflexible partitioning of the project into distinct stages makes it **difficult to respond to changing customer requirements.**
 - Therefore, this model is only appropriate when the requirements are **well-understood** and changes will be fairly limited during the design process.
 - Few business systems have **stable requirements.**
- The waterfall model is mostly used for large systems engineering projects where a **system is developed at several sites.**
 - In those circumstances, the plan-driven nature of the waterfall model helps coordinate the work.

The following illustration depicts the different phases in a V-Model of the SDLC.



The Incremental Model



The Incremental Model Benefits

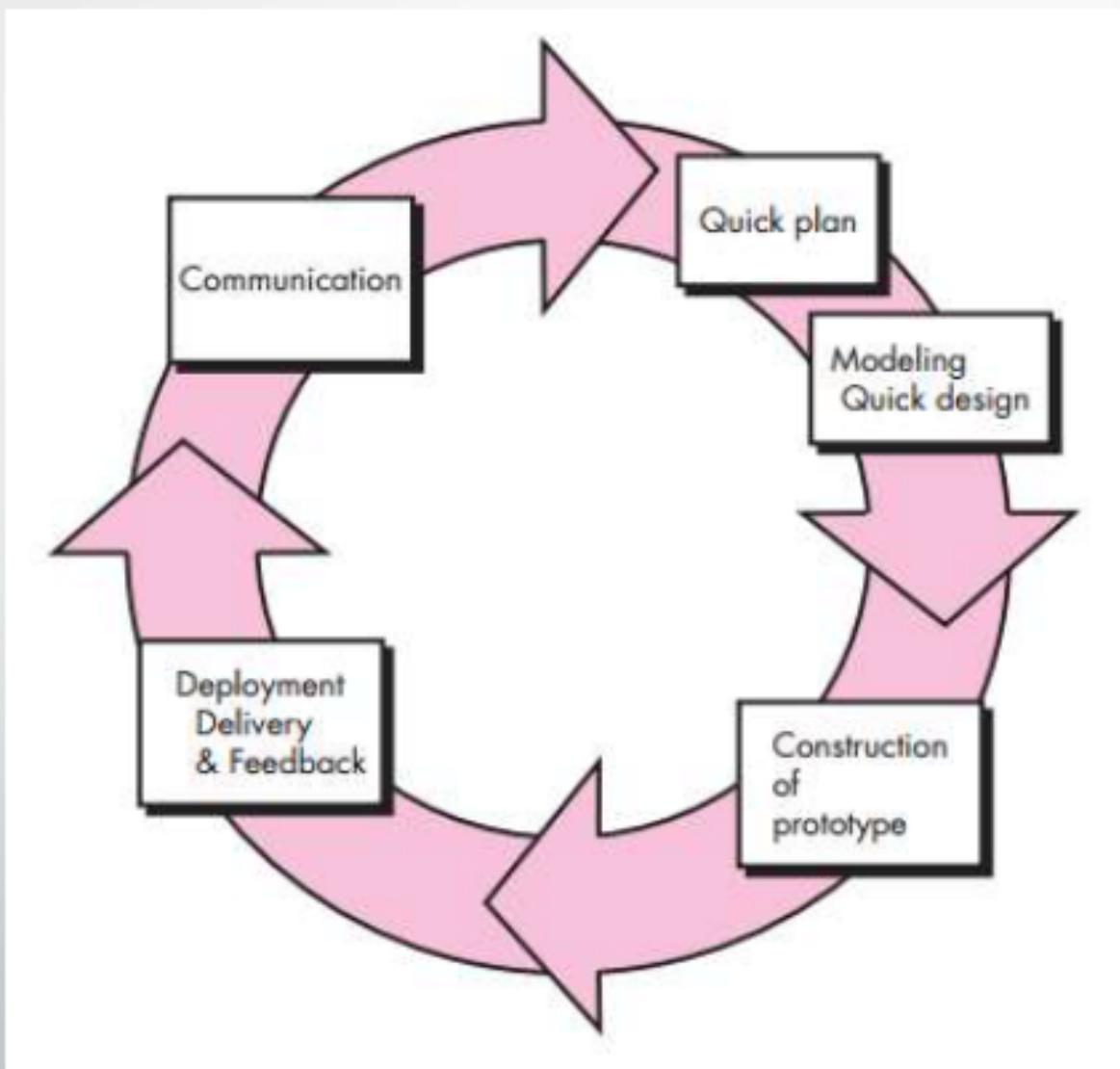
- The cost of **accommodating changing customer requirements** is **reduced**. The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.
- It is **easier to get customer feedback** on the development work that has been done. Customers can comment on demonstrations of the software and see how much has been implemented.
- **More rapid delivery and deployment** of useful software to the customer is possible. Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

The Incremental Model Problems

- The process is not visible.
 - Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.
- System structure tends to degrade as new increments are added.
 - Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.
- The problems of incremental development become particularly acute for large, complex, long-lifetime systems, where different teams develop different parts of the system.

End for Sec A

Evolutionary Model: Prototyping



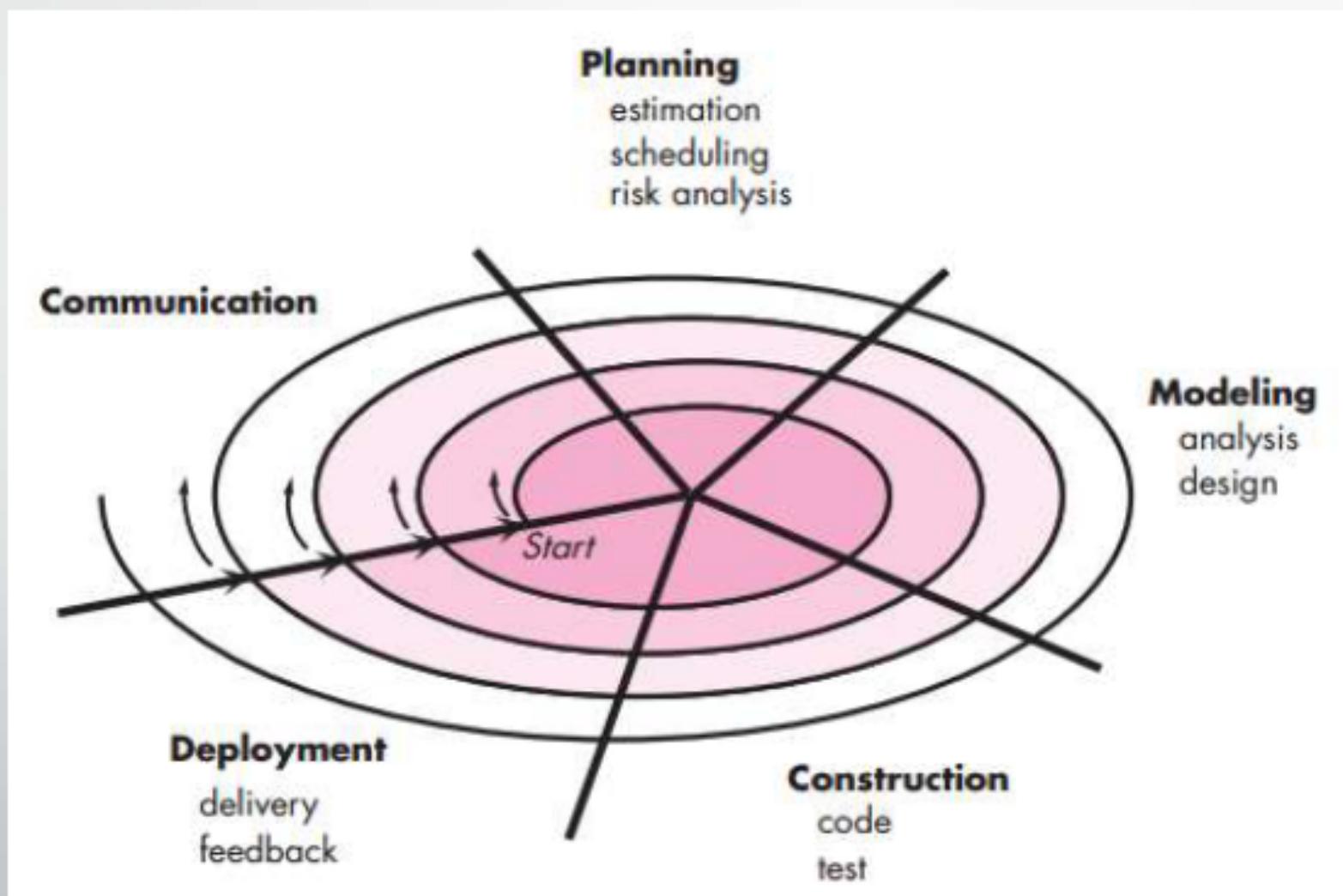
The Prototyping Benefits

- Improved system usability.
- A closer match to users' real needs.
- Improved design quality.
- Improved maintainability.
- Reduced development effort.

The Prototyping Problems

- Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven't considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high levels of quality can be maintained, stakeholders cry foul and demand that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents.
- As a software engineer, you often make implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, you may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system

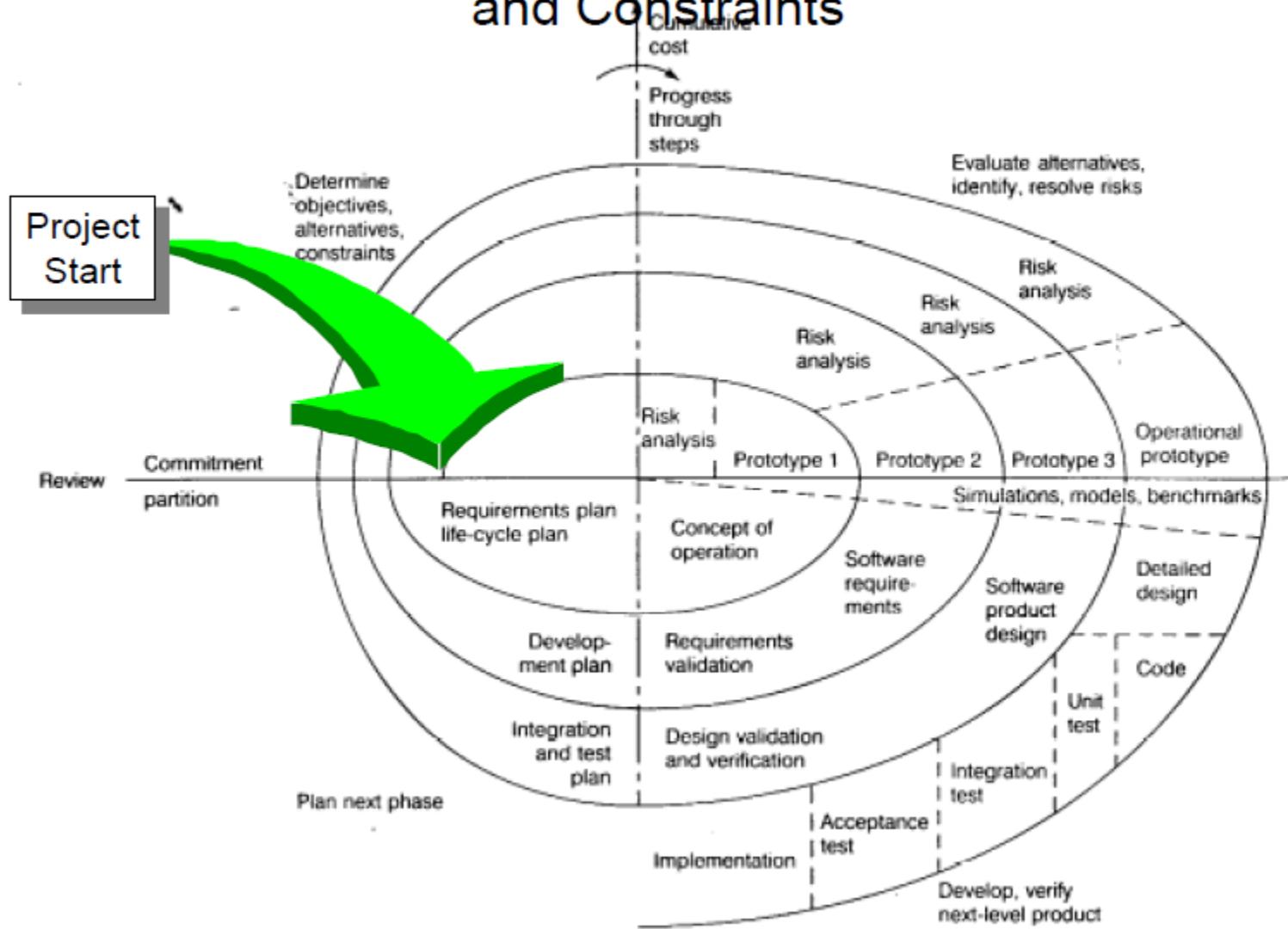
Evolutionary Model: The Spiral



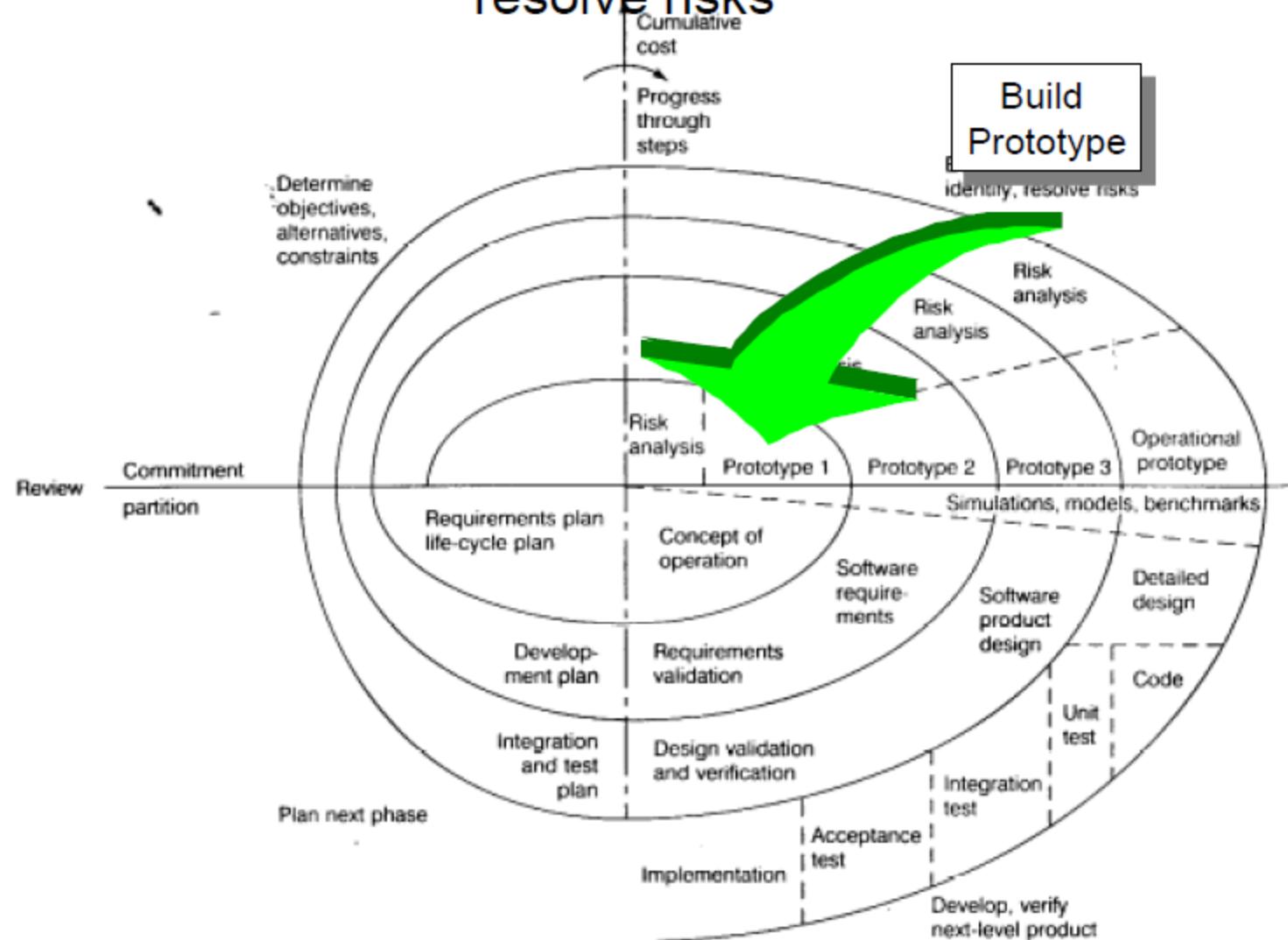
4 Quadrants of spiral model

- 1st Quadrant:
 - The objectives are investigated, elaborated and analysed.
 - Risks are also identified.
 - Alternative Solutions are proposed
- 2nd Quadrant:
 - Alternative solutions are evaluated to select best.
- 3rd Quadrant:
 - Developing and verifying the next level of the product
- 4th Quadrant:
 - Reviewing the results of the stages traversed so far with the customer.
 - Planning the next iteration around the spiral.
- :

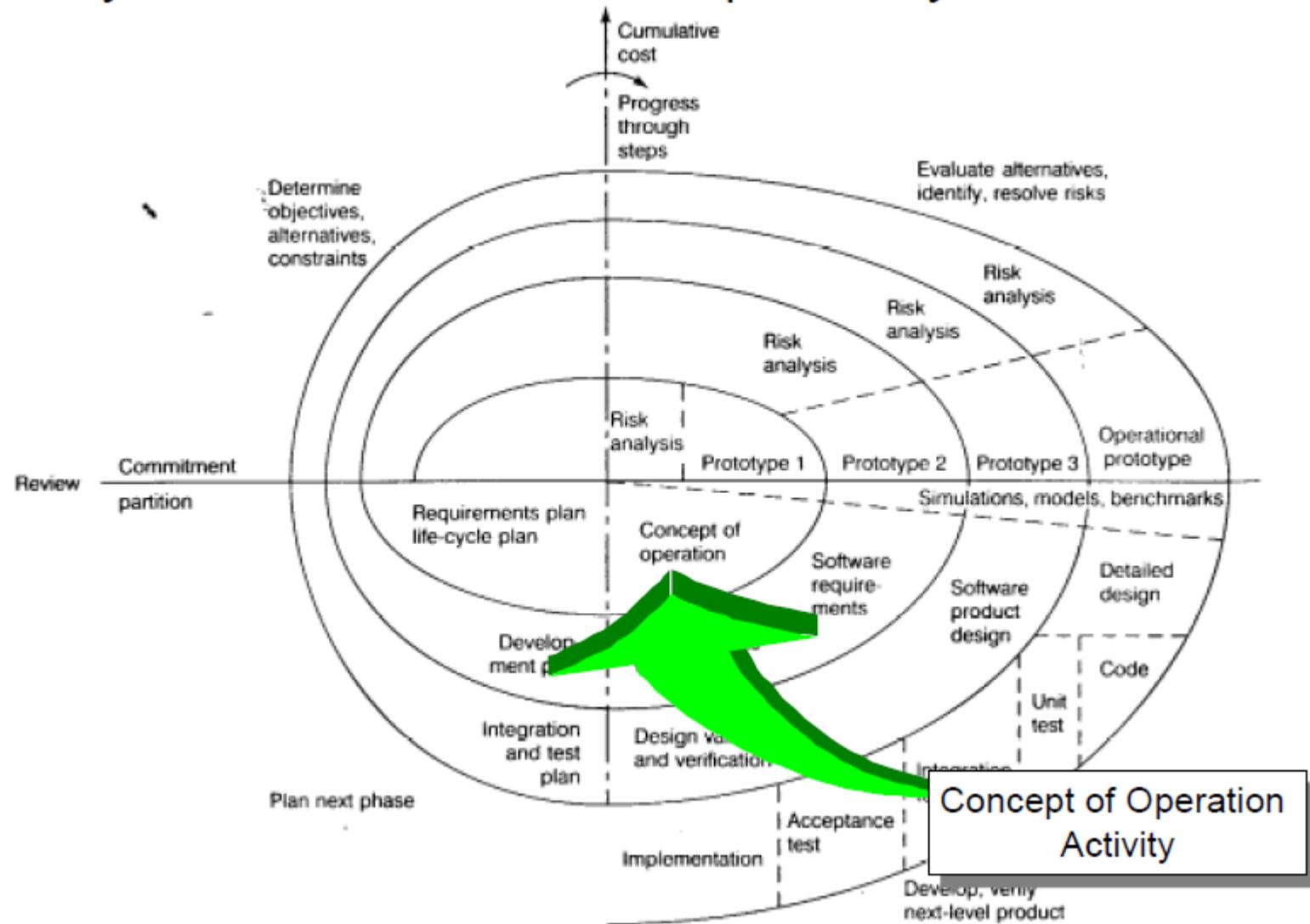
Cycle 1, Quadrant IV: Determine Objectives, Alternatives and Constraints



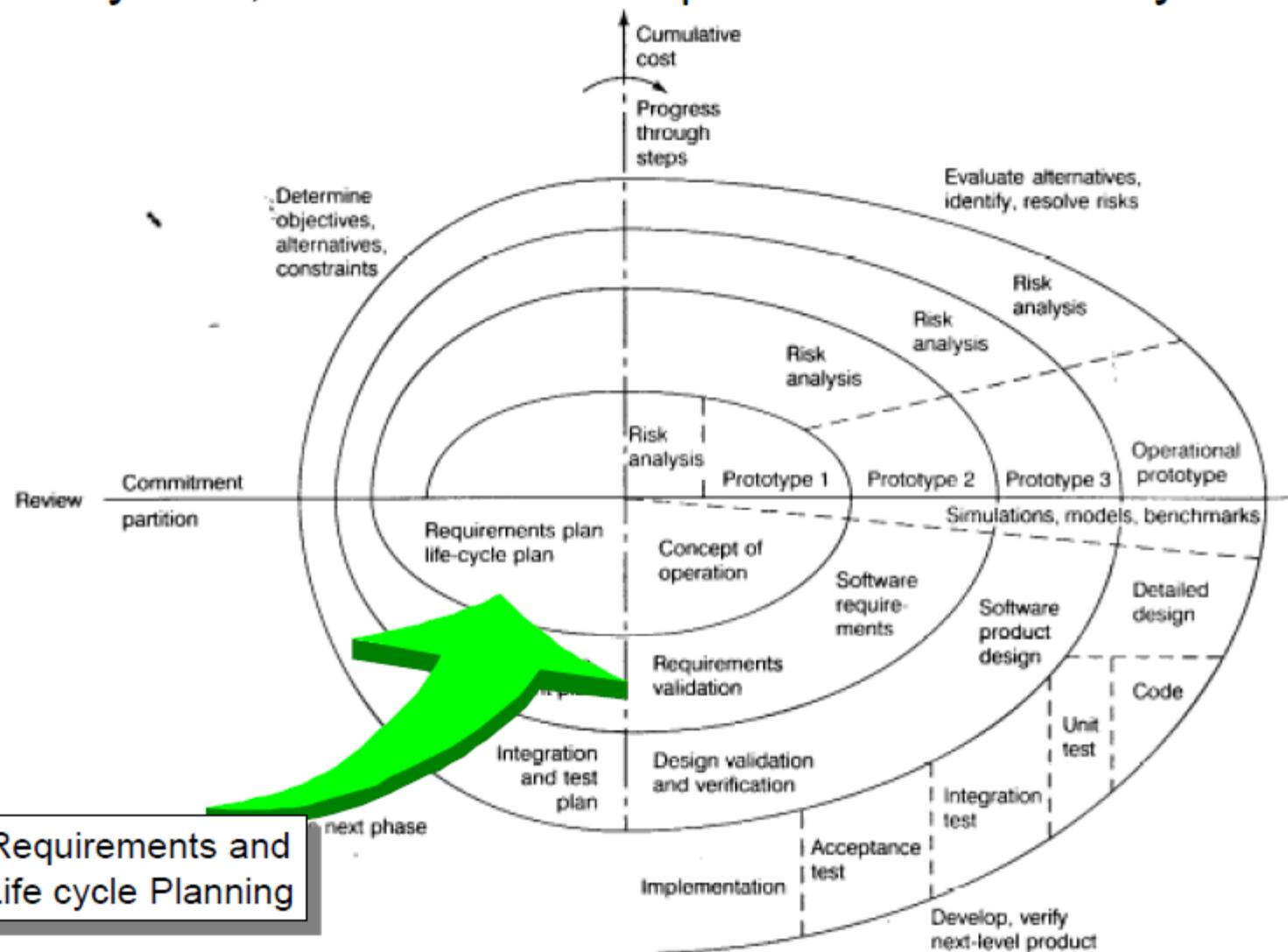
Cycle 1, Quadrant I: Evaluate Alternatives, Identify, resolve risks



Cycle 1, Quadrant II: Develop & Verify Product



Cycle 1, Quadrant III: Prepare for Next Activity



Requirements and
Life cycle Planning

Evolutionary Model: The Spiral

A spiral model is divided into a set of framework activities defined by the software engineering team.

The software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center.

Anchor point milestones—a combination of work products and conditions that are attained along the path of the spiral—are noted for each evolutionary pass.

The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.

Cost and schedule are adjusted based on feedback derived from the customer after delivery.

Evolutionary Model: The Spiral

Each loop in the spiral is split into four sectors:

Objective setting. Specific objectives for that phase of the project are defined

Risk assessment and reduction. For each of the identified project risks, a detailed analysis is carried out.

Development and validation. After risk evaluation, a development model for the system is chosen.

Planning. The project is reviewed and a decision made whether to continue with a further loop of the spiral.

Selecting Software Models

Table Selections on the Basis of the Project Type and Associated Risks

Project Type and Associated Risks	Waterfall	Prototype	Spiral	RAD	Formal Methods
Reliability requirements	No	No	Yes	No	Yes
Stable funds	Yes	Yes	No	Yes	Yes
Reuse components	No	Yes	Yes	Yes	Yes
Tight project schedule	No	Yes	Yes	Yes	No
Scarcity of resources	No	Yes	Yes	No	No

Selecting Software Models

Table Selection on the Basis of the Requirements of the Project

Requirements of the Project	Waterfall	Prototype	Spiral	RAD	Formal Methods
Requirements are defined early in SDLC	Yes	No	No	Yes	No
Requirements are easily defined and understandable	Yes	No	No	Yes	Yes
Requirements are changed frequently	No	Yes	Yes	No	Yes
Requirements indicate a complex System	No	Yes	Yes	No	No

Selecting Software Models

Table Selection on the Basis of the Users

User Involvement	Waterfall	Prototype	Spiral	RAD	Formal Methods
Requires Limited User Involvement	Yes	No	Yes	No	Yes
User participation in all phases	No	Yes	No	Yes	No
No experience of participating in similar projects	No	Yes	Yes	No	Yes

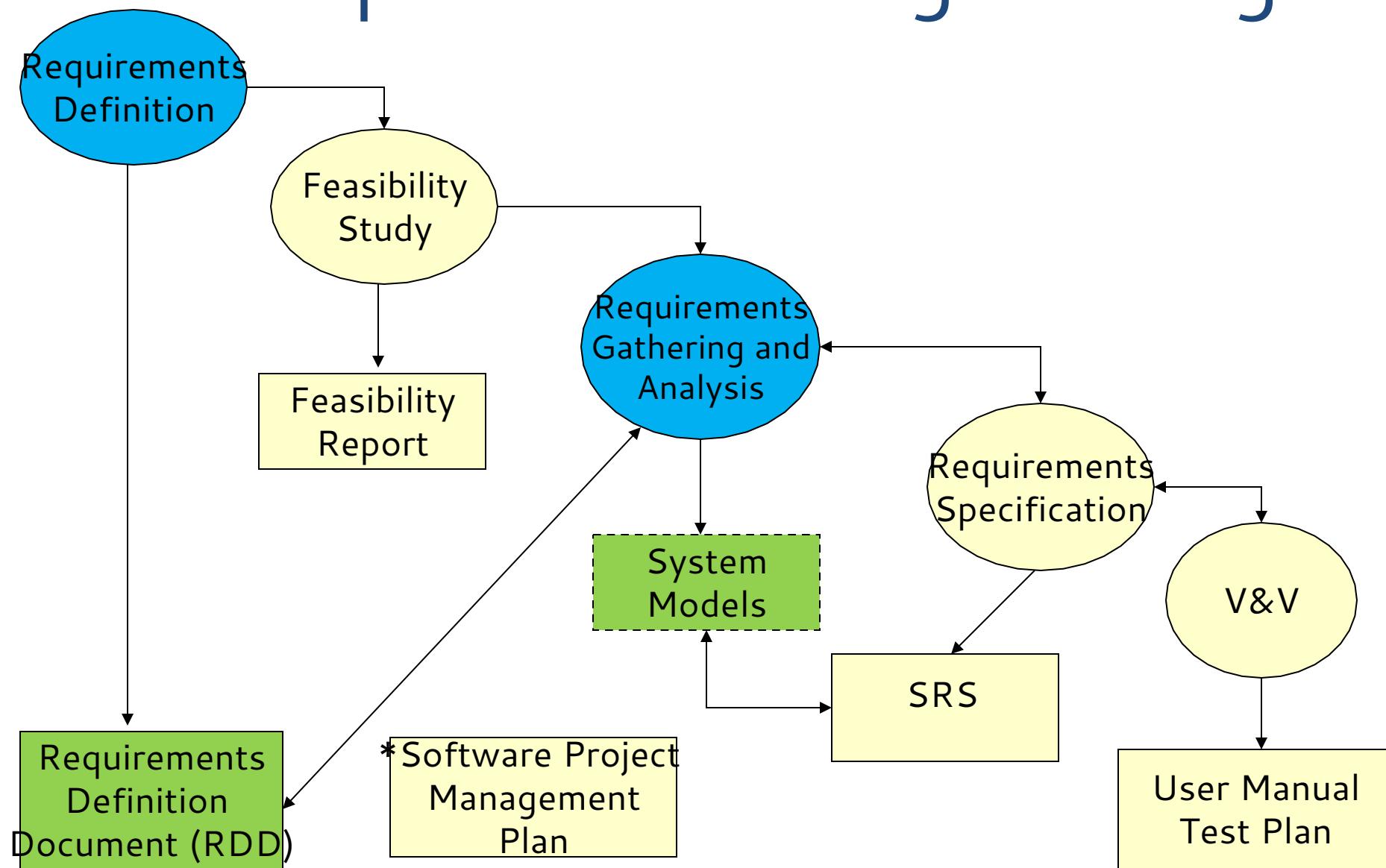
Features	Original water fall	Iterative water fall	Prototyping model	Spiral model
Requirement Specification	Beginning	Beginning	Frequently Changed	Beginning
Understanding Requirements	Well Understood	Not Well understood	Not Well understood	Well Understood
Cost	Low	Low	High	Expensive
Availability of reuseable component	No	Yes	yes	yes
Complexity of system	Simple	simple	complex	complex
Risk Analysis	Only at beginning	No Risk Analysis	No Risk Analysis	yes
User Involvement in all phases of SDLC	Only at beginning	Intermediate	High	High
Guarantee of Success	Less	High	Good	High
Overlapping Phases	No overlapping	No Overlapping	Yes Overlapping	Yes Overlapping
Implementation time	long	Less	Less	Depends on project
Flexibility	Rigid	Less Flexible	Highly Flexible	Flexible

Unit 3- Requirement Phase

Contents

- Requirement Engineering.
 - Requirement Gathering
 - Types of Requirement
 - Comparison
 - Task
 - Use Case
- Feasibility Study
 - Types of Feasibility
- Software Requirement Document
 - Template

Requirements Engineering



Requirement Engineering

- The broad spectrum of tasks and techniques that lead to an understanding of requirements is called *requirements engineering*.
- Requirements engineering provides the appropriate mechanism for understanding **what the customer wants**, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system [Tha97].
- It encompasses seven distinct tasks: **inception, elicitation, elaboration, negotiation, specification, validation, and management**. It is important to note that some of these tasks occur in parallel and all are adapted to the needs of the project

- **Inception:** At project inception, you establish a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team.
- **Elicitation.**
 - Problems of scope.
 - Problems of understanding.
 - Problems of volatility.
- **Elaboration:** Elaboration is driven by the creation and refinement of **user scenarios** that describe how the end user (and other actors) will interact with the system. Each user scenario is parsed to extract analysis classes—business domain entities that are visible to the end user.
- **Negotiation.**
- **Specification:** A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.
- **Validation:**

Why do we need Requirements?

- When 38 IT professionals in the UK were asked about which project stages caused failure, respondents mentioned “requirements definition” more than any other phase.

What are requirements?

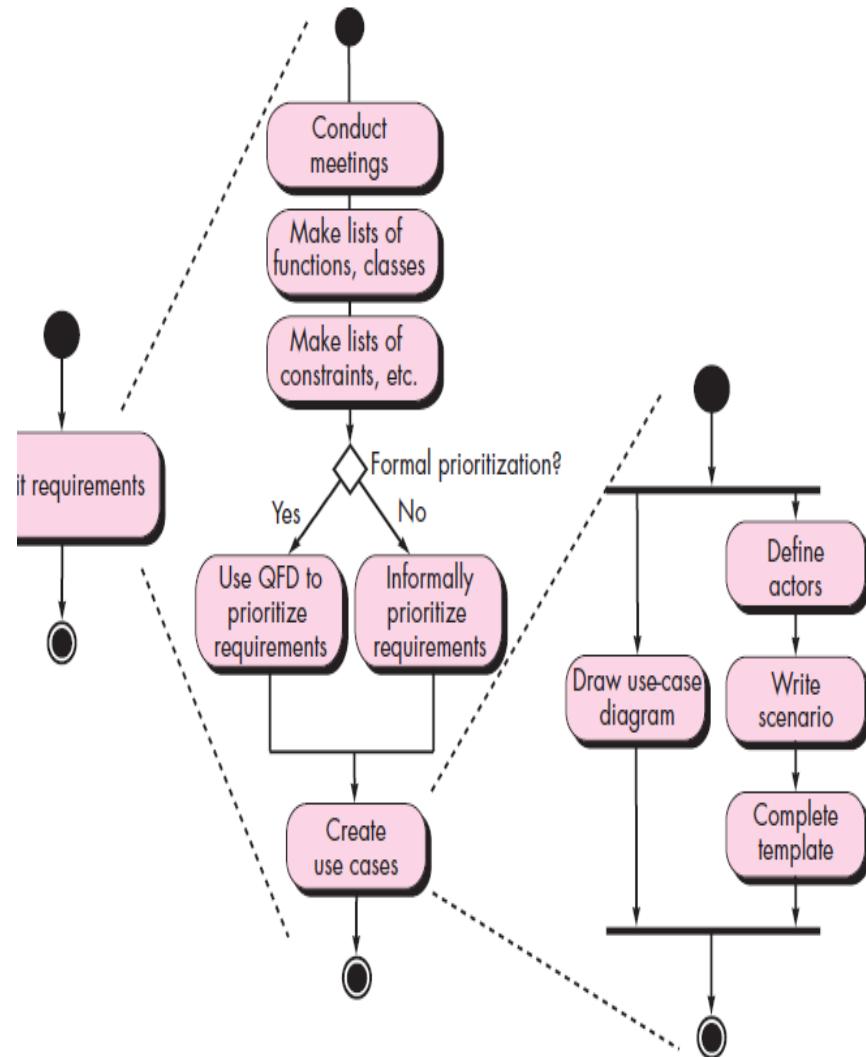
- A requirement is a statement about an intended product that specifies what it should do or how it should perform.
- Goal: To make as specific, unambiguous, and clear as possible.

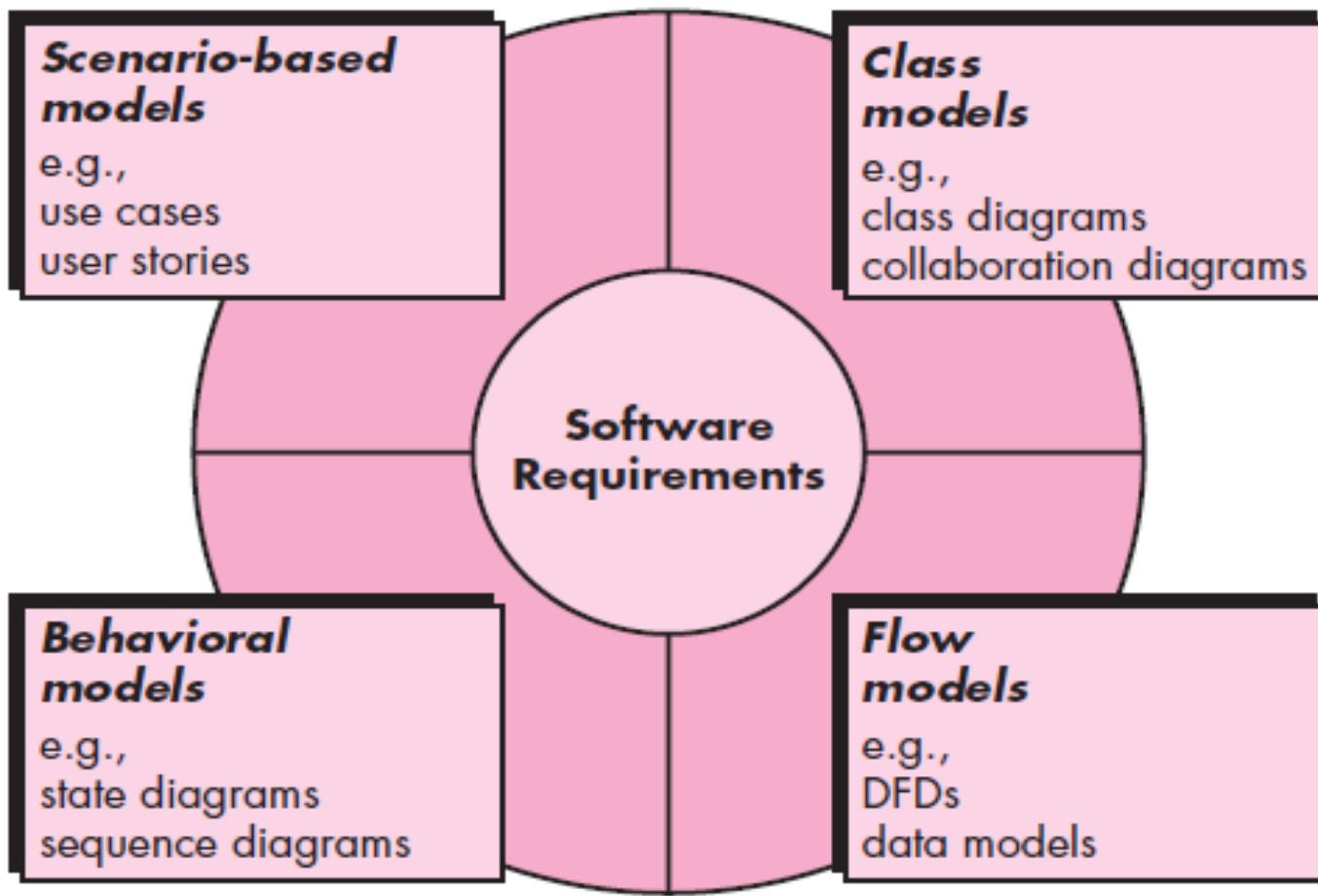
What requirements should be gathered?

- Functional: What the product should do.
- Data requirements: Capture the type, volatility, size/amount, persistence, accuracy and the amounts of the required data.
- Environmental requirements: a) context of use b) Social environment (eg. Collaboration and coordination) c) how good is user support likely to be d) what technologies will it run on
- User Requirements: Capture the characteristics of the intended user group.
- Usability Requirement: Usability goals associated measures for a particular product (More info on Chapter

Requirement Modeling

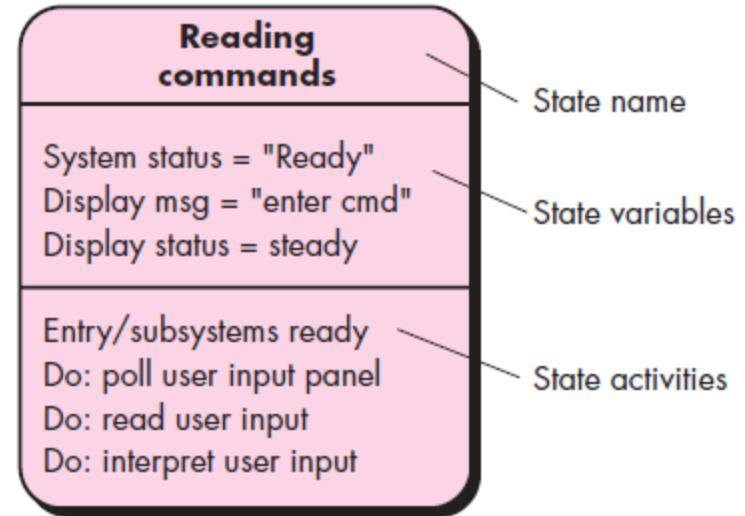
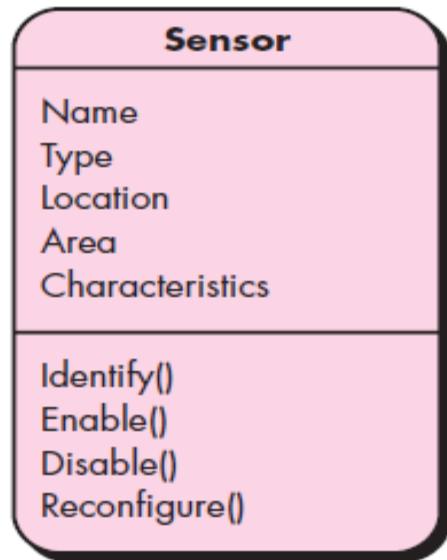
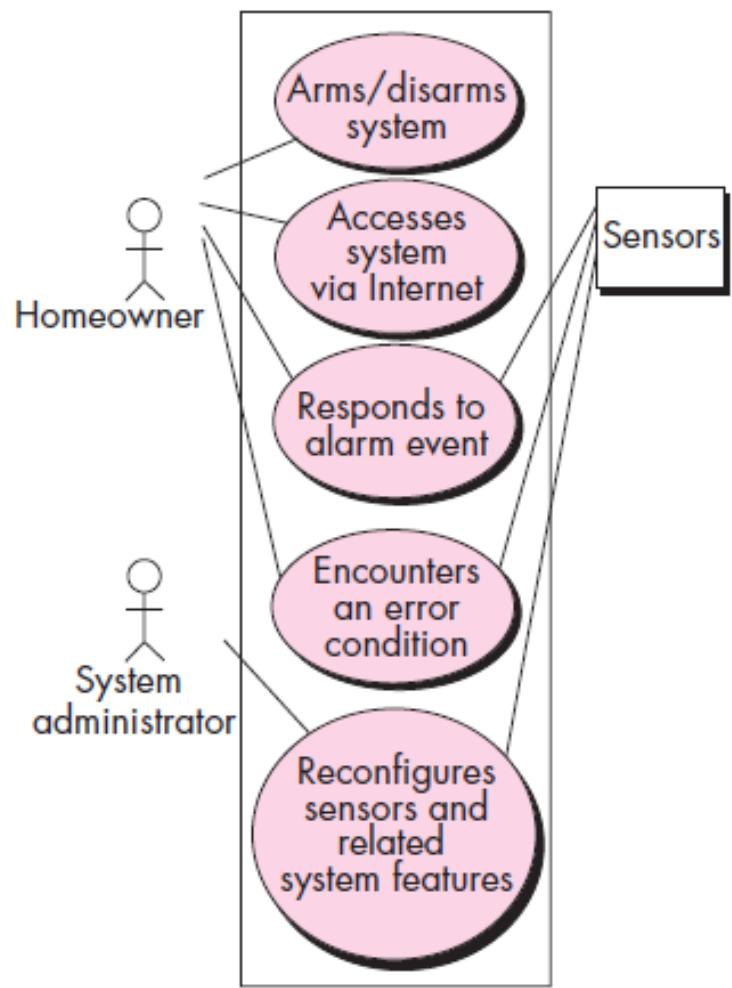
- Scenario-based models of requirements from the point of view of various system "actors"
- Data models that depict the information domain for the problem
- Class-oriented models that represent object-oriented classes (attributes and operations) and the manner in which classes collaborate to achieve system requirements
- Flow-oriented models that represent the functional elements of the system and how they transform data as it moves through the system
- Behavioral models that depict how the software behaves as a consequence of external "events"





Requirement Modelling

- Who is the primary actor, the secondary actor(s)?
- What are the actor's goals?
- What preconditions should exist before the story begins?
- What main tasks or functions are performed by the actor?
- What exceptions might be considered as the story is described?
- What variations in the actor's interaction are possible?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?



Illustration

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Actor: homeowner

If I'm at a remote location, I can use any PC with appropriate browser software to log on to the *SafeHome Products* website. I enter my user ID and two levels of passwords and once I'm validated, I have access to all functionality for my installed *SafeHome* system. To access a specific camera view, I select "surveillance" from the major function buttons displayed. I then select "pick a camera" and the floor plan of the house is displayed. I then select the camera that I'm interested in. Alternatively, I can look at thumbnail snapshots from all cameras simultaneously by selecting "all cameras" as my viewing choice. Once I choose a camera, I select "view" and a one-frame-per-second view appears in a viewing window that is identified by the camera ID. If I want to switch cameras, I select "pick a camera" and the original viewing window disappears and the floor plan of the house is displayed again. I then select the camera that I'm interested in. A new viewing window appears.

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Actor: homeowner

1. The homeowner logs onto the *SafeHome Products* website.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects the “surveillance” from the major function buttons.
6. The homeowner selects “pick a camera.”
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the “view” button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

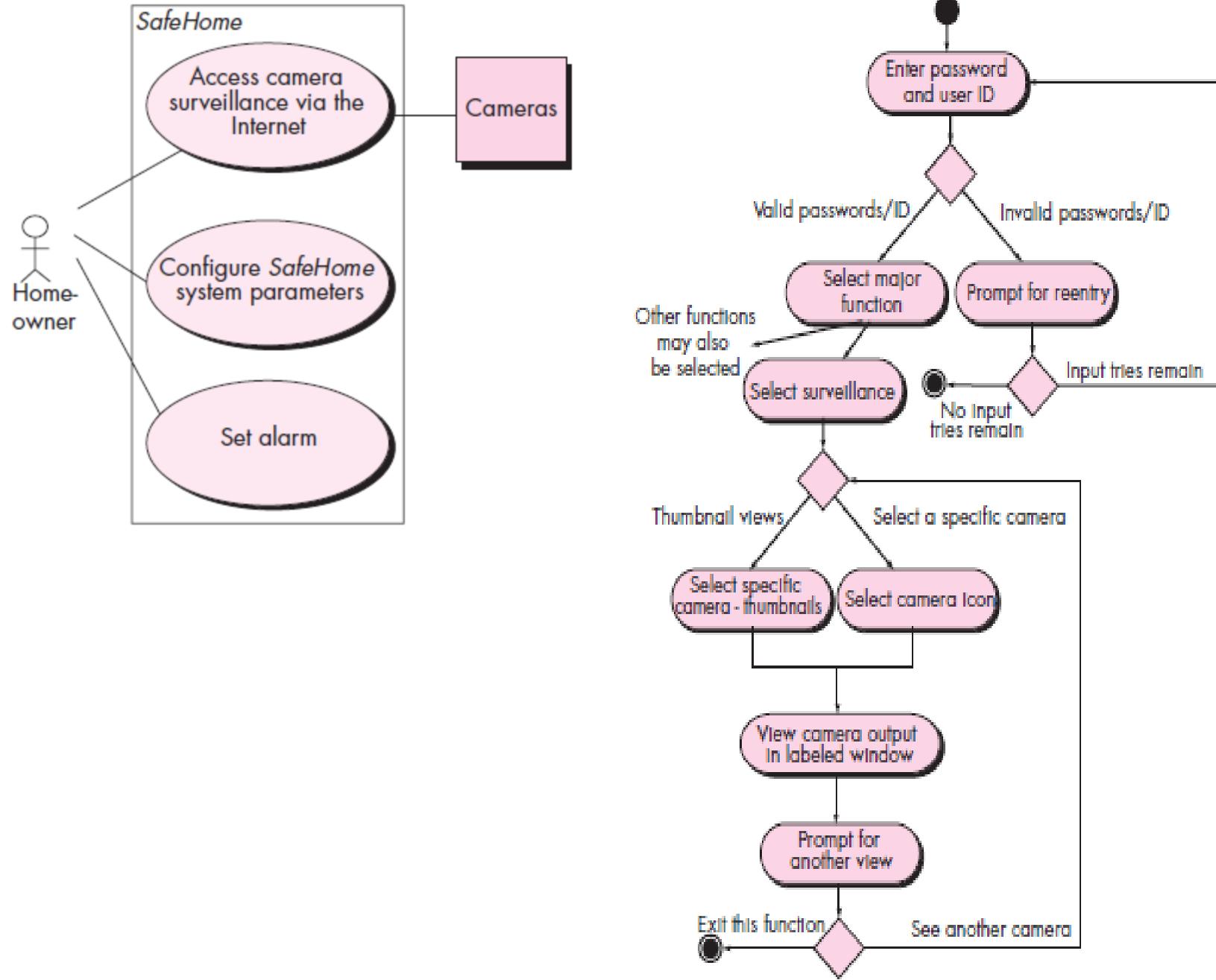
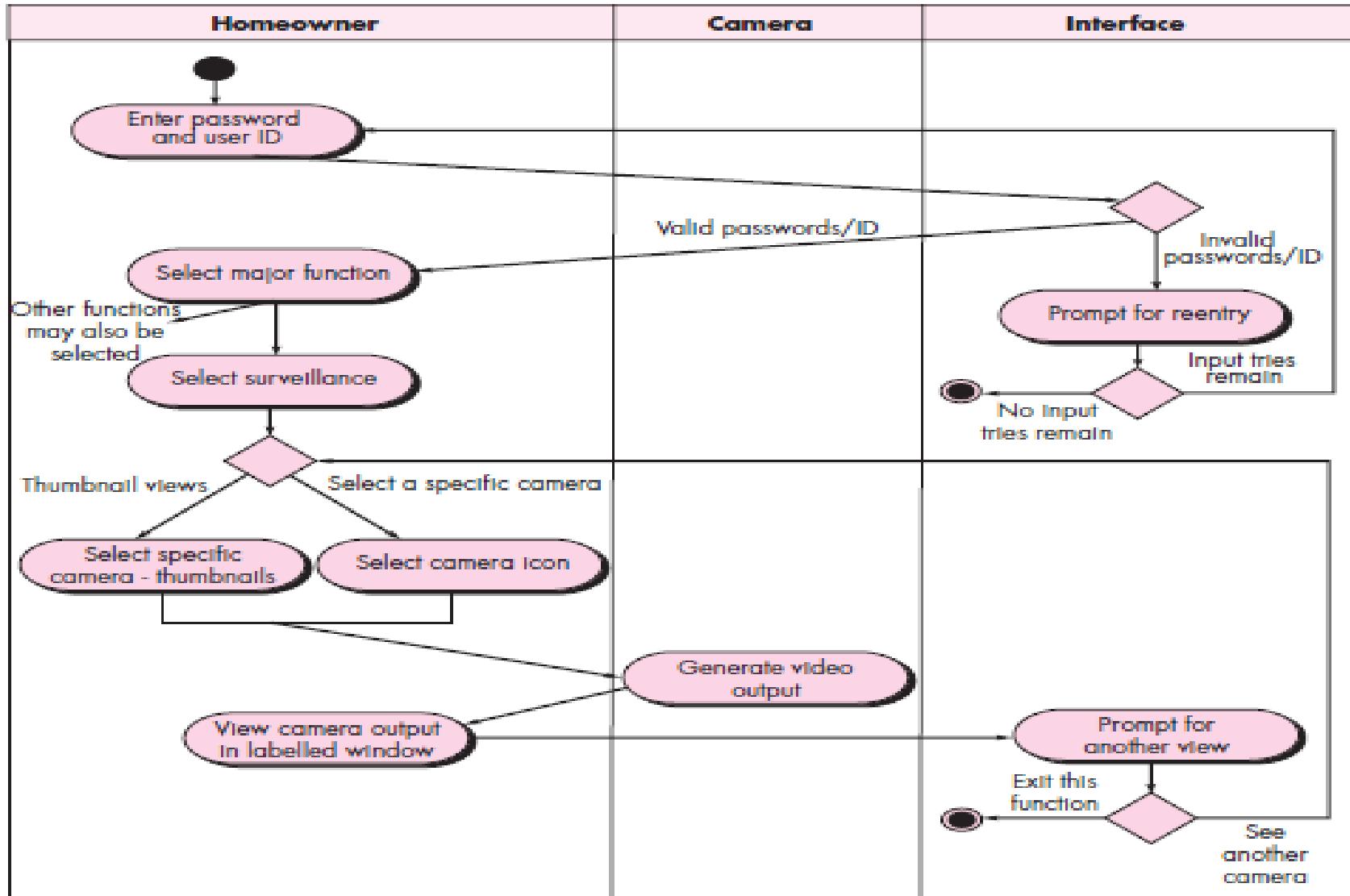


FIGURE 6.6 Swimlane diagram for Access camera surveillance via the Internet—display camera views function



Purpose of SRS document?

SRS establishes basis of agreement between the user and the supplier.

Users needs have to be satisfied, but user may not understand software

Developers will develop the system, but may not know about problem domain

SRS is

the medium to bridge the communications gap, and

specifies user needs in a manner both can understand

Need for SRS...

Helps user understand his needs.

users do not always know their needs
must analyze and understand the
potential

The requirement process helps clarify
needs

SRS provides a reference for validation of
the final product

Clear understanding about what is
expected.

Need for SRS...

- High quality SRS essential for high Quality SW
 - Requirement errors get manifested in final sw
 - To satisfy the quality objective, must begin with high quality SRS
 - Requirements defects cause later problems
 - 25% of all defects in one study; 54% of all defects found after user testing

Need for SRS...

Good SRS reduces the development cost

SRS errors are expensive to fix later

Req. changes can cost a lot (up to 40%)

Good SRS can minimize changes and errors

Substantial savings; extra effort spent during req. saves multiple times that effort

An Example

Cost of fixing errors in req. , design ,
coding, acceptance testing and operation

Requirements Process

- Basic activities:
 - Problem or requirement analysis
 - Requirement specification
 - Validation
- Analysis involves elicitation and is the hardest

Need for SRS...

Good SRS reduces the development cost

SRS errors are expensive to fix later

Req. changes can cost a lot (up to 40%)

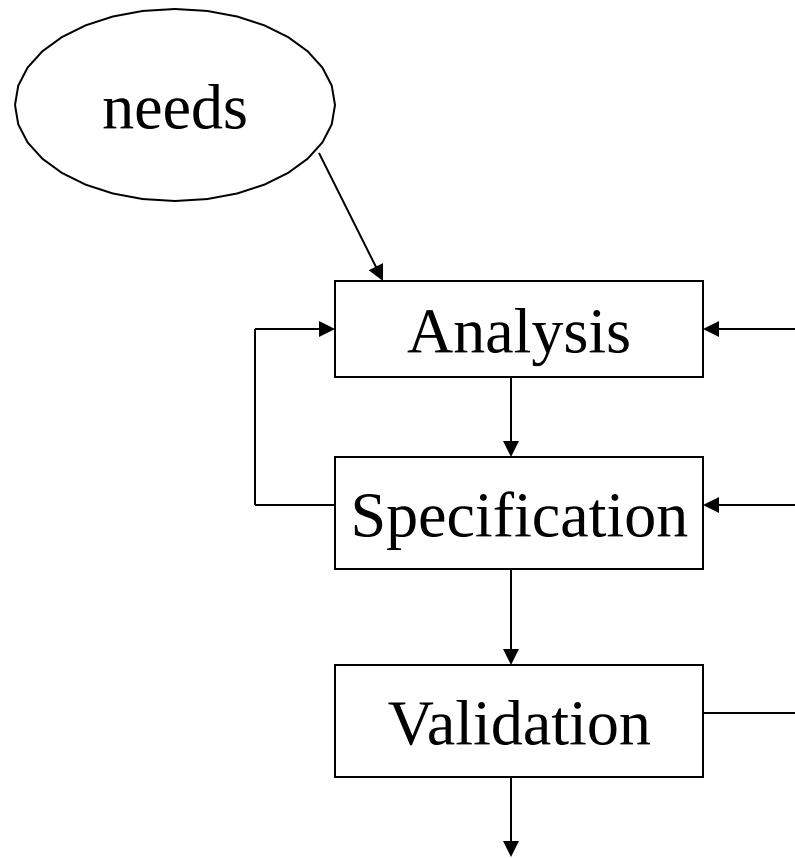
Good SRS can minimize changes and errors

Substantial savings; extra effort spent during req. saves multiple times that effort

An Example

Cost of fixing errors in req. , design ,
coding, acceptance testing and operation

Requirement process..



Process is not linear, it is iterative and parallel

Overlap between phases – some parts may be analyzed and specified

Specification itself may help analysis

Validation can show gaps that can lead to further analysis and spec



Software Requirements Specification Template

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

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

Table of Contents

Revision History

1. Introduction

- 1.1 Purpose
- 1.2 Document Conventions
- 1.3 Intended Audience and Reading Suggestions
- 1.4 Project Scope
- 1.5 References

2. Overall Description

- 2.1 Product Perspective

2.2 Product Features

2.3 User Classes and Characteristics

2.4 Operating Environment

2.5 Design and Implementation Constraints

2.6 User Documentation

2.7 Assumptions and Dependencies

3. System Features

3.1 System Feature 1

3.2 System Feature 2 (and so on)

4. External Interface Requirements

4.1 User Interfaces

4.2 Hardware Interfaces

4.3 Software Interfaces

4.4 Communications Interfaces

5. Other Nonfunctional Requirements

5.1 Performance Requirements

5.2 Safety Requirements

5.3 Security Requirements

5.4 Software Quality Attributes

6. Other Requirements

Appendix A: Glossary

Appendix B: Analysis Models

Appendix C: Issues List

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



Requirements Validation Checklist

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

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

Characteristics of an SRS

- Correct
- Complete
- Unambiguous
- Consistent
- Verifiable
- Traceable
- Modifiable
- Ranked for importance and/or stability

Characteristics...

- Correctness
 - Each requirement accurately represents some desired feature in the final system
- Completeness
 - All desired features/characteristics specified
 - Hardest to satisfy
 - Completeness and correctness strongly related
- Unambiguous
 - Each req has exactly one meaning
 - Without this errors will creep in
 - Important as natural languages often used

Characteristics...

- Verifiability
 - There must exist a cost effective way of checking if sw satisfies requirements
- Consistent
 - two requirements don't contradict each other
- Traceable
 - The origin of the req, and how the req relates to software elements can be determined
- Ranked for importance/stability
 - Needed for prioritizing in construction
 - To reduce risks due to changing requirements

the Feasibility Study

- ↪ What is a feasibility study?

- ↳ What to study and conclude?

- ↪ Types of feasibility

- ↳ Technical
 - ↳ Economic
 - ↳ Schedule
 - ↳ Operational

- ↪ Quantifying benefits and costs

- ↳ Payback analysis
 - ↳ Net Present Value Analysis
 - ↳ Return on Investment Analysis

- ↪ Comparing alternatives

Exploring Feasibility

› The “PIECES” framework

- ↳ Useful for identifying operational problems to be solved, and their urgency
- ↳ Performance
 - Is current throughput and response time adequate?
- ↳ Information
 - Do end users and managers get timely, pertinent, accurate and usefully formatted information?
- ↳ Economy
 - Are services provided by the current system cost-effective?
 - Could there be a reduction in costs and/or an increase in benefits?
- ↳ Control
 - Are there effective controls to protect against fraud and to guarantee information accuracy and security?
- ↳ Efficiency
 - Does current system make good use of resources: people, time, flow of forms,...?
- ↳ Services
 - Are current services reliable? Are they flexible and expandable?



Four Types of feasibility

Technical feasibility

Is the project possible with current technology?

What technical risk is there?

Availability of the technology:

Is it available locally?

Can it be obtained?

Will it be compatible with other systems?

Economic feasibility

Is the project possible, given resource constraints?

What are the benefits?

Both tangible and intangible

Quantify them!

What are the development and operational costs?

Are the benefits worth the costs?

Schedule feasibility

Is it possible to build a solution in time to be useful?

What are the consequences of delay?

Any constraints on the schedule?

Can these constraints be met?

Operational feasibility

If the system is developed, will it be used?

Human and social issues...

Potential ~~labour~~ objections?

Manager resistance?

Organizational conflicts and policies?

Social acceptability?

Legal aspects and government regulations?

Economic Feasibility

- Can the bottom line be quantified yet?
 - Very early in the project...
 - a judgement of whether solving the problem is worthwhile.
 - Once specific requirements and solutions have been identified...
 - ...the costs and benefits of each alternative can be calculated
- Cost-benefit analysis
 - Purpose – answer questions such as:
 - Is the project justified (I.e. will benefits outweigh costs)?
 - What is the minimal cost to attain a certain system?
 - How soon will the benefits accrue?
 - Which alternative offers the best return on investment?
 - Examples of things to consider:
 - Hardware/software selection
 - Selection among alternative financing arrangements (rent/lease/purchase)
 - Difficulties
 - benefits and costs can both be intangible, hidden and/or hard to estimate
 - ranking multi-criteria alternatives

Benefits

- Tangible Benefits
 - Readily quantified as \$ values
 - Examples:
 - increased sales
 - cost/error reductions
 - increased throughput/efficiency
 - increased margin on sales
 - more effective use of staff time
- Intangible benefits
 - Difficult to quantify
 - But maybe more important!
 - business analysts help estimate \$ values
 - Examples:
 - increased flexibility of operation
 - higher quality products/services
 - better customer relations
 - improved staff morale
- How will the benefits accrue?
 - When – over what timescale?
 - Where in the organization?

Costs

- Development costs (OTO)
 - Development and purchasing costs:
 - Cost of development team
 - Consultant fees
 - software used (buy or build)?
 - hardware (what to buy, buy/lease)?
 - facilities (site, communications, power,...)
 - Installation and conversion costs:
 - installing the system,
 - training personnel,
 - file conversion,....
- Operational costs (on-going)
 - System Maintenance:
 - hardware (repairs, lease, supplies,...),
 - software (licenses and contracts),
 - facilities
 - Personnel:
 - For operation (data entry, backups,...)
 - For support (user support, hardware and software maintenance, supplies,...)
 - On-going training costs

Example: costs for small Client–Server project



Analyzing Costs vs. Benefits

↪ Identify costs and benefits

- ↳ Tangible and intangible, one-time and recurring
- ↳ Assign values to costs and benefits

↪ Determine Cash Flow

- ↳ Project the costs and benefits over time, e.g. 3-5 years
- ↳ Calculate Net Present Value for all future costs/benefits
 - determines future costs/benefits of the project in terms of today's dollar values
 - A dollar earned today is worth more than a potential dollar earned next year

↪ Do cost/benefit analysis

- ↳ Calculate Return on Investment:
 - Allows comparison of lifetime profitability of alternative solutions.

$$\text{ROI} = \frac{\text{Total Profit}}{\text{Total Cost}} = \frac{\text{Lifetime benefits} - \text{Lifetime costs}}{\text{Lifetime costs}}$$

- ↳ Calculate Break-Even point:

- how long will it take (in years) to pay back the accrued costs:
@T (Accrued Benefit > Accrued Cost)



Calculating Present Value

➲ A dollar today is worth more than a dollar tomorrow...

- ↳ Your analysis should be normalized to "current year" dollar values.

➲ The discount rate

- ↳ measures opportunity cost:
 - Money invested in this project means money not available for other things
 - Benefits expected in future years are more prone to risk
- ↳ This number is company- and industry-specific.
 - "what is the average annual return for investments in this industry?"

➲ Present Value:

- ↳ The "current year" dollar value for costs/benefits n years into the future
 - ... for a given discount rate i

$$\text{Present_Value}(n) = \frac{1}{(1 + i)^n}$$

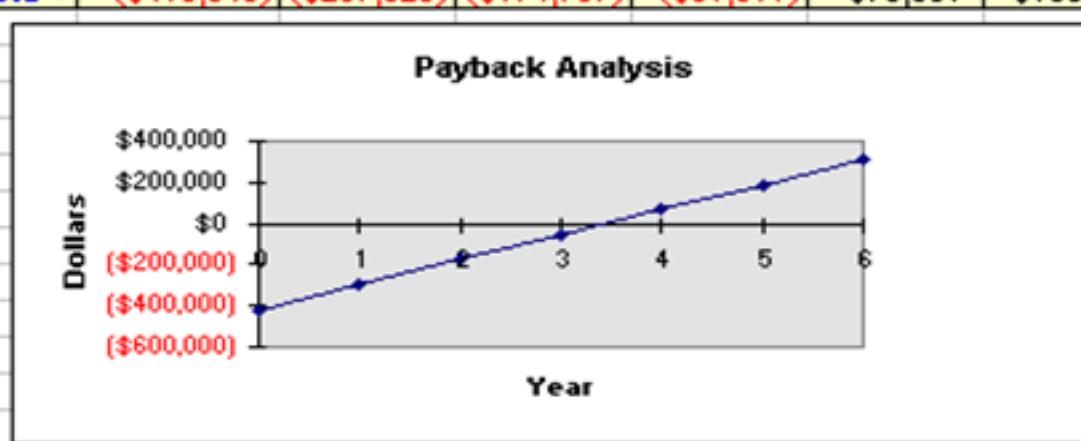
- ↳ E.g. if the discount rate is 12%, then

- $\text{Present_Value}(1) = 1/(1 + 0.12)^1 = 0.893$
- $\text{Present_Value}(2) = 1/(1 + 0.12)^2 = 0.797$

Payback Analysis for Client-Server System Alternative

(Numbers rounded to nearest \$1)

	Cash flow description	Year 0	Year 1	Year 2	Year 3	Year 4	Year 5	Year 6
5	Development cost:	(\$418,040)						
6	Operation & maintenance cost:		(\$15,045)	(\$16,000)	(\$17,000)	(\$18,000)	(\$19,000)	(\$20,000)
7	Discount factors for 12%:	1.000	0.893	0.797	0.712	0.636	0.567	0.507
8	Time-adjusted costs (adjusted to present)	(\$418,040)	(\$13,435)	(\$12,752)	(\$12,104)	(\$11,448)	(\$10,773)	(\$10,140)
9	Cumulative time-adjusted costs over	(\$418,040)	(\$431,475)	(\$444,227)	(\$456,331)	(\$467,779)	(\$478,552)	(\$488,692)
10								
11	Benefits derived from operation of new	\$0	\$150,000	\$170,000	\$190,000	\$210,000	\$230,000	\$250,000
12	Discount factors for 12%:	1.000	\$0.89	\$0.80	\$0.71	\$0.64	\$0.57	\$0.51
13	Time-adjusted benefits (current of present)	\$0	\$133,950	\$135,490	\$135,280	\$133,560	\$130,410	\$126,750
14	Cumulative time-adjusted benefits over	\$0	\$133,950	\$269,440	\$404,720	\$538,280	\$668,690	\$795,440
15		0	1	2	3	4	5	6
16	Cumulative lifetime time-adjusted costs +	(\$418,040)	(\$297,525)	(\$174,787)	(\$51,611)	\$70,501	\$190,138	\$306,748



Schedule Feasibility

- How long will it take to get the technical expertise?
 - We may have the technology, but that doesn't mean we have the skills required to properly apply that technology.
 - May need to hire new people
 - Or re-train existing systems staff
 - Whether hiring or training, it will impact the schedule.
- Assess the schedule risk:
 - Given our technical expertise, are the project deadlines reasonable?
 - If there are specific deadlines, are they mandatory or desirable?
 - If the deadlines are not mandatory, the analyst can propose several alternative schedules.
- What are the real constraints on project deadlines?
 - If the project overruns, what are the consequences?
 - Deliver a properly functioning information system two months late...
 - ...or deliver an error-prone, useless information system on time?
 - Missed schedules are bad, but inadequate systems are worse!

Software

Engineering

LECTURE 15: Software Complexity Metrics

*Ivan Marsic
Rutgers University*

Topics

- Measuring Software Complexity
- Cyclomatic Complexity

Measuring Software Complexity

- Software complexity is difficult to operationalize complexity so that it can be measured
- Computational complexity measure big O (or big Oh), $O(n)$
 - Measures software complexity from the *machine's viewpoint* in terms of how the size of the input data affects an algorithm's usage of computational resources (usually running time or memory)
- Complexity measure in software engineering should measure complexity from the *viewpoint of human developers*
 - Computer time is cheap; human time is expensive

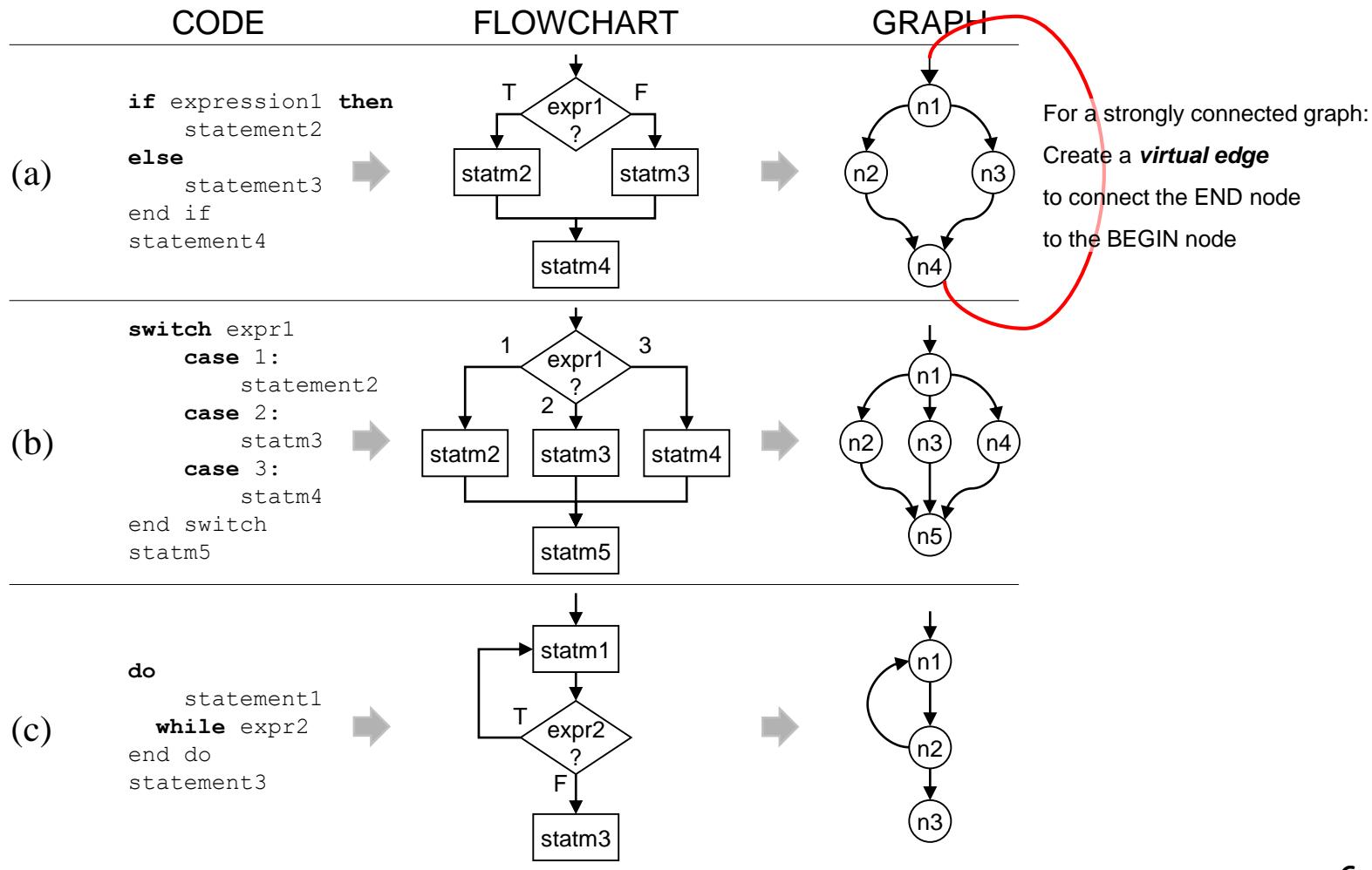
Desirable Properties of Complexity Metrics

- **Monotonicity:** adding responsibilities to a module cannot decrease its complexity
 - If a responsibility is added to a module, the modified module will exhibit a complexity value that is the same as or higher than the complexity value of the original module
- **Ordering** (“representation condition” of measurement theory):
 - Metric produces the same ordering of values as intuition would
 - Cognitively more difficult should be measured as greater complexity
- **Discriminative power** (sensitivity): modifying responsibilities should change the complexity
 - Discriminability is expected to increase as:
 - 1) the number of distinct complexity values increases and
 - 2) the number of classes with equal complexity values decreases
- **Normalization:** allows for easy comparison of the complexity of different classes

Cyclomatic Complexity

- Invented by Thomas McCabe (1974) to measure the complexity of a program's conditional logic
 - Counts the number of decisions in the program, under the assumption that decisions are difficult for people
 - Makes assumptions about decision-counting rules and linear dependence of the total count to complexity
- Cyclomatic complexity of graph G equals
 $\#edges - \#nodes + 2$
 - $V(G) = e - n + 2$
- Also corresponds to the number of linearly independent paths in a program (described later)

Converting Code to Graph

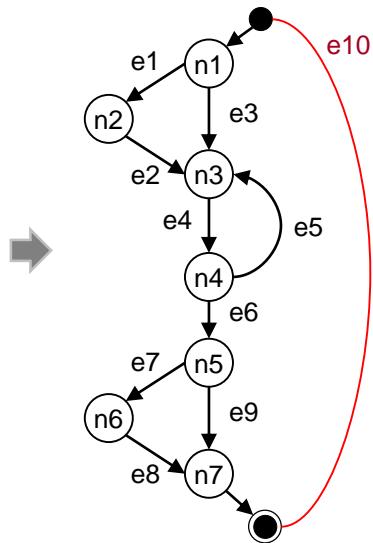


Paths in Graphs (1)

- A graph is **strongly connected** if for any two nodes x, y there is a path from x to y and vice versa
- A **path** is represented as an n -element vector where n is the number of edges
 $\langle \square, \square, \dots, \square \rangle$
- The i -th position in the vector is the number of occurrences of edge i in the path

Example Paths

```
if expression1 then  
    statement2  
end if  
  
do  
    statement3  
    while expr4  
end do  
  
if expression5 then  
    statement6  
end if  
statement7
```



Paths:

- P1 = e1, e2, e4, e6, e7, e8
- P2 = e1, e2, e4, e5, e4, e6, e7, e8
- P3 = e3, e4, e6, e7, e8, e10
- P4 = e6, e7, e8, e10, e3, e4
- P5 = e1, e2, e4, e6, e9, e10
- P6 = e4, e5
- P7 = e3, e4, e6, e9, e10
- P8 = e1, e2, e4, e5, e4, e6, e9, e10

e1	e2	e3	e4	e5	e6	e7	e8	e9	e10
1, 1, 0, 1, 0, 1, 1, 1, 0, 0									
1, 1, 0, 2, 1, 1, 1, 1, 0, 0									
0, 0, 1, 1, 0, 1, 1, 1, 0, 1									
0, 0, 1, 1, 0, 1, 1, 1, 0, 1									
1, 1, 0, 1, 0, 1, 0, 0, 1, 1									
0, 0, 0, 1, 1, 0, 0, 0, 0, 0									
0, 0, 1, 1, 0, 1, 0, 0, 1, 1									
1, 1, 0, 2, 1, 1, 0, 0, 1, 1									

Paths P3 and P4 are the same, but with different start and endpoints

NOTE: A path does not need to start in node n1 and does not need to begin and end at the same node.

E.g.,

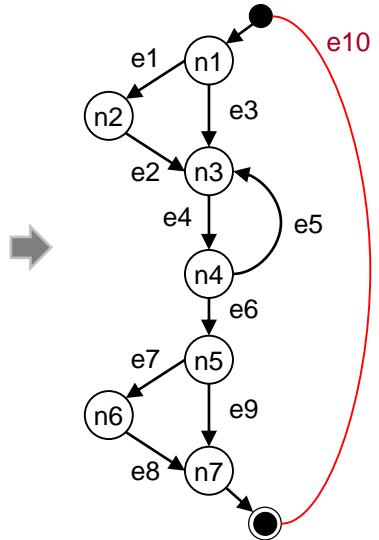
- Path P4 starts (and ends) at node n4
- Path P1 starts at node n1 and ends at node n7

Paths in Graphs (2)

- A **circuit** is a path that begins and ends at the same node
 - e.g., $P_3 = \langle e_3, e_4, e_6, e_7, e_8, e_{10} \rangle$ begins and ends at node n_1
 - $P_6 = \langle e_4, e_5 \rangle$ begins and ends at node n_3
- A **cycle** is a circuit with no node (other than the starting node) included more than once

Example Circuits & Cycles

```
if expression1 then  
    statement2  
end if  
  
do  
    statement3  
    while expr4  
end do  
  
if expression5 then  
    statement6  
end if  
statement7
```



Circuits:

- P3 = e3, e4, e6, e7, e8, e10
- P4 = e6, e7, e8, e10, e3, e4
- P5 = e1, e2, e4, e6, e9, e10
- P6 = e4, e5
- P7 = e3, e4, e6, e9, 10
- P8 = e1, e2, e4, e5, e4, e6, e9, e10
- P9 = e3, e4, e5, e4, e6, e9, 10

e1	e2	e3	e4	e5	e6	e7	e8	e9	e10
0, 0, 1, 1, 0, 1, 1, 1, 0, 1									
0, 0, 1, 1, 0, 1, 1, 1, 0, 1									
1, 1, 0, 1, 0, 1, 0, 0, 1, 1									
0, 0, 0, 1, 1, 0, 0, 0, 0, 0									
0, 0, 1, 1, 0, 1, 0, 0, 1, 1									
1, 1, 0, 2, 1, 1, 0, 0, 1, 1									
0, 0, 1, 2, 1, 1, 0, 0, 1, 1									

Cycles:

- P3 = e3, e4, e6, e7, e8, e10
- P5 = e1, e2, e4, e6, e9, e10
- P6 = e4, e5
- P7 = e3, e4, e6, e9, 10

P4, P8, P9 are not cycles

Linearly Independent Paths

- A path p is said to be a **linear combination** of paths p_1, \dots, p_n if there are integers a_1, \dots, a_n such that $p = \sum a_i \cdot p_i$ (a_i could be negative, zero, or positive)
- A set of paths in a strongly connected graph is **linearly independent** if no path in the set is a linear combination of any other paths in the set
 - A **linearly independent path** is any path *through* the program (“complete path”) that introduces at least one *new edge* that is not included in any other linearly independent paths.
 - A path that is subpath of another path is not considered to be a linearly independent path.
- A **basis set of cycles** is a maximal linearly independent set of cycles
 - In a graph with e edges and n nodes, the basis has $e - n + 1$ cycles
 - +1 is for the virtual edge, introduced to obtain a strongly connected graph
- Every path is a linear combination of basis cycles

Baseline method for finding the basis set of cycles

- Start at the source node
(the first statement of the program/module)
- Follow the leftmost path until the sink node is reached
- Repeatedly retrace this path from the source node, but change decisions at every node with out-degree ≥ 2 , starting with the decision node earliest in the path

T.J. McCabe & A.H. Watson, *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*, NIST Special Publication 500-235, 1996.

Linearly Independent Paths (1)

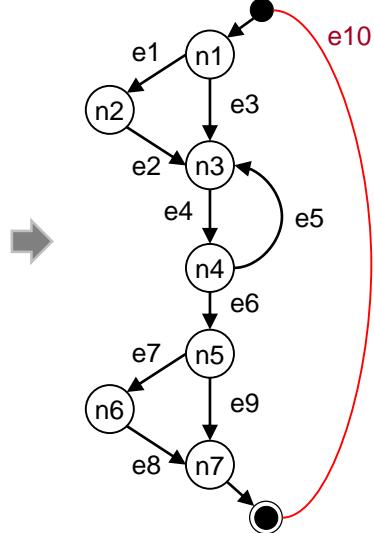
```

if expression1 then
    statement2
end if

do
    statement3
    while expr4
end do

if expression5 then
    statement6
end if
statement7

```



$$V(G) = e - n + 2 = 9 - 7 + 2 = 4$$

Or, if we count e_{10} , then $e - n + 1 = 10 - 7 + 1 = 4$

Cycles:

$P_3 = e_3, e_4, e_6, e_7, e_8, e_{10}$

$P_5 = e_1, e_2, e_4, e_6, e_9, e_{10}$

$P_6 = e_4, e_5$

$P_7 = e_3, e_4, e_6, e_9, 10$

Example paths:

$P_1 = e_1, e_2, e_4, e_6, e_7, e_8$

$P_2 = e_1, e_2, e_4, e_5, e_4, e_6, e_7, e_8$

$P_3 = e_3, e_4, e_6, e_7, e_8, e_{10}$

$P_4 = e_6, e_7, e_8, e_{10}, e_3, e_4$

$P_5 = e_1, e_2, e_4, e_6, e_9, e_{10}$

$P_6 = e_4, e_5$

$P_7 = e_3, e_4, e_6, e_9, 10$

$P_8 = e_1, e_2, e_4, e_5, e_4, e_6, e_9, e_{10}$

e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}
1, 1, 0, 1, 0, 1, 1, 1, 0, 0									
1, 1, 0, 2, 1, 1, 1, 1, 0, 0									
0, 0, 1, 1, 0, 1, 1, 1, 0, 1									
0, 0, 1, 1, 0, 1, 1, 1, 0, 1									
1, 1, 0, 1, 0, 1, 0, 0, 1, 1									
0, 0, 0, 1, 1, 0, 0, 0, 0, 0									
0, 0, 1, 1, 0, 1, 0, 0, 1, 1									
1, 1, 0, 2, 1, 1, 0, 0, 1, 1									

EXAMPLE #1: $P_5 + P_6 = P_8$

$$\begin{aligned}
 P_5 & \{1, 1, 0, 1, 0, 1, 0, 0, 1, 1\} \\
 + P_6 & \{0, 0, 0, 1, 1, 0, 0, 0, 0, 0\} \\
 = P_8 & \{1, 1, 0, 2, 1, 1, 0, 0, 1, 1\}
 \end{aligned}$$

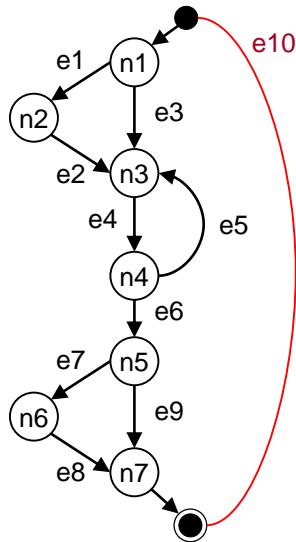
EXAMPLE #2: $2 \times P_3 - P_5 + P_6 =$

$$\begin{aligned}
 2 \times P_3 & \{0, 0, 2, 2, 0, 2, 2, 2, 0, 2\} \\
 - P_5 & \{1, 1, 0, 1, 0, 1, 0, 0, 1, 1\} \\
 \hline
 & \{-1, -1, 2, 1, 0, 1, 2, 2, -1, 1\} \\
 + P_6 & \{0, 0, 0, 1, 1, 0, 0, 0, 0, 0\} \\
 = P? & \{-1, -1, 2, 2, 1, 1, 2, 2, -1, 1\}
 \end{aligned}$$

→ Problem: The arithmetic doesn't work for *any* paths

— it works *always* only for *linearly independent paths!*

Linearly Independent Paths (2)



$$V(G) = e - n + 2 = 9 - 7 + 2 = 4$$

Linearly Independent Paths:

(by enumeration)

$$P1' = e1, e2, e4, e6, e7, e8, e10$$

$$P2' = e1, e2, e4, e5, e4, e6, e7, e8, e10$$

$$P3' = e3, e4, e6, e7, e8, e10$$

$$P4' = e1, e2, e4, e6, e9, e10$$

$$P1 = \rightarrow$$

$$P2 = \rightarrow$$

$$P3 = \rightarrow$$

$$(P4 \text{ same as } P3) \quad P4 = \rightarrow$$

$$P5 = \rightarrow$$

$$P6 = \rightarrow$$

$$P7 = \rightarrow$$

$$P8 = \rightarrow$$

e1	e2	e3	e4	e5	e6	e7	e8	e9	e10
1, 1, 0, 1, 0, 1, 1, 1, 0, 0									
1, 1, 0, 2, 1, 1, 1, 1, 0, 0									
0, 0, 1, 1, 0, 1, 1, 1, 0, 1									
0, 0, 1, 1, 0, 1, 1, 1, 0, 1									
1, 1, 0, 1, 0, 1, 0, 0, 1, 1									
0, 0, 0, 1, 1, 0, 0, 0, 0, 0									
0, 0, 1, 1, 0, 1, 0, 0, 1, 1									
1, 1, 0, 2, 1, 1, 0, 0, 1, 1									

EXAMPLE #3: $P6 = P2' - P1'$

$$\begin{aligned} P2' &= \{1, 1, 0, 2, 1, 1, 1, 1, 0, 0\} \\ - P1' &= \{1, 1, 0, 1, 0, 1, 1, 1, 0, 0\} \\ = P6 &= \{0, 0, 0, 1, 1, 0, 0, 0, 0, 0\} \end{aligned}$$

EXAMPLE #4: $P7 = P3' + P4' - P1'$

$$\begin{aligned} P3' &= \{0, 0, 1, 1, 0, 1, 1, 1, 0, 1\} \\ + P4' &= \{0, 0, 1, 1, 0, 1, 1, 1, 0, 1\} \\ - P1' &= \{1, 1, 0, 1, 0, 1, 1, 1, 0, 0\} \\ = P7 &= \{0, 0, 1, 1, 0, 1, 0, 0, 1, 1\} \end{aligned}$$

EXAMPLE #5: $P8 = P2' - P1' + P4'$

$$\begin{aligned} P2' &= \{1, 1, 0, 2, 1, 1, 1, 1, 0, 0\} \\ - P1' &= \{1, 1, 0, 1, 0, 1, 1, 1, 0, 0\} \\ + P4' &= \{0, 0, 1, 1, 0, 1, 1, 1, 0, 1\} \\ = P8 &= \{1, 1, 0, 2, 1, 1, 0, 0, 1, 1\} \end{aligned}$$

Q: Note that $P2' = P1' + P6$, so why not use $P1'$ and $P6$ instead of $P2'$?

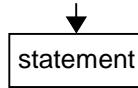
A: Because $P6$ is not a “complete path”, so it cannot be a linearly independent path

Unit Testing: Path Coverage

- Finds the number of distinct paths through the program to be traversed at least once
- Minimum number of tests necessary to cover all edges is equal to the **number of independent paths** through the control-flow graph
- (Recall the lecture on Unit Testing)

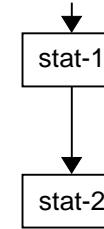
Issues (1)

Single statement:



= CC =

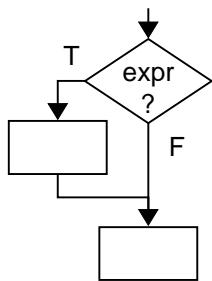
Two (or more) statements:



Cyclomatic complexity (CC) remains the same for a linear sequence of statements regardless of the sequence length
—insensitive to complexity contributed by the multitude of statements
(Recall that discriminative power (sensitivity) is a desirable property of a metric)

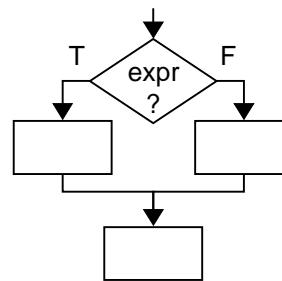
Issues (2)

Optional action:



= CC =

Alternative choices:

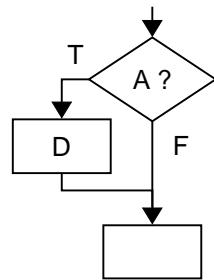


Optional action versus alternative choices –
the latter is psychologically more difficult

Issues (3)

Simple condition:

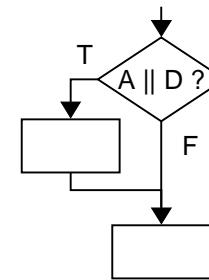
if (A) then D;



= CC =

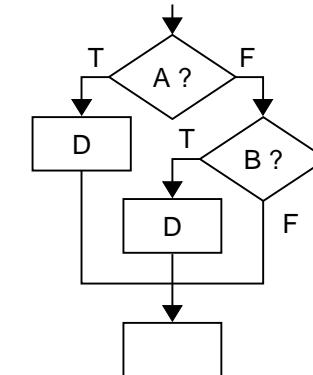
Compound condition:

if (A OR B) then D;



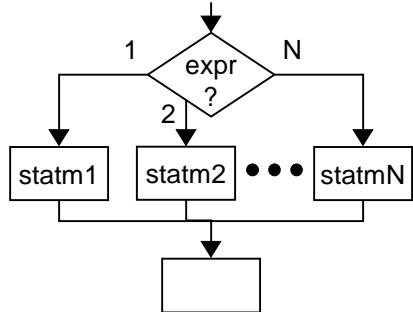
BUT, compound condition can be written
as a nested IF:

if (A) then D;
else if (B) then D;



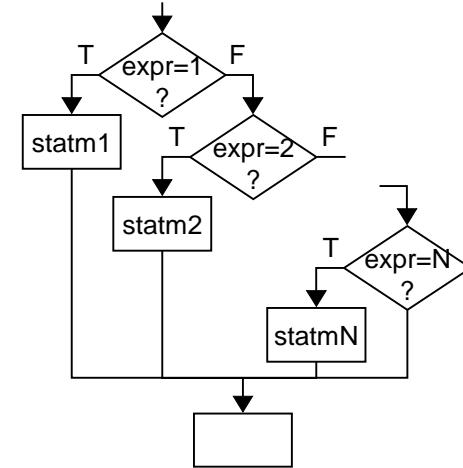
Issues (4)

Switch/Case statement:



= CC =

N-1 predicates:



Counting a switch statement:

—as a single decision

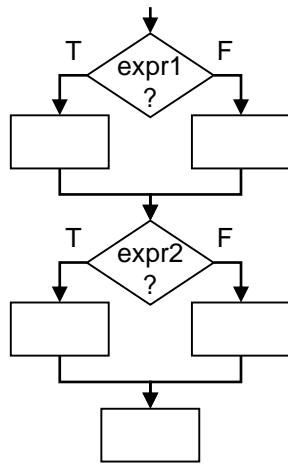
proposed by W. J. Hansen, "Measurement of program complexity by the pair (cyclomatic number, operator count)," *SIGPLAN Notices*, vol.13, no.3, pp.29-33, March 1978.

—as $\log_2(N)$ relationship

proposed by V. Basili and R. Reiter, "Evaluating automatable measures for software development," *Proceedings of the IEEE Workshop on Quantitative Software Models for Reliability, Complexity and Cost*, pp.107-116, October 1979.

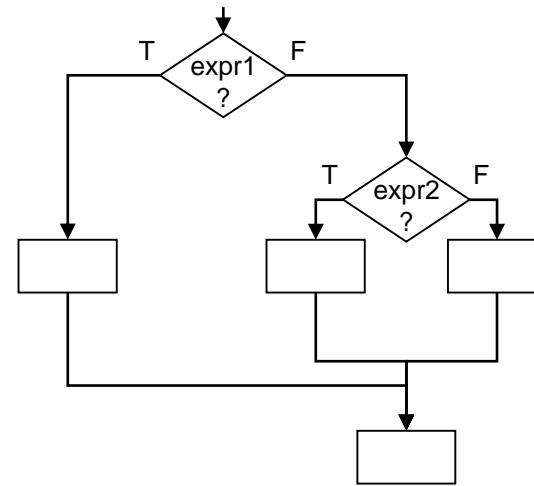
Issues (5)

Two sequential decisions:



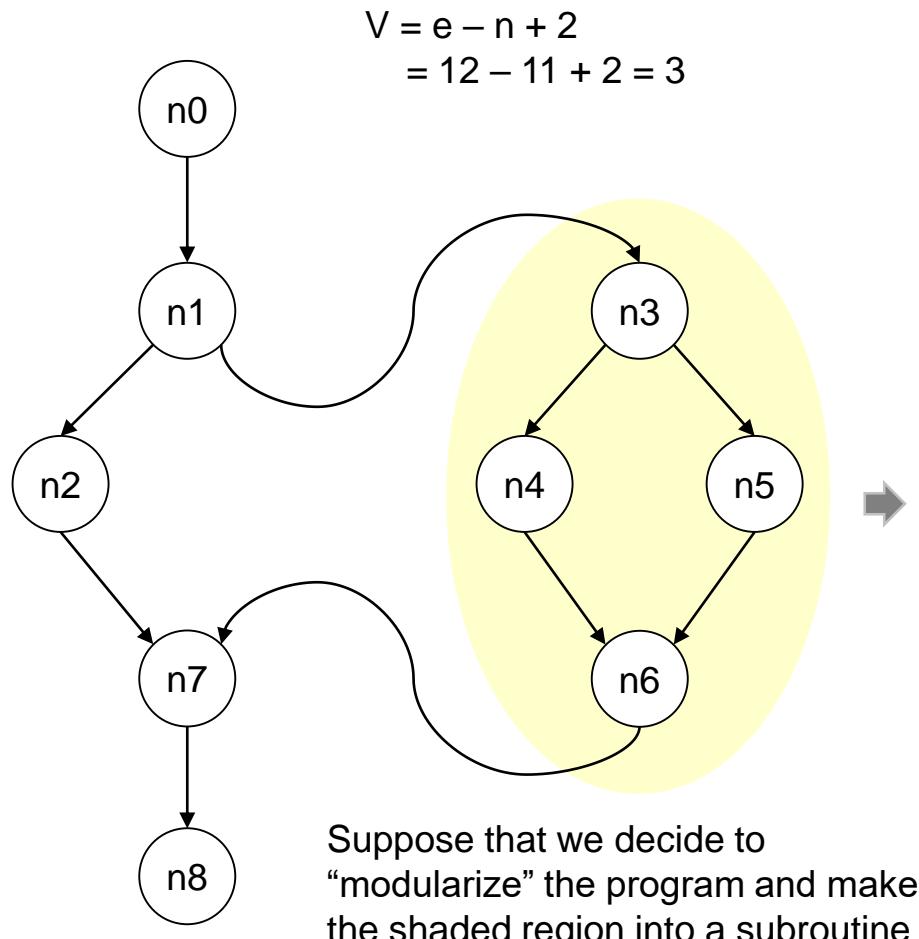
= CC =

Two nested decisions:

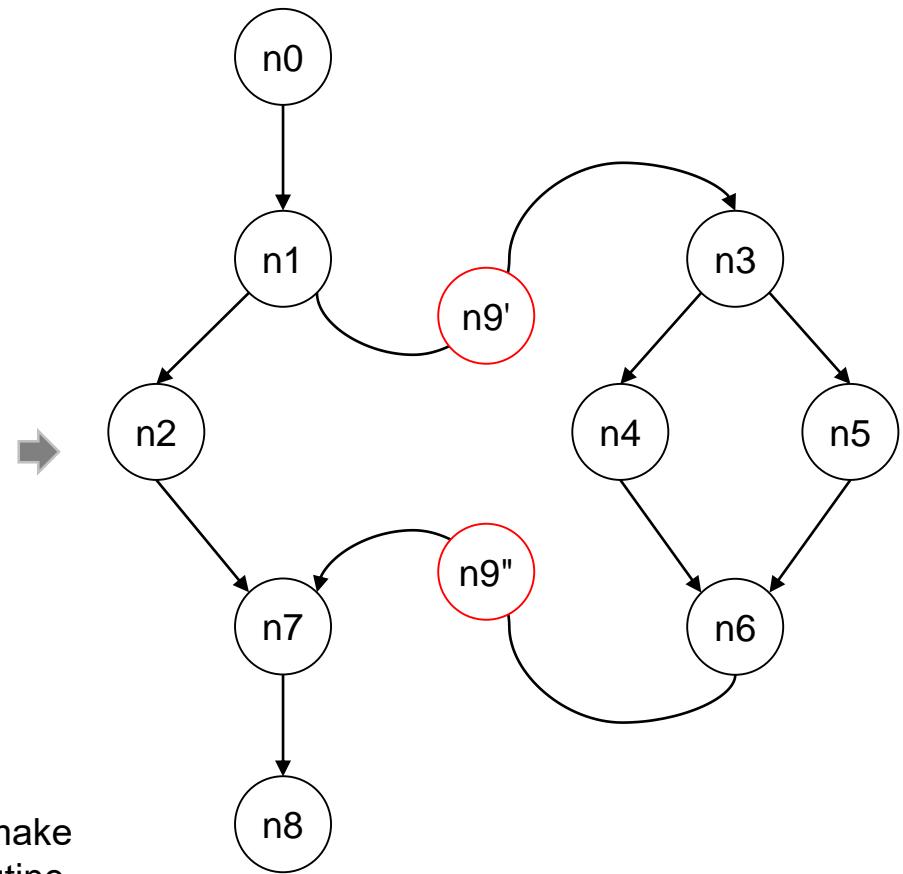


But, it is known that people find nested decisions more difficult ...

CC for Modular Programs (1)

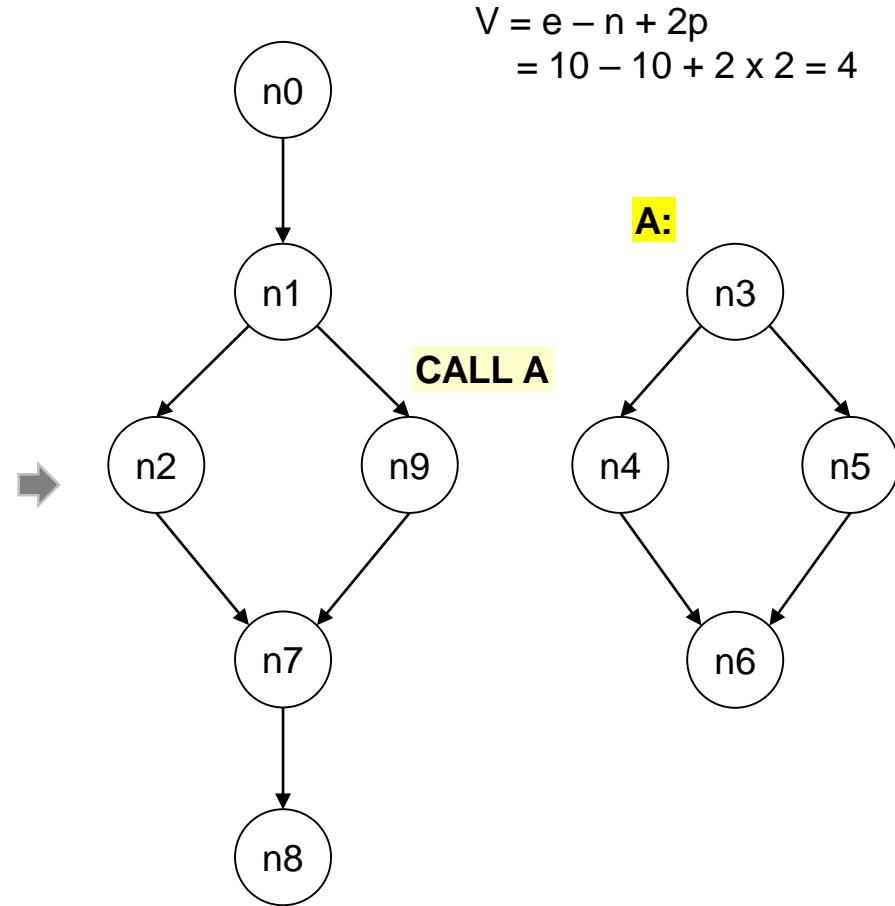
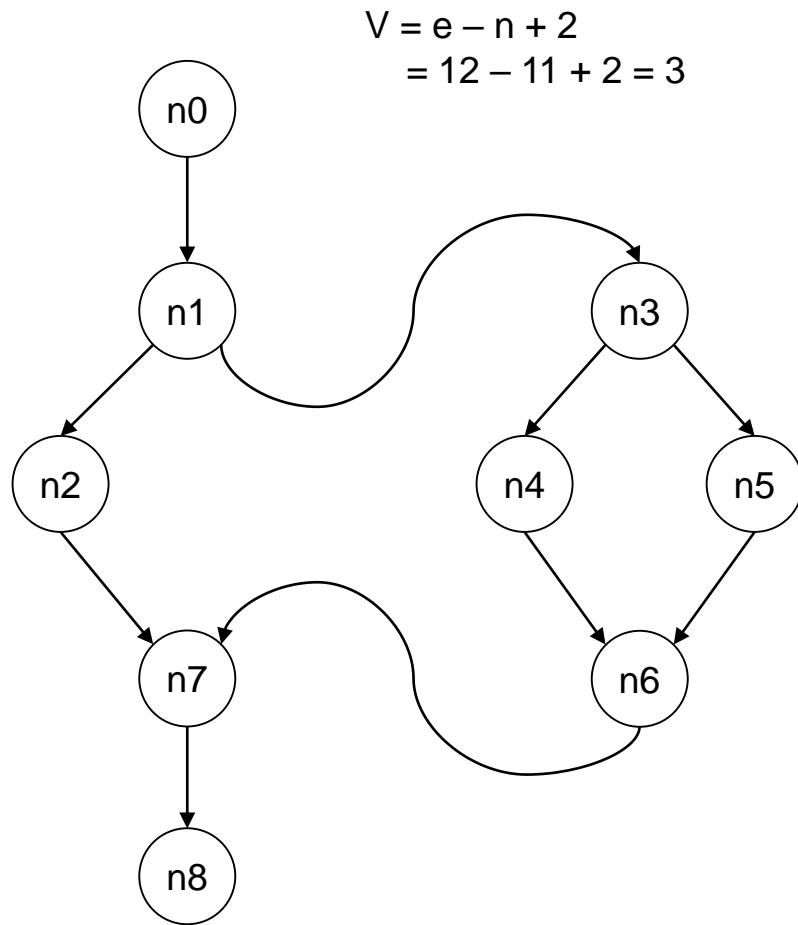


Adding a sequential node does not change CC:



CC for Modular Programs (2)

Intuitive expectation:
Modularization should not increase complexity

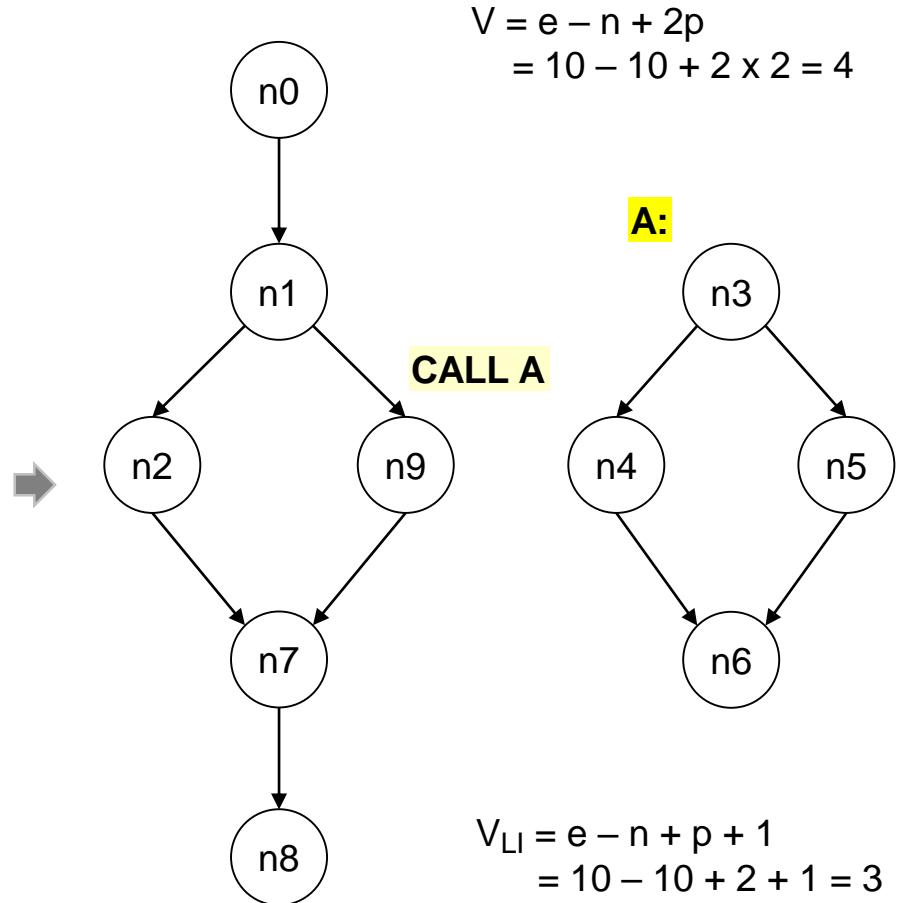
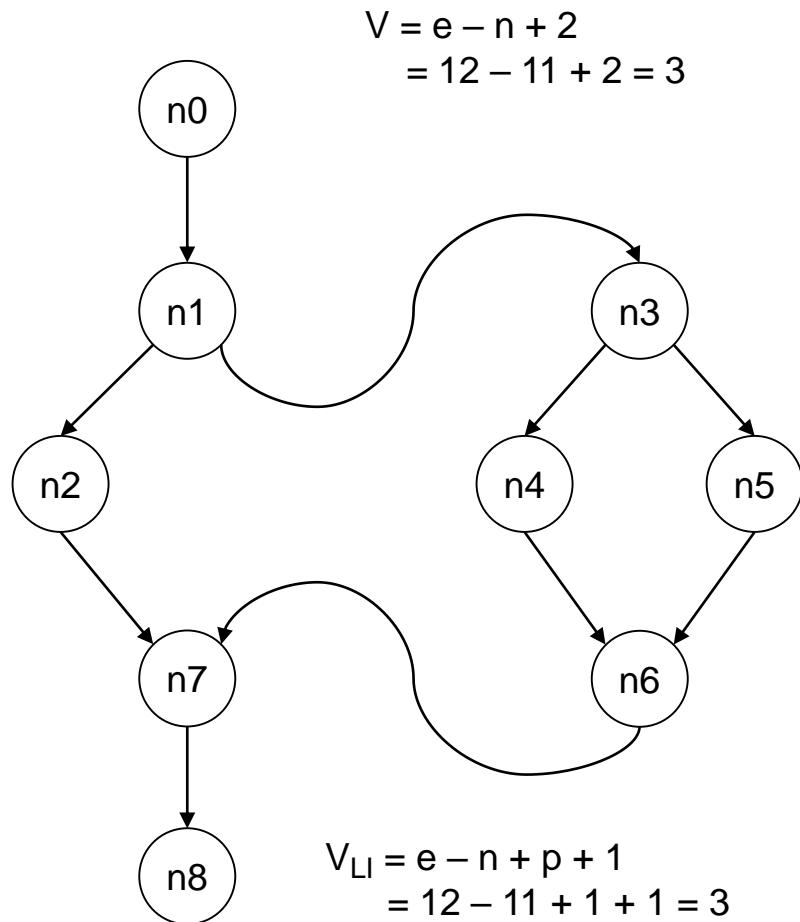


Modified CC Measures

- Given p connected components of a graph:
 - $V(G) = e - n + 2p$ (1)
 - $V_{LI}(G) = e - n + p + 1$ (2)
 - Eq. (2) is known as *linearly-independent* cyclomatic complexity
 - V_{LI} does not change when program is modularized into p modules

CC for Modular Programs (3)

Intuitive expectation:
Modularization should not increase complexity



Practical SW Quality Issues (1)

- No program module should exceed a cyclomatic complexity of 10
 - Originally suggested by McCabe
 - P. Jorgensen, *Software Testing: A Craftman's Approach, 2nd Edition*, CRC Press Inc., pp.137-156, 2002 .
- Software refactorings are aimed at reducing the complexity of a program's conditional logic

- ◆ *Refactoring: Improving the Design of Existing Code*
by Martin Fowler, et al.; Addison-Wesley Professional, 1999.
- ◆ *Effective Java (2nd Edition)*
by Joshua Bloch; Addison-Wesley, 2008.

Practical SW Quality Issues (2)

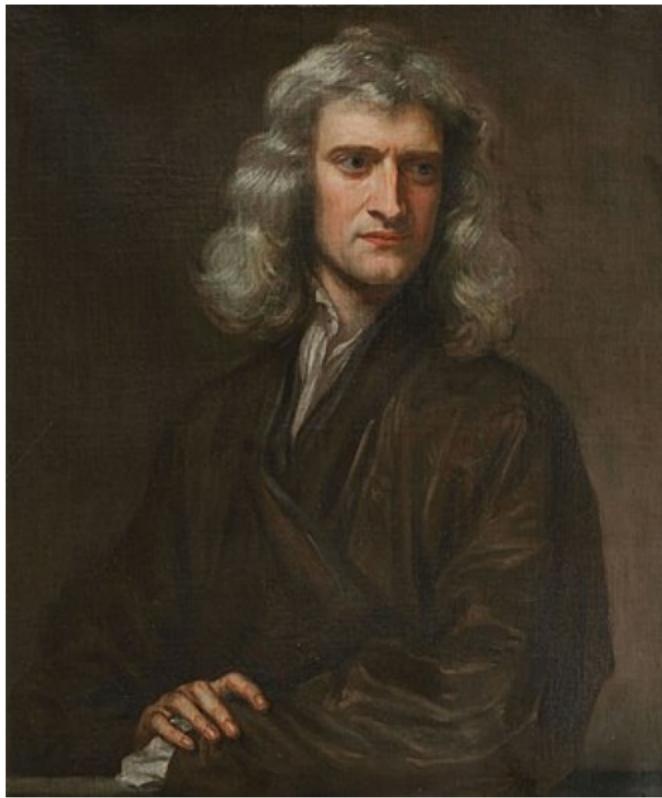
- Cyclomatic complexity is a screening method, to check for potentially problematic code.
- As any screening method, it may turn false positives and false negatives
- Will learn about more screening methods (cohesion, coupling, ...)

Software Engineering Shoulders of Giants

Omkarendra Tiwari

**Department of IT
IIIT Allahabad**

February 2, 2023



If I have seen further it
is by standing on the
shoulders of Giants.

Outline

- Giants
- Design and Architecture

Part I: Giants

Programming



The art of programming is the art of organizing complexity.

Software Engineering



Apollo 11

Moon landing

almost did not
happen.

Test Driven Development



Listening, Testing,
Coding, Designing.

Thats all there is to
software.

*I am not a great programmer. I'm just a good
programmer with great habits.*

Clean code



Would you rather
Test-first,
or debug-Later.

Good Practices in software development



Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Version Control System



...there's almost always a range of responses, and

"it depends" is almost always the right answer in any big question.

A picture worth thousand words: Giants behind UML



Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change. —Grady Booch

The name of a class should reflect its intrinsic nature and not a role that it plays in an association. For example, Owner would be a poor name for a class in a car manufacturer's database. What if a list of drivers is added later? What about persons who lease cars? The proper class is Person (or possibly Customer), which assumes various different roles, such as owner, driver, and lessee.—James Rumbaugh

When a user uses the system, she or he will perform a behaviorally related sequence of transactions in a dialogue with the system. We call such a special sequence a use case.—Ivar Jacobson

Part II: Design and Architecture

Journey Ahead

Analysis

Design

Development

Maintenance and Evolution

Analysis: An Overview

Representing elicited requirements

- Text or diagrams are used
- Easier to communicate
- Quick to review for correctness, completeness and consistency

Analysis approaches

- Structured Analysis
 - ▶ Data and process transforming data are considered as two entities
- Object-oriented Analysis
 - ▶ Classes (objects) and their association/collaboration is modeled

Object-oriented Analysis

Diagrams

- Activity diagrams
- Use-case diagrams
- Class diagrams
- CRC model

Software Elements

- Statement
- Block
- Method
- Class
- Package
- Component
- Library/executables

Design

Plan, organize or structure

- Every software has a design
- Each design offers certain qualities (desirable or not)
- Design decisions are made throughout the SDLC
- A software is acceptable only if it satisfies user requirements

Requirements and Design

- **User Requirements** Objectives/tasks that the software should achieve
- **Functional Requirements** Functionalities/services a software system offers
- **Non-functional Requirements** How well the software accomplishes the task
- **Requirements** Elicitate it. Validate it. Change still arrives.
- **Design** Some decisions are taken early, some are delayed. Trade-offs are involved. You can not have all good qualities.

Requirements Dictate Design

Bicycle: Handlebar, suspension, tires

- Thick tire; Straight Handles; Front suspension for shock absorption;^a
- Curve and drop handlebars; Light and aerodynamic alloy frames; ^b
- Straight handlebars and fat tire; ^c

^aMountain Bikes

^bRoad Bikes

^cSnow/Sand Bikes

Object-oriented Thinking

Class

- Encapsulation of Data and Operations
- Abstraction of a real world entity or a phenomenon
- Represents a unit/module/component/subsystem/system/service

Class Elements

- **Field** State of the object
- **Public Methods** Behavior of the object

Design Decisions at Class Level¹

Best Practices

- Keep the fields as Private
- Provide a constructor
- Initialize the fields via constructor
- Avoid getter/setter methods
- Five to seven public methods, not more
- Define *interface* and then *implement* in a class
- Check for anti-patterns and refactor the code

¹Find more in 'Further Readings' slide

Object Orientation

- Abstraction
- Encapsulation
- Composition
- Aggregation

- Class
- Instance
- Polymorphism
- Inheritance

Object-oriented Design Principle

SOLID

- Single Responsibility Principle
- Open-close Principle
- Liskov's Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Architectures and Requirements

Performance

Keep large components; avoid distributed over networks.

Security

Keep Layers. Add authentication mechanism.

Availability

Keep redundant components that can be replaced quickly.

Maintainability

Keep components smaller; separate data component with consumer components.

Further Readings

Highly Recommended

- <https://www.yegor256.com/2014/11/20/seven-virtues-of-good-object.html>

Software Engineering

Design: What, Why, and How

Omkarendra Tiwari

**Department of IT
IIIT Allahabad**

March 3, 2023

What and Why

- Organization of software system elements and their interaction/association/collaboration, to achieve
 - ▶ Reduced complexity in maintaining and evolving the system
 - ▶ Satisfying the non-functional requirements
 - ▶ Developing cost-effective solution

Design: Get it Right

Look for a suitable Pattern/Style

Monolithic

All components are packed and deployed together.

Microservices

Identify various services an application is providing; package, deploy, and evolve each service independent of the other.

N-tier Architecture

Divide application into components and organize their interaction in layered fashion to hide/secure crucial functionalities/data.

Client-Server

Server, a software designed to receive requests and provide services connects with other clients over the network.

Microkernel

An application with a core and extensible supporting functionalities.

Pipe-filter

An application designed to transform its input and feed forward to another filter for transformation.

Design: Not Quite Right

Code Smells/Anti-pattern

Code smells are design flaws that impact maintainability, evolvability, readability and other aspects of the code.

Large Class/God Class

A class with too many responsibilities.

Feature Envy

A method which is more interested in the data/members of other class.

Long Method/Brain Method

A method with more than one responsibility.

code clone

Duplicate logic/code present at multiple places.

Design : Make it right

Refactoring

External behavior preserving transformations

Extract Class

Divide the set of methods into two sets and extract one set along with fields into a new class.

Extract Method

Decompose the method into multiple smaller methods with cohesive method body.

Move Method

Move the envied method to a proper class.

Lets Design

Design at Method Level

```
void FiboPrime() {  
    int i, n, a, b, t;  
A.    printf("Value of N:");  
B.    scanf("%d",&n);  
C.    a = 0;  
D.    if (n ==1)  
E.        printf("Nth Term:%d",a);  
F.    else {  
G.        b = 1;  
H.        for (i=3;i <=n;i++) {  
I.            t = a + b;  
J.            a = b;  
K.            b = t;  
        }  
    }  
L.    printf("Nth Term:%d",b);  
M.    for (i=2; i<=b/2;i++){  
N.        if (b%i == 0)  
O.            break;  
    }  
P.    if (b<=1 || i <=b/2)  
Q.        printf("Not Prime");  
R.    else  
S.        printf("Prime");  
} Fibonacci Prime Code
```

Design at Method Level

```
void FiboPrime() {  
    int i, n, a, b, t;  
A.    printf("Value of N:");           - Get Input : {A,B}  
B.    scanf("%d",&n);  
C.    a = 0;  
D.    if (n ==1)  
E.        printf("Nth Term:%d",a);  
F.    else {  
G.        b = 1;  
H.        for (i=3;i <=n;i++){  
I.            t = a + b;           - Compute Nth Fibonacci Term : {C-K}  
J.            a = b;  
K.            b = t;  
    }  
L.    printf("Nth Term:%d",b);       - Print the Fibonacci Term : {L}  
M.    for (i=2; i<=b/2;i++){  
N.        if (b%i == 0)           - Has a divisor other than one and self : {M-O}  
O.            break;  
P.    if (b<=1 || i <=b/2)  
Q.        printf("Not Prime");  
R.    else  
S.        printf("Prime");       - Prime Checker : {P-S}  
} Fibonacci Prime Code
```

Design at Method Level

```
int fibonacci() {
    int n, a, b, t;
    A. printf("Value of N:");
    B. scanf("%d",&n);
    C. a = 0;
    D. if (n ==1)
        E.     printf("Nth Term:%d",a);
    F. else {
        G.     b = 1;
        H.     for (i=3;i <=n;i++) {
        I.         t = a + b;
        J.         a = b;
        K.         b = t;
    }
}
L. printf("Nth Term:%d",b);
X. return(b);
}
```

```
void isPrime(int b) {
    int i;
    M.    for (i=2; i<=b/2;i++){
    N.        if (b%i == 0)
    O.            break;
    }
    P.    if (b<=1 || i <=b/2)
    Q.        printf("Not Prime");
    R.    else
    S.        printf("Prime");
}
Decomposition #1
```

```
void fiboPrime() {
    int n;
    A.    printf("Value of N:");
    B.    scanf("%d",&n);
    X.    isPrime(fibonacci(n));
}
int fibonacci(int n) {
    int a, b, t;
    C.    a = 0;
    D.    if (n ==1)
        E.        printf("Nth Term:%d",a);
    F.    else {
        G.        b = 1;
        H.        for (i=3;i <=n;i++) {
        I.            t = a + b;
        J.            a = b;
        K.            b = t;
    }
}
L.    printf("Nth Term:%d",b);
X.    return(b);
}
```

```
void isPrime(int b) {
    int i;
    M.    for (i=2; i<=b/2;i++){
    N.        if (b%i == 0)
    O.            break;
    }
    P.    if (b<=1 || i <=b/2)
    Q.        printf("Not Prime");
    R.    else
    S.        printf("Prime");
}
Decomposition #2
```

Design at Class Level

Class version 1.0

```
public class Student {  
    String Name;  
    String EnrollmentNo;  
    Date dob;  
    int age;  
    String address;  
    String email;  
    long int mobile;  
    String semester;  
    String department;  
    String branch;  
    String compulsory_courses;  
    String elective_courses;  
    String blood_grp;  
    String father_name;  
    String mother_name;  
    long int bank_account;  
    long int aadhar;  
    float fee_due;  
    String library_card;  
    float gpa;  
}
```

Design at Class Level

Class version 1.0

```
public class Student {  
    String Name;  
    String EnrollmentNo;  
    Date dob;  
    int age;  
    String address;  
    String email;  
    long int mobile;  
    String semester;  
    String department;  
    String branch;  
    String compulsory_courses;  
    String elective_courses;  
    String blood_grp;  
    String father_name;  
    String mother_name;  
    long int bank_account;  
    long int aadhar;  
    float fee_due;  
    String library_card;  
    float gpa;  
}
```

Class version 1.1

```
public class Student {  
    String EnrollmentNo;  
    PersonalData personalDetails;  
    AcademicData academicDetails;  
    Responsibility responsibility;  
    FinancialData financialDetails;  
    Contact contact;  
}  
public class PersonalData {  
    String Name;  
    Date dob;  
    String blood_grp;  
    long int aadhar;  
}  
public class Contact {  
    String address;  
    String email;  
    long int mobile;  
    String father_name;  
    String mother_name;  
}  
public class FinancialData {  
    long int bank_account;  
    float fee_due;  
}  
public class AcademicData {  
    String department;  
    String branch;  
    String semester;  
    String compulsory_courses;  
    String elective_courses;  
    String library_card;  
    float gpa;  
}
```

Design at Class Level

Class version 1.0

```
public class Student {  
    String Name;  
    String EnrollmentNo;  
    Date dob;  
    int age;  
    String address;  
    String email;  
    long int mobile;  
    String semester;  
    String department;  
    String branch;  
    String compulsory_courses;  
    String elective_courses;  
    String blood_grp;  
    String father_name;  
    String mother_name;  
    long int bank_account;  
    long int aadhar;  
    float fee_due;  
    String library_card;  
    float gpa;  
}
```

Class version 1.1

```
public class Student {  
    String EnrollmentNo;  
    PersonalData personalDetails;  
    AcademicData academicDetails;  
    Responsibility responsibility;  
    FinancialData financialDetails;  
    Contact contact;  
}  
public class PersonalData {  
    String Name;  
    Date dob;  
    String blood_grp;  
    long int aadhar;  
}  
public class Contact {  
    String address;  
    String email;  
    long int mobile;  
    String father_name;  
    String mother_name;  
}  
public class FinancialData {  
    long int bank_account;  
    float fee_due;  
}  
public class AcademicData {  
    String department;  
    String branch;  
    String semester;  
    String compulsory_courses;  
    String elective_courses;  
    String library_card;  
    float gpa;  
}
```

Class version 1.2

```
public class TeachingStaff {  
    String EmpId;  
    PersonalData personalDetails;  
    AcademicData academicDetails;  
    Responsibility responsibility;  
    FinancialData financialDetails;  
    Contact contact;  
}
```

Class version 1.2

```
public class NonTeachingStaff {  
    String EmpId;  
    PersonalData personalDetails;  
    AcademicData academicDetails;  
    Responsibility responsibility;  
    FinancialData financialDetails;  
    Contact contact;  
}
```

Design at Class Level : Inheritance

Class version 1.3

```
public class Student extends  
InstituteMember {  
    String EnrollmentNo;  
    AcademicData academicDetails;  
}  
public class PersonalData {  
    String Name;  
    Date dob;  
    String blood.grp;  
    long int aadhar;  
}  
public class Contact {  
    Address address;  
    String email;  
    long int mobile;  
    Parents parent;  
}  
public class FinancialData {  
    long int bank.account;  
    float fee.due;  
}  
public class AcademicData {  
    String department;  
    String branch;  
    String semester;  
    String compulsory.courses;  
    String elective.courses;  
    String library.card;  
    float gpa;  
}
```

Design at Class Level : Inheritance

Class version 1.3

```
public class Student extends InstituteMember {  
    String EnrollmentNo;  
    AcademicData academicDetails;  
}  
  
public class PersonalData {  
    String Name;  
    Date dob;  
    String blood_grp;  
    long int aadhar;  
}  
  
public class Contact {  
    Address address;  
    String email;  
    long int mobile;  
    Parents parent;  
}  
  
public class FinancialData {  
    long int bank_account;  
    float fee_due;  
}  
  
public class AcademicData {  
    String department;  
    String branch;  
    String semester;  
    String compulsory_courses;  
    String elective_courses;  
    String library_card;  
    float gpa;  
}
```

Class version 1.3

```
public abstract class InstituteMember {  
    PersonalData personalDetails;  
    Responsibility responsibility;  
    FinancialData financialDetails;  
    Contact contact;  
}
```

Class version 1.3

```
public class TeachingStaff extends InstituteMember{  
    String EmpId;  
    AcademicData academicDetails;  
}
```

Class version 1.3

```
public class NonTeachingStaff  
extends InstituteMember{  
    String EmpId;  
    AcademicData academicDetails;  
}
```

Class version 1.3

```
public class Address {  
    String HouseNo;  
    String Street;  
    String district;  
    String state;  
    String country;  
    long int pin;  
}
```

Further Increments

- AcademicData (history, current, grade-generation, courses)
- Library
- Gymkhana (Sports, Swimming Pool, Gym, Cultural)
- Projects (Social or Financial)
- Department

Caution with Inheritance

Substitutability

- Use *Inheritance* only when substitutability is required
- A derived class can not demand more resources and can not offer fewer services
- A user of base class should be able to use the derived class without knowing the difference
- If the criteria does not meet use *composition/delegation* instead

Class Diagram: Communicating Design and Architecture

Shapes and association

- Class: A rectangle with three parts
- Association: A straight line connecting two elements
 - ▶ Aggregation: An arrow with a hollow diamond head
 - ▶ Composition: An arrow with a filled diamond head
- Inheritance: An arrow with a triangle head
- Access Specifiers: + (public), # (protected), -(private)

Design Principles

Basic Principles

- YGNI: You Aren't Gonna Need It
- DRY: Don't Repeat Yourself

SOLID Principles

- SRP: Single Responsibility Principle
- OCP: Open-close Principle—Open for Extension but closed for modification
- LSP: Liskov's Substitution Principle—Inheritance should be used only for substitutability
- ISP: Interface Segregation Principle
- DIP: Dependency Inversion Principle

Further Readings

Highly Recommended

- <https://www.yegor256.com/2014/11/20/seven-virtues-of-good-object.html>
- <https://martinfowler.com/architecture/>
- <https://microservices.io/patterns/monolithic.html>



The Background of Software Testing

Please Answer Yes or No



- Q1. Purpose of Testing is to detect all defects and show correctness of software
- Q2. Test with all combination of input and preconditions should be conducted
- Q3. It is easy and effective to start testing after programming.
- Q4. Some of module/ software have more bugs than others
- Q5. Same test set should be used again and again in a project.
- Q6. Result of testing must be same in any time.
- Q7. After detecting all bugs and fixing them, user will accept software and system.

The Background of Software Testing



Q1. Purpose of Testing is to detect all defects and show correctness of software

- **Principle 1 – Testing shows presence of defects**
 - Testing can show that defects are present, but cannot prove that there are no defects.
 - *Testing reduces the probability of undiscovered defects remaining in the software*

The Background of Software Testing



Q2. Test for all combination of input and preconditions should be conducted

- **Principle 2 – Exhaustive testing is impossible**
 - Testing everything (all combinations of inputs and preconditions) is not feasible .
 - Tester should design appropriate test case to detect many bugs.
 - Risk analysis and priorities should be used to focus testing efforts.

The Background of Software Testing



Q3. It is easy and effective to start testing after programming.

- **Principle 3 – Early testing**

- Testing activities should start as early as possible in the software or system development life cycle.
- Detecting earlier , Fixing with lower cost.
- Test should start in user requirement phase.

The Background of Software Testing



Q4. Some of module/ software have more bugs than other ones

- **Principle 4 – Defect clustering**

- A small number of modules contain most of the defects discovered during pre-release testing.
- Tester should recognize what parts are weak.

Software process models are an ideal ... not reality



- No software development effort follows a process model perfectly. Why?
 - The specification never corresponds to the customers needs perfectly.
 - There is never enough time to perform all of the testing.
- Nevertheless, an ideal model is helpful to make progress.
 - Trade offs and concessions are unavoidable.

The Background of Software Testing



Q5. Same test set should be used again and again in a project.

- **Principle 5 – Pesticide paradox**

- If the same tests are repeated over and over again, eventually the same set of test cases will no longer find any new defects.
- To overcome this “pesticide paradox”, the test cases need to be regularly reviewed and revised

The Background of Software Testing



Q6. Result of testing must be same any time.

- **Principle 6 – Testing is context dependent**
 - Testing is done differently in different contexts.
 - Test should be done in all possible situations

The Background of Software Testing



Q7. After detecting all bugs and fixing them, user will accept software and system.

- **Principle 7 – Absence-of-errors fallacy**
 - Finding and fixing defects does not help if the system built is unusable and does not fulfill the users' needs and expectations.
 - So Verification & Validation is required.

Software testing axioms



What is an axiom anyway?



- An **axiom** is a sentence that is not proved or demonstrated and is considered as obvious or as an initial necessary consensus for a theory building.
- Therefore, it is taken for granted as true, and serves as a starting point for deducing and inferring other (theory dependent) truths.

Axiom 1



It is impossible to test a program completely

Axiom 1



It is impossible to test a program completely

- How many test cases do you need to exhaustively test:
 - Power point
 - A calculator
 - MS Word
 - Any interesting software!
- The only way to be absolutely sure software works is to run it against all possible inputs and observe all of its outputs ...
- Oh, and the specification must be correct and complete.

Axiom 1 (cont'd)



It is impossible to test a program completely

- The number of possible inputs is very large.
- The number of possible outputs is very large.
- The number of paths through the software is very large.
- The software specification open to interpretation.

Axiom 2

Software testing is a risk-based exercise



Axiom 2



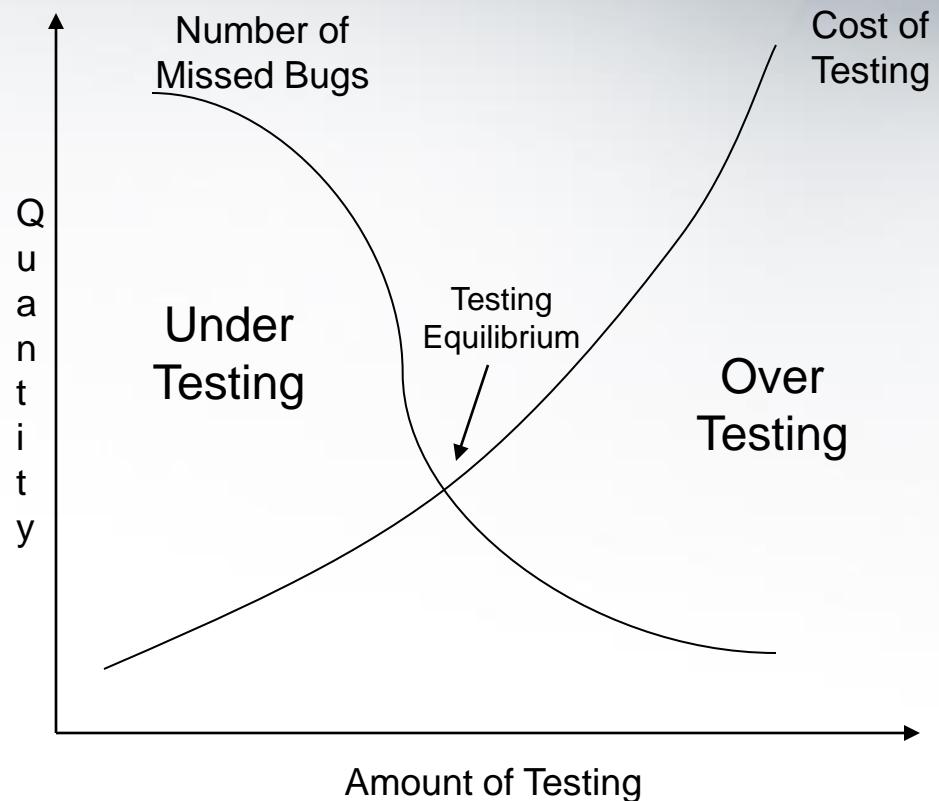
Software testing is a risk-based exercise

- If you do not test the software for all inputs (a wise choice) you take a risk.
- Hopefully you will skip a lot of inputs that work correctly.
- What if you skip inputs that cause a fault?
 - Risk: financial loss, security, loss of money, loss of life!
 - That is a lot of pressure for a tester!
- This course is all about techniques and practices to help reduce the risk without breaking the bank.

Axiom 2 (cont'd)

Software testing is a risk-based exercise

- If you try to test too much, the development cost becomes prohibitive.
- If you test too little, the probability of software failure increases and as we discussed ... software failures can cost us big time!



Axiom 2 (cont'd)

Software testing is a risk-based exercise



What about Murphy's Law?

- *"If there's more than one possible outcome of a job or task, and one of those outcomes will result in disaster or an undesirable consequence, then somebody will do it that way."*

Axiom 3

- **Testing cannot show the absence of bugs**



Axiom 3



- **Testing cannot show the absence of bugs**
- “*Program testing can be used to show the presence of bugs, but never to show their absence!*”
- - *Edsger Wybe Dijkstra*

Axiom 4



- **The more bugs you find, the more bugs there are**

Axiom 4



- **The more bugs you find, the more bugs there are**
- Bugs appear in groups, where you see one you will likely find more ... Why?
 - Programmers can have bad days
 - Programmers tend to make the same mistakes
 - Some bugs are just the tip of the iceberg.

Axiom 5

- Not all bugs found will be fixed



Axiom 5

Not all bugs found will be fixed



- Why wouldn't you fix a bug you knew about?
 - There's not enough time
 - Some deadlines cannot be extended (e.g., Y2K)
 - It's not really a bug
 - Specifications can be wrong
 - It's too risky to fix
 - "I'm not touching Murphy's code!"
 - It's just not worth it
 - Bugs in fringe features may have to wait
 - Why not charge the customer for bug fixes in the next release (sound familiar?) :-)

Axiom 6



- It is difficult to say when a bug is indeed a bug

Axiom 6



- **It is difficult to say when a bug is indeed a bug**
- If there is a problem in the software but no one ever discovers it ... is it a bug?
- What is your opinion? Does a bug have to be observable in order for it to be a bug?
- Bugs that are undiscovered are called *latent bugs*.

Axiom 7

- **Specifications are never final**





Axiom 7

Specifications are never final

- Building a product based on a “moving target” specification is fairly unique to software development.
 - Competition is aggressive
 - Very rapid release cycles
 - Software is “easy” to change
- Not true in other engineering domains
 - E.g., the Brooklyn Bridge could not be adjusted to allow train traffic to cross it once its construction started.

A Story ...



Dear Mr. Architect,
Please design and build me a house. I am not quite sure of
what I need, so you should use your discretion.

My house should have between two and forty-five bedrooms.
Just make sure the plans are such that bedrooms can be easily
added or deleted.

When you bring the blueprints to me, I will make the final
decision of what I want. Also bring me the cost breakdown
for each configuration so that I can arbitrarily pick one.

A Story ... (Cont'd)



Keep in mind that the house I ultimately chose must cost less than the one I am currently living in. Make sure, however, that you correct all the deficiencies that currently exist in my house (the floor of my kitchen vibrates when I walk across it, and the walls don't have nearly enough insulation in them).

Also keep in mind as you design this house that I wish to keep yearly maintenance cost as low as possible. This should mean the incorporation of extra-cost features like aluminum or vinyl siding. If you chose not to specify aluminum, be prepared to explain in detail.

A Story ... (Cont'd)



Please take care that modern design practices and the latest materials are used in construction of the house. The house should be really nice. However, be alerted that the kitchen should be designed to accommodate among other things, my 1952 Gibson refrigerator.

To assure that you are building the correct house for our family, make sure that you contact each of the children and also the in-laws. My mother-in-law will have very strong feelings about how the house ought to be designed since she visits with us at least once a year. Make sure that you weigh all these options carefully and make the right decision.

I, however, retain the right to override any decision you come up with.

A Story ... (Cont'd)



Please don't bother me with small details right now. Your job is to develop the overall plans for this house. Get the big picture. It is not appropriate at this time to be choosing the color of the carpet. However, keep in mind that my wife likes green.

Also do not worry at this time about acquiring resources to build this house. Your first priority is to develop detailed plans and specifications. However, once I accept these plans, I will expect to have the house under roof within 48 hours.

A Story ... (Cont'd)



While you are designing this house specifically for me, keep in mind that sooner or later I will have to sell this house. It should have appeal to potential buyers. Please make sure that before you finalize the plans, there is a consensus of the population in my area that they like the features this house has.

You are advised to run up and look at my neighbor's house he had constructed last year. We like it a great deal. It has many features that we would like to have in our new home, particularly the 75-foot swimming pool. With careful engineering I believe that you can design this into our new house without impacting the construction cost.

A Story ... (Cont'd)



Please prepare a complete set of blueprints. It is not necessary at this time to do the real design since these blueprints will be used only for construction bids. Please be advised however, that any increase of cost in the future as a result of design changes will result in you getting your hands slapped.

You must be thrilled to be working on such an interesting project such as this. To be able to use new kinds of construction and to be given such freedom in your designs is something that doesn't happen very often. Contact me as rapidly as possible with your design ideas. I am enthusiastic about seeing what you can come up with.

A Story ... (Cont'd)



P.S. My wife has just told me that she disagrees with many on the instructions I've given you in this letter. As architect it is your responsibility to resolve these issues. I have tried in the past and have been unable to accomplish this. If you can't handle this, I'll have to look for a new architect.

Axiom 8



- **Software testers are not the most popular members of a project**

Axiom 8

Software testers are not the most popular members of a project

- Goal of a software tester:
 - Find bugs
 - Find bugs early
 - Make sure bugs get fixed
- Tips to avoid becoming unpopular:
 - Find bugs early
 - Temper your enthusiasm ... act in a professional manner
 - Don't report just the bad news



Axiom 9



- **Software testing is a disciplined and technical profession**

Axiom 9



- **Software testing is a disciplined and technical profession**
- When software was simpler and more manageable software testers were often untrained and testing was not done methodically.
- It is now too costly to build buggy software. As a result testing has matured as a discipline.
 - Sophisticated techniques
 - Tool support
 - Rewarding careers

Software testing axioms



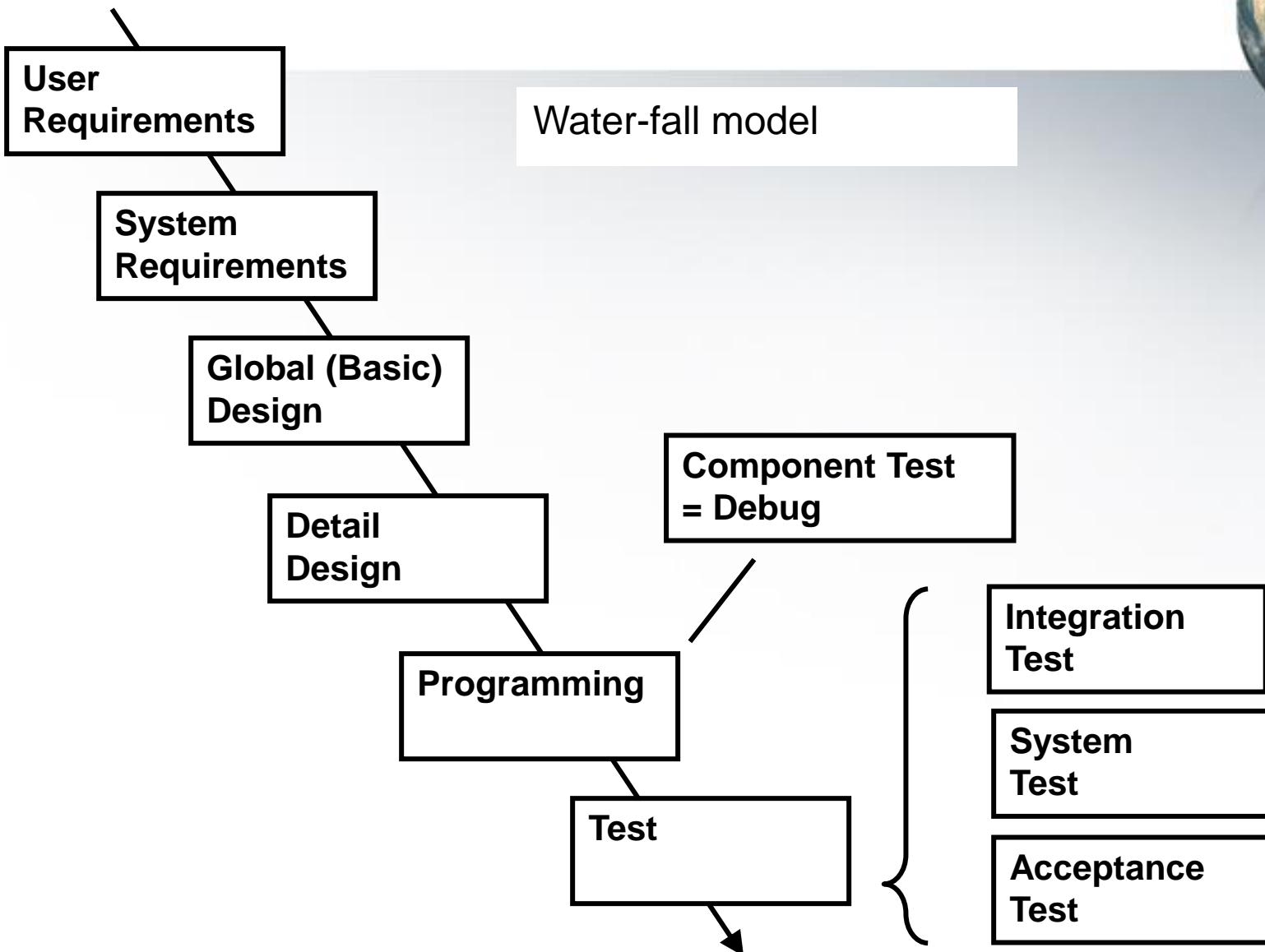
1. It is impossible to test a program completely.
2. Software testing is a risk-based exercise.
3. Testing cannot show the absence of bugs.
4. The more bugs you find, the more bugs there are.
5. Not all bugs found will be fixed.
6. It is difficult to say when a bug is indeed a bug.
7. Specifications are never final.
8. Software testers are not the most popular members of a project.
9. Software testing is a disciplined and technical profession.

Testing Cycle

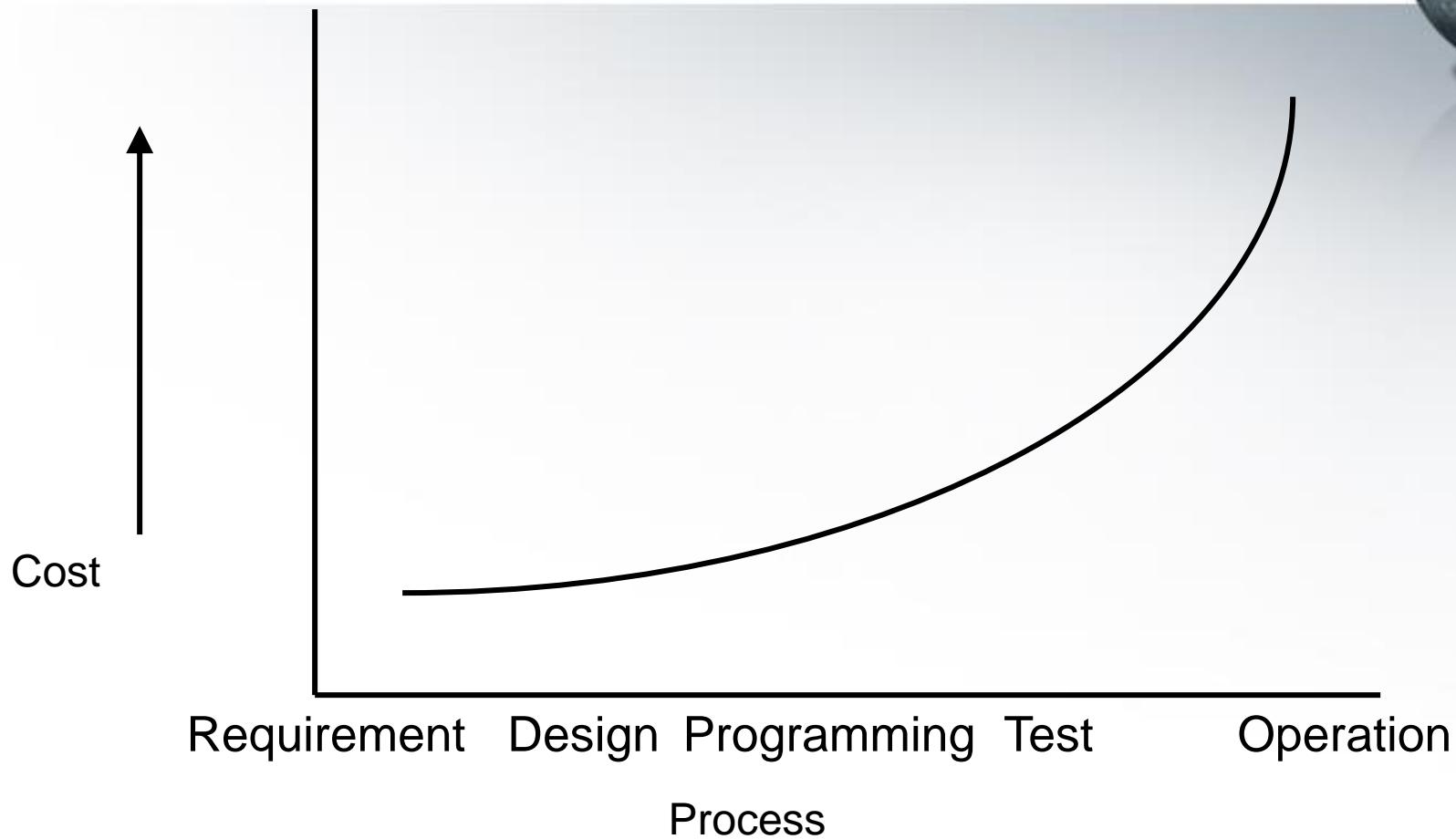


- When the testing starts
- How it should start

Old view of Testing



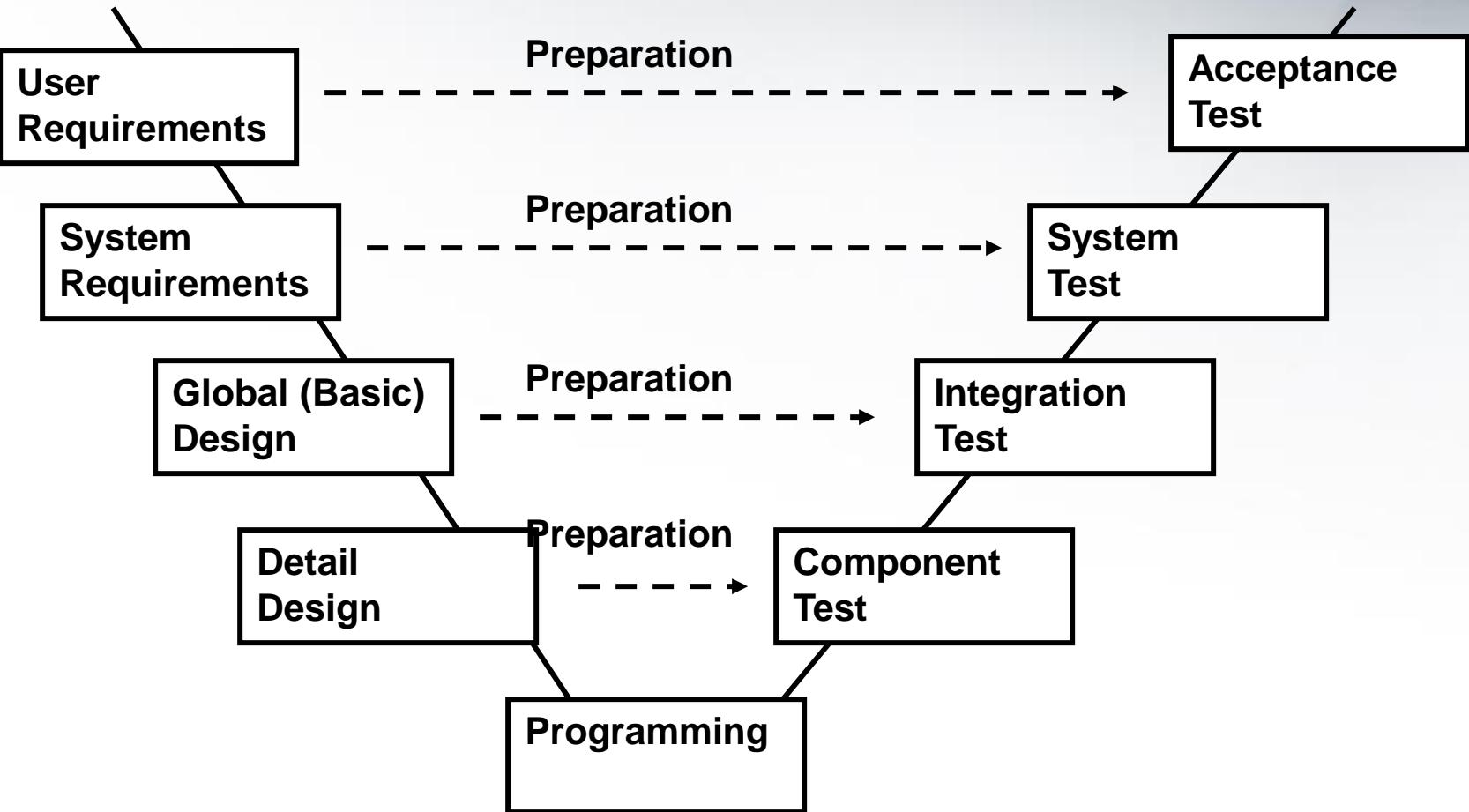
Reason 1: Cost of Fixing bugs



Principle 3 – Early testing

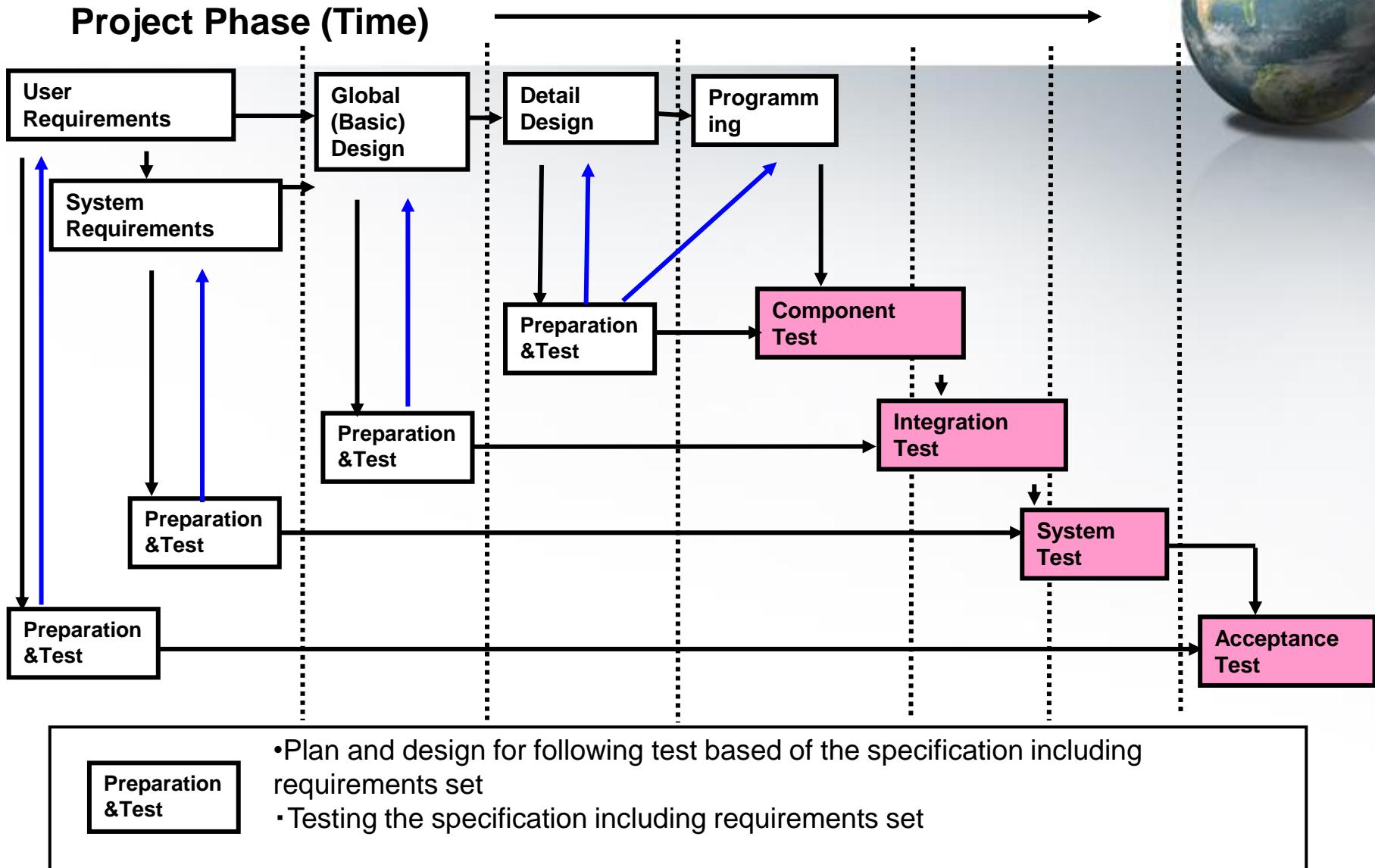


New view of Testing V-model

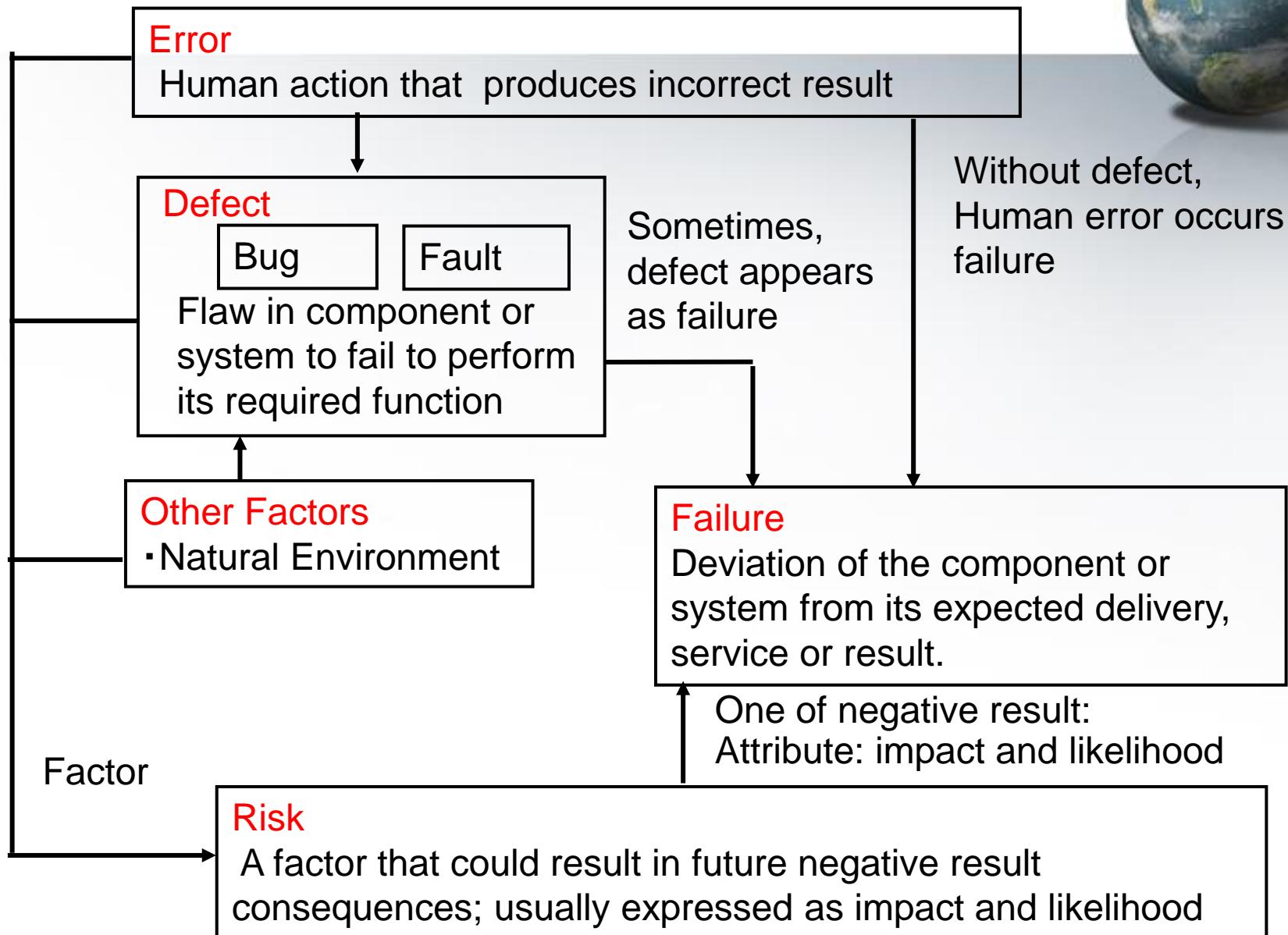


V-model (sequential development model)

Real Time line of V-model



Definition of basic terms related bug, error,

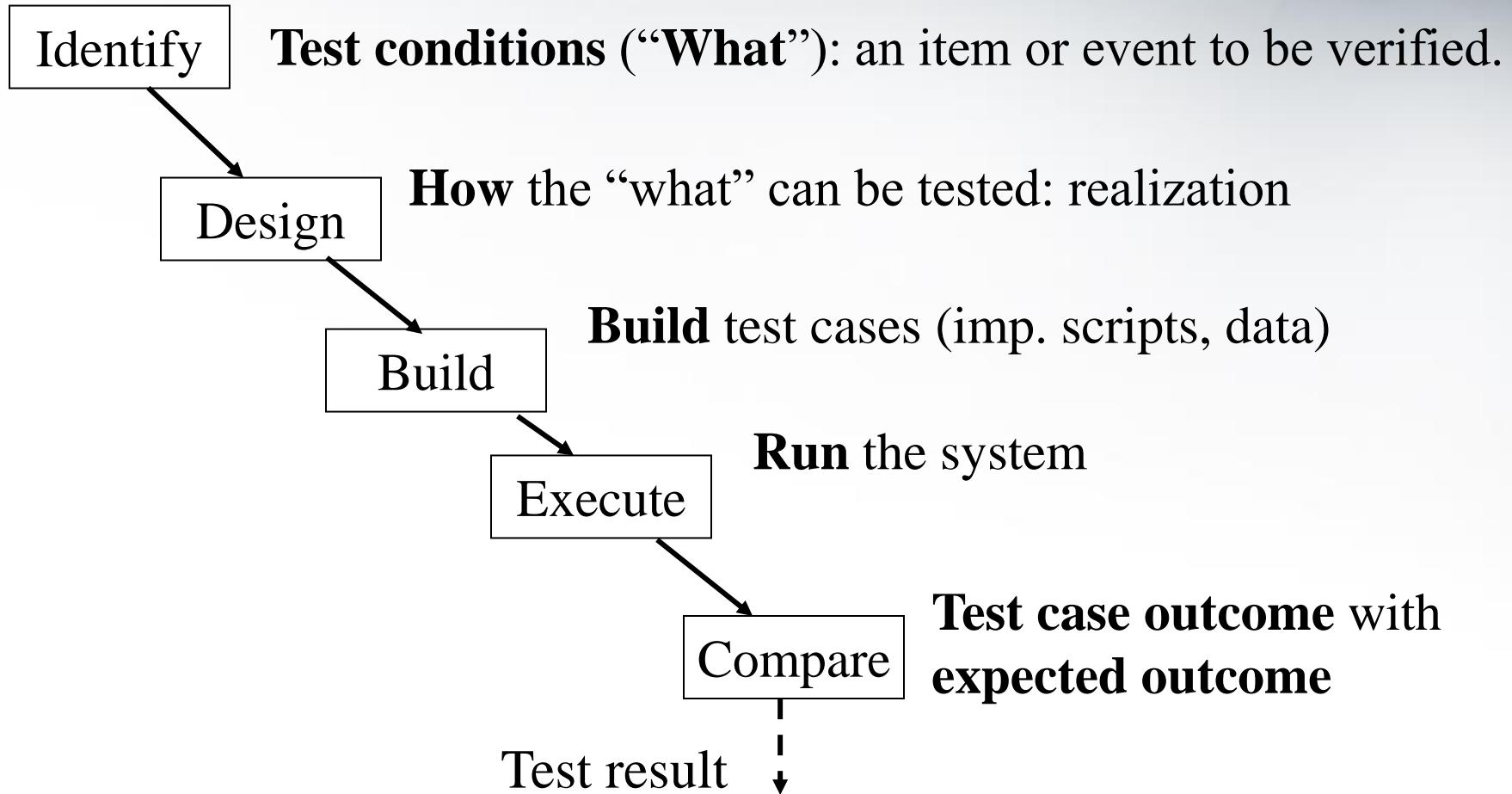


Testing Cycle

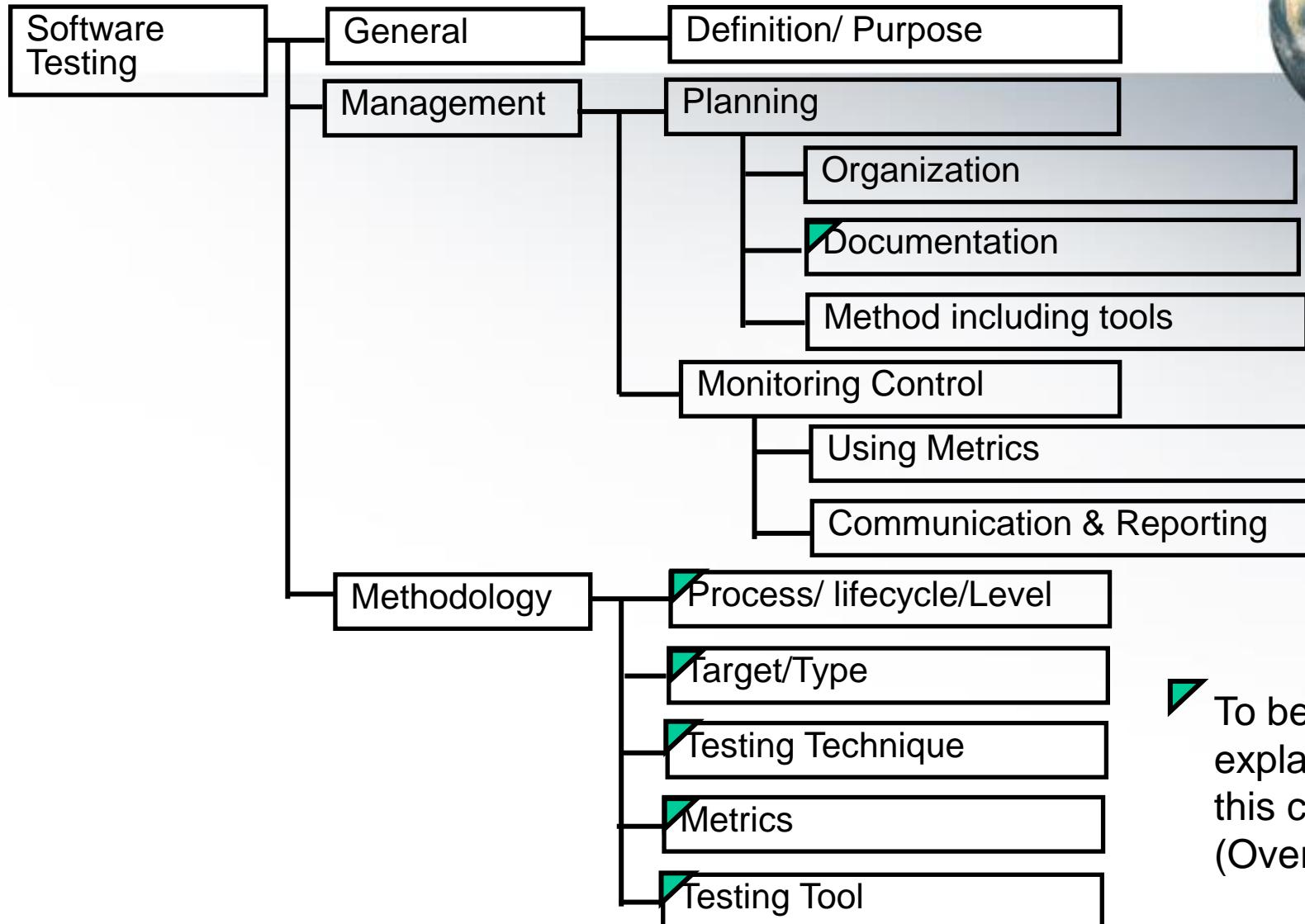


- Although testing varies between organizations, there is a cycle to testing:
 - Requirements Analysis: Testing should begin in the requirements phase of the software life cycle (SDLC).
 - Design Analysis: During the design phase, testers work with designers in determining what aspects of a design are testable and under what parameter those testers work.
 - Test Planning: Test Strategy, Test Plan(s), Test Bed creation.
 - Test Development: Test Procedures, Test Scenarios, Test Cases, Test Scripts to use in testing software.
 - Test Execution: Testers execute the software based on the plans and report any errors found to the development team.
 - Test Reporting: Once testing is completed, testers generate metrics and make final reports on their test effort and whether or not the software tested is ready for release.
 - Retesting the Defects

Testing Activities



Attribute of Software Testing

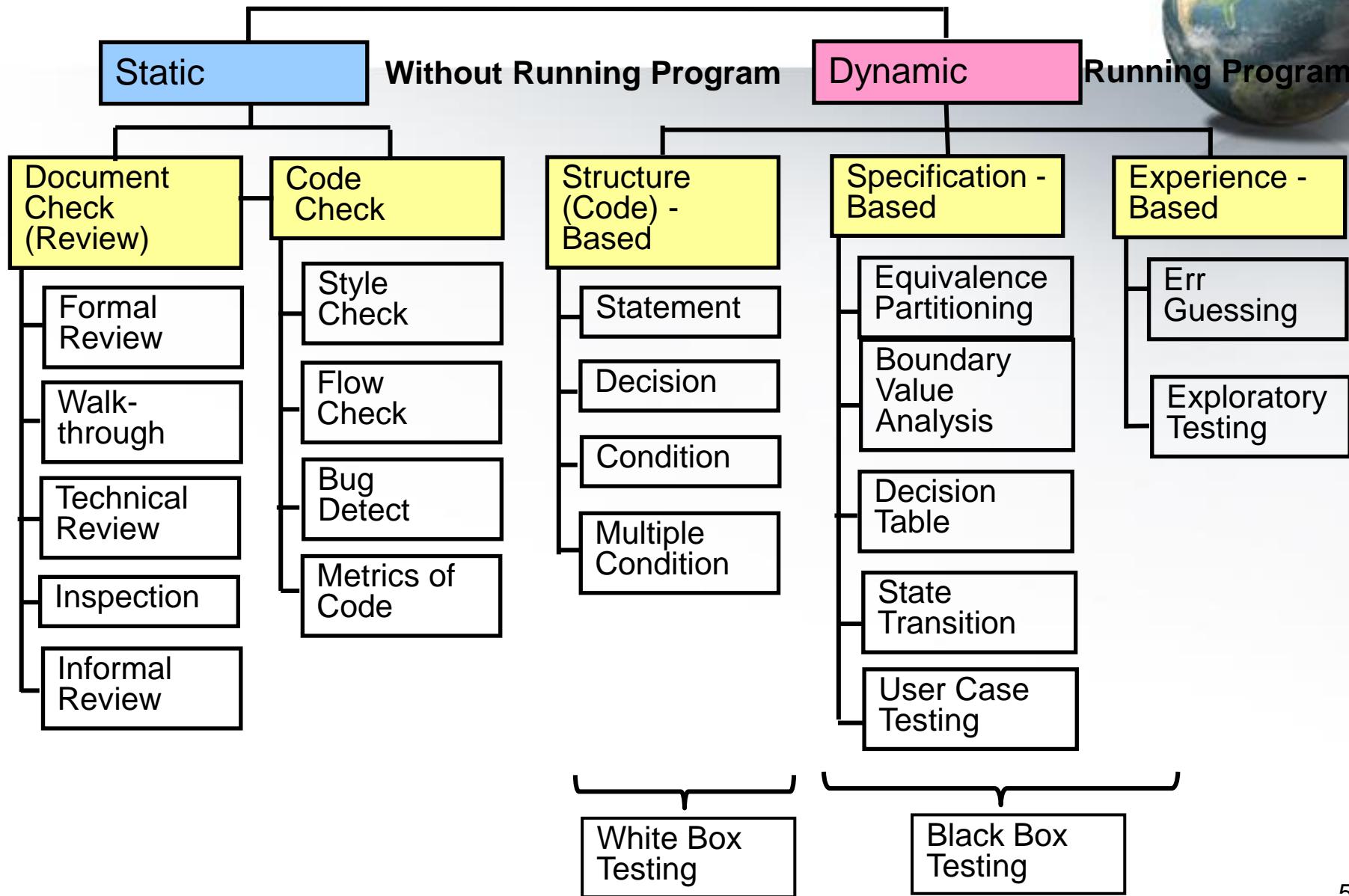


► To be explained in this chapter (Overview)

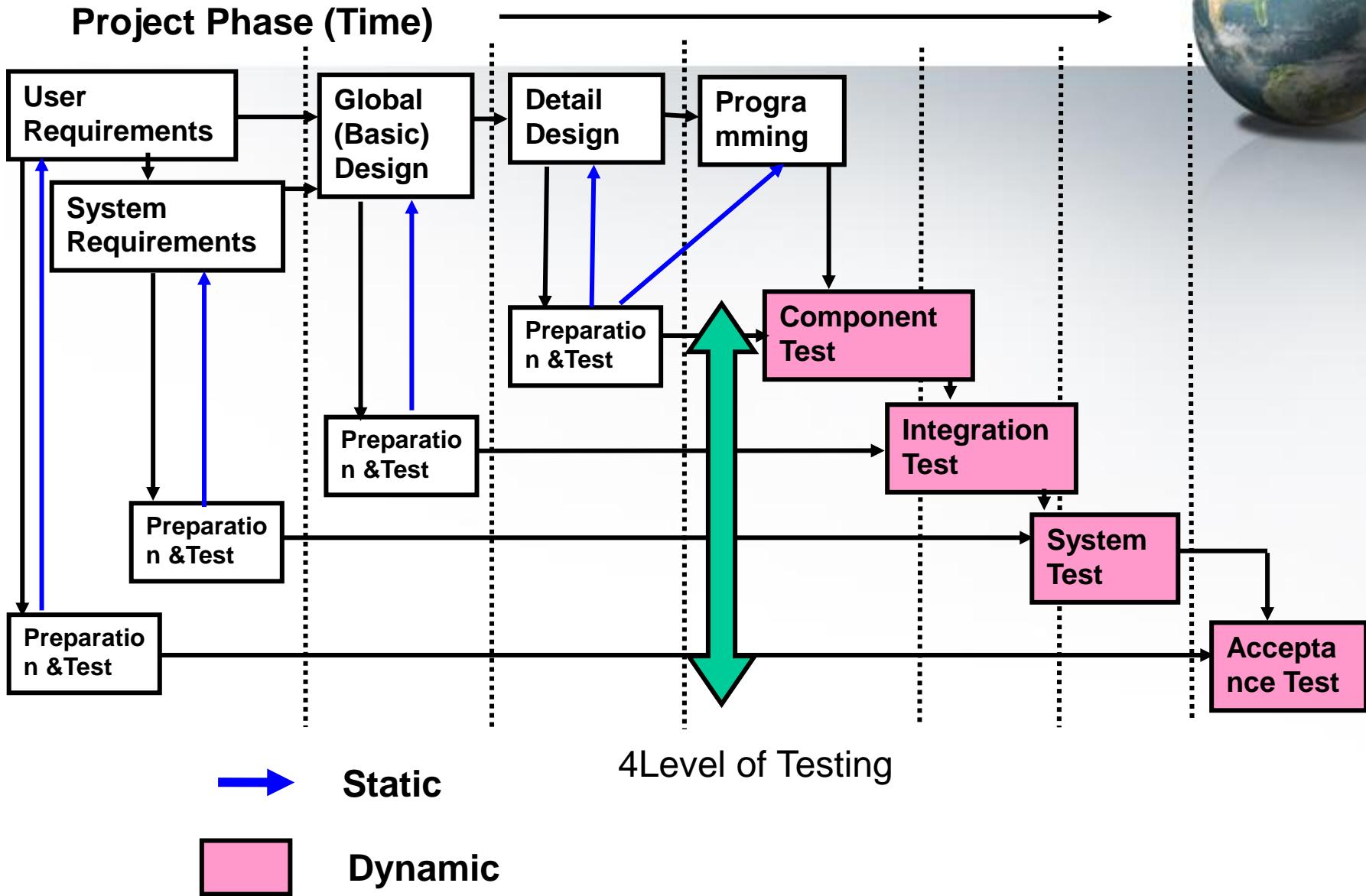


Software Testing has many concepts and Terms, You should understand them not alone, but with relation among them.

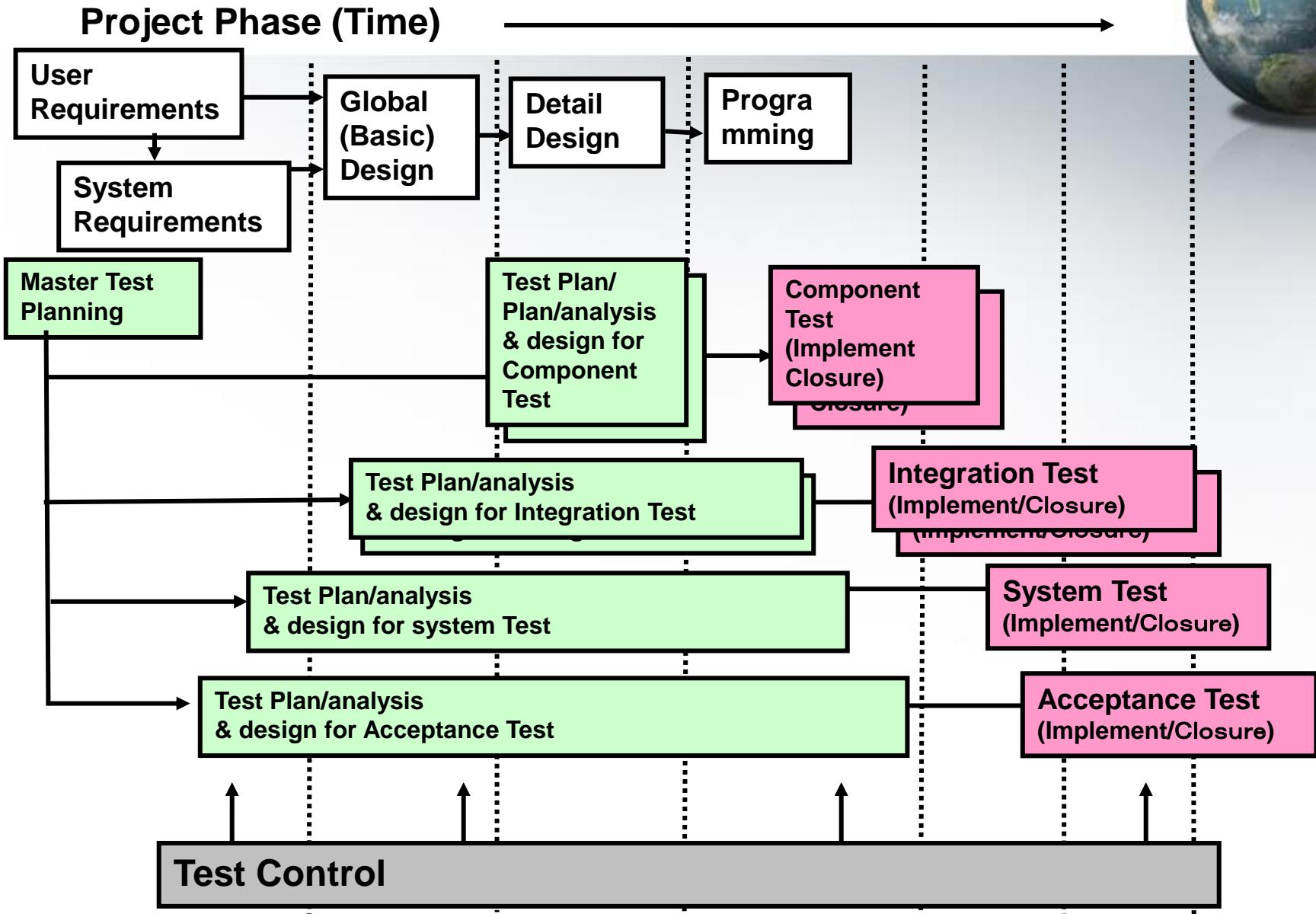
Overview of Testing Techniques



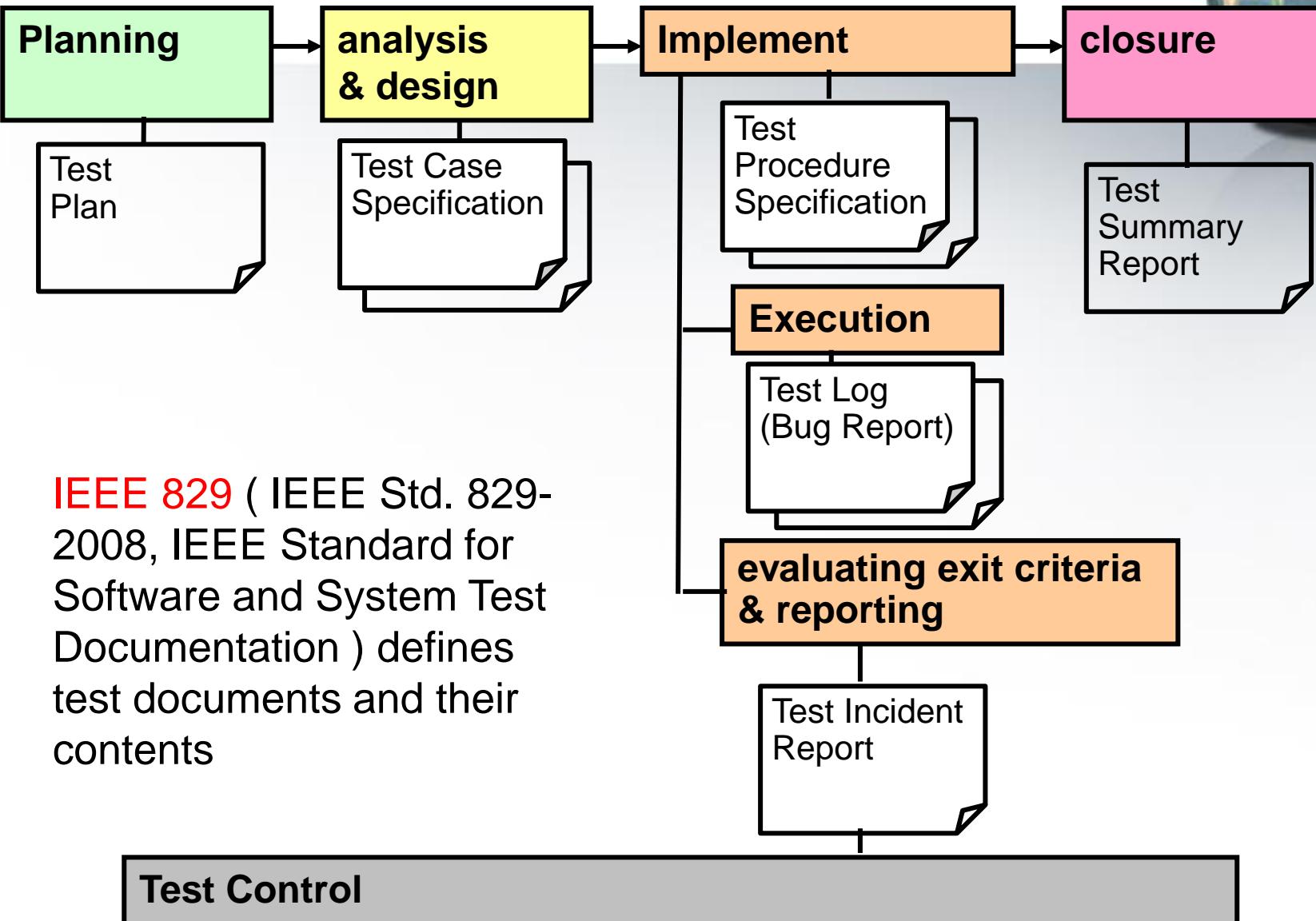
Overview of Process/ lifecycle/Level in a Project



Example Test Process in the Project process (Middle size)



Overview of Test Documents



Some Terminology



- Verification
- Validation
- Software Quality Assurance

Some Terminology



- Verification
 - “Are we building the product right?”
 - Does the software meet the specification?
- Validation
 - “Are we building the right product?”
 - Does the software meet the user requirements?
- Software Quality Assurance
 - Not the same as software testing ...
 - Create and enforce standards and methods to improve the development process and to prevent bugs from occurring.

What is Testability



What is Testability



- Operability
- Observe-ability
- Controllability
- Decomposability
- Stability
- Understandability

Software Testing Definitions



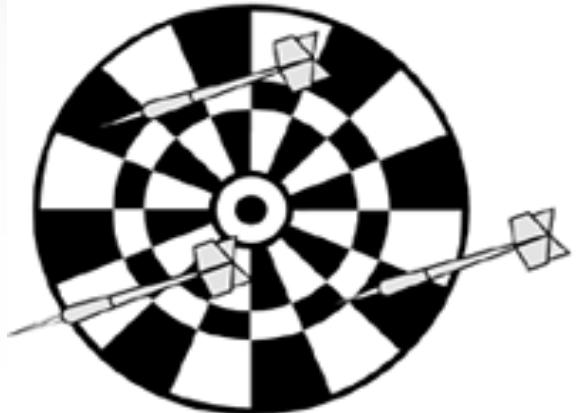
- **Accurate:**
- **Precise:**

Software Testing Definitions

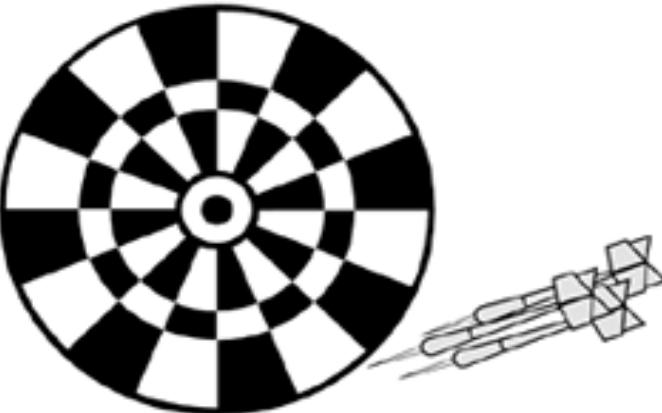


- **Accurate:** the degree of closeness of measurements of a quantity to its actual (true) value
- **Precise:** the degree to which repeated measurements under unchanged conditions show the same results

Accurate and Precise



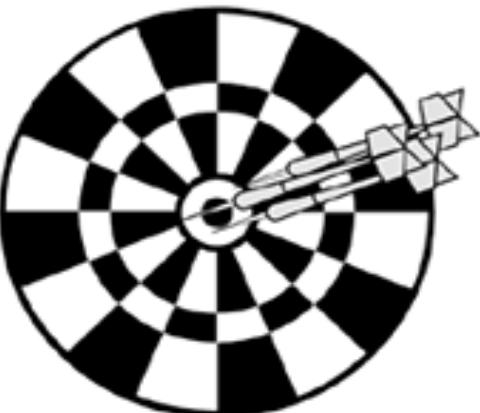
Neither Accurate nor Precise



Precise, but not Accurate



Accurate, but not Precise



Accurate and Precise

Software Testing Definitions



- **Static verification**
- **Dynamic verification**
- **Defect testing**
- **Debugging**

Software Testing Definitions



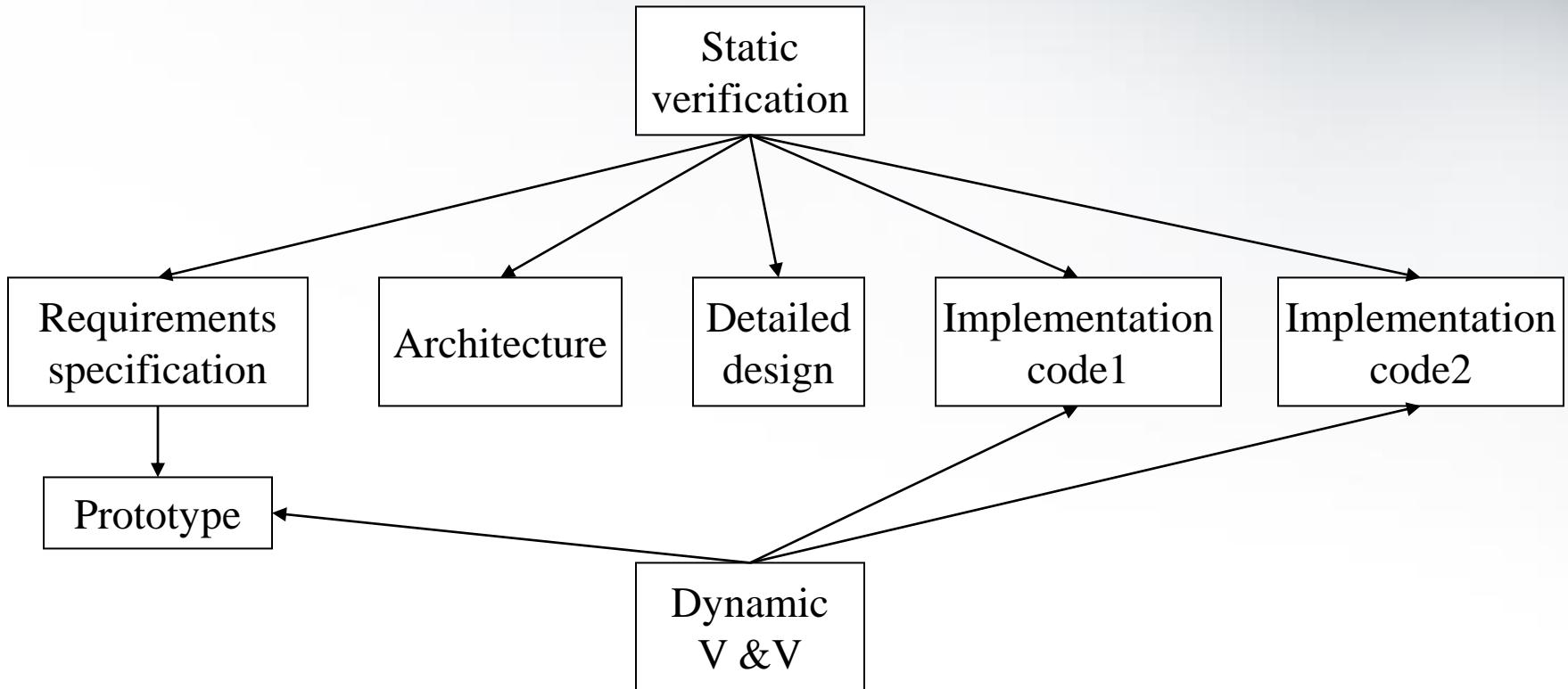
- **Static verification** refers to testing something that is not running—examining and reviewing it
- **Dynamic verification** Concerned with exercising and observing product behaviour
 - The system is executed with test data and its operational behaviour is observed
- **Defect testing** is concerned with confirming the presence of errors
- **Debugging** is concerned with locating and repairing these errors

Testing stages



- Unit testing
 - Testing of individual components
- Integration testing
 - Testing to expose problems arising from the combination of components
- System testing
 - Testing the complete system prior to delivery
- Acceptance testing
 - Testing by users to check that the system satisfies requirements. Sometimes called alpha testing

Static and dynamic V&V



QUALITY PRINCIPLES



QUALITY PRINCIPLES



Quality - the most important factor affecting an organization's long-term performance.

Quality - the way to achieve improved productivity and competitiveness in any organization.

Quality - saves. It does not cost.

Quality - is the solution to the problem, not a problem.



Cost of Quality

Prevention Cost

- Amount spent before the product is actually built. Cost incurred on establishing methods and procedures, training workers, acquiring tools and planning for quality.

Appraisal cost

- Amount spent after the product is built but before it is shipped to the user. Cost of inspection, testing, and reviews.



Failure Cost

- Amount spent to repair failures.
- Cost associated with defective products that have been delivered to the user or moved into production, costs involve repairing products to make them fit as per requirement.



Quality Assurance	Quality Control
A planned and systematic set of activities necessary to provide adequate confidence that requirements are properly established and products or services conform to specified requirements.	The process by which product quality is compared with applicable standards; and the action taken when non-conformance is detected.
An activity that establishes and evaluates the processes to produce the products.	An activity which verifies if the product meets pre-defined standards.



Quality Assurance	Quality Control
Helps establish processes.	Implements the process.
Sets up measurements programs to evaluate processes.	Verifies if specific attributes are in a specific product or Service
Identifies weaknesses in processes and improves them.	Identifies defects for the primary purpose of correcting defects.

Responsibilities of QA and QC



QA is the responsibility of the entire team.	QC is the responsibility of the tester.
Prevents the introduction of issues or defects	Detects, reports and corrects defects
QA evaluates whether or not quality control is working for the primary purpose of determining whether or not there is a weakness in the process.	QC evaluates if the application is working for the primary purpose of determining if there is a flaw / defect in the functionalities.

Responsibilities of QA and QC

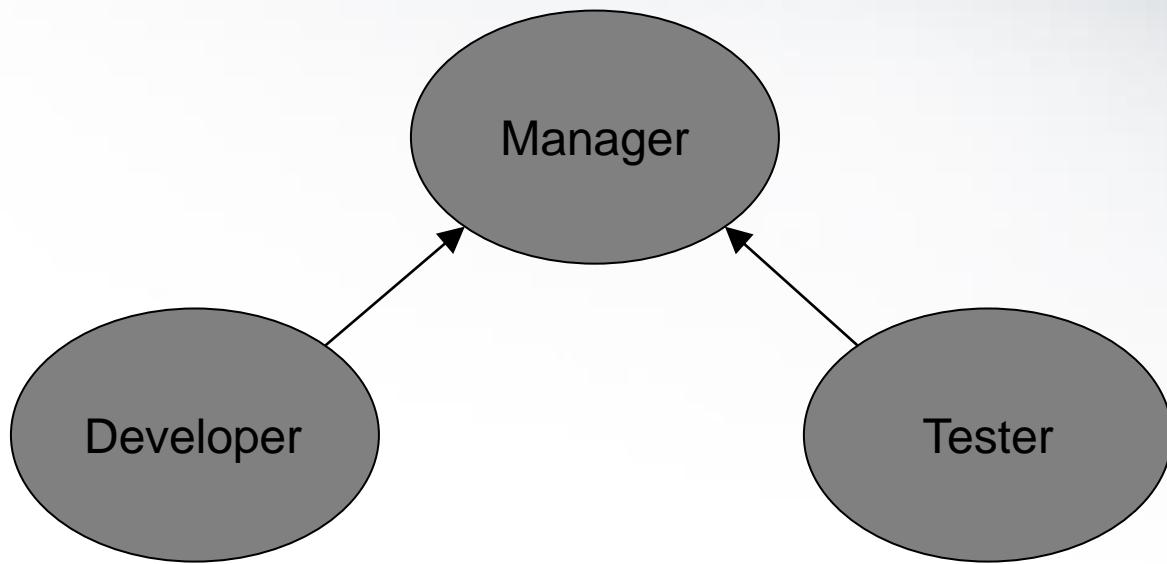


<p>QA improves the process that is applied to multiple products that will ever be produced by a process.</p>	<p>QC improves the development of a specific product or service.</p>
<p>QA personnel should not perform quality control unless doing it to validate quality control is working.</p>	<p>QC personnel may perform quality assurance tasks if and when required.</p>

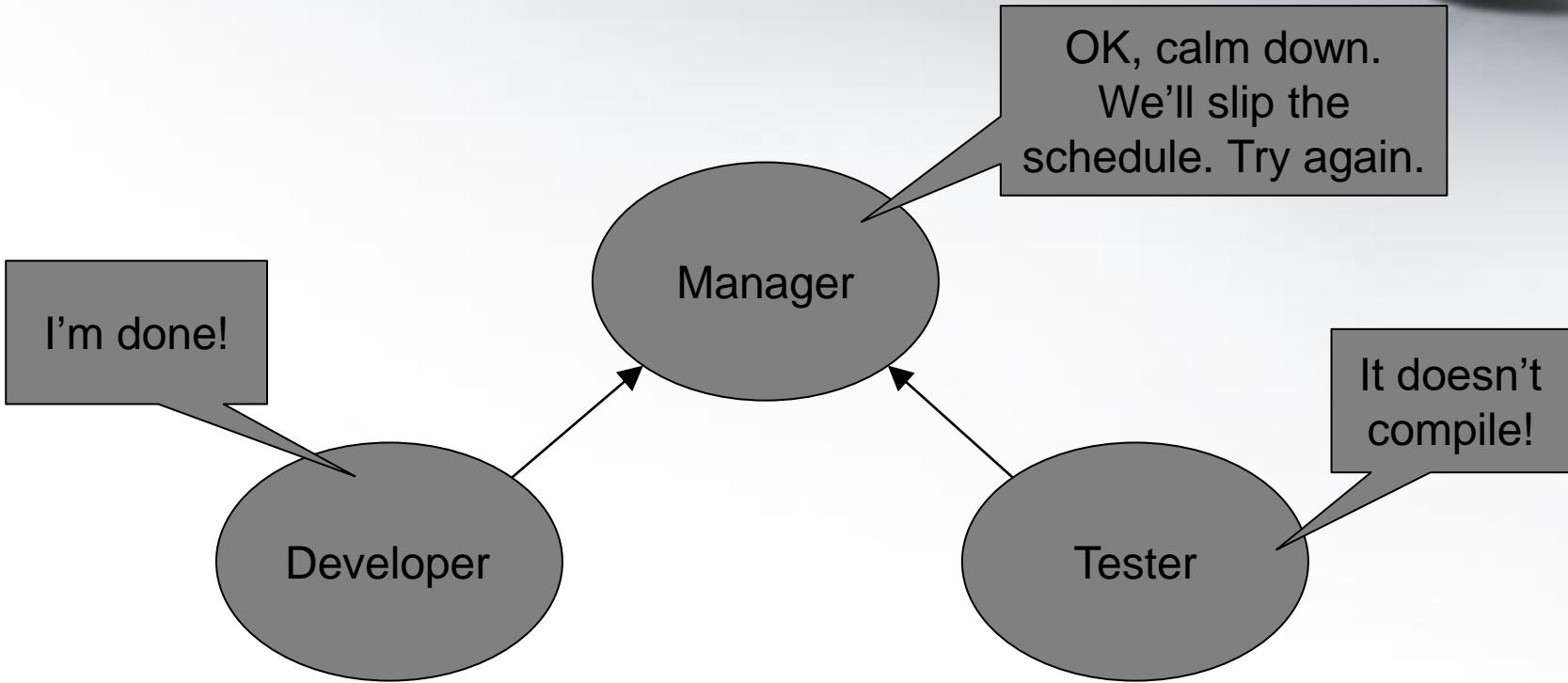
Software Testing and Quality Management



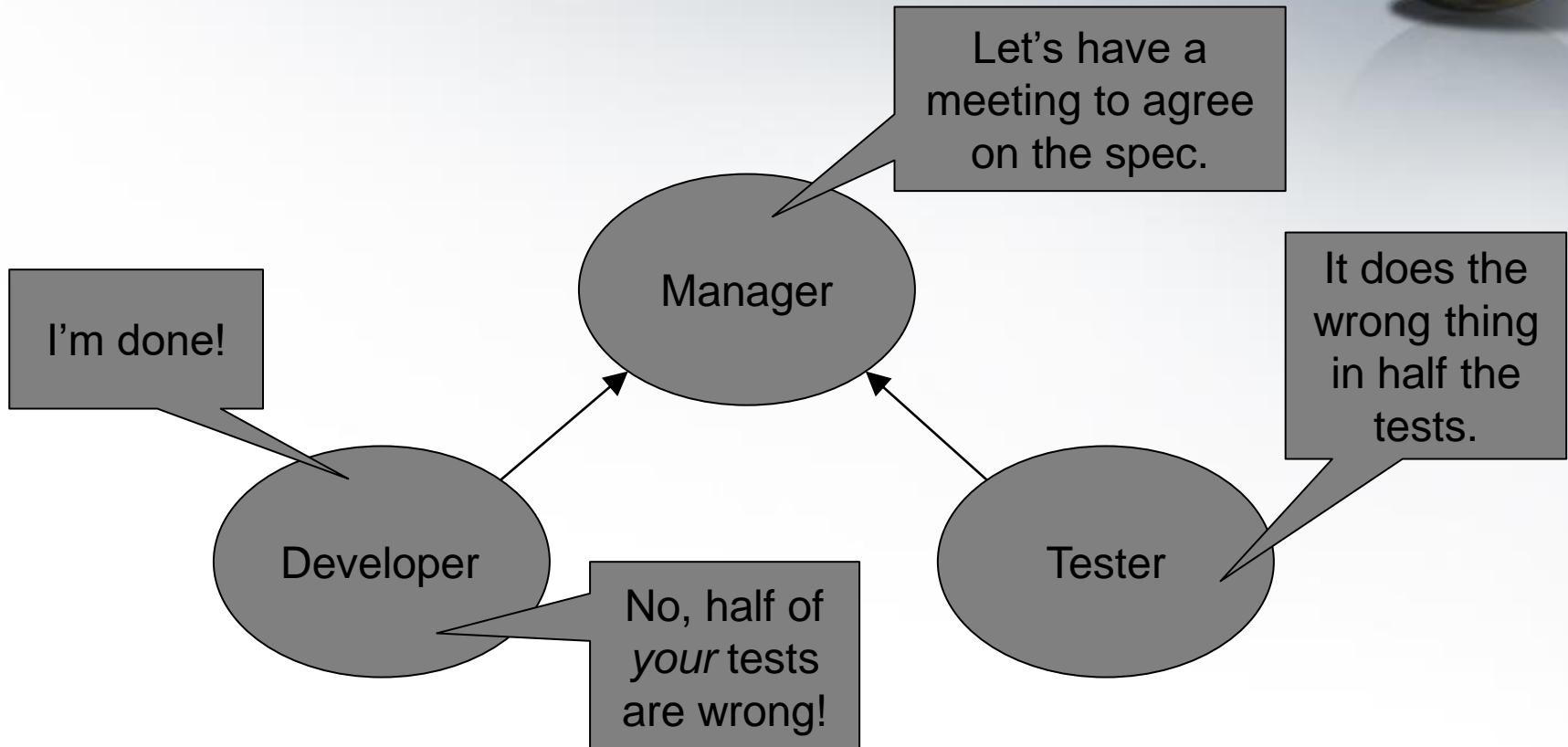
Software Development Today



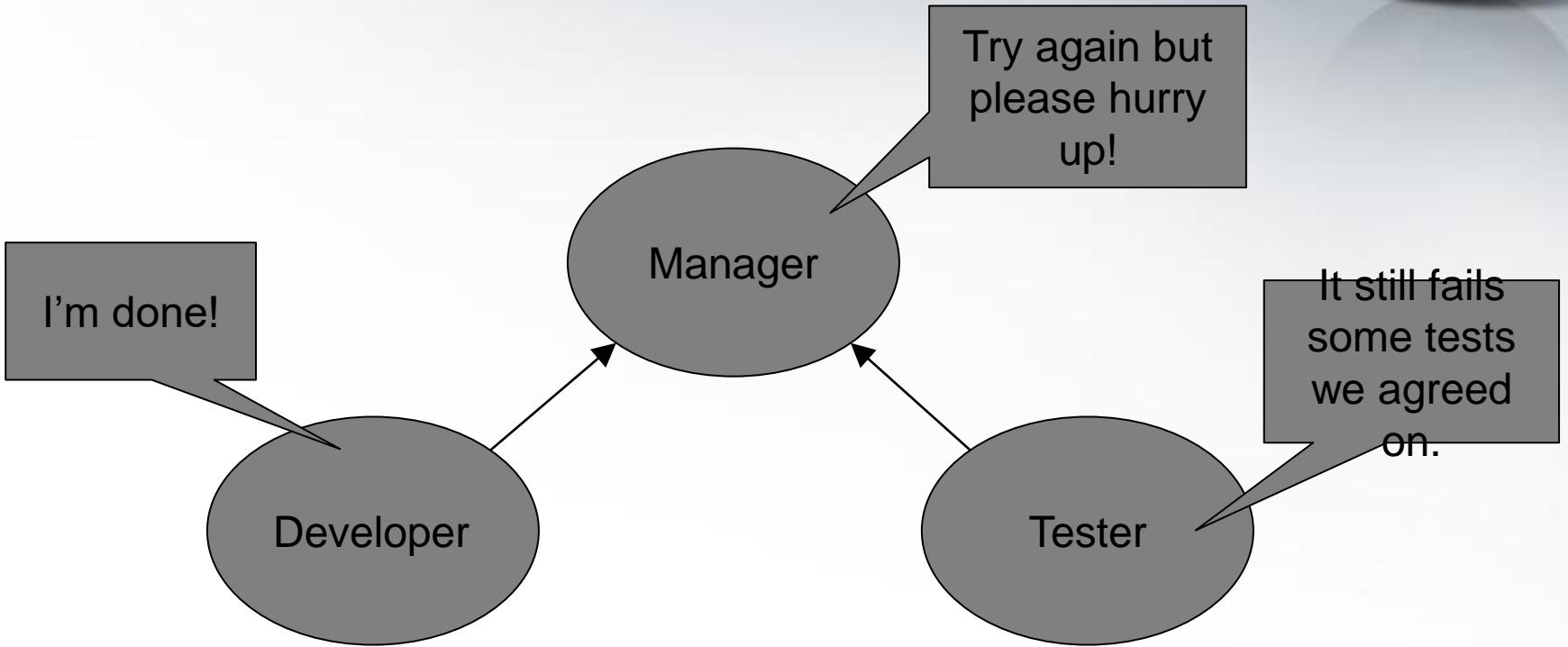
A Typical Scenario



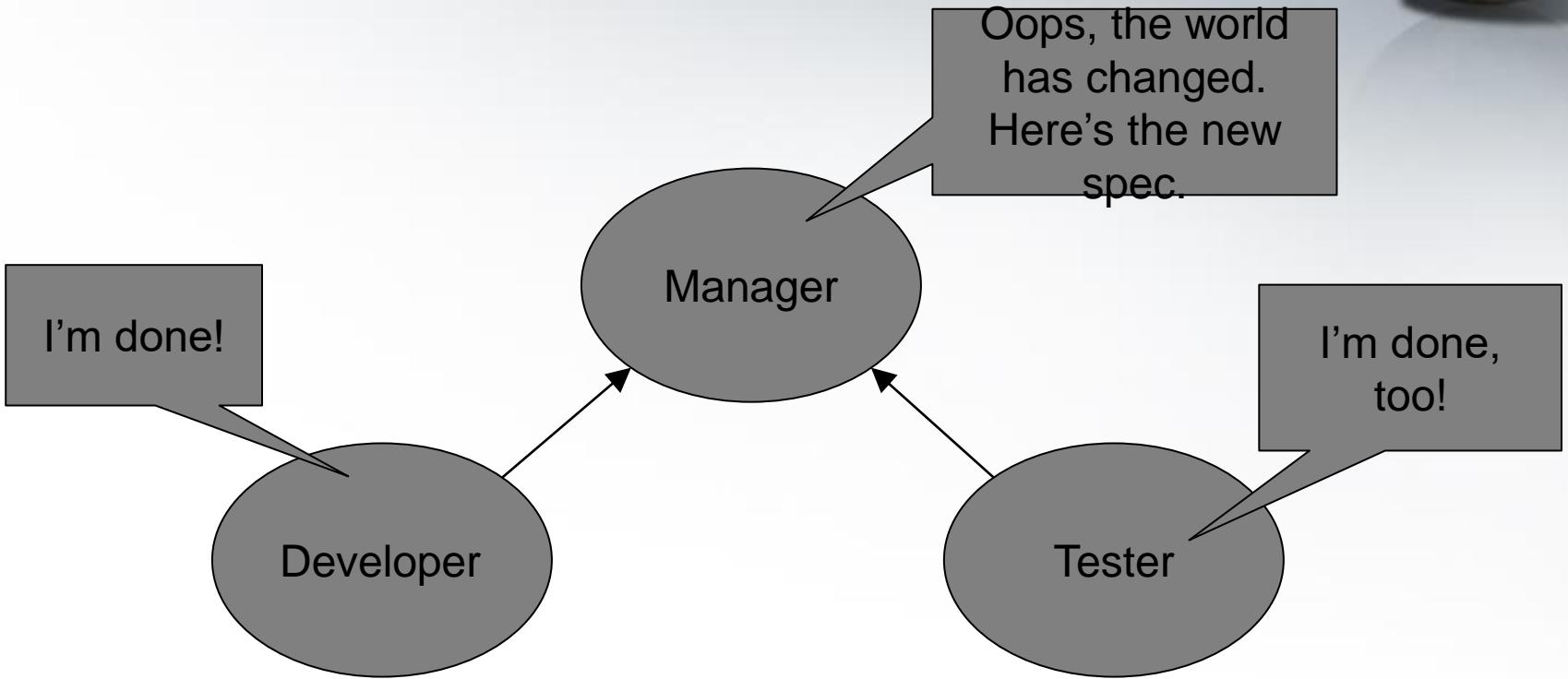
A Typical Scenario



A Typical Scenario



A Typical Scenario



QUALITY PRINCIPLES



QUALITY PRINCIPLES



Quality - the most important factor affecting an organization's long-term performance.

Quality - the way to achieve improved productivity and competitiveness in any organization.

Quality - saves. It does not cost.

Quality - is the solution to the problem, not a problem.

What is the cost of quality





Cost of Quality

Prevention Cost

- Amount spent before the product is actually built. Cost incurred on establishing methods and procedures, training workers, acquiring tools and planning for quality.

Appraisal cost

- Amount spent after the product is built but before it is shipped to the user. Cost of inspection, testing, and reviews.



Failure Cost

- Amount spent to repair failures.
- Cost associated with defective products that have been delivered to the user or moved into production, costs involve repairing products to make them fit as per requirement.

Quality Basic Terms



- Quality Assurance
- Quality Control

Quality Assurance versus Quality Control



Quality Assurance	Quality Control
A planned and systematic set of activities necessary to provide adequate confidence that requirements are properly established and products or services conform to specified requirements.	The process by which product quality is compared with applicable standards; and the action taken when non-conformance is detected.
An activity that establishes and evaluates the processes to produce the products.	An activity which verifies if the product meets pre-defined standards.

Quality Assurance versus Quality Control



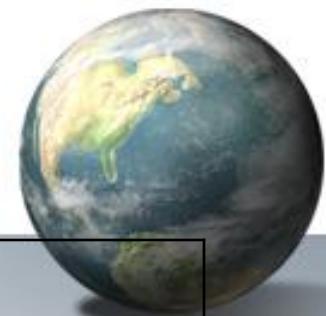
Quality Assurance	Quality Control
Helps establish processes.	Implements the process.
Sets up measurements programs to evaluate processes.	Verifies if specific attributes are in a specific product or Service
Identifies weaknesses in processes and improves them.	Identifies defects for the primary purpose of correcting defects.

Responsibilities of QA and QC



QA is the responsibility of the entire team.	QC is the responsibility of the tester.
Prevents the introduction of issues or defects	Detects, reports and corrects defects
QA evaluates whether or not quality control is working for the primary purpose of determining whether or not there is a weakness in the process.	QC evaluates if the application is working for the primary purpose of determining if there is a flaw / defect in the functionalities.

Responsibilities of QA and QC



<p>QA improves the process that is applied to multiple products that will ever be produced by a process.</p>	<p>QC improves the development of a specific product or service.</p>
<p>QA personnel should not perform quality control unless doing it to validate quality control is working.</p>	<p>QC personnel may perform quality assurance tasks if and when required.</p>

Testing is NOT

- Code inspections
- Design reviews
- Configuration management
- Bug tracking



*These are, along with **testing**, are part of Software Quality Assurance (SQA). Together these improve the quality of the product*

Quality and testing



- “Errors should be found and fixed as close to their place of origin as possible.” Fagan
- “Trying to improve quality by increasing testing is like trying to lose weight by weighing yourself more often.” tMcConnell

Tools for Improving Quality



- Formal specification
- Self-checking code
- Program verification and validation
- **Testing**
- Deploy with capabilities to repair

What is Testing?

Testing

- Testing is the process of executing software in a controlled manner, to answer the question:
“does the software behave as specified?”
- Implies that we have a specification (or possibly the tests are it)
- (or) Implies that we have some property we wish to test for independently of the specification
 - e.g., “*all paths in the code are reachable (no dead code)*”
- Testing is often associated with the words validation and verification

Verification vs. Validation

Verification

- Verification is the checking or testing of software (or anything else) for conformance and consistency with a given specification – answers the question:
“Are we doing the job right?”

Validation

- Validation is the process of checking that what has been specified is what the user actually wanted – answers the question:
“Are we doing the right job?”

Verification vs. Validation

Verification

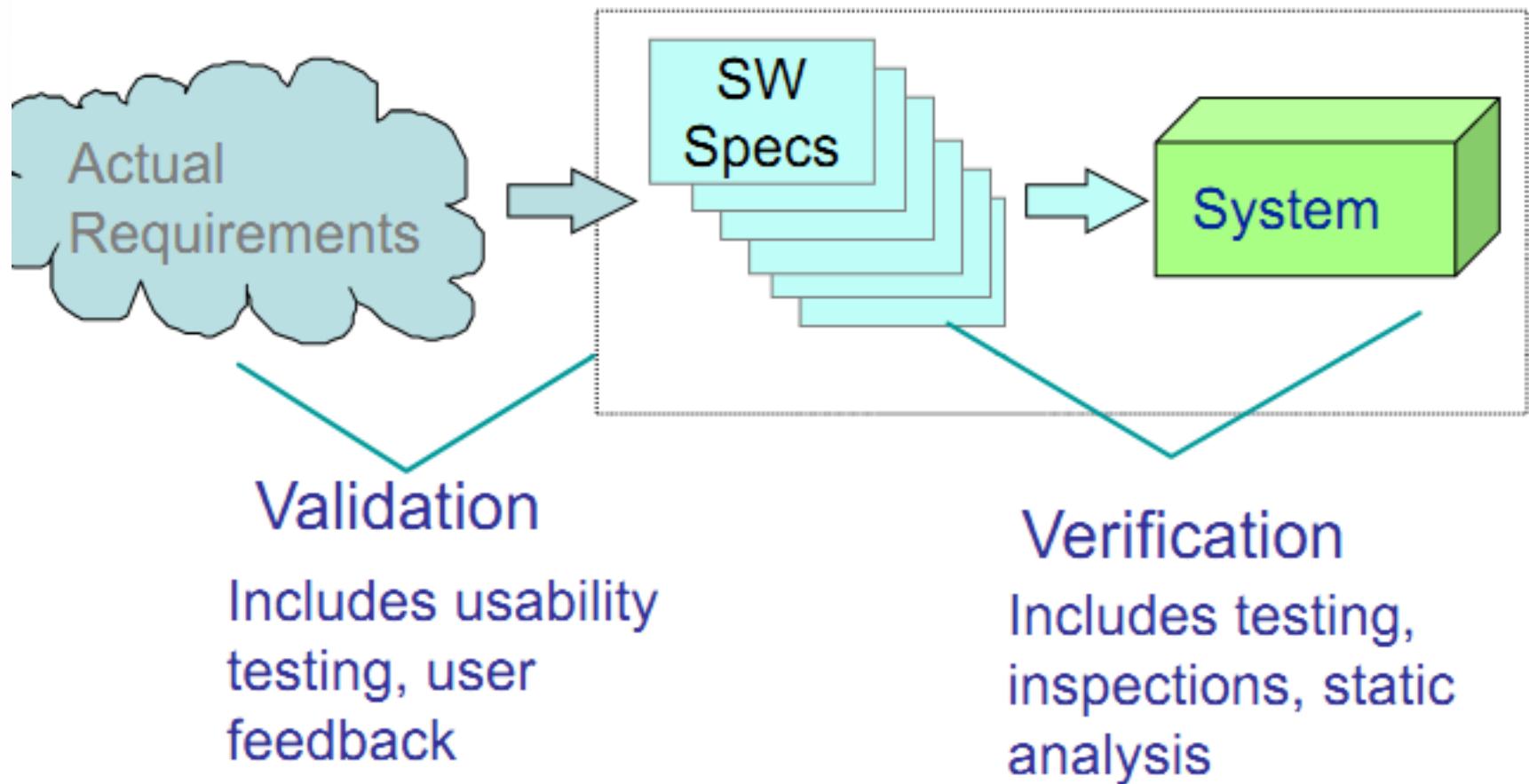
- Testing is most useful in verification – checking software (or anything else) for conformance and consistency with a given specification,
- However, testing is just one part of it analysis – inspection and measurement are also important

Validation

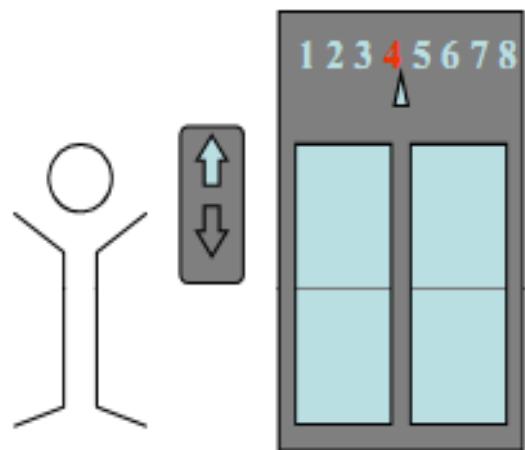
- Checking that what has been specified is what the user actually wanted usually involves meetings, reviews and discussions
- Testing is less useful in validation, although it can have a role



Validation and Verification



Verification or validation depends on the specification



Example: elevator response

Unverifiable (but validatable) spec: ... if a user presses a request button at floor i, an available elevator must arrive at floor i soon...

Verifiable spec: ... if a user presses a request button at floor i, an available elevator must arrive at floor i within 30 seconds...

Testing vs. Debugging

Debugging is not Testing!

Debugging:

the process of analyzing and locating bugs when the software does not behave as expected.

Testing:

the process of methodically searching for and exposing bugs (not just fixing those that happen to show up) – much more comprehensive.

- Debugging therefore supports testing, but cannot replace it
 - However, no amount of testing* is guaranteed to find all bugs
-

Types of Testing:



- Scope
 - Unit
 - Module
 - Integration
 - Interface
 - Coverage
- Purpose
 - Functional
 - Stress
- Execution
 - Manual
 - Automated
- Phases/Triggers
 - Nightly Builds
 - Defect Removal
 - Change Order Completion
 - Alpha
 - Beta

Who Tests?



- Developers
 - Unit testing
- QA Team
 - Module testing, automated testing
- Software
 - Automated testing
- Designers
 - Engineers, Analysts
- Users
 - Unpleasant but true

Key Observations



- Specifications must be explicit
- Independent development and testing
- Resources are finite
- Specifications evolve over time

The Need for Specifications



- Testing checks whether program implementation agrees with program specification
- Without a specification, there is nothing to test!
- Testing a form of consistency checking between implementation and specification
 - Recurring theme for software quality checking approaches
 - What if both implementation and specification are wrong?

Developer != Tester



- Developer writes **implementation**, tester writes **specification**
- Unlikely that both will independently make the same mistake
- Specifications useful even if written by developer itself
 - Much simpler than implementation
 - => specification unlikely to have same mistake as implementation

Other Observations



- Resources are finite
=> Limit how many tests are written
- Specifications evolve over time
=> Tests must be updated over time
- An Idea: Automated Testing
=> No need for testers!?

Other Observations



- In reality there are many trade-offs software testing effort faces.
- In the current day scenario, it is less likely that clients are able to determine every requirement analysis aspect in one-go.

Other Observations



- Requirements can keep changing during the course of time. Hence there are more chances for some of the following realities to occur.
 - a) The specification might not correspond to the customer's needs perfectly.
 - b) Many a time, the time available for testing would not be comprehensive to cover all aspects of testing.
 - c) Tradeoffs and concessions are expected.

Other Observations



Realities of Software Testing

- A detailed specification which can perfectly meet the needs of a customer is not given and there is insufficient time to test the software in its entirety.
- However, if one aims to become a good software tester, he or she needs to:
 - (a) Identify the ideal process involved.
 - (b) Identify the bugs and problems and realize how they affect the project.

What is Systematic Testing?

Systematic Testing

- Systematic testing is an explicit discipline or procedure (a system) for:
 - choosing and creating test cases
 - executing the tests and documenting the results
 - evaluating the results, possibly automatically
 - deciding when we are done (enough testing)
- Because in general it is impossible to ever test completely, systematic methods choose a particular point of view, and test only from that point of view (the test criterion)
 - e.g., test only that every decision (if statement) can be executed either way

The Role of Specification

The Need for Specification

- Validation and verification activities, such as testing, cannot be meaningful unless we have a specification for the software
- The software we are building could be a single module or class, or could be an entire system
- Depending on the size of the project and the development methods, specifications can range from a single page to a complex hierarchy of interrelated documents

Levels of Specifications

- There are usually at least **three levels** of software specification documents in large systems:
 1. **Functional specifications** (or requirements) give a precise description of the required behaviour of the system – *what the software should do, not how it should do it* – *may also describe constraints on how this can be achieved*
 2. **Design specifications** describe the architecture of the design to implement the functional specification – *the components of the software and how they are to relate to one another*
 3. **Detailed design specifications** describe how each component of the architecture is to be implemented – *down to the individual code units*

Levels of Testing

- Given the hierarchy of specifications, it is usual to structure testing into three (or more) corresponding levels:
 3. Unit testing addresses the verification that individual components meet their detailed design specification
 2. Integration testing verifies that the groups of units corresponding to architectural elements of the design specification can be integrated to work as a whole
 1. System testing verifies that the integrated total product has the functionality specified in the functional specification
- To these levels it is usual to add the additional test level:
 0. Acceptance testing, in which the actual customers validate that the software meets their real intentions as well as what has been functionally specified, and accept the result

Using Tests

Evaluating Tests

- Within each level of testing, once the tests have been applied, test results are evaluated
- If a problem is encountered, then either:
 - a) the tests are revised and applied again, if the tests are wrong, or
 - b) the software is fixed and the tests are applied again, if the software is wrong
- In either case, the tests are applied again, and so on, until no more problems are found.
- Only when no more problems are found does development proceed to the next level of testing

Test Evolution

Tests Don't Die!

- As we have already seen with XP, testing does not end when the software is accepted by the customer
- Tests must be repeated, modified and extended to insure that no existing functionality has been broken, and that any new functionality is implemented according to the revised specifications and design
- Maintenance of the tests for a system is a major part of the effort to maintain and evolve a software system while retaining a high level of quality
- In order to make this continual testing practical, automation plays a large role in software testing methods

Testing in the Software Life Cycle

Kinds of Tests

- Testing has a role at every stage of the software life cycle
- As we have seen, tests play a role in:
 - the development of code ([unit testing](#)),
 - the integration of the units into subsystems ([integration testing](#)) and
 - the acceptance of the first version of the software system ([system testing](#))



Testing in the Software Life Cycle

Kinds of Tests

- We will divide these tests into:

**Black Box
Testing
Methods**

- **Black box** methods – cannot see the software code (it may not exist yet!) – can only base their tests on the requirements or specifications

**White Box
Testing
Methods**

- **White box** (aka glass box) methods – can see the software's code – can base their tests on the software's actual **architecture** or **code**

Testing in the Software Life Cycle

Regression Tests

- In addition, as the system is maintained, other kinds of tests based on *past behaviour* come into play
- Once a system is stable and in production, we build and maintain a set of *regression tests* to insure that when a change is made the existing behaviour has not been broken
- These often consist of a set of actual observed production inputs and their archived outputs from past versions of the system

Testing in the Software Life Cycle

Failure Tests

- As failures are discovered and fixed, we also maintain a set of **failure tests** to insure that we have really fixed the observed failures, and to make sure that we don't cause them again
- These consist of a set of actual observed inputs that caused the failures and their archived outputs after the system is fixed

Test Design

Design of Tests

- The design of tests for a system is a difficult and tricky engineering problem as important as design of the software itself
- The design of effective tests requires a set of stages from an initial high level test strategy down to detailed test procedures
- Typical test design stages are:
 - test strategy, test planning, test case design, test procedure

Test Design

(1) Test Strategy

- A test strategy is a statement of the overall approach to testing for a software development organization
- Specifies the levels of testing to be done as well as the methods, techniques and tools to be used
- Part of the project's overall quality plan, to be followed and reported by all members of the project

Test Strategy Examples

Big Bang Testing Strategies

- Test the entire software once it is complete.

Incremental Testing Strategies

- Test the software in phases (unit testing, integration testing, system testing)
- This testing strategy is what we use in **XP**
- Incremental testing can occur **bottom-up** (using drivers) or **top-down** (using stubs)
- In general bottom-up is easier to perform but means the whole program behavior is observed at a later stage of development

Test Strategy Examples

Big Bang vs. Incremental

- Big bang testing only works with a very small and simple program
- In general, incremental testing has several advantages:
 - Error identification ✓
 - Easier to identify more errors
 - Error correction ✓
 - Simpler and requires less resources

Test Design

(2) Test Plans

- A test plan for a development project specifies in detail how the test strategy will be carried out for the project
- In particular, it specifies:
 - the **items** to be tested
 - the **level** they will be tested at
 - the **order** they will be tested in
 - the test **environment**
- May be project wide, or may be structured into separate plans for unit, integration, system and acceptance testing

Test Design

(3) Test Case Design

- Once we have a plan, we need to specify a set of **test cases** for each item to be tested at each level
- Each test case specifies **how** the implementation of a particular functional requirement or design unit is to be tested and how we will **know** if the test is successful
- It is important to include test cases to test both that the software does what it should (**positive** testing) and that it doesn't do what it shouldn't (**negative** testing)
- Test cases are specified **separately** at each level: unit, integration, system and acceptance – and their documentation forms a **test specification** for the level

Test Design

(4) Test Procedures

- The final stage of test design is the test procedure, which specifies the process for conducting test cases
- For each item or set of items to be tested at each level of testing, the test procedure specifies the process to be followed in running and evaluating the test cases for the item
- Often this includes the use of test harnesses (programs written solely to exercise the software or parts of it on the test cases), test scripts (automated procedures for running sets of test cases), or commercial testing tools

Test Reports

Documenting Test Results

- Output of test execution should be saved in a test results file, and summarized in a readable report
- Test reports should be designed to be concise, easy to read and to clearly point out failures or unexpectedly changed results
- Test result files should be saved in a standardized form, for easy comparison with future test executions

Types of Black Box Testing

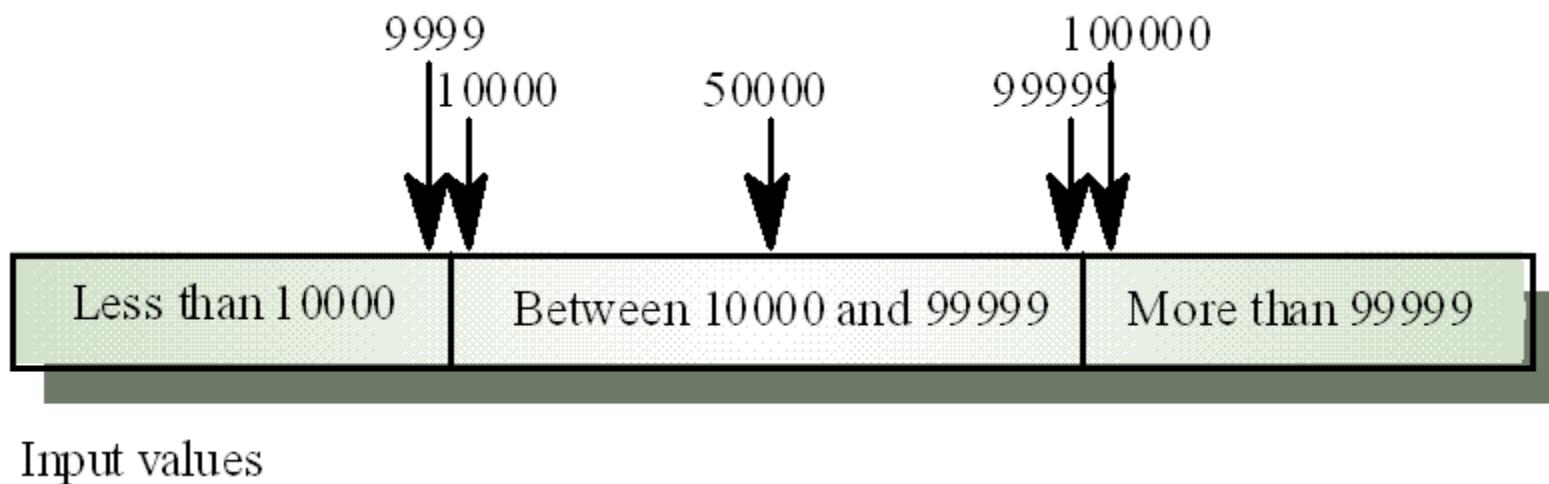


- Equivalence Partitioning
- Boundary value analysis
- Graph-Based Testing Methods
- Requirements-Based Testing
- Random Testing

Equivalence Partitioning

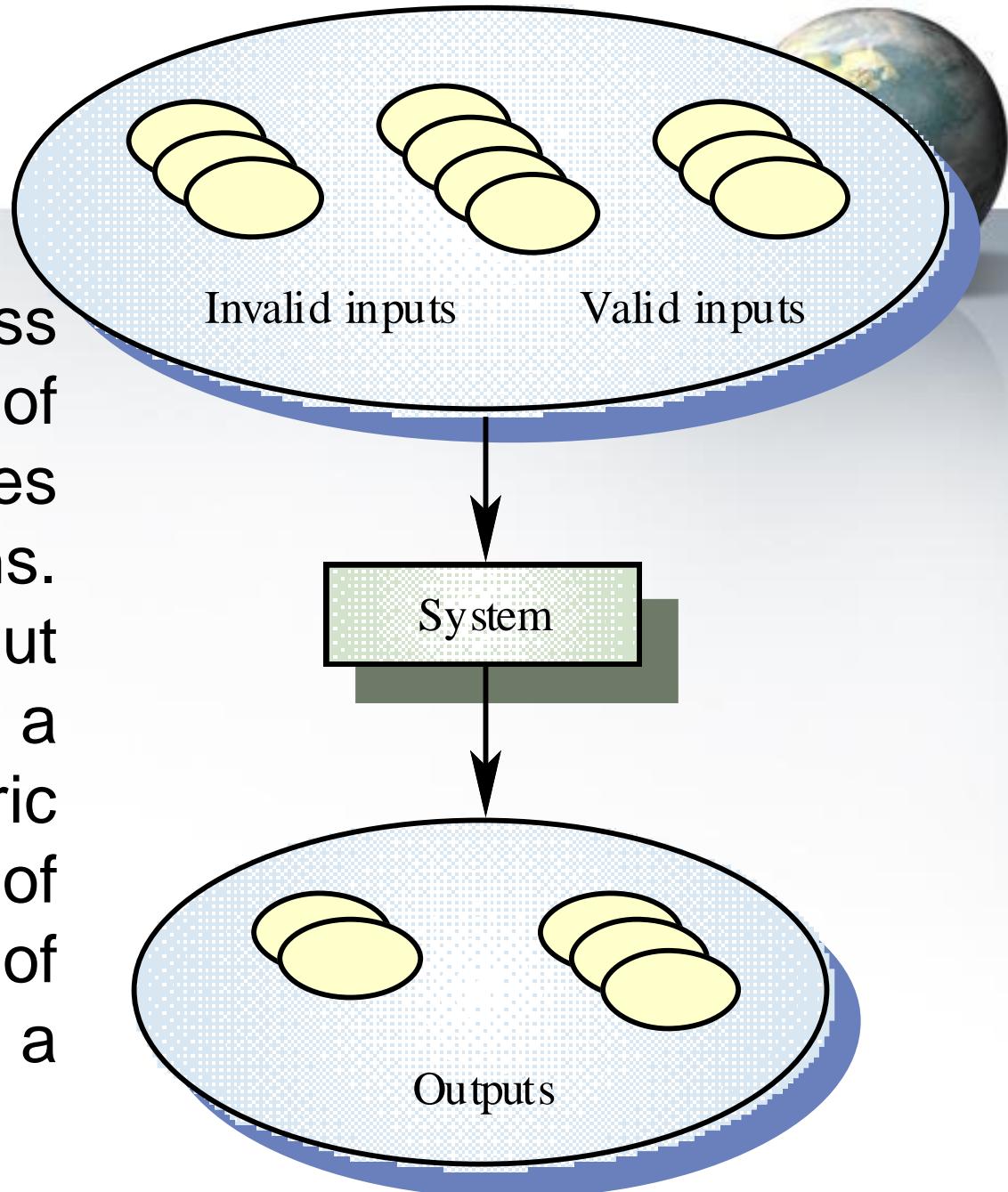


- Partition system inputs and outputs into “equivalence sets”:
 - If input is a 5-digit integer between 10,000 and 99,999, equivalence partitions are



Equivalence Partitioning

An equivalence class represents a set of valid or invalid states for input conditions. Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition.



Equivalence partitioning



Test selection using **equivalence partitioning** allows a tester to subdivide the input domain into a relatively small number of sub-domains.

Each subset is known as an **equivalence class**.

Aim



- Equivalence class testing strategy is developed with the following aims
 - – To test as few input values as possible
 - – To spot and eliminate redundant tests
 - – To tell how many tests are necessary to test a given piece of software to a known level

Example 1



Consider an application A that takes an integer denoted by age as input. Let us suppose that the only legal values of age are in the range [1..120]. The set of input values is now divided into a set E containing all integers in the range [1..120] and a set U containing the remaining integers.





Example 1 (contd.)

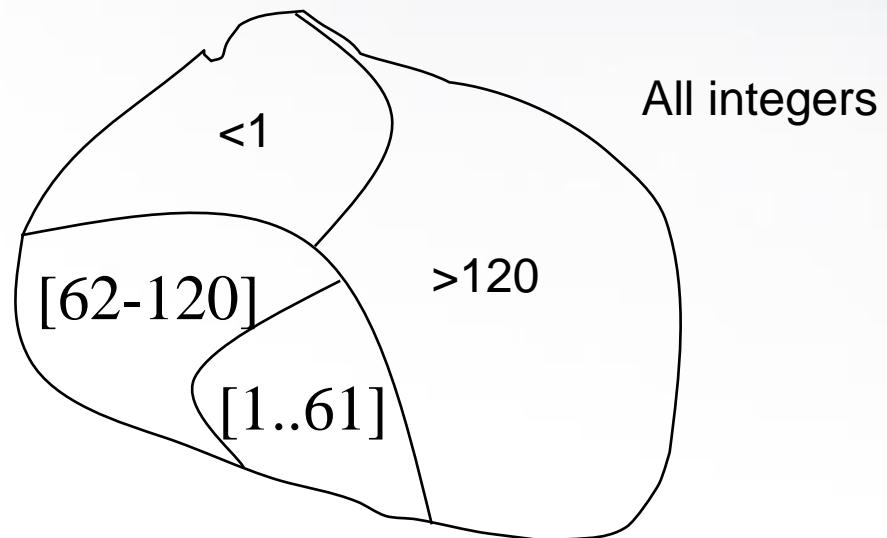
Further, assume that the application is required to process all values in the range [1..61] in accordance with requirement R1 and those in the range [62..120] according to requirement R2.

Thus E is further subdivided into two regions depending on the expected behavior.

Similarly, it is expected that all invalid inputs less than or equal to 1 are to be treated in one way while all greater than 120 are to be treated differently. This leads to a subdivision of U into two categories.



Example 1 (contd.)





Example 1 (contd.)

Tests selected using the equivalence partitioning technique aim at targeting faults in the application under test with respect to inputs in any of the four regions, i.e. two regions containing expected inputs and two regions containing the unexpected inputs.

It is expected that any single test selected from the range [1..61] will reveal any fault with respect to R1. Similarly, any test selected from the region [62..120] will reveal any fault with respect to R2. A similar expectation applies to the two regions containing the unexpected inputs.

Types of Equivalence classes

Partition



- Uni-dimensional
- Multi-dimensional



Unidimensional partitioning

One way to partition the input domain is to consider one input variable at a time. Thus each input variable leads to a partition of the input domain. We refer to this style of partitioning as **unidimensional** equivalence partitioning or simply **unidimensional** partitioning.

This type of partitioning is commonly used.



Multidimensional partitioning

Another way is to consider the input domain I as the set product of the input variables and define a relation on I . This procedure creates one partition consisting of several equivalence classes. We refer to this method as **multidimensional** equivalence partitioning or simply **multidimensional** partitioning.

Multidimensional partitioning leads to a large number of equivalence classes that are difficult to manage manually.



Partitioning Example

Consider an application that requires two integer inputs x and y . Each of these inputs is expected to lie in the following ranges: $3 \leq x \leq 7$ and $5 \leq y \leq 9$.

For unidimensional partitioning we apply the partitioning guidelines to x and y individually. This leads to the following six equivalence classes.



Partitioning Example (contd.)

$$E1: x < 3$$

$$E2: 3 \leq x \leq 7$$

$$E3: x > 7$$

← *y ignored.*

$$E4: y < 5$$

$$E5: 5 \leq y \leq 9$$

$$E6: y > 9$$

← *x ignored.*

For multidimensional partitioning we consider the input domain to be the set product $X \times Y$. This leads to 9 equivalence classes.



Partitioning Example (contd.)

E1: $x < 3, y < 5$

E2: $x < 3, 5 \leq y \leq 9$

E3: $x < 3, y > 9$

E4: $3 \leq x \leq 7, y < 5$

E5: $3 \leq x \leq 7, 5 \leq y \leq 9$

E6: $3 \leq x \leq 7, y > 9$

E7: $x > 7, y < 5$

E8: $x > 7, 5 \leq y \leq 9$

E9: $x > 7, y > 9$

Exhaustive testing



- The large size of the input domain prevents a tester from exhaustively testing the program under test against all possible inputs. By ``exhaustive'' testing we mean testing the given program against every element in its input domain.
- The complexity makes it harder to select individual tests.

Large input domain



Consider program P that is required to sort a sequence of integers into ascending order. Assuming that P will be executed on a machine in which integers range from -32768 to 32767, the input domain of pr consists of all possible sequences of integers in the range [-32768, 32767].

If there is no limit on the size of the sequence that can be input, then the input domain of P is infinitely large and P can never be tested exhaustively.

Complex input domain



Consider a procedure P in a payroll processing system that takes an employee record as input and computes the weekly salary. For simplicity, assume that the employee record consists of the following items with their respective types and constraints:

ID: int;	ID is 3-digits long from 001 to 999.
name: string;	name is 20 characters long; each character belongs to the set of 26 letters and a space character.
rate: float;	rate varies from \$5 to \$10 per hour; rates are in multiples of a quarter.
hoursWorked: int;	hoursWorked varies from 0 to 60.



Boundary value analysis

Errors at the boundaries



Experience indicates that programmers make mistakes in processing values at and near the **boundaries of equivalence classes**.

For example, suppose that method M is required to compute a function f1 when $x \leq 0$ is true and function f2 otherwise. However, M has an error due to which it computes f1 for $x < 0$ and f2 otherwise.

Obviously, this fault is revealed, though not necessarily, when M is tested against $x=0$ but not if the input test set is, for example, $\{-4, 7\}$ derived using equivalence partitioning. In this example, the value $x=0$, lies at the boundary of the equivalence classes $x \leq 0$ and $x > 0$.

Boundary value analysis (BVA)



Boundary value analysis is a test selection technique that targets faults in applications at the boundaries of equivalence classes.

While equivalence partitioning selects tests from within equivalence classes, boundary value analysis focuses on tests at and near the boundaries of equivalence classes.

Certainly, tests derived using either of the two techniques may **overlap**.

BVA: Procedure



- 1 **Partition the input domain** using unidimensional partitioning. This leads to as many partitions as there are input variables. Alternately, a single partition of an input domain can be created using multidimensional partitioning. We will generate several sub-domains in this step.
- 2 **Identify the boundaries** for each partition. Boundaries may also be identified using special relationships amongst the inputs.
- 3 **Select test data** such that each boundary value occurs in at least one test input.

BVA: Example: 1. Create equivalence classes



Assuming that an item **code** must be in the range 99-999 and **quantity** in the range 1-100,

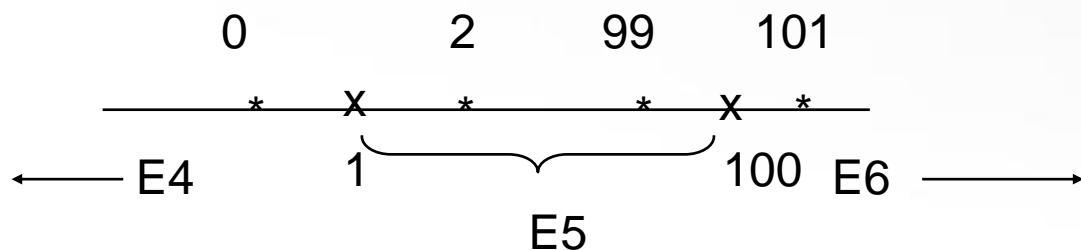
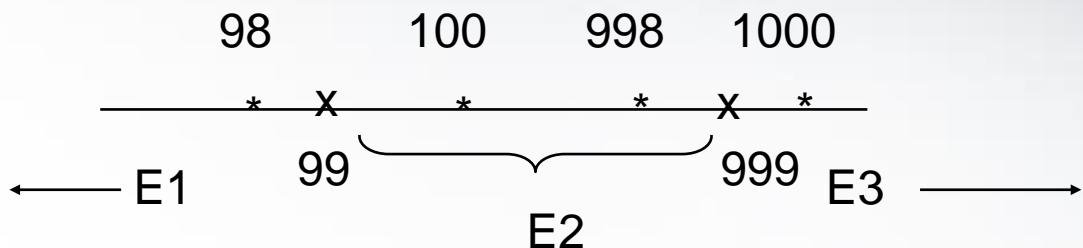
Equivalence classes for code:

- E1: Values less than 99.
- E2: Values in the range.
- E3: Values greater than 999.

Equivalence classes for qty:

- E4: Values less than 1.
- E5: Values in the range.
- E6: Values greater than 100.

BVA: Example: 2. Identify boundaries



Equivalence classes and boundaries for **findPrice**. Boundaries are indicated with an x. Points near the boundary are marked *.

BVA: Example: 3. Construct test set



Test selection based on the boundary value analysis technique requires that tests must include, for each variable, values at and around the boundary. Consider the following test set:

```
T={      t1: (code=98, qty=0),  
          t2: (code=99, qty=1),  
          t3: (code=100, qty=2),  
          t4: (code=998, qty=99),  
          t5: (code=999, qty=100),  
          t6: (code=1000, qty=101)  
    }
```

Illegal values of code and qty included.

Black Box testing in more detail





Black Box Methods

Black Box Methods

- In a black box method, we choose our test cases based solely on the requirements, specification or (sometimes) design documents
- **Advantage:** we can do it independently of the software – on a large project, black box tests can be developed in parallel with the development of the software, saving time
- Normally, black box testing is based on the functional specification (requirements) for the software system

Black Box Methods

Functional Specifications

- Functional specifications can be formal (mathematical), or more often informal (in a natural language such as English)
- In either case, the functional specification usually contains at least three kinds of information:
 1. the intended inputs
 2. the corresponding intended actions
 3. the corresponding intended outputs
- Focusing on each one of these separately gives us three different black box systems for testing

Black Box Methods

Three Kinds of Black Box Methods

- Systematic black box methods can be divided into three classes corresponding to these three kinds of information:
 1. input coverage tests, which are based on an analysis of the intended inputs, independent of their actions or outputs
 2. output coverage tests, which are based on an analysis of the intended outputs, independent of their inputs or actions
 3. functionality coverage tests, which are based on an analysis of the intended actions, with or without their inputs and outputs

Systematic Functionality Testing

An Example

- We begin with the third of these, functionality coverage testing
- Functionality coverage attempts to partition the functional specification of the software into a set of small, separate requirements

Example: Suppose that the informal requirements for a program we are to write are as follows:

"Given as input two integers x and y, output all the numbers smaller than or equal to x that are evenly divisible by y. If either x or y is zero, then output zero."

Systematic Functionality Testing

Requirements Partitioning

- Our first step is to physically **partition** the functional specification into separate requirements
- In this system we **model** the separate requirements as **independent**, even though they are not

Example: Suppose that the **informal** requirements for a program we are to write are as follows:

"Given as input two integers x and y, output all the numbers smaller than or equal to x that are evenly divisible by y. If either x or y is zero, then output zero."

Requirements Partitioning

"Given as input two integers x and y "

R1. Accept two integers as input.

"output ... the numbers"

R2. Output zero or more (integer) numbers.

"smaller than or equal to x "

R3. All numbers output must be less than or equal to the first input number.

"evenly divisible by y "

R4. All numbers output must be evenly divisible by the second number.

"all the numbers"

R5. Output must contain all numbers that meet both R3 and R4.

"If either x or y is zero, then output zero."

R6. Output must be zero (only) in the case where either first or second input integer is zero.

Test Case Selection

Test Cases for Each Requirement

- We model each partitioned requirement as independent
- We create separate test cases for each partitioned requirement

Example: For the partitioned requirement: "*If either x or y is zero, then output zero.*" - R6. Output must be zero (only) in the case where either first or second input integer Is zero.

We might choose the test cases:

R6T1.	0	0	(both zero)
R6T2.	0	1	(x zero, y not)
R6T3.	1	0	(y zero, x not)
R6T4.	1	1	(neither zero)

Test Case Selection

Notice these test inputs are the **simplest possible** and make no attempt to be exhaustive (more on this later)

Example: For the partitioned requirement: "*If either x or y is zero, then output zero.*" - R6. Output must be zero (only) in the case where either first or second input integer Is zero.

We might choose the test cases:

- R6T1. 0 0 (*both zero*)
- R6T2. 0 1 (*x zero, y not*)
- R6T3. 1 0 (*y zero, x not*)
- R6T4. 1 1 (*neither zero*)

A Systematic Method

Black Box Functionality Coverage

- Functionality coverage gives us a **system** for creating functionality test cases
- It tells us when we are **done** (i.e., when we have test cases for every partitioned requirement)
- But notice this is **not** the same as **acceptance** testing – because it treats functional requirements as if they were completely **separate**, when in fact they are tightly **related**
 - So it does not replace acceptance testing, we (or the customer) must do that as well
 - Unlike acceptance testing, it is a **systematic** method, but like other systematic methods, it is only a **partial** test



Choosing Test Inputs

An Experiment

- Black box testing performs an experiment on the software system
 - We have a hypothesis that the software has certain properties, and we test the hypothesis with our test cases
 - We then observe the results and draw conclusions, in classic scientific method style

Choosing Test Inputs

Experimental Design

- A principle of experimental design is the isolation of “variables”
- This refers to the fact that we should design the experiment such that each possible cause that may affect the outcome (each experimental “variable”) can be observed independently
- Thus when an effect is observed we can tell which cause is at work
- The usual way to do this is to design the experiment in steps that only vary one “variable” (possible cause) at a time, keeping everything else constant

Guidelines for Choosing Test Inputs

Choosing Inputs

- For test inputs, this principle means that we should help isolate failure causes, by as much as possible:
 - (1) Consistently choosing the **simplest** input values possible, in order not to introduce arbitrary variations
 - (2) Keeping everything **constant** between test cases, varying only **one input value at a time** (don't try to be "clever" introducing random input variations)
- These principles hold for **all** systematic test methods, not just this one

Functionality Coverage

Functionality Coverage Methods Review

- Functionality coverage partitions the functional specification into separate requirements to test
- Isolate causes by keeping test input values simple and varying one input value at a time



Input Coverage Testing

Input Coverage

- The second kind of **black box** testing
- **Idea:** Analyze all the possible inputs allowed by the **functional specifications** (requirements), create test sets based on the analysis
- Input coverage methods:
 - **exhaustive**, input **partitioning**, **shotgun**, (**robustness**) **boundary**
- **Objective:** Show software correctly handles **all** allowed **inputs**
- **Question:** What does “**all**” mean?



Exhaustive Testing

What does “all” mean?

- Ideally, input coverage testing should try **every possible** input to the program
- This is called **exhaustive** testing
- Involves testing the program with **every possible** input – yields a strong result: virtually **certain** that the program is correct
- Easy **system** for test cases, and obvious when **done**
- But usually **impractical**, even for very small programs



Input Partition Testing

Input Partitioning

- However, cases where exhaustive testing is practical are **extremely** rare
- So we must choose another way to decide when we are done testing inputs
- The most common way is to **partition** all the possible inputs into **equivalence classes** which characterize sets of inputs with something in common

Example: Recall our example gcd functional specification.

"Given as input two integers x and y , output all the numbers smaller than or equal to x that are evenly divisible by y . If either x or y is zero, then output zero."



Input Partition Testing

Input Partitioning

- The gcd functional specification identifies three special cases for input:
 - the case where $x=0$,
 - the case where $y=0$, and
 - the case where neither is zero
- Since the input set is to be **integers**, we can further partition into **negative** and **positive** cases for x and y , giving us the set of input partitions on the next slide



Input Partition Testing

Example:

Partition	x input	y input
P1	0	<i>non-zero</i>
P2	<i>non-zero</i>	0
P3	0	0
P4	<i>less than zero</i>	<i>less than zero</i>
P5	<i>less than zero</i>	<i>greater than zero</i>
P6	<i>greater than zero</i>	<i>less than zero</i>
P7	<i>greater than zero</i>	<i>greater than zero</i>

Input Partition Testing

Covering Partitions

- The partitions give us our **test cases** – all we must do now is design test cases to cover each partition
- For the reasons we saw last time, for each case we choose the **simplest** input values and vary them **as little as possible**



Input Partition Testing

Advantages of Input Partition Testing

- Input partitioning is what many of us think of **intuitively** for testing – that is, test the response to each kind of input
- It is generally **easy** to identify a set of partitions given the functional specification (although it may require **insight**)
- It is **easy** to say when we are **done** (when we have run at least one representative test for each partition)
- It gives us confidence that the program is at least **capable** of handling (one example of) each different **kind** of input correctly



Black Box Shotgun Testing

Shotgun Testing

- Black box **shotgun** testing consists of choosing **random** values for inputs (with or without worrying about legality) over a large number of test runs
- We then verify that the outputs are correct for the **legal** inputs, and that program simply did not crash for **illegal** ones
- More practically, we usually choose inputs from the **legal** set and inputs from the **illegal** set as **separate** sets of shotgun tests



Black Box Shotgun Testing

Shotgun Testing

Example:

<u>Test</u>	<u>x input</u>	<u>y input</u>
T1	375	15554
T2	-76	1763
T3	273334	-9762
... and so on ...		



Black Box Shotgun Testing

Systematic?

- Black box **shotgun** testing is still a black box method - we don't need the code to invent appropriate inputs at random
- But it is not really systematic – although there is a **system** for choosing test cases (randomly), there is no **completion criterion**
- So to gain any confidence, we must run a **very large** number of test cases



Input Partition Shotgun Testing

A Hybrid Method

- Shotgun testing is nevertheless interesting, because it tries lots of different inputs
- We can use it to strengthen [input partition testing](#) by applying the shotgun method to choose random input values within each partition
 - That way we both have the confidence of input partition testing and the [additional](#) confidence that our simple input values are not the only ones that work
 - However, we [still](#) need automated output verification to be practical
 - And we must be careful about [experimental design](#) – we should run the ordinary simple input partition tests first, [then](#) load the shotgun



Input Robustness Testing

Robustness

- Robustness is the property that a program doesn't crash or halt unexpectedly, no matter what the input
 - Robustness testing tests for this property
- Two kinds of robustness testing:
 - (1) Shotgun robustness testing (random garbage input)
 - (2) Boundary value robustness testing



Input Boundary Testing

Boundary Values

- Even when programs behave well with input values well outside their expected range, it is typical that failures come at the **boundaries** of the legal or expected range of values

Example:

If a sort program expects a list of numbers to sort, it often fails with lists of length **one** or **zero**, and with lists exactly as large as the largest allowed (the **end of array problem**)



Input Boundary Testing

Boundary Values

- For this reason, black box testers often create **boundary value** tests to check that the program is robust with inputs on the edge
- Unlike shotgun testing, boundary value testing is a **systematic** test method, because it has both
 - an easy way to **choose** test cases, and
 - an easy way to know when we are **done** (when all boundary values have been tested)



Input Coverage Methods

Input Coverage Methods Review

- **Exhaustive** testing is usually impractical, but we can approximate it using **input partitioning**
- **Shotgun** testing can be added to input partitioning to give additional confidence
- **Robustness** testing checks for crashes on unexpected or unusual input, such as the boundaries of the input range



Overview

- Today we look at the third kind of black box method, **output coverage testing**, and begin to consider the role of black box methods in **unit** and **integration** testing
- We'll look at:
 - **Exhaustive** output testing
 - **Output partitioning**
- Testing multiple input or output streams
- Black box testing at the unit and integration levels (“**gray box**” testing)



Output Coverage Testing

Output Coverage

- The third kind of **black box** testing
- **Idea:** Analyze all the possible **outputs** specified in the **functional specification** (requirements), create tests to cause each one
- More **difficult** than input coverage: must analyze requirements to figure out what **input** is required to produce each **output** – this can be a complex and time consuming analysis
- **But** can be very effective in finding problems, because it requires a **deep understanding** of the requirements



Output Coverage Testing

Different from Input Coverage

- Output coverage testing is definitely different from input coverage

Example: suppose the requirements say: “*Output 1 if two input integers are equal, 0 otherwise*”

This specification allows two integer inputs - so if we do **input partitioning**, then we have the test cases:

- numbers equal
- numbers not equal
- first number zero / positive / negative
- second number zero / positive / negative

Whereas we can do **exhaustive** output testing with only **two** test cases
– output **1** & output **0**



Exhaustive Output Testing

More Practical than Input

- **Exhaustive** output testing makes one test for every possible output
- Practical more often than exhaustive **input** testing, because programs are often written to **reduce** or **summarize** input data (like the previous example)
- But still impractical in general - **most** programs have an infinite number of different possible outputs



Output Partition Testing

Output Partitioning

- Output **partitioning** is like input partitioning, only we analyze the possible **outputs**
- In a fashion similar to input partitioning, we partition all the possible **outputs** into a set of **equivalence classes** which have something in common



Output Partition Testing

Designing Inputs

- Once we have the output partitions, we must design **inputs** to cause outputs in each class
- This is a difficult and time consuming task - the biggest drawback to output coverage testing
- Sometimes, we discover that we cannot find such an input – this implies an error or oversight in either the **requirements** or in the **partition analysis**



“Gray” Box Testing

If We Already Have a Design ...

- If we allow ourselves the luxury of waiting until we have a architectural (**class** level) design, or even a detailed (**method** level) design, then we can use black box testing at each of those levels as well
- Since we can see a **part** of the software (its design), black box testing at these levels is not really “pure” black box – for that reason it is sometimes called “gray box” testing

Gray Box
Testing

Black Box Testing of Single Method

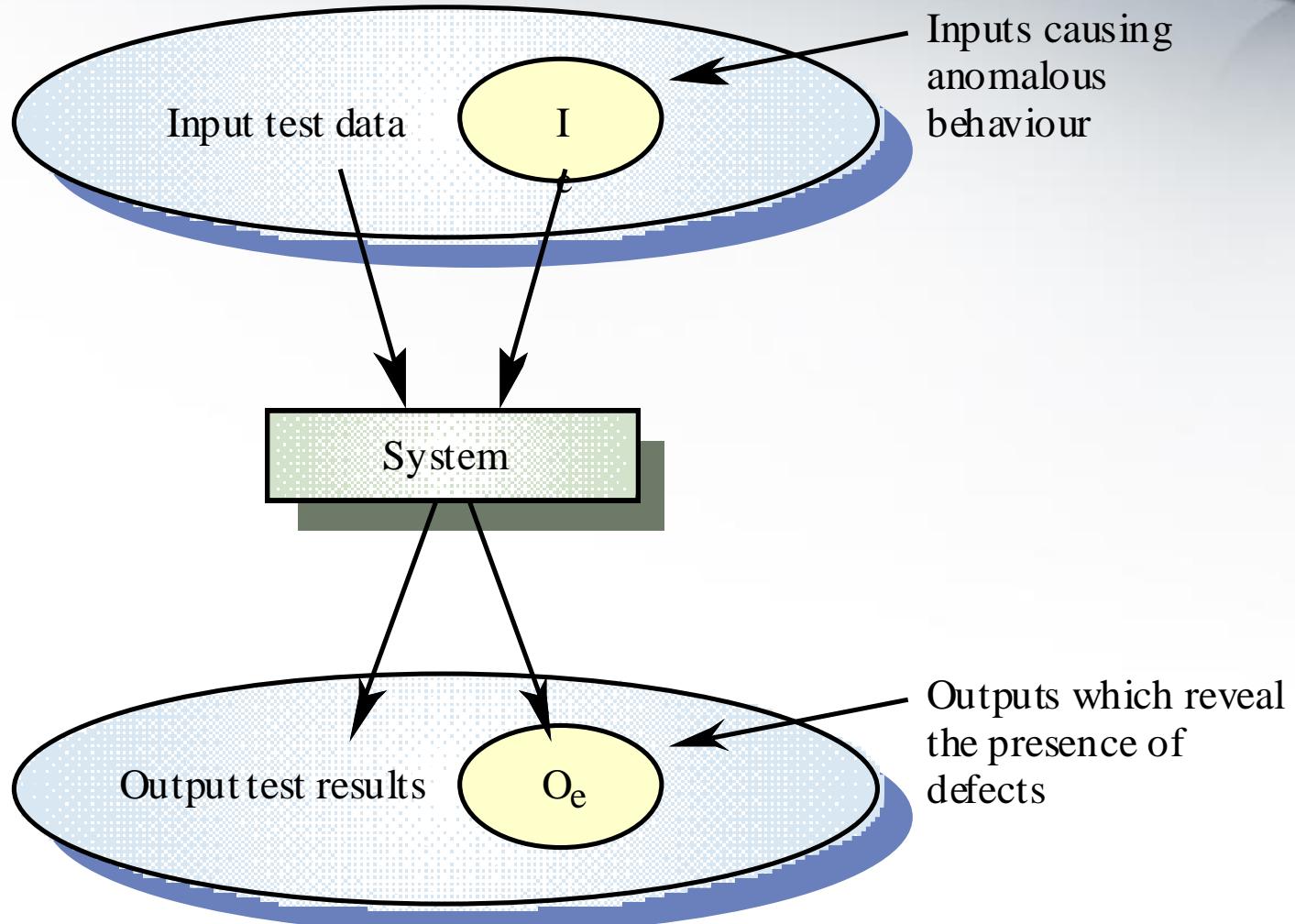
Method Testing

- To use black box testing on code units, we need to figure out what corresponds to requirements, inputs and outputs
- If the unit we are testing is a method (procedure or function), then:
 - The "requirements" are the specification of the method
 - The "input" is the value of parameters and global environment variables used by the method
 - The "output" is the value of return values, global environment variables, and exceptions thrown (together these are referred to as the "outcome" in unit testing)
- Once we know these, test cases can be created according to any of the black box testing criteria :
 - Functionality coverage, input coverage or output coverage



BLACK BOX TESTING

Black-box Testing



Black-box Testing



- Black-box testing, also called behavioral testing, focuses on the functional requirements of the software.
- That is, black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program.

Black-box Testing



- Black-box testing is not an alternative to white-box techniques.
- Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods.

Black-box Testing



- Black-box testing attempts to find errors in the following categories:
 - (1) incorrect or missing functions,
 - (2) interface errors,
 - (3) errors in data structures or external data base access,
 - (4) behavior or performance errors, and
 - (5) initialization and termination errors.

Black-box Testing



- Unlike white-box testing, which is performed early in the testing process, black-box testing tends to be applied during later stages of testing.
- Because black-box testing purposely disregards control structure, attention is focused on the information domain.

Black-box Testing



- Tests are designed to answer the following questions:
 - How is functional validity tested?
 - How is system behavior and performance tested?
 - What classes of input will make good test cases?
 - Is the system particularly sensitive to certain input values?
 - How are the boundaries of a data class isolated?
 - What data rates and data volume can the system tolerate?
 - What effect will specific combinations of data have on system operation?

Graph-Based Testing Methods



- The first step in black-box testing is to understand the objects that are modeled in software and the relationships that connect these objects.
- Once this has been accomplished, the next step is to define a series of tests that verify “all objects have the expected relationship to one another [BEI95].”

Graph-Based Testing Methods

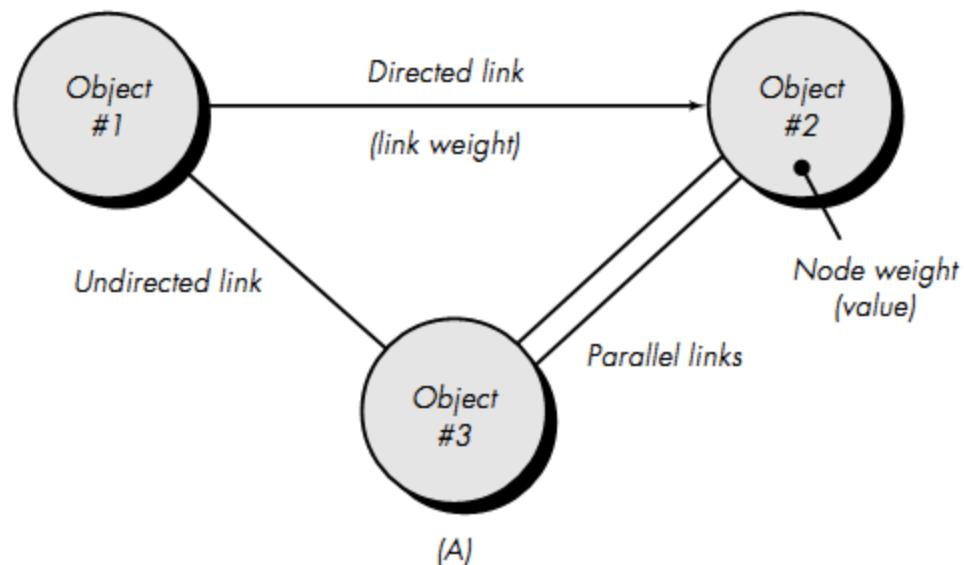


- Stated in another way, software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.

Graph-Based Testing Methods



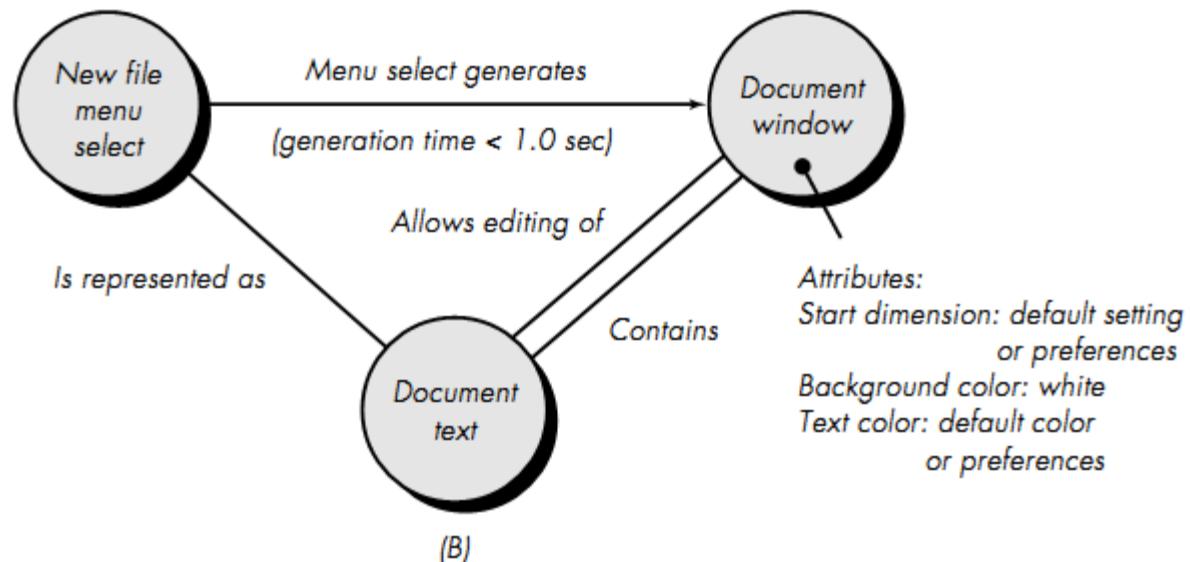
- Nodes are represented as circles connected by links that take a number of different forms. A directed link (represented by an arrow) indicates that a relationship moves in only one direction. A bidirectional link, also called a symmetric link, implies that the relationship applies in both directions. Parallel links are used when a number of different relationships are established between graph nodes.



Graph-Based Testing Methods



- Object #1 = new file menu select
- Object #2 = document window
- Object #3 = document text



Transaction flow modeling



- The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an on-line service), and the links represent the logical connection between steps.
- The data flow diagram can be used to assist in creating graphs of this type.

Finite state modeling



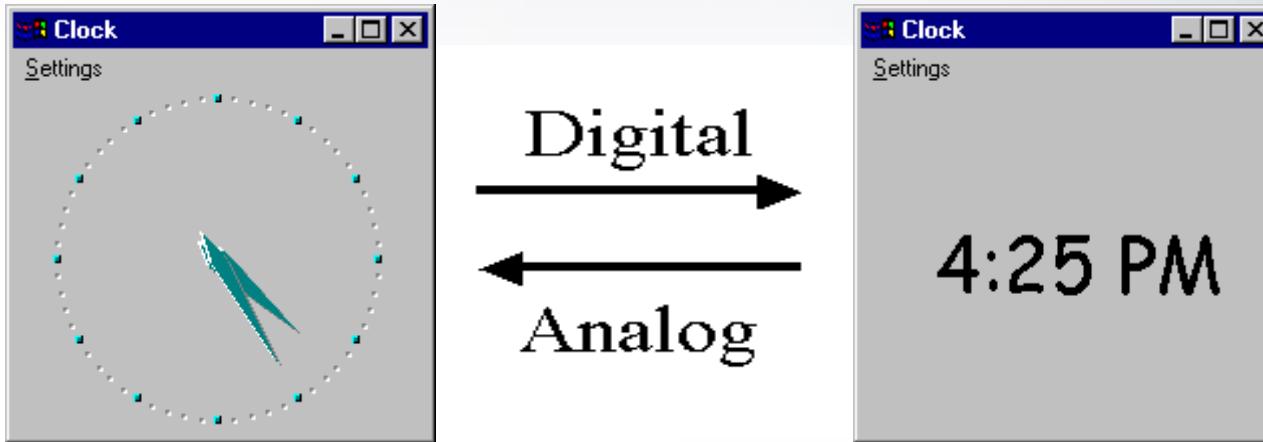
- The nodes represent different user observable states of the software (e.g., each of the “screens” that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state.

Timing modeling



- The nodes are program objects and the links are the sequential connections between those objects.
- Link weights are used to specify the required execution times as the program executes.

Example: A Very Simple Finite State Model of the Clock



Test cases



- We could use this very simple state model as a basis for tests, where following a path in the model is equivalent to running a test:
 - Setup:
 - Put the Clock into its Analog display mode
 - Action:
 - Click on “Settings\Digital”
 - Outcome:
 - Does the Clock correctly change to the Digital display?

Actions



- the following actions in the Clock:
 - **Start** the Clock application
 - **Stop** the Clock application
 - Select **Analog** setting
 - Select **Digital** setting

The rules for actions



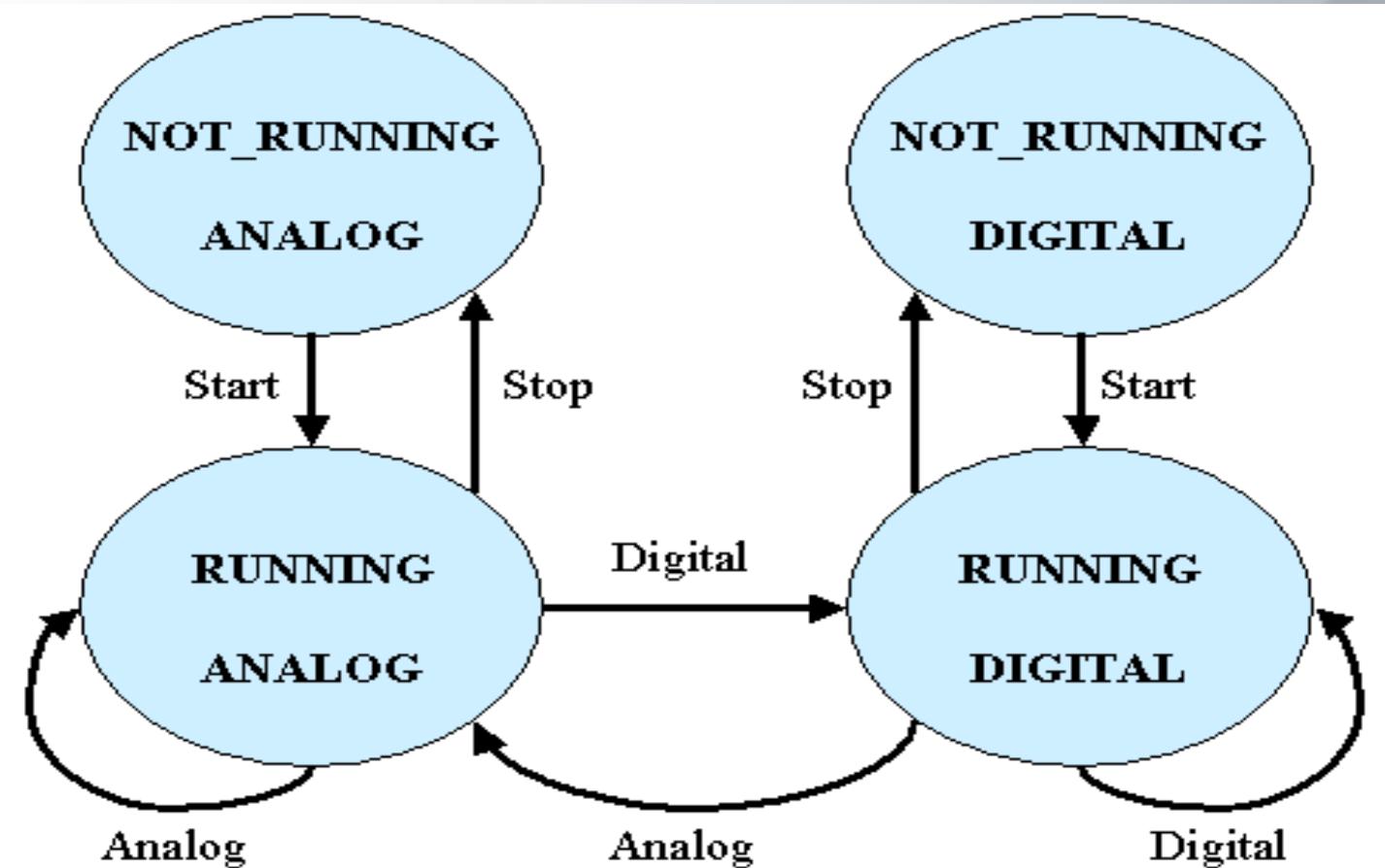
- The rules for these actions in the Clock application are as follows:
- **Start**
 - If the application is NOT running, the user can execute the **Start** command.
 - If the application is running, the user cannot execute the **Start** command.
 - After the **Start** command executes, the application is running.
- **Stop**
 - If the application is NOT running, the user cannot execute the **Stop** command.
 - If the application is running, the user can execute the **Stop** command.
 - After the **Stop** command executes, the application is not running.
-
- **Analog**
 - If the application is NOT running, the user cannot execute the **Analog** command.
 - If the application is running, the user can execute the **Analog** command.
 - After the **Analog** command executes, the application is in Analog display mode.
- **Digital**
 - If the application is NOT running, the user cannot execute the **Digital** command.
 - If the application is running, the user can execute the **Digital** command.
 - After the **Digital** command executes, the application is in Digital display mode.

Operational modes



- Two modes of operations:
 - system mode
 - NOT_RUNNING means Clock is not running
 - RUNNING means Clock is running
 - setting mode
 - ANALOG means Analog display is set
 - DIGITAL means Digital display is set

State Transition Diagram for the Clock Model



Types of Testing



- **Exhaustive testing**
- **Boundary value analysis**
- **Test generation from predicates**
- **Web Sites Testing**
- **Grey Box Testing**
- **GUI Testing**
- **System Testing**
- **Integration Testing**
- **Configuration Testing**

White Box Testing

Software quality assurance
and testing in
Structured programming

Structured Programming

- **structured programming:** A technique for organizing and coding computer programs in which a hierarchy of modules is used.
- Each having a single entry and a single exit point, and in which control is passed downward through the structure.

STRUCTURED PROGRAMMING

- In structured Programming each construct has a predictable logical structure.
- This could enabling a reader to follow procedural flow more easily.
- The constructs are sequence, condition, and repetition.

STRUCTURED PROGRAMMING

- Sequence implements processing steps that are essential in the specification of any algorithm.
- Condition provides the facility for selected processing based on some logical occurrence.
- Repetition allows for looping.
- These three constructs are fundamental to structured programming—an important component-level design technique.

STRUCTURED PROGRAMMING

- The structured constructs were proposed to limit the procedural design of software to a small number of predictable operations.
- Complexity metrics indicate that the use of the structured constructs reduces program complexity and thereby enhances readability, testability, and maintainability.

STRUCTURED PROGRAMMING

- The use of a limited number of logical constructs also contributes to a human understanding process that is known as chunking.
- To understand this process, consider the way in which you are reading this page.
- You do not read individual letters but rather recognize patterns or chunks of letters that form words or phrases.

STRUCTURED PROGRAMMING

- The structured constructs are logical chunks that allow a reader to recognize procedural elements of a module, rather than reading the design or code line by line.
- Understanding is enhanced when readily recognizable logical patterns are encountered.

STRUCTURED PROGRAMMING

- Any program, regardless of application area or technical complexity, can be designed and implemented using only the three structured constructs.
- It should be noted, however, that strict use of only these constructs can sometimes cause practical difficulties.

Advantages of Structured Programming

- **Easy to write:**

Modular design increases the programmer's productivity by allowing them to look at the big picture first and focus on details later.

Several Programmers can work on a single, large program, each working on a different module

Studies show structured programs take less time to write than standard programs.

Procedures written for one program can be reused in other programs requiring the same task. A procedure that can be used in many programs is said to be **reusable**

Advantages of Structured Programming

- **Easy to debug**

Since each procedure is specialized to perform just one task, a procedure can be checked individually. Older unstructured programs consist of a sequence of instructions that are not grouped for specific tasks. The logic of such programs is cluttered with details and therefore difficult to follow.

Easy to Understand

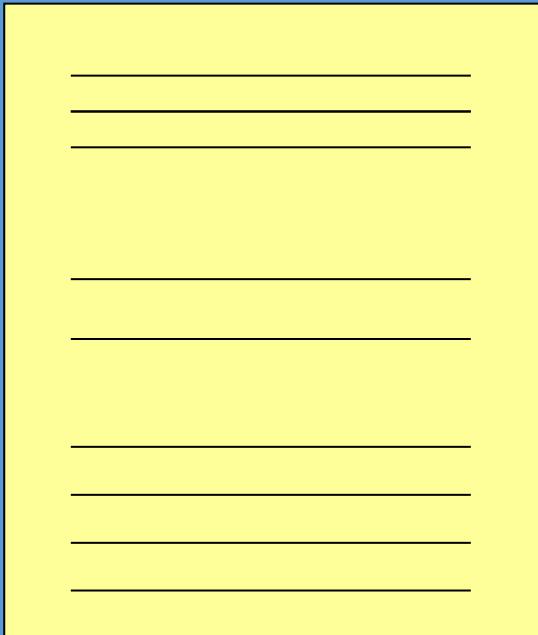
The relationship between the procedures shows the modular design of the program.

Easy to Change

Since a correctly written structured program is self-documenting, it can be easily understood by another programmer.

Unstructured programming

All the program code written in a single continuous *main program*.



Many disadvantages for large programs

- difficult to follow logic
- if something needs to be done more than once must be re-typed
- hard to incorporate other code
- not easily modified
- difficult to test particular portions of the code
- ...

Graphical Design Notation

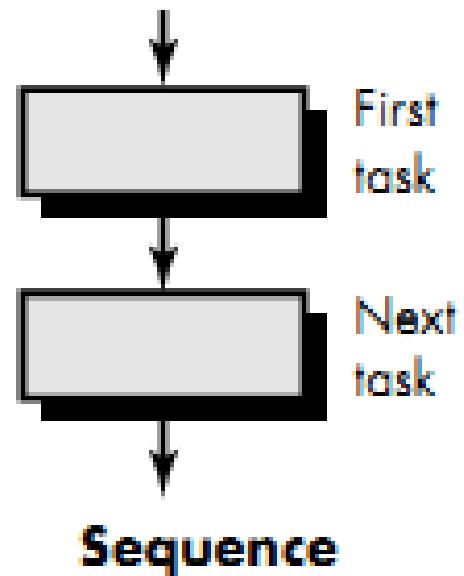
- "A picture is worth a thousand words," but it's rather important to know which picture and which 1000 words.
- There is no question that graphical tools, such as the flowchart or box diagram, provide useful pictorial patterns that readily depict procedural detail.
- However, if graphical tools are misused, the wrong picture may lead to the wrong software.

Graphical Design Notation

- A flow chart is quite simple pictorially.
- A box is used to indicate a processing step.
- A diamond represents a logical condition, and arrows show the flow of control.

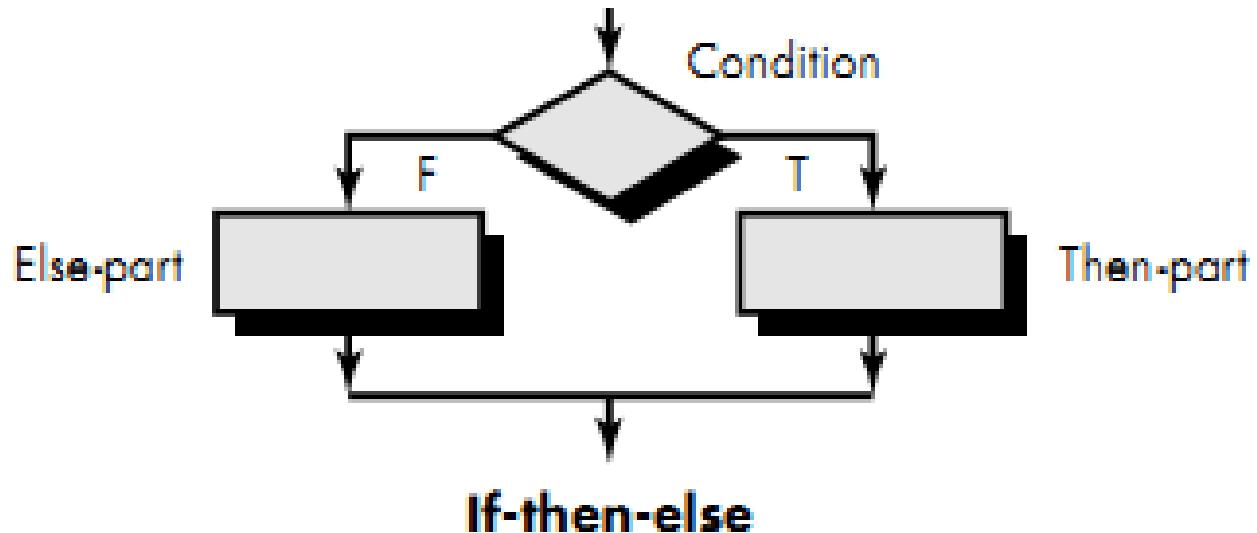
Graphical Design Notation

- The sequence is represented as two processing boxes connected by an line (arrow) of control.



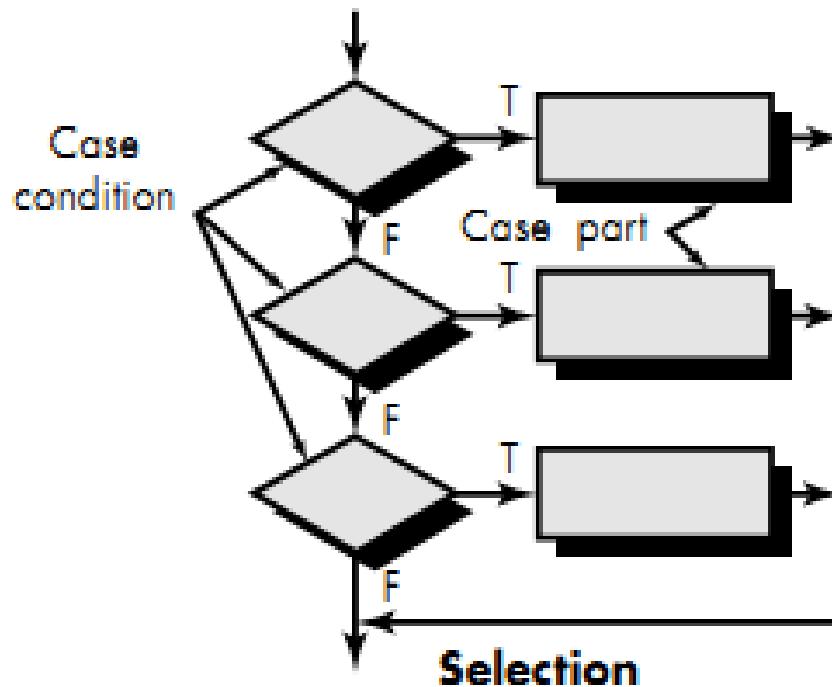
Graphical Design Notation

- Condition, also called if-then-else, is shown as a decision diamond that if true, causes then-part processing to occur, and if false, invokes else-part processing.



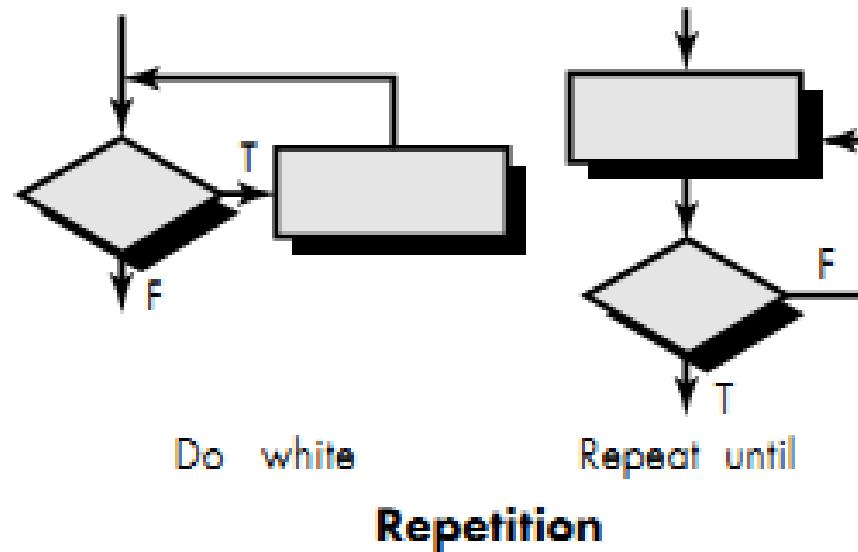
Graphical Design Notation

- The selection (or select-case) construct shown in the figure is actually an extension of the if-then-else. A parameter is tested by successive decisions until a true condition occurs and a case part processing path is executed.



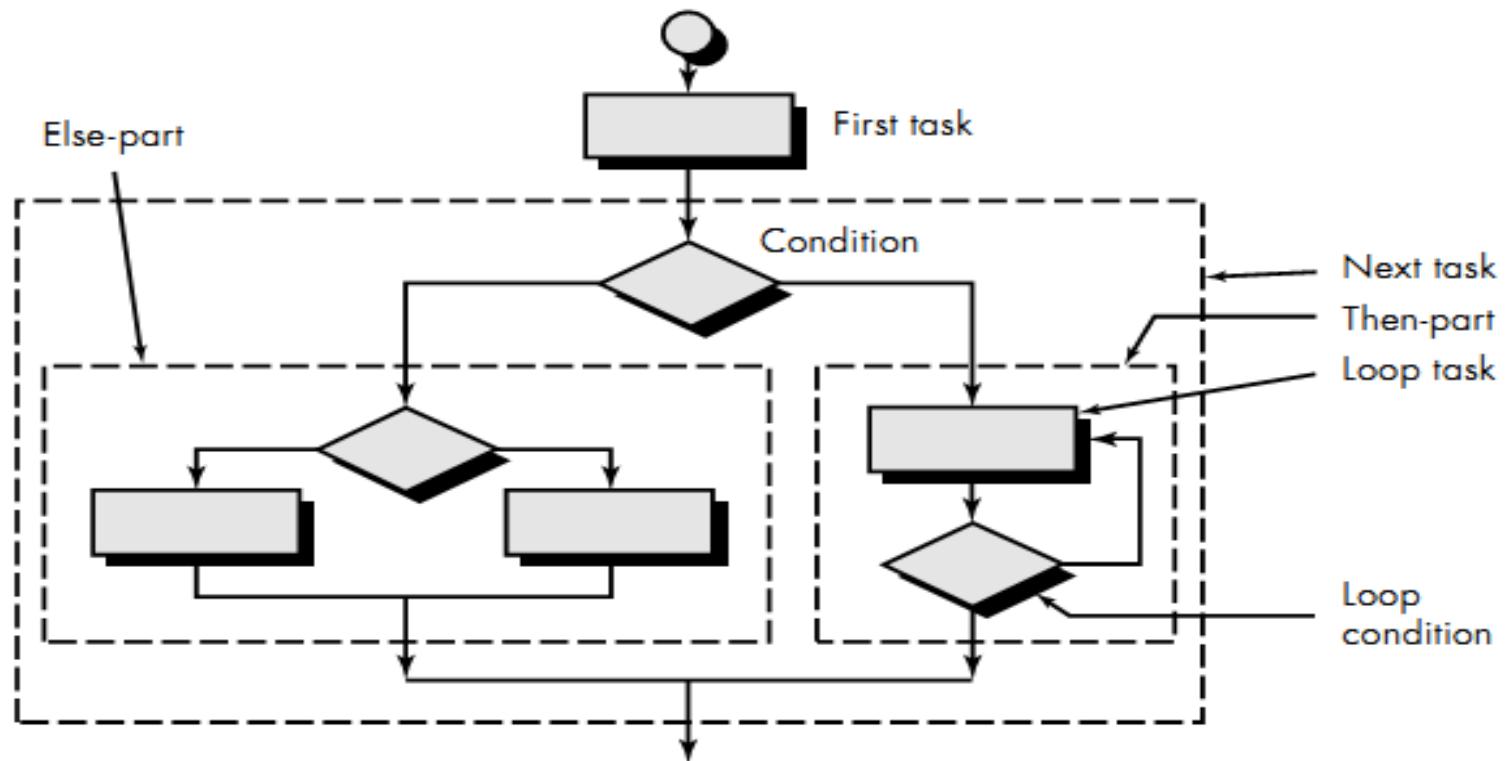
Graphical Design Notation

- Repetition is represented using two slightly different forms. The do while tests a condition and executes a loop task repetitively as long as the condition holds true.
- A repeat until executes the loop task first, then tests a condition and repeats the task until the condition fails.



Graphical Design Notation

- The structured constructs may be nested within one another as shown in Figure



Structure Programming and testing

- (1) guarantee that all independent paths within a module have been exercised at least once,
- (2) exercise all logical decisions on their true and false sides,
- (3) execute all loops at their boundaries and within their operational bounds, and
- (4) exercise internal data structures to ensure their validity.

Structure Programming and testing

- Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed.
- Typographical errors are random. When a program is translated into programming language source code, it is likely that some typing errors will occur.

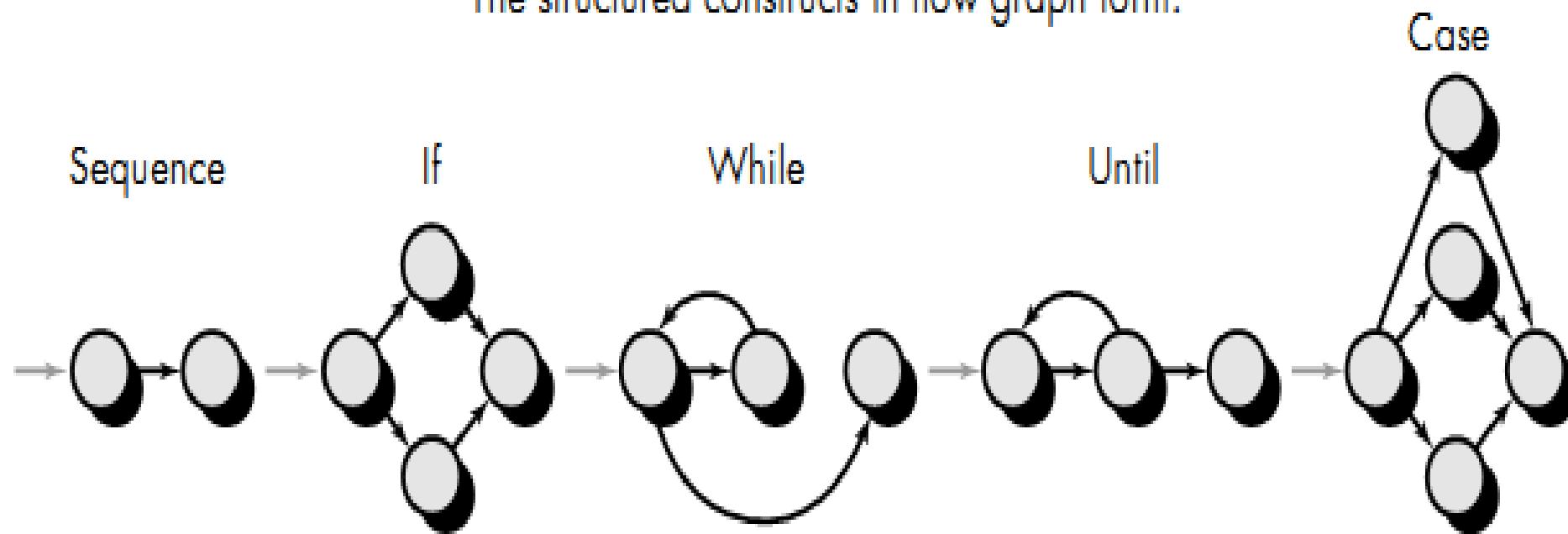
Structure Programming and testing

- **BASIS PATH TESTING :** Test cases derived to check the execution of every statement in the program at least one time during testing.
 - Flow Graph Notation
 - Cyclomatic Complexity
 - Deriving Test Cases
 - Graph Matrices

Flow Graph Notation

- Draw a flow graph when the logical control structure of a module is complex. The flow graph enables you to trace program paths more readily.

The structured constructs in flow graph form:



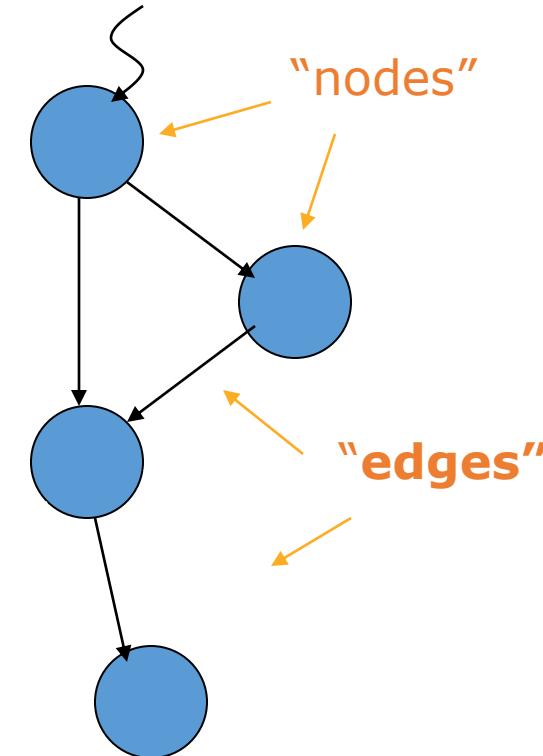
Types of Logic Coverage

- **Statement:** each statement executed at least once
- **Branch:** each branch traversed (and every entry point taken) at least once
- **Condition:** each condition True at least once and False at least once
- **Branch/Condition:** both Branch and Condition coverage achieved

Types of Logic Coverage (cont'd)

- **Compound Condition:** all combinations of condition values at every branch statement covered (and every entry point taken)
- **Path:** all program paths traversed at least once

Control Flow Graphs



Statement Coverage

- Statement Coverage requires that each statement will have been executed at least once.
- Simplest form of logic coverage.
- Also known as Node Coverage.

Statement Coverage

- Statement should be executed completely or not at all.
- For example:
- IF a THEN b ENDIF
- is considered as more than 1 statement since b may or may not be executed depending on the condition a.

Branch Coverage

- Branch Coverage requires that each branch will have been traversed, and that every program entry point will have been taken, at least once.
- Also known as *Edge Coverage*.

Branch Coverage

- Attempting to cover all the paths in the software is called **path coverage**
- The simplest form of path testing is called *branch testing*

```
Print "Hello World"
IF Date == "01-01-2005" THEN
    Print "Happy New Year"
ELSE
    Print "Have a good day!"
END IF
Print "The date is: "+Date
Print "The time is: "+Time
END
```

-
- To test each branch you would need to execute 2 test cases, one with the date set to 1st Jan 2005, and another with any other date.

Condition Coverage

- A branch predicate may have more than one condition.

```
input(X,Y)
if (Y<=0) or (X=0) then
    Y := -Y
end_if
while (Y>0) and (not EOF) do
    input(X)
    Y := Y-1
end_while
```

Condition Coverage (cont'd)

- Condition Coverage requires that each condition will have been True at least once and False at least once.

Condition Coverage

```
Print "Hello World"  
IF Date == "01-01-2005" AND Time =="00:00:00" THEN  
    Print "Happy New Year"  
ELSE  
    Print "Have a good day!"  
END IF  
Print "The date is: "+Date  
Print "The time is: "+Time  
END
```

-
- Condition Coverage takes extra conditions into consideration
 - To achieve full condition coverage you would need 4 test cases

Branch/Condition Coverage

- **Branch/Condition Coverage requires that both Branch AND Condition Coverage will have been achieved.**
- Therefore, Branch/Condition Coverage subsumes both Branch Coverage and Condition Coverage.

Compound Condition Coverage (cont'd)

- Compound Condition Coverage requires that all *combinations* of condition values at every branch statement will have been covered, and that every entry point will have been taken, at least once.
- Also know as *Multiple Condition Coverage*.
- Subsumes Branch/Condition Coverage, regardless of the order in which conditions are evaluated.

Path Coverage

- **Path Coverage requires that all program paths will have been traversed at least once.**
- Often described as the “strongest” form of logic coverage?
- Path Coverage is usually impossible when loops are present. (How many test cases would be required to cover all paths in the example below?)

Loop Coverage

- **Loop Coverage requires that the body of loops be executed 0, 1, 2, t, max, and max+1 times, where possible.**

Loop Coverage (cont'd)

- Rationale: (cont'd)
 - Check loop re-initialization.
 - Check *typical* number of iterations.
 - Check upper (valid) bound on number of times body may be executed.
 - If the maximum can be exceeded, what behavior results?

Flow Graph Notation

- Each circle, called a flow graph node, represents one or more procedural statements.
- A sequence of process boxes and a decision diamond can map into a single node.
- The arrows on the flow graph, called edges or links, represent flow of control and are analogous to flowchart arrows.

Flow Graph Notation

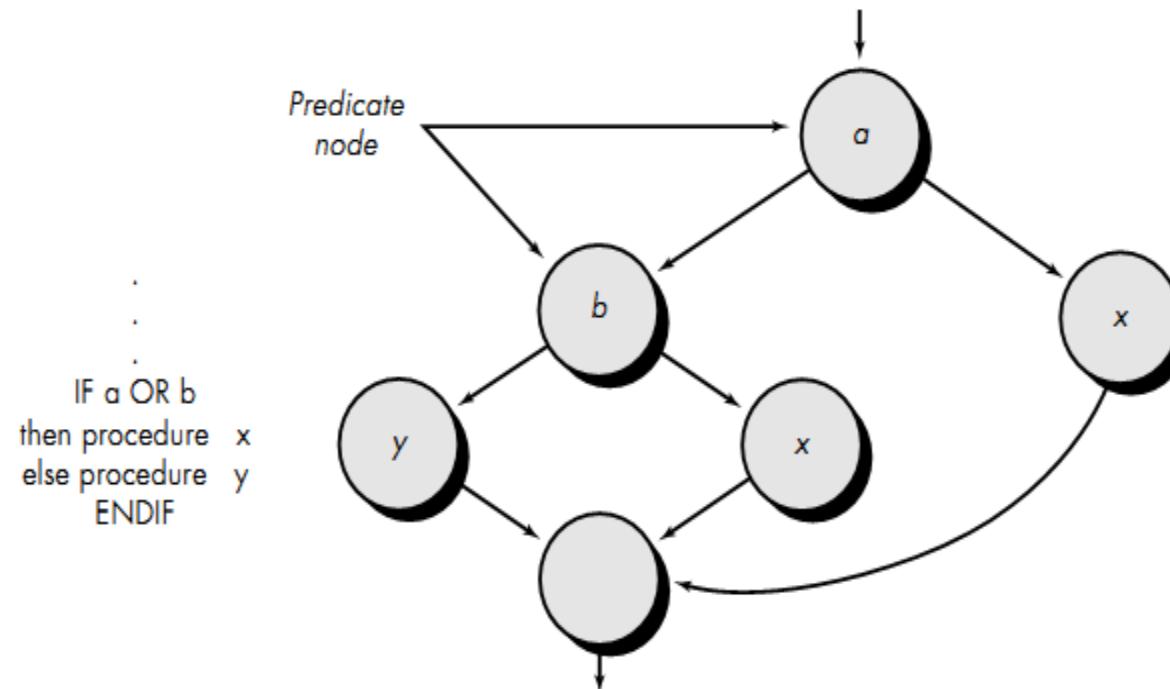
- An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the symbol for the if-then-else construct).
- Areas bounded by edges and nodes are called regions.
- When counting regions, we include the area outside the graph as a region.

Flow Graph Notation

- When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated.
- A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement.

Flow Graph Notation

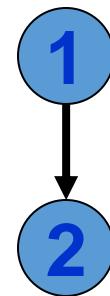
- Note that a separate node is created for each of the conditions a and b in the statement IF a OR b.
- Each node that contains a condition is called a predicate node and is characterized by two or more edges emanating from it.



How to draw Control flow graph?

- Sequence:

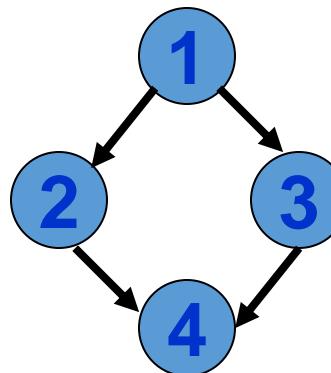
- 1 a=5;
- 2 b=a*b-1;



How to draw Control flow graph?

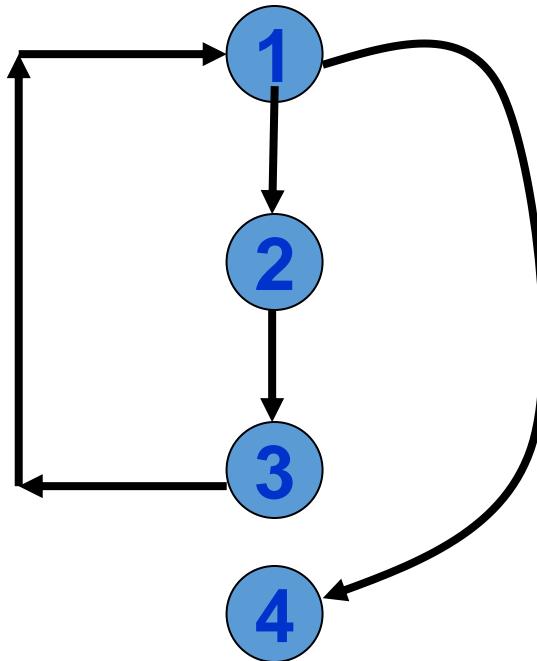
- Selection:

- 1 if($a > b$) then
- 2 $c = 3;$
- 3 else $c = 5;$
- 4 $c = c * c;$



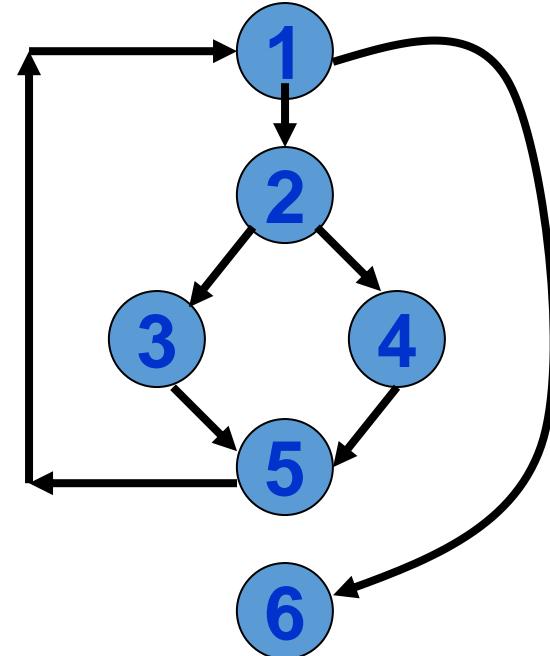
How to draw Control flow graph?

- Iteration:
 - 1 while($a > b$) {
 - 2 $b = b * a$;
 - 3 $b = b - 1$;
 - 4 $c = b + d$;



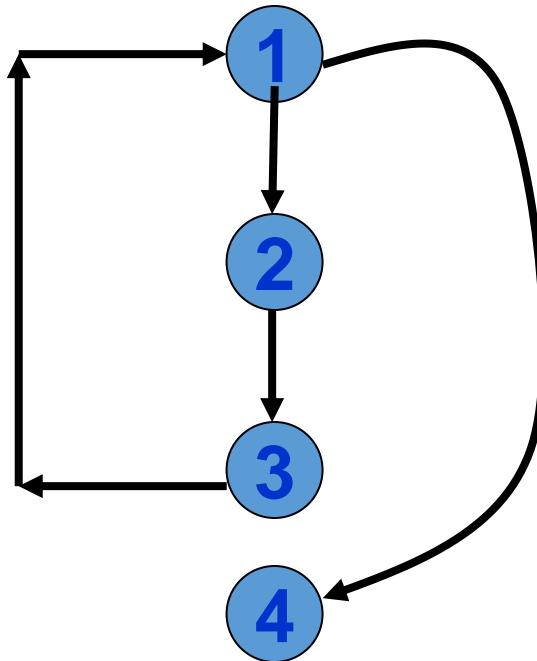
Example

- int f1(int x,int y){
- 1 while (x != y){
- 2 if (x>y) then
- 3 x=x-y;
- 4 else y=y-x;
- 5 }
- 6 return x; }



How to draw Control flow graph?

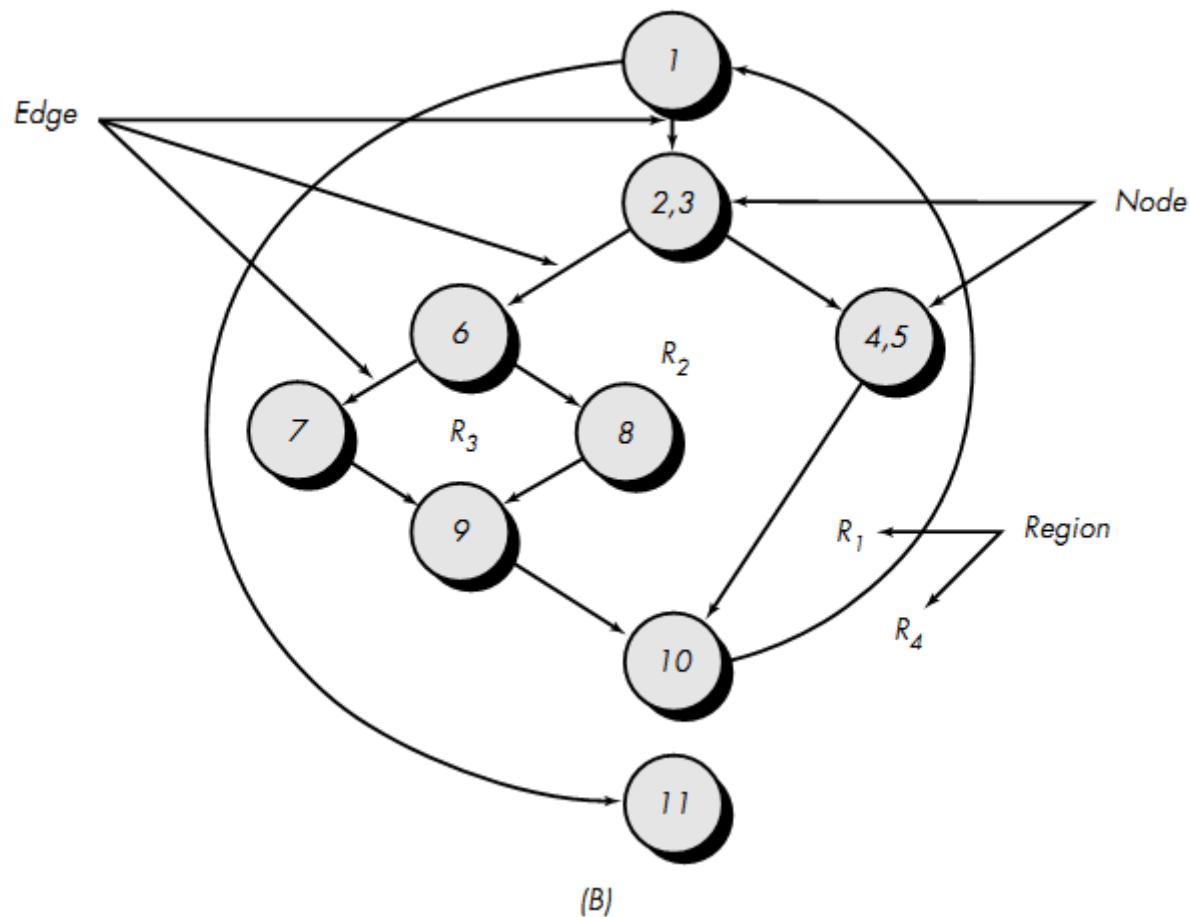
- Iteration:
 - 1 while($a > b$) {
 - 2 $b = b * a;$
 - 3 $b = b - 1;$ }
 - 4 $c = b + d;$



Cyclomatic Complexity

- Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program.
- It defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

Cyclomatic Complexity



- path 1: 1-11
- path 2: 1-2-3-4-5-10-1-11
- path 3: 1-2-3-6-8-9-10-1-11
- path 4: 1-2-3-6-7-9-10-1-11

Complexity is computed in one of three ways

- The number of regions of the flow graph correspond to the cyclomatic complexity.
- Cyclomatic complexity, $V(G)$, for a flow graph, G , is defined as

$$V(G) = E - N + 2$$

- where E is the number of flow graph edges, N is the number of flow graph nodes.

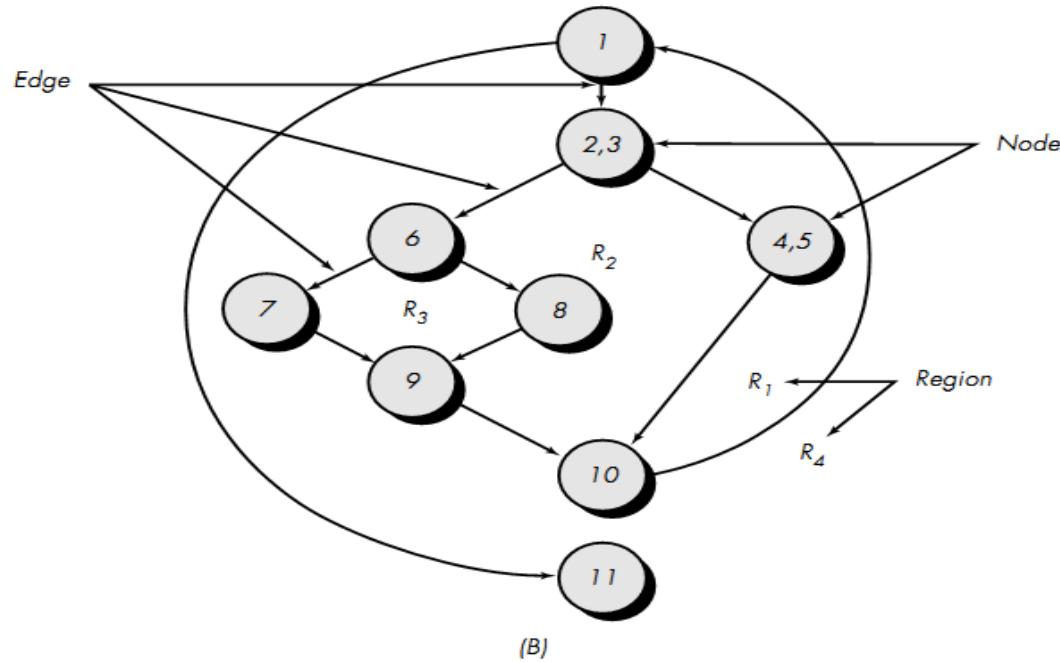
Complexity is computed in one of three ways

- Cyclomatic complexity, $V(G)$, for a flow graph, G , is also defined as

$$V(G) = P + 1$$

- where P is the number of predicate nodes contained in the flow graph G .

Complexity is computed in one of three ways



1. The flow graph has four regions.
2. $V(G) = 11 \text{ edges} - 9 \text{ nodes} + 2 = 4$.
3. $V(G) = 3 \text{ predicate nodes} + 1 = 4$.

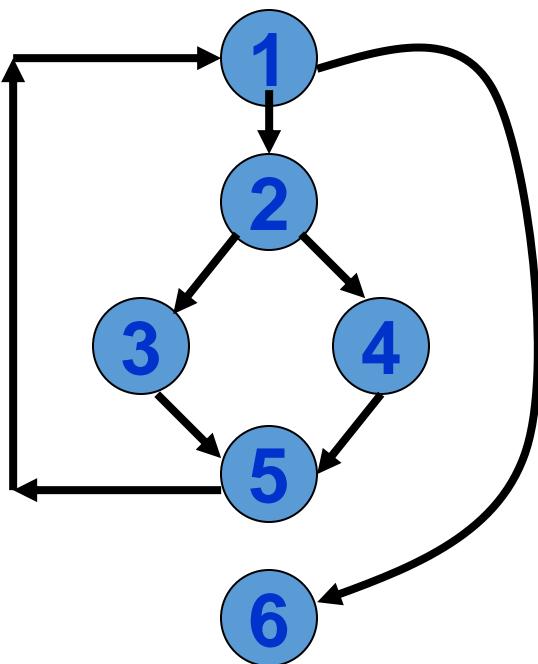
How to decide no. of test cases

- More important, the value for $V(G)$ provides us with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

How to derive test cases

- Using the design or code as a foundation, draw a corresponding flow graph.
- Determine the cyclomatic complexity of the resultant flow graph.
- Determine a basis set of linearly independent paths.
- Prepare test cases that will force execution of each path in the basis set.

Example Control Flow Graph

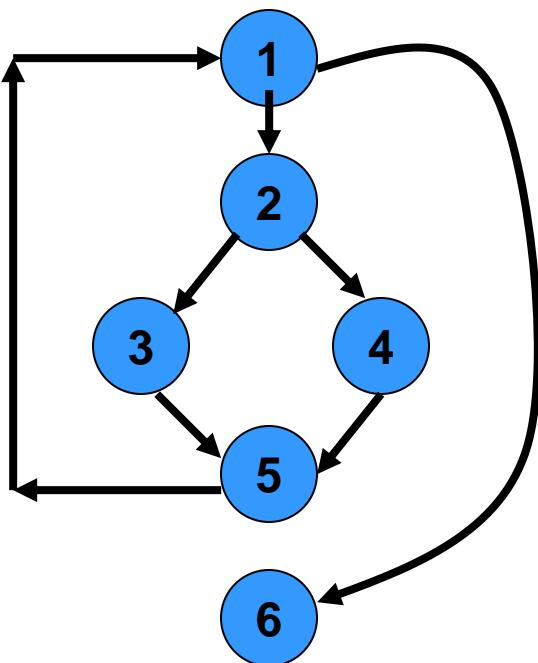


**Cyclomatic complexity = E-N+2
7-6+2 = 3.**

Cyclomatic complexity

- Another way of computing cyclomatic complexity:
 - inspect control flow graph
 - determine number of bounded areas in the graph
- $V(G) = \text{Total number of bounded areas} + 1$

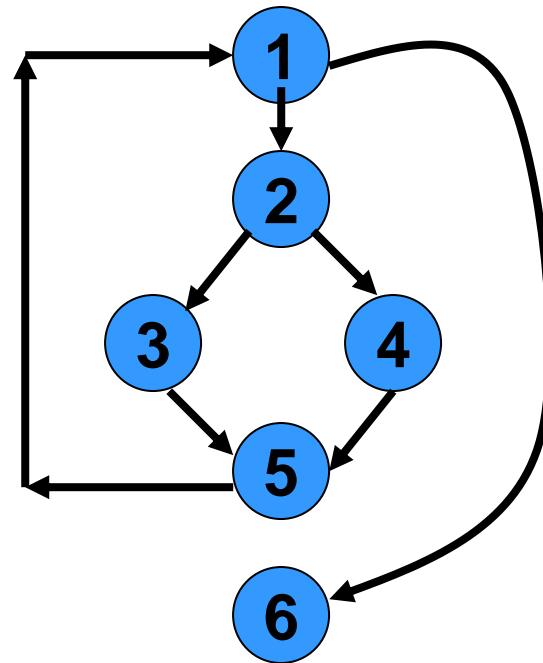
Example Control Flow Graph



**From a visual examination of the CFG:
the number of bounded areas is 2.
cyclomatic complexity = $2+1=3$.**

Example

- int f1(int x,int y){
- 1 while (x != y){
- 2 if (x>y) then
- 3 x=x-y;
- 4 else y=y-x;
- 5 }
- 6 return x; }

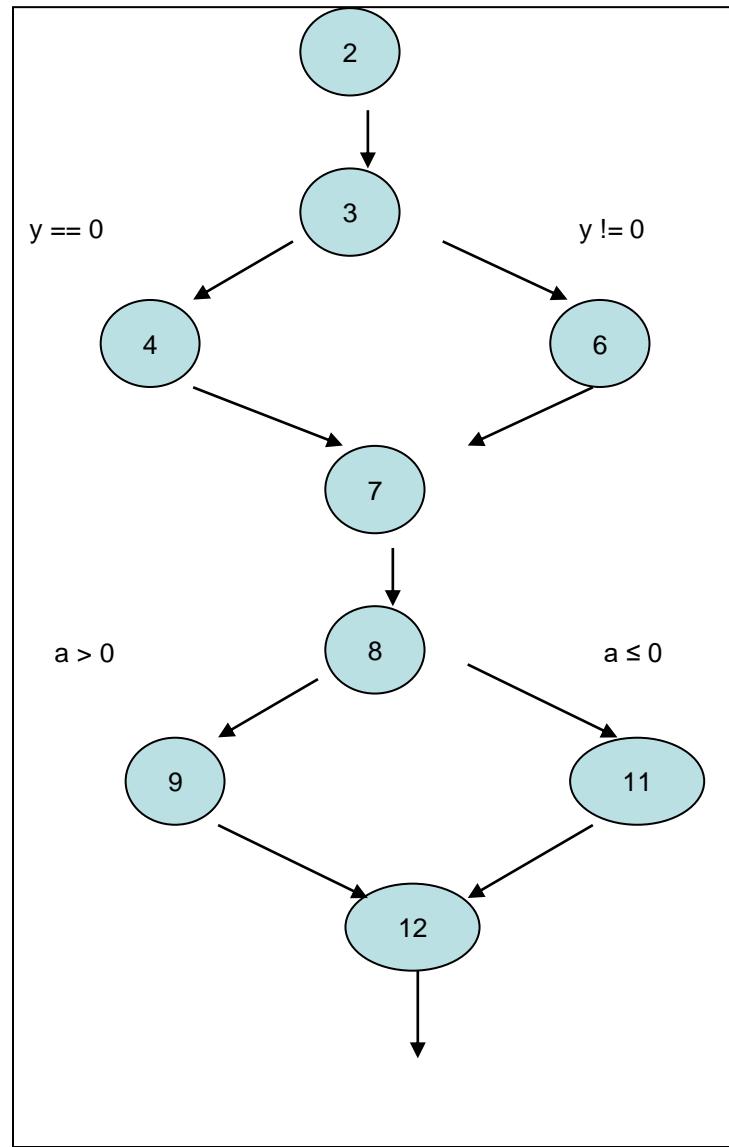


- Number of independent paths: 3
 - 1,6 test case (x=1, y=1)
 - 1,2,3,5,1,6 test case(x=1, y=2)
 - 1,2,4,5,1,6 test case(x=2, y=1)

```

1. void myflow(int x, int y)
{
2.     int a;
3.     if (y == 0) {
4.         a = 1;
5.     } else {
6.         a = y;
7.     }
8.     y = y - 1;
9.     if (a > 0) {
10.        a = x;
11.    } else {
12.        a = x + 1;
}
13. printf("%d\n", a);
}

```

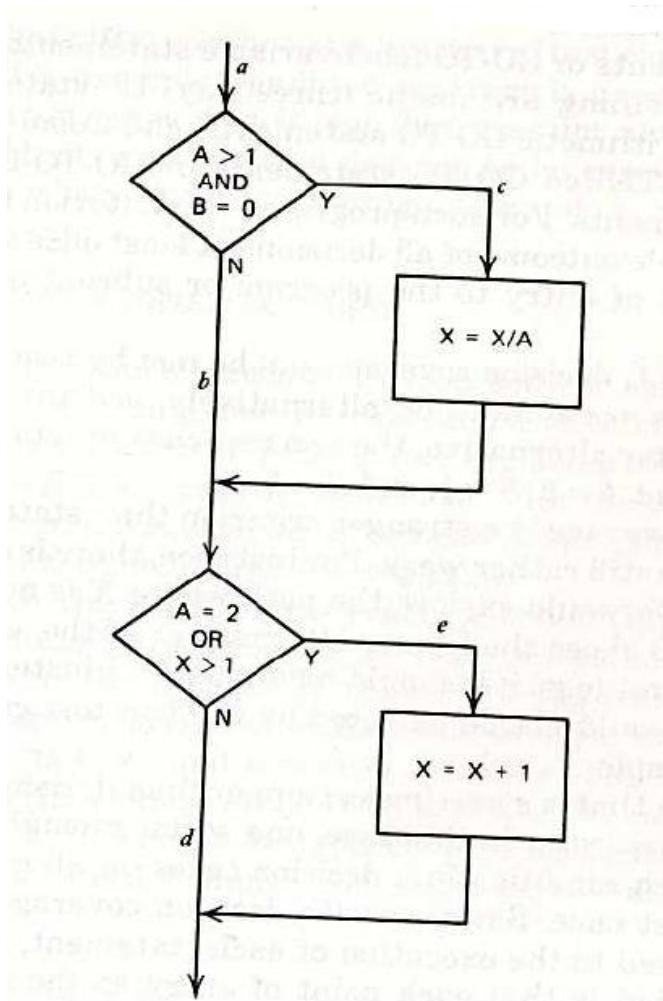


White-box Test Methods

- Statement Coverage
- Decision/Branch Coverage
- Condition Coverage
- Decision/Condition Coverage
- Path Coverage

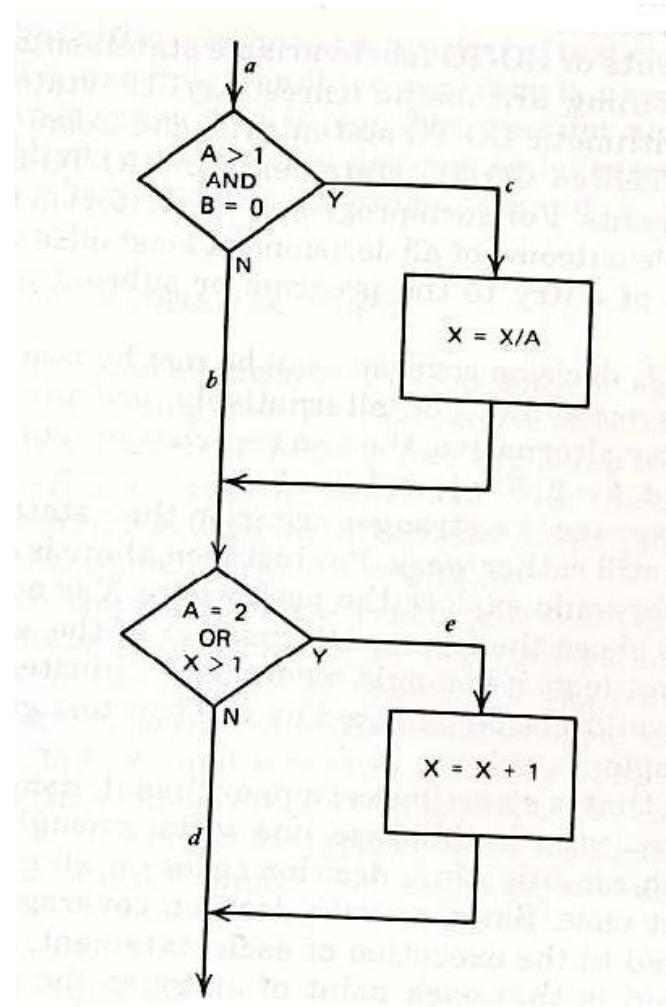
Example Code Fragment

- If ((A>1) & (B=0)) then Do;
 - X=X/A;
 - END;
- If ((A==2) | (X>1)) then Do;
 - X=X+1;
 - END;
- END;



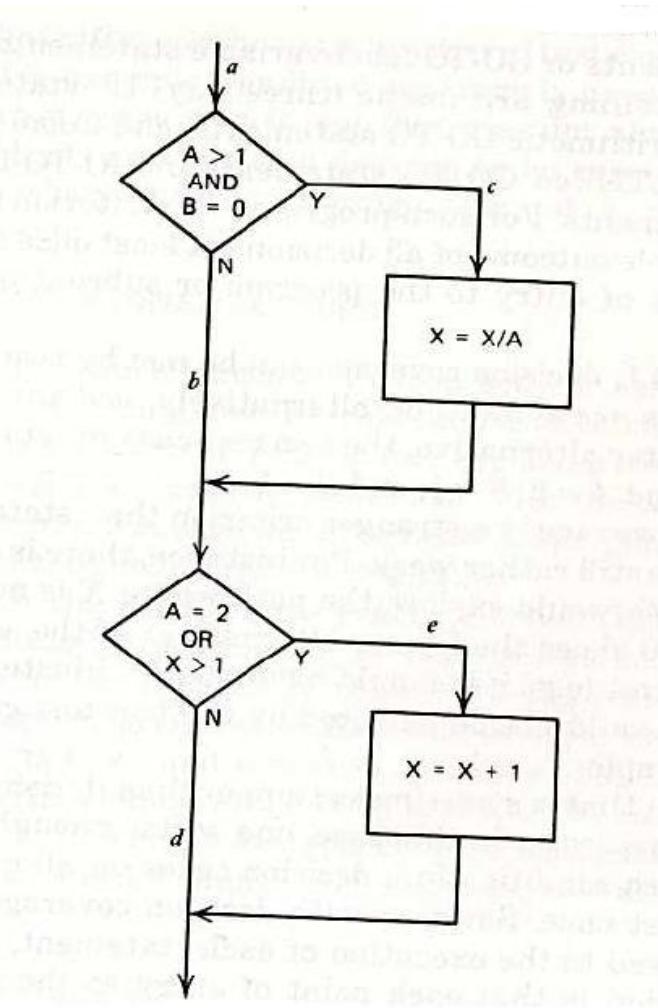
Statement Coverage

- Exercise all statements at least once
- How many test cases?
 - A=2 and B=0 (ace)



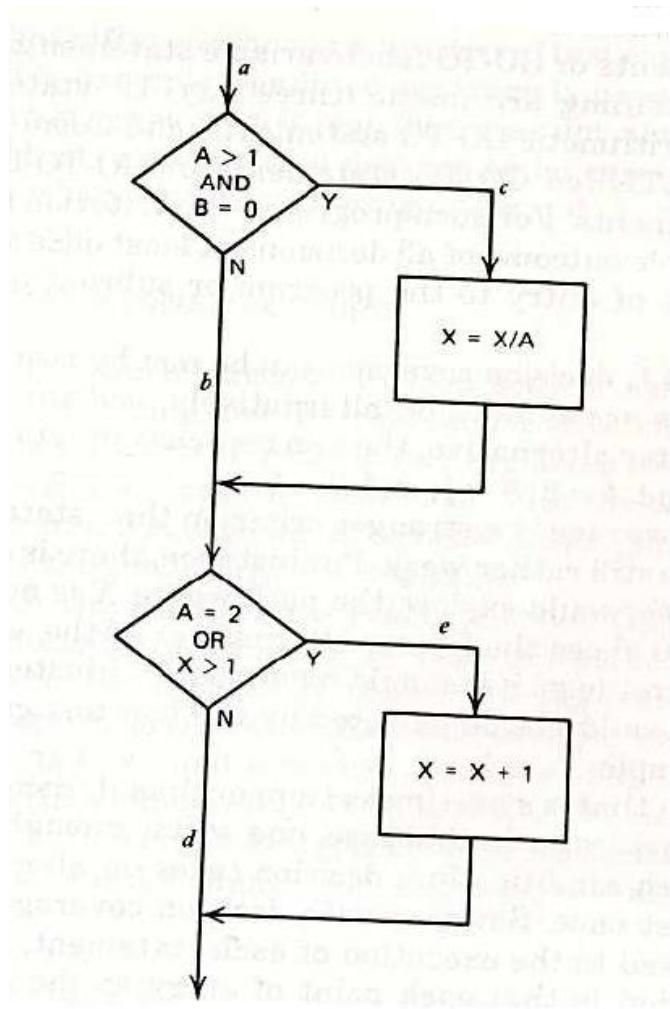
Decision/Branch Coverage

- Each decision has a true and a false outcome at least once
- How many test cases?
 - A=2 and B=0 (ace)
 - A=1 and X=1 (abd)



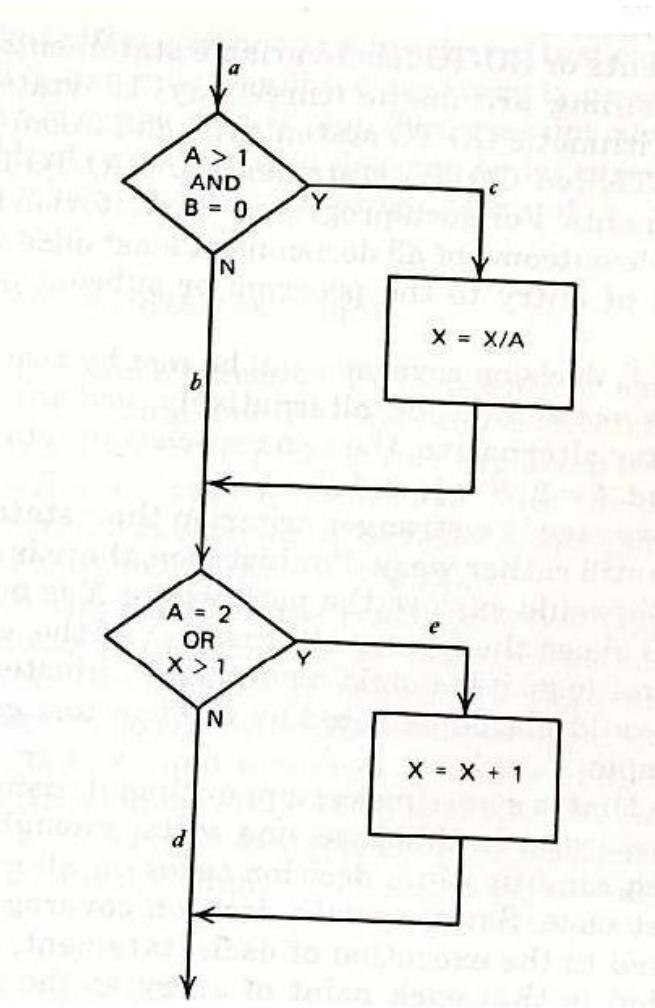
Condition Coverage

- Each condition in a decision takes on all possible outcomes at least once
- Conditions: $A > 1$, $B = 0$, $A = 2$, $X > 1$
- How many test cases?
 - $A=2$, $B=0$, and $X=4$ (ace)
 - $A=1$, $B=1$, and $X=1$ (abd)



Decision/Condition Coverage

- Each condition in a decision takes on all possible outcomes at least once, and each decision takes on all possible outcomes at least once
- How many test cases?
 - A=2, B=0, and X=4 (ace)
 - A=1, B=1, and X=1 (abd)
- What about these?
 - A=1, B=0, and X=3
 - A=2, B=1, and X=1
(abe)
(abe)



Multiple Condition Coverage

- Exercise all possible combinations of condition outcomes in each decision
- Conditions:

$A > 1, B = 0$

$A > 1, B \neq 0$

$A \leq 1, B = 0$

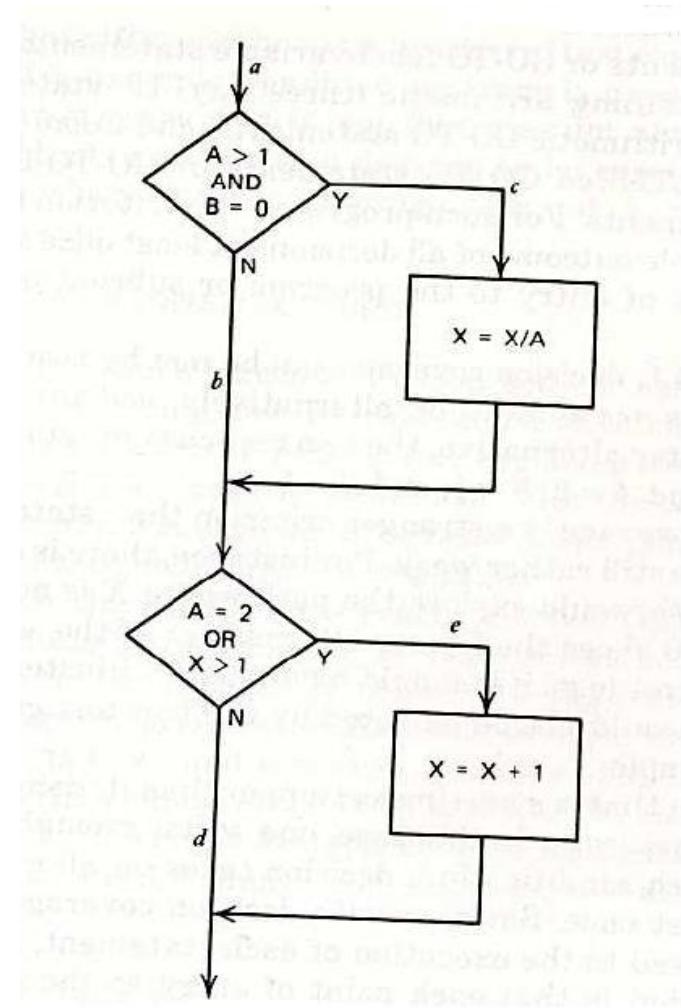
$A \leq 1, B \neq 0$

$A = 2, X > 1$

$A = 2, X \leq 1$

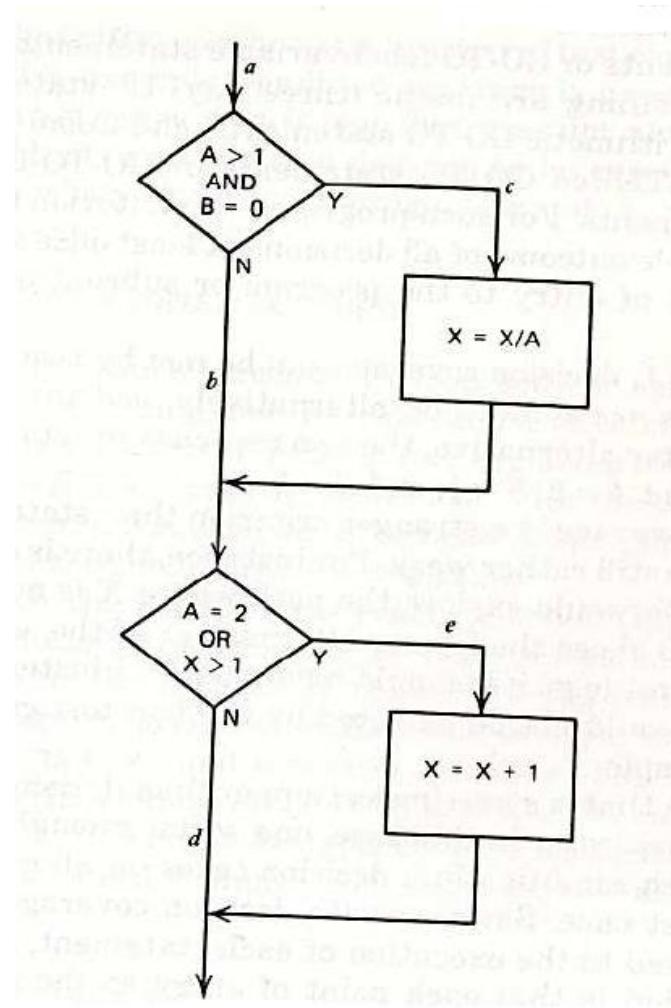
$A \neq 2, X > 1$

$A \neq 2, X \leq 1$



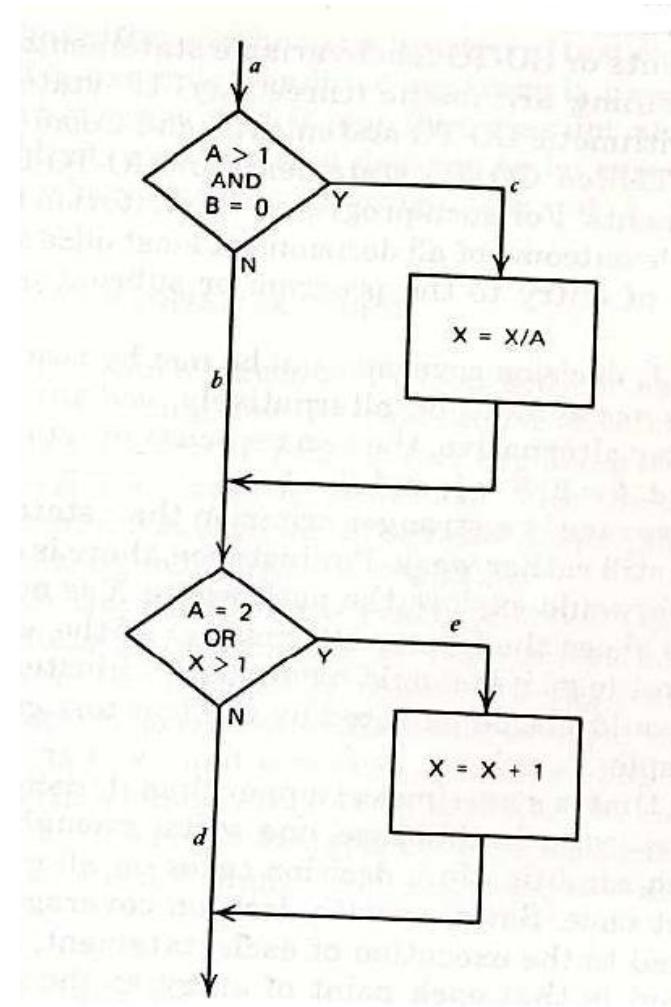
Multiple Condition Coverage

- How many test cases?
 - A=2, B=0, X=4
 - A=2, B=1, X=1 (ace)
 - A=1, B=0, X=2 (abe)
 - A=1, B=1, X=1 (abe)
 - (abd)



Path Coverage

- Every unique path through the program is executed at least once
- How many test cases?
 - A=2, B=0, X=4 (ace)
 - A=2, B=1, X=1 (abe)
 - A=3, B=0, X=1 (acd)
 - A=1, B=1, X=1 (abd)



STRUCTURED PROGRAMMING

Graph Matrices for testing

- A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph.
- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.

Structured Programming and Testing

1. Loop Testing
2. Data Flow Testing

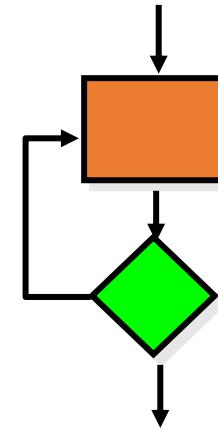
Loop testing

Four loops can be defined:

- Simple loops
- Concatenate loops
- Nested loops and
- Unstructured loops

Simple loop

- 7 different test cases made:
 - Skip loop entirely
 - Only one pass through loop
 - Two passes through loop
 - m passes through loop ($m < n$, where n is the maximum number)
 - $n - 1$
 - n
 - $n + 1$



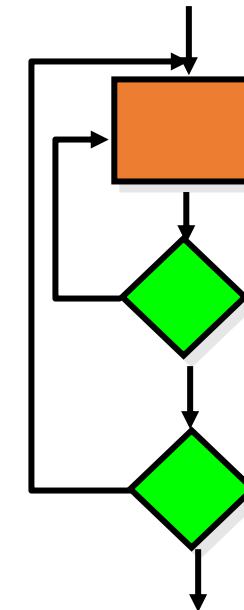
Simple loop example

```
int i;  
while (i >= 1 && i <= 10) {  
    i++;  
}
```

i	Number of iterations
0 or 11	0 (Skip loop entirely)
10	1 (one pass)
9	2 (two passes)
5	6 (pass M times)
2	9 (pass n - 1 times)
1	10 (pass n times)
0	0 (pass n + 1 times)

Nested loops

- Start with inner loop. Set all other loops to minimum values.
- Conduct simple loop testing on inner loop
- Work outwards
- Continue until all loops tested

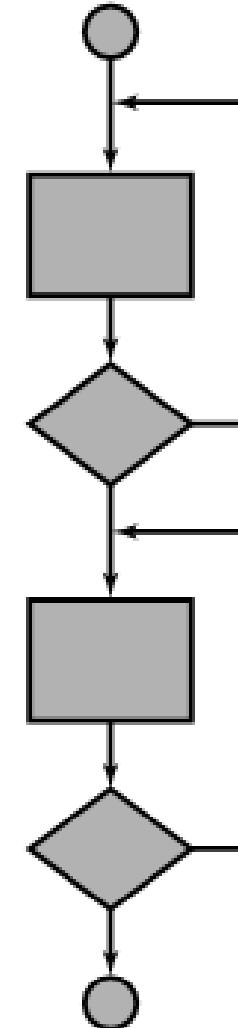


Nested loops example

```
int n = 10;  
for (int i = 0; i < n; i++) {  
    ...  
    for (int j = 0; j < n; j++) {  
        ...  
    }  
    ...  
}
```

Concatenated loops

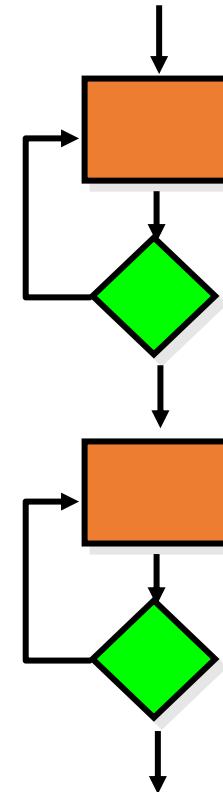
- Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other.
- However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent.
- When the loops are not independent, the approach applied to nested loops is recommended.



Concatenated
loops

Concatenated Loops

- If independent loops, use simple loop testing
- If dependent, treat as nested loops

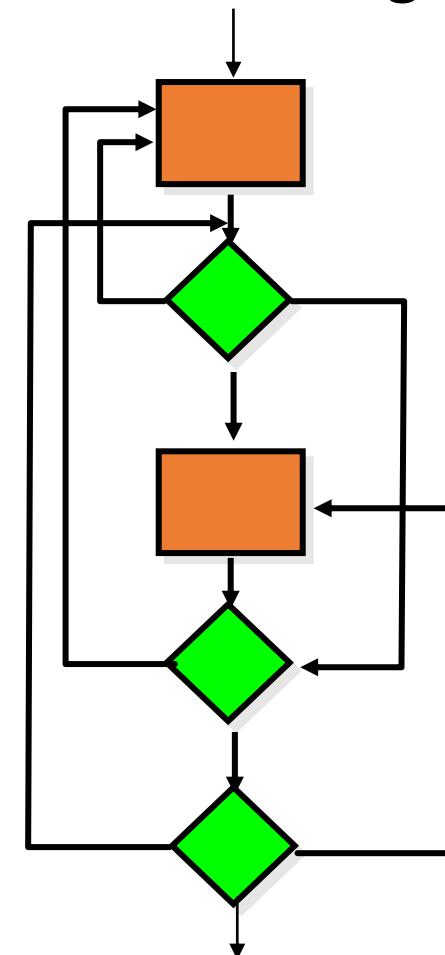


Concatenated loops example

```
int n = 10;  
int m = 20;  
while (i < n) {  
    ...  
}  
int j = i;  
while (j < m) {  
    ...  
}
```

Unstructured loops

- This kind of loop must be redesigned instead of testing.



Testing predicates

Example

A condition is represented formally as a predicate, also known as a Boolean expression. For example, consider the requirement

``if the printer is ON and has paper then send document to printer.''

This statement consists of a **condition** part and an **action** part. The following predicate represents the condition part of the statement.

$$p_r: (\text{printerstatus}=\text{ON}) \wedge (\text{printertray} \neq \text{empty})$$

Predicates

Relational operators (relop): { $<$, \leq , $>$, \geq , $=$, \neq .}
= and == are equivalent.

Boolean operators (bop): { $!$, \wedge , \vee , xor} also known as
{not, AND, OR, XOR}.

Relational expression: e1 relop e2. (e.g. $a+b < c$)
e1 and e2 are expressions whose values
can be compared using relop.

Simple predicate: A Boolean variable or a relational
expression. ($x < 0$)

Compound predicate: Join one or more simple predicates
($\text{gender} == \text{"female"} \wedge \text{age} > 65$)

Data Flow Testing

- The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program
- Data flow testing is a powerful tool to detect improper use of data values due to coding errors
 - Incorrect assignment or input statement
 - Definition is missing (use of null definition)
 - Predicate is faulty (incorrect path is taken which leads to incorrect definition)

Data Object Categories

- (d) Defined, Created, Initialized
- (k) Killed, Released
- (u) Used:
 - (c) Used in a calculation
 - (p) Used in a predicate

(d) Defined Objects

- An object (*e.g.*, variable) is **defined** when it:
 - appears in a data declaration
 - is assigned a new value
 - is a file that has been opened
 - is dynamically allocated
 - ...

(u) Used Objects

- An object is **used** when it is part of a computation or a predicate.
- A variable is used for a computation (**c**) when it appears on the RHS (sometimes even the LHS in case of array indices) of an assignment statement.
- A variable is used in a predicate (**p**) when it appears directly in that predicate.

Example: Definition and Uses

What are the *definitions* and *uses* for the program below?

1. `read (x, y);`
2. `z = x + 2;`
3. `if (z < y)`
4. `w = x + 1;`
- `else`
5. `y = y + 1;`
6. `print (x, y, w,`
- `z);`

Example: Definition and Uses

1. `read (x, y);`
2. `z = x + 2;`
3. `if (z < y)`
4. `w = x + 1;`
- `else`
5. `y = y + 1;`
6. `print (x, y, w, z);`

<i>Def</i>	<i>C-use</i>	<i>P-use</i>
x, y		
z	x	
		z, y
w	x	
y	y	
	$x, y,$	
	w, z	

Static vs Dynamic Anomaly Detection

- **Static Analysis** is analysis done on source code without actually executing it.
 - *E.g.*, Syntax errors are caught by static analysis.

Anomaly Detection Using Compilers

- Compilers are able to detect several data-flow anomalies using static analysis.
- *E.g.*, By forcing declaration before use, a compiler can detect anomalies such as:
 - **-u**
 - **ku**

Data-Flow Modeling

- Data-flow modeling is based on the control flowgraph.
- Each link is annotated with:
 - symbols (*e.g.*, **d**, **k**, **u**, **c**, **p**)
 - sequences of symbols (*e.g.*, **dd**, **du**, **ddd**)
- that denote the sequence of data operations on that link with respect to the variable of interest.

Data Flow Testing

- Variables that contain data values have a defined life cycle: created, used, killed (destroyed).
- The "scope" of the variable

```
{      // begin outer block
    int x; // x is defined as an integer within this outer block
    ...; // x can be accessed here
    {
        // begin inner block
        int y; // y is defined within this inner block
        ...; // both x and y can be accessed here
        } // y is automatically destroyed at the end of
          // this block
        ...; // x can still be accessed, but y is gone
    } // x is automatically destroyed
```

Data Flow Anomaly

- Anomaly: It is an abnormal way of doing something.
 - Example 1: The second definition of x overrides the first.
 $x = f1(y);$
 $x = f2(z);$
- Three types of abnormal situations with using variable.
 - Type 1: Defined and then defined again
 - Type 2: Undefined but referenced
 - Type 3: Defined but not referenced

Data flow analysis (DFA)'s anomalies

- Detects Defines/reference anomalies, such as
 - A variable that is defined but never used
 - (unused variable)
 - A variable that is used but never defined
 - (undefined variable)
 - A variable that is defined twice before it is used
 - (redundant operations)
 - Data reads from location not previously written to
 - (uninitialized variables)
- The anomalies can be discovered by “static analysis”

Data Flow Testing

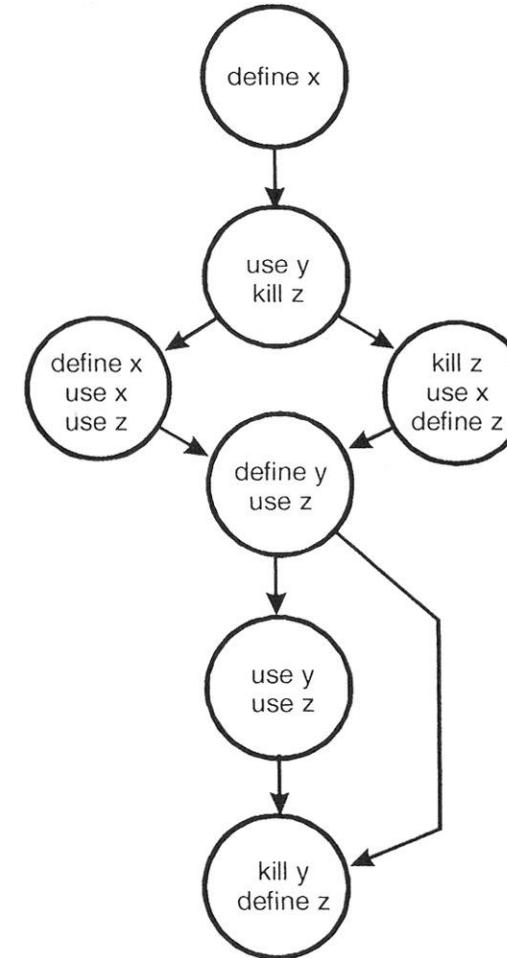
- Three possibilities exist for the first occurrence of a variable through a program path:
 - $\sim d$ - the variable does not exist (indicated by the \sim), then it is defined (d)
 - $\sim u$ - the variable does not exist, then it is used (u)
 - $\sim k$ - the variable does not exist, then it is killed or destroyed (k)

Data Flow Testing

- Time-sequenced pairs of defined (d), used (u), and killed (k):
 - dd - Defined and defined again—not invalid but suspicious. Probably a programming error.
 - du - Defined and used—perfectly correct. The normal case.
 - dk - Defined and then killed—not invalid but probably a programming error.
 - uu - Used and used again—acceptable.
 - uk - Used and killed—acceptable.
 - kd - Killed and defined—acceptable. A variable is killed and then redefined.
 - ku - Killed and used—a serious defect. Using a variable that does not exist or is undefined is always an error.
 - kk - Killed and killed—probably a programming error

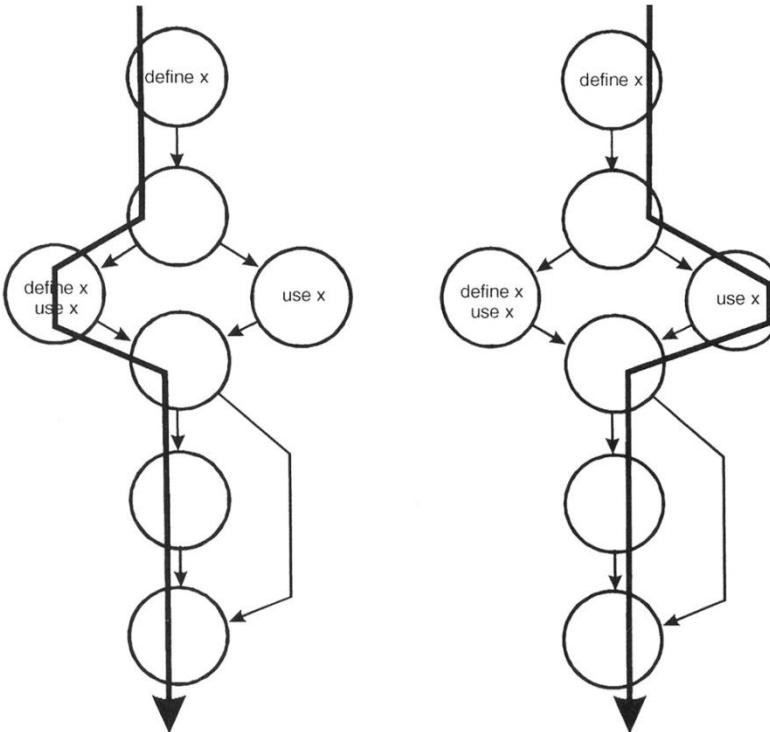
Data Flow Testing

- A data flow graph is similar to a control flow graph in that it shows the processing flow through a module.
In addition, it details the definition, use, and destruction of each of the module's variables.
- Technique
 - Construct diagrams
 - Perform a static test of the diagram
 - Perform dynamic tests on the module



Data Flow Testing

- Perform a static test of the diagram
 - For each variable within the module we will examine define-use-kill patterns along the control flow paths
 - The define-use-kill patterns for x (taken in pairs as we follow the paths) are:
 - ~define - correct, the normal case
 - define-define - suspicious, perhaps a programming error
 - define-use - correct, the normal case
 -



Example: Test enhancement using data flow

```
1 begin
2     int x, y; float z;
3     input (x, y);
4     z=0;
5     if (x!=0)
6         z=z+y;
7     else z=z-y;
8     if (y!=0) ← This condition should be (y!=0 and x!=0)
9         z=z/x;
10    else z=z*x;
11    output(z);
12 end
```

Test	x	y	z
t_1	0	0	0.0
t_2	1	1	1.0

Example (contd.)

```
1 begin
2   int x, y; float z;
3   input(x, y);
4   z=0;
5   if (x!=0)
6     z=z+y;
7   else z=z-y;
8   if (y!=0) ← This condition should be (y!=0 and x!=0)
9     z=z/x;
10  else z=z*x;
11  output(z);
12 end
```

Neither of the two tests force the use of z defined on line 6, at line 9. To do so one requires a test that causes conditions at lines 5 and 8 to be true.

This test does not force the execution of this path, hence the divide by zero error is not revealed.

Example (contd.)

Verify that the following test set covers all **def-use pairs** of z and reveals the error.

Test	x	y	z	*def-use pairs covered
t_1	0	0	0.0	(4, 7), (7,10)
t_2	1	1	1.0	(4, 6), (6,9)
t_3	0	1	0.0	(4, 7), (7,9)
t_4	1	0	1.0	(4, 6), (6,10)

*In the pair (l_1, l_2) , z is defined in l_1 and used in line l_2 .

C-use

Uses of a variable that occurs within an expression as part of an assignment statement, in an output statement, as a parameter within a function call, and in subscript expressions, are classified as **c-use, where the “c” in c-use stands for computational.**

How many c-uses of x can you find in the following statements?

Answer: 5

`z=x+1;`

`A[x-1]= B[2];`

`foo(x*x)`

`output(x);`

p-use

The occurrence of a variable in an expression used as a condition in a branch statement, such as an ***if*** or a ***while***, is considered a **p-use**.
The “p” in p-use stands for **predicate**.

How many **p-uses** of **z** and **x** can you find in the following statements?

```
if(z>0){output (x)};  
while(z>x){...};
```

Answer: 3 (2 of z and 1 of x)

SolveQuadratic

Quadratic equation:

$$Ax^2 + Bx + C = 0$$

It can have up to two real (i.e. not complex) solutions.

First test to see whether the real solutions exist, and how many:

if $B^2 - 2AC > 0$ there are two solutions

if $B^2 - 2AC = 0$ there is one solution

if $B^2 - 2AC < 0$ there are no real solutions

If solutions exist, compute:

$$x_1 = -B + \sqrt{B^2 - 2AC}/2A$$

$$x_2 = -B - \sqrt{B^2 - 2AC}/2A$$

```
SolveQuadratic           // Ax2 + Bx + C = 0
float A, B, C            // input
float x1, x2             // output
boolean is_complex        // output
float D
input( A, B, C )
D = B*B - 4*A*C
if D < 0.0 then
    is_complex = T
else
    is_complex = F
endif
if not is_complex then
    x1 = (-B + sqrt( D )) / (2.0*A)
    x2 = (-B - sqrt( D )) / (2.0*A)
endif
end SolveQuadratic
```

```
SolveQuadratic           // Ax2 + Bx + C = 0
float A, B, C            // input
float x1, x2             // output
boolean is_complex        // output
float D
input( A, B, C )
D = B*B - 4*A*C
if D < 0.0 then
    is_complex = T
else
    is_complex = F
endif
if not is_complex then
    x1 = (-B + sqrt( D )) / (2.0*A)
    x2 = (-B - sqrt( D )) / (2.0*A)
endif
end SolveQuadratic
```

}

Declarations

```
SolveQuadratic           // Ax2 + Bx + C = 0
float A, B, C            // input
float x1, x2             // output
boolean is_complex        // output
float D
input( A, B, C )          ←
D = B*B - 4*A*C          ←
if D < 0.0 then           ←
    is_complex = T         ←
else
    is_complex = F         ←
endif
if not is_complex then
    x1 = (-B + sqrt( D )) / (2.0*A)
    x2 = (-B - sqrt( D )) / (2.0*A)
endif
end SolveQuadratic
```

Definitions

```
SolveQuadratic           // Ax2 + Bx + C = 0
float A, B, C            // input
float x1, x2             // output
boolean is_complex        // output
float D
input( A, B, C )
D = B*B - 4*A*C ←
if D < 0.0 then ←
    is_complex = T
else
    is_complex = F
endif
if not is_complex then ←
    x1 = (-B + sqrt( D )) / (2.0*A)
    x2 = (-B - sqrt( D )) / (2.0*A)
endif
end SolveQuadratic
```

Computation use
(c-use)

Predicate use
(p-use)

```
SolveQuadratic           // Ax2 + Bx + C = 0
float A, B, C            // input
float x1, x2             // output
boolean is_complex        // output
float D
input( A, B, C )
D = B*B - 4*A*C
if D < 0.0 then
    is_complex = T
else
    is_complex = F
endif
if not is_complex then
    x1 = (-B + sqrt( D )) / (2.0*A)
    x2 = (-B - sqrt( D )) / (2.0*A)
endif
end SolveQuadratic
```

du-pairs

Data Flow Testing

- **Variable definition**
 - Occurrences of a variable where a variable is given a new value (assignment, input by the user, input from a file, etc.)
- **Variable uses**
 - Occurrences of a variable where a variable is not given a new value (variable DECLARATION is NOT its use)

Data Flow Testing

- **Variable uses**
- Occurrences of a variable where a variable is not given a new value
 - p-uses (predicate uses)
- Occur in the predicate portion of a decision statement such as
- if-then-else, while-do etc.
- – c-uses (computation uses)
- All others, including variable occurrences in the right hand side of an assignment statement, or an output statement du-path
- A sub-path from a variable definition to its use

Data Flow testing

- Checks the correctness of the du-paths in a control
- Flow Graph of a program
- Test case definitions based on four groups of
- coverage
 - – All definitions
 - – All c-uses
 - – All p-uses
 - – All du-paths

Data Flow testing: key steps

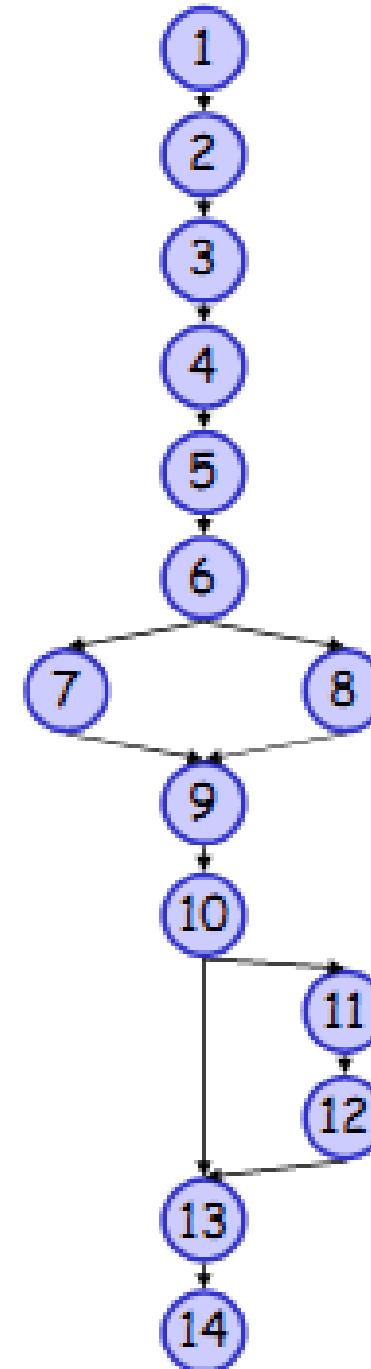
Given a code (program or pseudo-code)

1. Number the lines
2. List the variables
3. List occurrences & assign a category to each variable
4. Identify du-pairs and their use (p- or c-)
5. Define test cases, depending on the required coverage

Data flow testing

1: number the lines

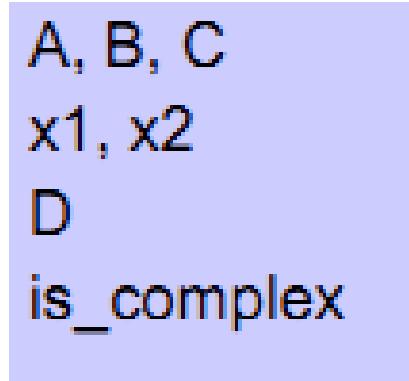
```
1  SolveQuadratic
2  float A, B, C, D, x1, x2
3  boolean is_complex
4  input( A, B, C )
5  D = B*B - 4*A*C
6  if D < 0.0
7    then is_complex = T
8    else is_complex = F
9  endif
10 if not is_complex
11   then x1 = (-B + sqrt( D )) / (2.0*A)
12     x2 = (-B - sqrt( D )) / (2.0*A)
13  endif
14 end SolveQuadratic
```



Data flow testing

2: list the variables

```
1  SolveQuadratic
2  float A, B, C, D, x1, x2
3  boolean is_complex
4  input( A, B, C )
5  D = B*B - 4*A*C
6  if D < 0.0
7      then is_complex = T
8      else is_complex = F
9  endif
10 if not is_complex
11     then x1 = (-B + sqrt( D )) / (2.0*A)
12         x2 = (-B - sqrt( D )) / (2.0*A)
13 endif
14 end SolveQuadratic
```



A, B, C
x1, x2
D
is_complex

Data flow testing

3:list occurrences & assign a category to each variable

line	category		
	definition	c-use	p-use
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			

```
1 SolveQuadratic
2 float A, B, C, D, x1, x2
3 boolean is_complex
4 input( A, B, C )
5 D = B*B - 4*A*C
6 if D < 0.0
7     then is_complex = T
8     else is_complex = F
9 endif
10 if not is_complex
11     then x1 = (-B + sqrt( D )) / (2.0*A)
12         x2 = (-B - sqrt( D )) / (2.0*A)
13 endif
14 end SolveQuadratic
```

Data flow testing

3:list occurrences & assign a category to each variable

line	category		
	definition	c-use	p-use
1			
2			
3			
4			
5	D		
6			D
7			
8			
9			
10			
11			
12			
13			
14			

```
1 SolveQuadratic
2 float A, B, C, D, x1, x2
3 boolean is_complex
4 input( A, B, C )
5 D = B*B - 4*A*C
6 if D < 0.0
7     then is_complex = T
8     else is_complex = F
9 endif
10 if not is_complex
11     then x1 = (-B + sqrt( D )) / (2.0*A)
12         x2 = (-B - sqrt( D )) / (2.0*A)
13 endif
14 end SolveQuadratic
```

Data flow testing

3: list occurrences & assign a category to each variable

line	category		
	definition	c-use	p-use
1			
2			
3			
4	A, B, C		
5	D	A, B, C	
6			D
7	is_complex		
8	is_complex		
9			
10			is_complex
11	x1	A, B, D	
12	x2	A, B, D	
13			
14			

1 SolveQuadratic
2 float A, B, C, D, x1, x2
3 boolean is_complex
4 input(A, B, C)
5 D = B*B - 4*A*C
6 if D < 0.0
7 then is_complex = T
8 else is_complex = F
9 endif
10 if not is_complex
11 then x1 = (-B + sqrt(D)) / (2.0*A)
12 x2 = (-B - sqrt(D)) / (2.0*A)
13 endif
14 end SolveQuadratic

Data flow testing

4: identify du-pairs and their use (p- or c-)

```
1 SolveQuadratic
2 float A, B, C, D, x1, x2
3 boolean is_complex
4 input( A, B, C )
5 D = B*B - 4*A*C
6 if D < 0.0
7     then is_complex = T
8     else is_complex = F
9 endif
10 if not is_complex
11     then x1 = (-B + sqrt( D )) / (2.0*A)
12         x2 = (-B - sqrt( D )) / (2.0*A)
13 endif
14 end SolveQuadratic
```

Data flow testing

4: identify du-pairs and their use (p- or c-)

```
1 SolveQuadratic
2 float A, B, C, D, x1, x2
3 boolean is_complex
4 input( A, B, C )
5 D = B*B - 4*A*C
6 if D < 0.0
7     then is_complex = T
8     else is_complex = F
9 endif
10 if not is_complex
11     then x1 = (-B + sqrt( D )) / (2.0*A)
12         x2 = (-B - sqrt( D )) / (2.0*A)
13 endif
14 end SolveQuadratic
```

Data flow testing

4: identify du-pairs and their use (p- or c-)

definition - use pair	variable(s)	
start line -> end line	c-use	p-use
4 -> 5	A	
4 -> 5	B	
4 -> 5	C	
4 -> 11	A	
4 -> 11	B	
4 -> 12	A	
4 -> 12	B	
5 -> 6		D
5 -> 11	D	
5 -> 12	D	
7 -> 10		is_complex
8 -> 10		is_complex

What about x1 and x2?

```
1 SolveQuadratic
2 float A, B, C, D, x1, x2
3 boolean is_complex
4 input( A, B, C )
5 D = B*B - 4*A*C
6 if D < 0.0
7   then is_complex = T
8   else is_complex = F
9 endif
10 if not is_complex
11   then x1 = (-B + sqrt( D )) / (2.0*A)
12     x2 = (-B - sqrt( D )) / (2.0*A)
13 endif
14 end SolveQuadratic
```

Data flow testing

5:define test cases

- The choice of test cases depends on the coverage type required
- Most common types of coverage
 - All definitions
 - All c-uses
 - All p-uses
 - All du-paths

Data flow testing

5:define test cases

All-definitions

To achieve 100% All-definitions data flow coverage at least one sub-path from **each variable definition** to **some use** of that definition (either c- or p- use) must be executed.

All-definitions testing: variable A

definition - use pair start line -> end line	variable(s)	
	c-use	p-use
4 -> 5	A	
4 -> 5	B	
4 -> 5	C	
4 -> 11	A	
4 -> 11	B	
4 -> 12	A	
4 -> 12	B	
5 -> 6		D
5 -> 11	D	
5 -> 12	D	
7 -> 10		is_complex
8 -> 10		is_complex

How many test cases?

1 test case
(some use)

All-definitions testing: variable is_complex

definition - use pair	variable(s)	
start line \rightarrow end line	C-use	P-use
4 \rightarrow 5	A	
4 \rightarrow 5	B	
4 \rightarrow 5	C	
4 \rightarrow 11	A	
4 \rightarrow 11	B	
4 \rightarrow 12	A	
4 \rightarrow 12	B	
5 \rightarrow 6		D
5 \rightarrow 11	D	
5 \rightarrow 12	D	
7 \rightarrow 10		is_complex
8 \rightarrow 10		is_complex

How many test cases?

2 test cases
(at least one
use for each
definition)

Data flow testing: All-definitions test cases

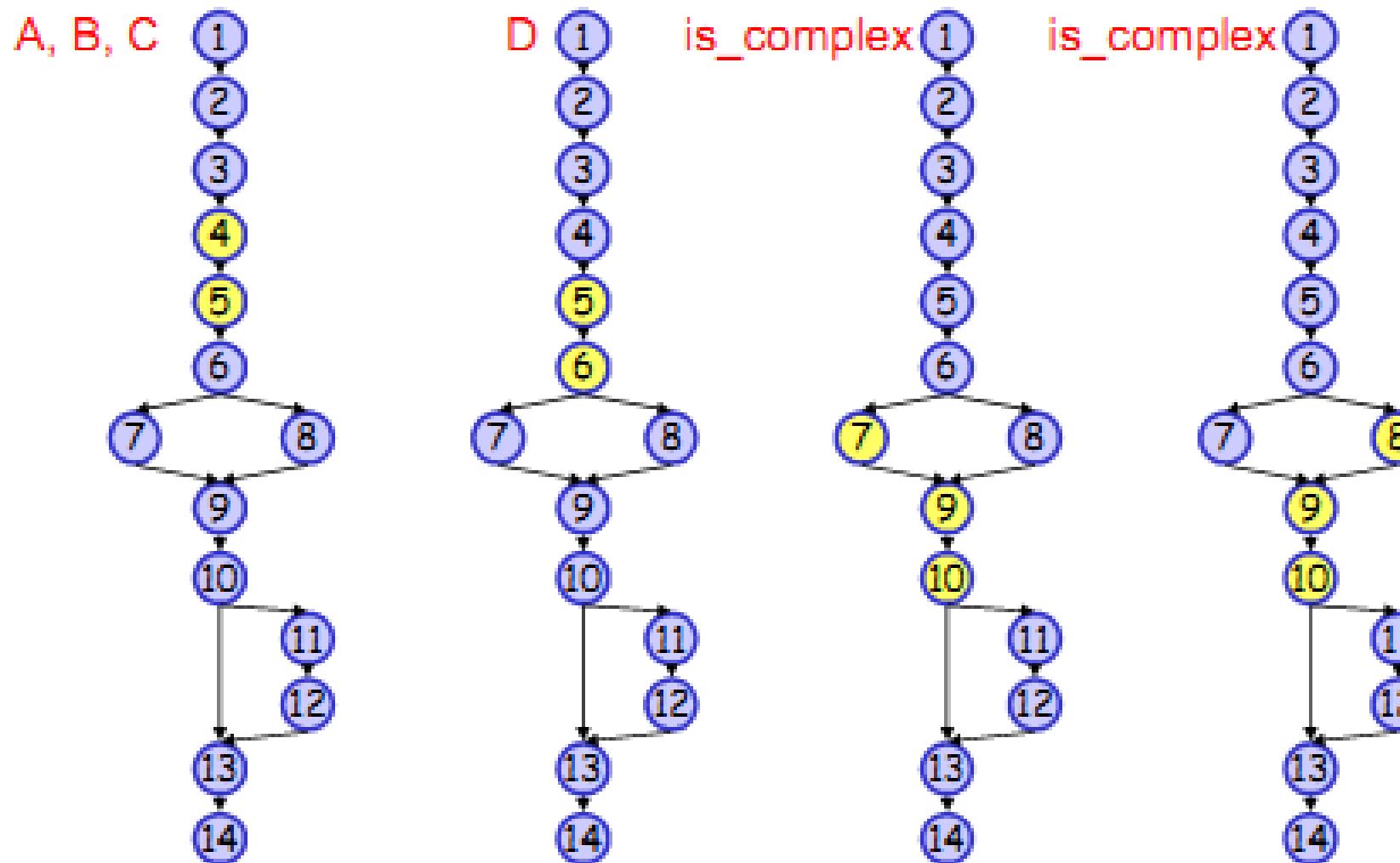
Data flow testing: All-definitions test cases

variable(s)	du-pair	sub-path	Inputs			Expected outcome		
			A	B	C	is_complex	x1	x2
A	4 -> 5	4-5	1	1	1	T	n/a	n/a
B	4 -> 5	4-5	1	1	1	T	n/a	n/a
C	4 -> 5	4-5	1	1	1	T	n/a	n/a
D	5 -> 6	5-6	1	1	1	T	n/a	n/a
is_complex	7 -> 10	7-10	1	1	1	T	n/a	n/a
is_complex	8 -> 10	8-10	1	2	1	F	-1	-1



These are just sample inputs

			Inputs			Expected outcome		
variable(s)	du-pair	sub-path	A	B	C	is_complex	x1	x2
A, B, C	4 \Rightarrow 5	4-5	1	1	1	T	n/a	n/a
D	5 \Rightarrow 6	5-6	1	1	1	T	n/a	n/a
is_complex	7 \Rightarrow 10	7-10	1	1	1	T	n/a	n/a
is_complex	8 \Rightarrow 10	8-10	1	2	1	F	-1	-1



Data flow testing

5:define test cases

All-c-uses

To achieve 100% All-c-uses data flow coverage at least one sub-path from **each variable definition** to **every c-use** of that definition must be executed.

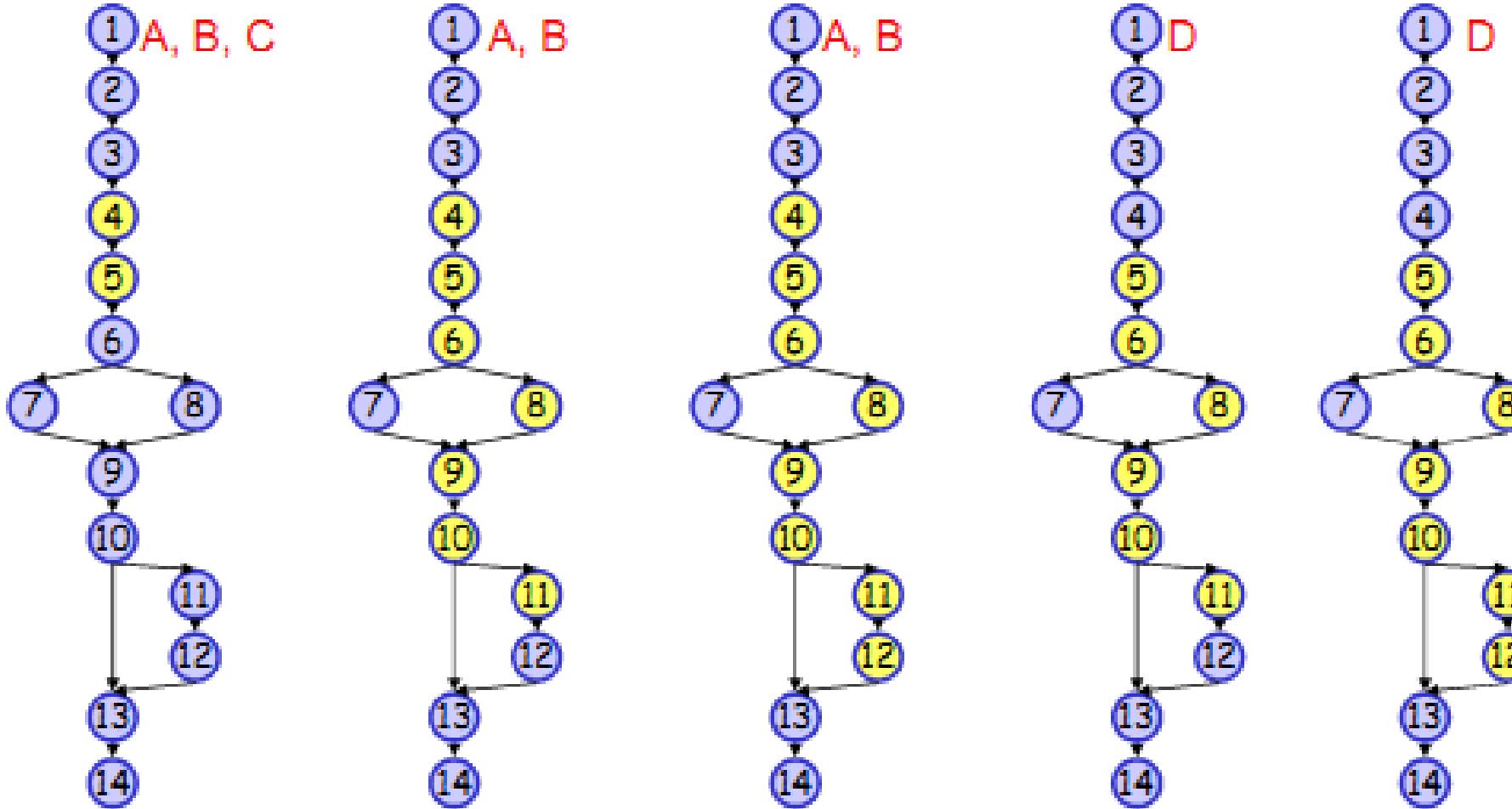
All-c-uses testing: variable A

definition - use pair start line \rightarrow end line	variable(s)	
	c-use	p-use
4 \rightarrow 5	A	
4 \rightarrow 5	B	
4 \rightarrow 5	C	
4 \rightarrow 11	A	
4 \rightarrow 11	B	
4 \rightarrow 12	A	
4 \rightarrow 12	B	
5 \rightarrow 6		D
5 \rightarrow 11	D	
5 \rightarrow 12	D	
7 \rightarrow 10		is_complex
8 \rightarrow 10		is_complex

How many test cases?

3 test cases
(every c-use)

variable(s)	du-pair	sub-path	A	B	C	is_complex	x1	x2
A, B, C	4 \Rightarrow 5	4-5	1	1	1	T	n/a	n/a
A, B	4 \Rightarrow 11	4-5-6-8-9-10-11	1	2	1	F	-1	-1
A, B	4 \Rightarrow 12	4-5-6-8-9-10-11-12	1	2	1	F	-1	-1
D	5 \Rightarrow 11	5-6-8-9-10-11	1	2	1	F	-1	-1
D	5 \Rightarrow 12	5-6-8-9-10-11-12	1	2	1	F	-1	-1



Mutation Testing

Mutation Testing (1)

- *Mutation testing is a structural testing method, i.e. we use the structure of the code to guide the test process.*

We cover the following aspects of Mutation Testing:

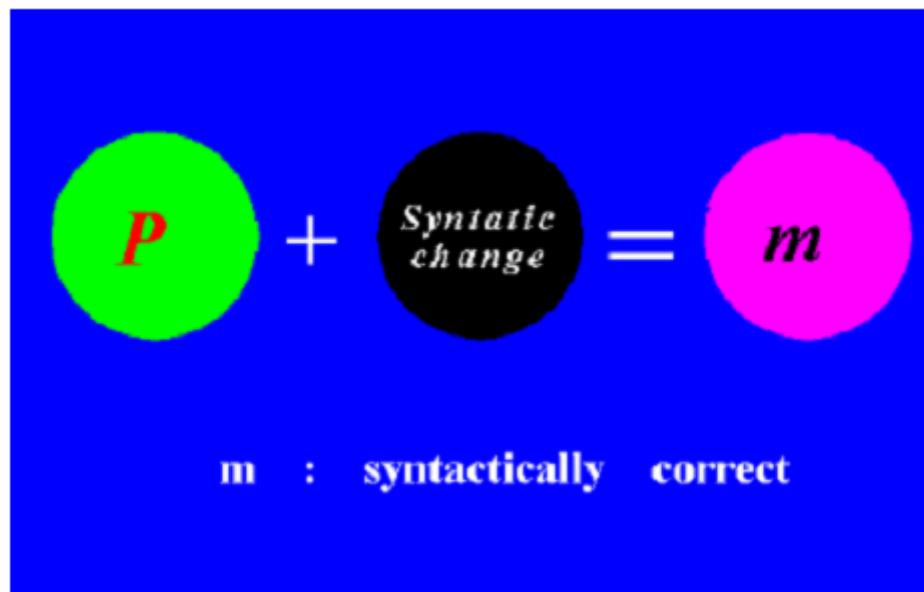
- What is a mutation?
- What is mutation testing?
- When should we use mutation testing?
- Mutations
- Examples
- Mutation testing tools

Mutation Testing (2)

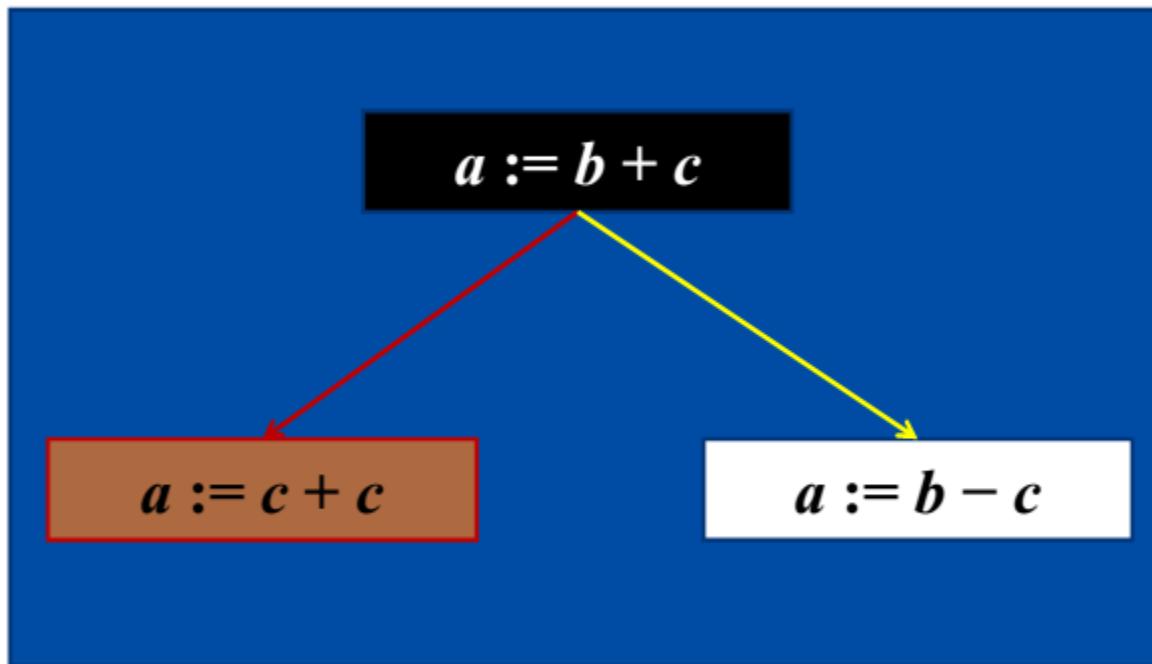
- It relies on the competent programmer hypothesis which is the following assumption: – Given a specification a programmer develops a program that is either correct or differs from the correct program by a combination of simple errors
- It also relies on “coupling effect” which suggests that – Test cases that detect simple types of faults are sensitive enough to detect more complex types of faults.

Mutant (1)

- Given a program P, a mutant of P is obtained by making a simple change in P

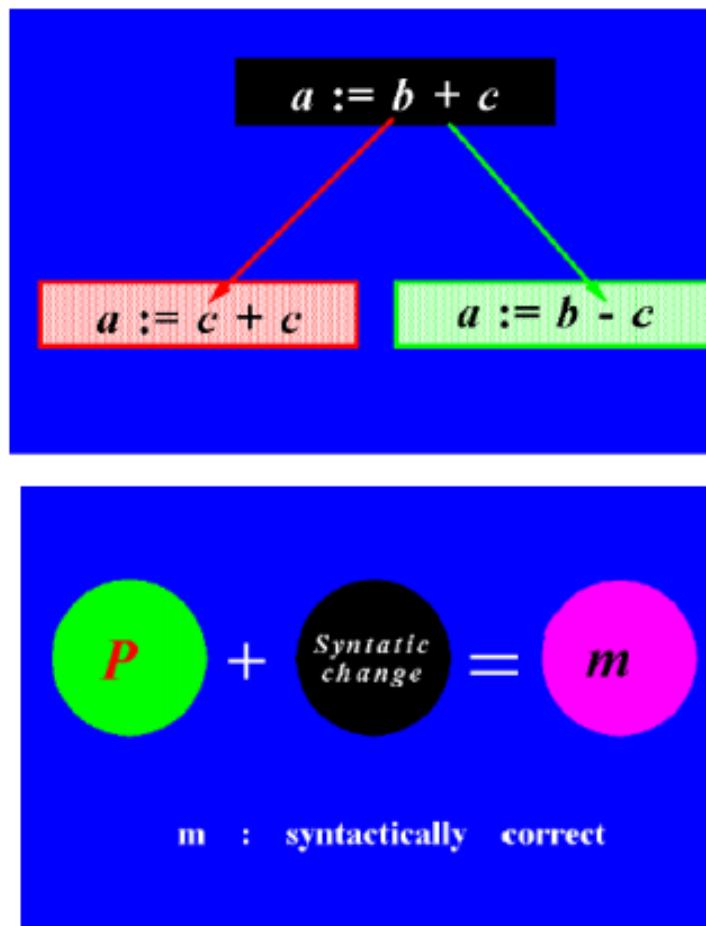


Mutant (2)



Definitions

- *Fault-based Testing*: directed towards "typical" faults that could occur in a program
- Basic idea:
 - Take a program and test data generated for that program
 - Create a number of *similar* programs (mutants), each differing from the original in one small way, i.e., each possessing a fault
 - e.g., replace addition operator by multiplication operator
 - The original test data are then run through the *mutants*
 - If test data detect all differences in mutants, then the mutants are said to be *dead*, and the test set is *adequate*



Example (1)

Program

```
1. int x, y;  
2. if (x != 0)  
3.     y = 5;  
4. else z = z - x;  
5. if (z > 1)  
6.     z = z/x;  
7. else  
8.     z = y;
```

Mutant

```
1. int x, y;  
2. if (x! = 0)  
3.     y = 5;  
4. else z = z - x;  
5. if (z > 1)  
6.     z = z/&x; ←  
7. else  
8.     z = y;
```

Example (2)

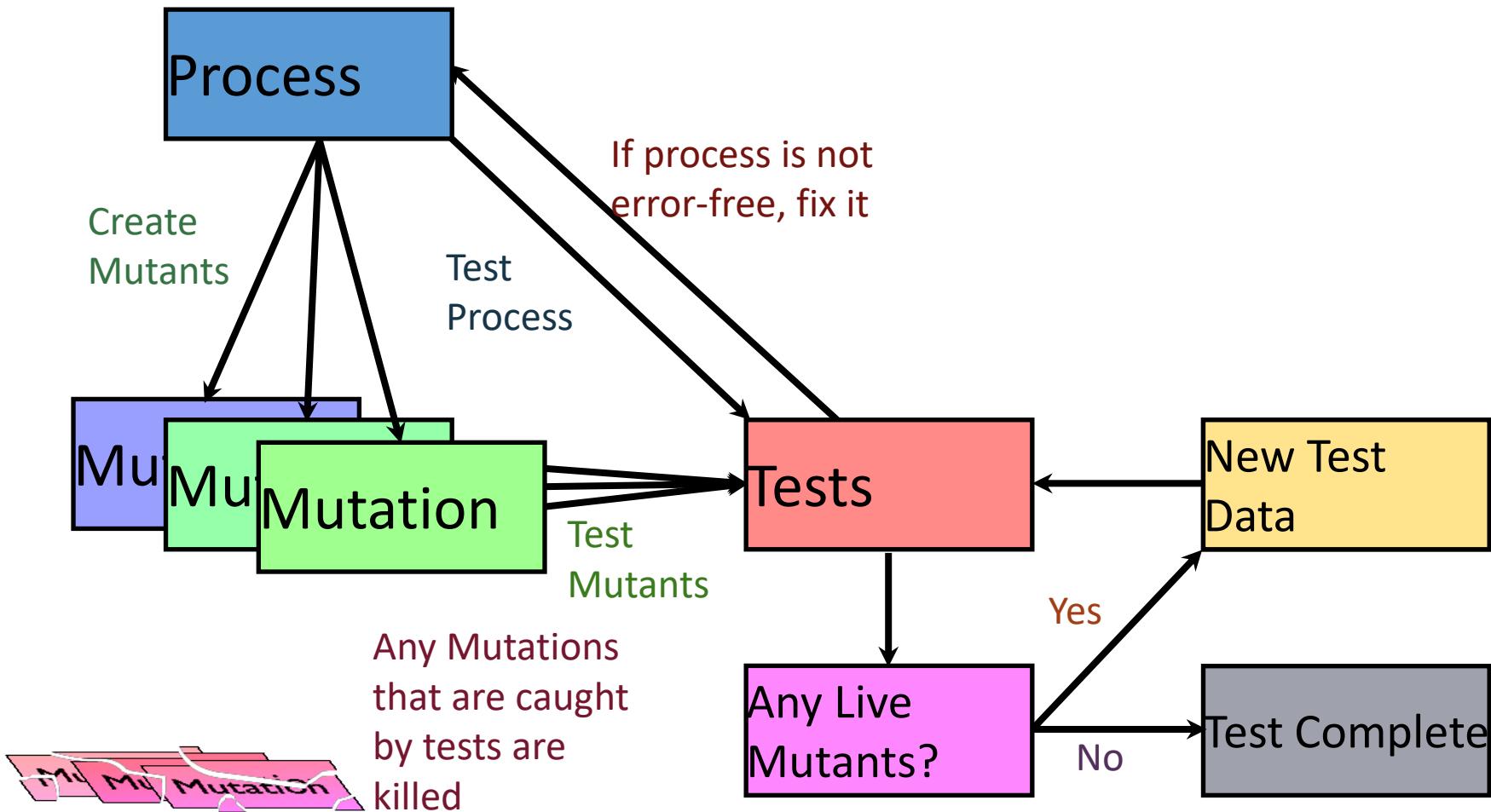
Program

```
1. int x, y;  
2. if (x != 0)  
3.     y = 5;  
4. else z = z - x;  
5. if (z > 1)  
6.     z = z/x;  
7. else  
8.     z = y;
```

Mutant

```
1. int x, y;  
2. if (x != 0)  
3.     y = 5;  
4. else z = z - x;  
5. if (z < 1) ←————  
6.     z = z/x;  
7. else  
8.     z = y;
```

The Mutation Process



Types of Mutations

- **Value Mutations:** these mutations involve changing the values of constants or parameters (by adding or subtracting values etc), e.g. loop bounds – being one out on the start or finish is a very common error.
- **Decision Mutations:** this involves modifying conditions to reflect potential slips and errors in the coding of conditions in programs, e.g. a typical mutation might be replacing $a >$ by $a <$ in a comparison.
- **Statement Mutations:** these might involve deleting certain lines to reflect omissions in coding or swapping the order of lines of code. There are other operations, e.g. changing operations in arithmetic expressions. A typical omission might be to omit the increment on some variable in a while loop.

Value Mutation

- Here we attempt to change values to reflect errors in reasoning about programs.
- Typical examples are:
 - Changing values to one larger or smaller (or similar for real numbers).
 - Swapping values in initialisations.
- The commonest approach is to change constants by one in an attempt to generate a one-off error (particularly common in accessing arrays).
- Coverage criterion: Here we might want to perturb all constants in the program or unit at least once or twice.

Decision Mutation

- Here again we design the mutations to model failures in reasoning about conditions in programs. As before this is a very limited model of programming error really modelling slips in coding rather than a design error.
- Typical examples are:
 - Modelling “one-off” errors by changing $<$ to \leq or vice versa (this is common in checking loop bounds).
 - Modelling confusion about larger and smaller, so changing $>$ to $<$ or vice versa.
 - Getting parenthesisation wrong in logical expressions e.g. mistaking precedence between $\&\&$ and $\|$
- Coverage Criterion: We might consider one mutation for each condition in the program. Alternatively we might consider mutating all relational operators (and logical operators e.g. replacing $\|$ by $\&\&$ and vice versa)

Statement Mutation

- Here the goal is primarily to model editing slips at the line level – these typically arise when the developer is cutting and pasting code. The result is usually omitted or duplicated code. In general we might consider arbitrary deletions and permutations of the code.
- Typical examples include:
 - Deleting a line of code
 - Duplicating a line of code
 - Permuting the order of statements.
- Coverage Criterion: We might consider applying this procedure to each statement in the program (or all blocks of code up to and including a given small number of lines).

Order of mutants

- First order mutants
 - One syntactic change
- Higher order mutants
 - Multiple syntactic changes
- Coupling effect

Types of mutants

- Distinguished mutants
- Live mutants
- Equivalent mutants
- Non-equivalent mutants

Types of mutants

- A mutant m is considered *distinguished* (or *killed*) by a test case $t \in T$ if
$$P(t) \neq m(t)$$
where $P(t)$ and $m(t)$ denote, respectively, the observed behavior of P and m when executed on test input t

- A mutant m is considered *equivalent* to P if

$$P(t) = m(t)$$

for any test case in the input domain

Distinguish a Mutant (1)

- Reachability
 - Execute the mutated statement
- Necessity
 - Make a state change
- Sufficiency
 - Propagate the change to output

Distinguish a Mutant (2)

Program P
read a
if ($a > 3$)
then
 $x = 5$
else
 $x = 2$
endif
print x

Program m
read a
if ($a \geq 3$)
then
 $x = 5$
else
 $x = 2$
endif
print x

Mutant m is distinguished by $a = 3$

Equivalent Mutant

Program P

read a, b

$a = b$

$x = a + b$

print x

Program m

read a, b

$a = b$

$x = a + a$

print x

P is equivalent to m

Basic Ideas (I)

In Mutation Testing:

1. We take a program and a test suite generated for that program (using other test techniques)
2. We create a number of *similar* programs (mutants), each differing from the original in one small way, i.e., each possessing a fault
 - E.g., replacing an addition operator by a multiplication operator
3. The original test data are then run on the *mutants*
4. If test cases detect differences in mutants, then the mutants are said to be *dead (killed)*, and the test set is considered *adequate*

Basic Ideas (II)

- A mutant remains *live* either
 - because it is equivalent to the original program (functionally identical although syntactically different - called an *equivalent mutant*) or,
 - the test set is inadequate to kill the mutant
- In the latter case, the test data need to be augmented (by adding one or more new test cases) to kill the *live* mutant
- For the automated generation of mutants, we use *mutation operators*, that is predefined program modification rules (i.e., corresponding to a fault model)

- Consider the following program P

```
int x, y, z;  
scanf (&x, &y);  
if (x>0)  
    x = x + 1; z = x × (y - 1);  
else  
    x = x - 1; z = x × (y - 1);
```

- Here z is considered the output of P
- Now suppose that a mutant of P is obtained by changing $x = x + 1$ to $x = \text{abs}(x) + 1$
- This mutant is **equivalent** to P as no test case can distinguish it from P

Mutation Coverage

- Complete coverage equals to killing all non-equivalent mutants (or random sample)
- The amount of coverage is also called “mutation score”
- We can see each mutant as a test requirement
- The number of mutants depends on the definition of mutation operators and the syntax/structure of the software
- Numbers of mutants tend to be large, even for small programs (hence random sampling)

Mutation Score

- During testing a mutant is considered *live* if it has not been distinguished or proven equivalent.
- Suppose that a total of \mathcal{M}_t mutants are generated for program P
- The *mutation score* of a test set T , designed to test P , is computed as:

$$MS(P, T) = \frac{\mathcal{M}_k}{\mathcal{M}_t - \mathcal{M}_q}$$

- \mathcal{M}_k – number of mutants killed
- \mathcal{M}_q – number of equivalent mutants
- \mathcal{M}_t – total number of mutants

Mutation Score

- Mutation score:

$$\frac{\text{Number of mutants distinguished}}{\text{Total number of non-equivalent mutants}}$$

- Data flow score:

$$\frac{\text{Number of blocks (decisions, p-uses, c-uses, all-uses) covered}}{\text{Total number of feasible blocks (decisions, p-uses, c-uses, all-uses)}}$$

Test Adequacy Criteria

- A test T is considered *adequate* with respect to the mutation criterion if its mutation score is 1
 - Equivalent mutants?
 - Which mutant operators are used?
- The number of mutants generated depends on P and the *mutant operators* applied on P
- A *mutant operator* is a rule that when applied to the program under test generates zero or more mutants

Mutant Operator

- Consider the following program:

```
int abs (x);  
int x;  
{  
    if ( $x \geq 0$ )  $x = 0 - x$ ;  
    return x;  
}
```

- Consider the following rule:

- Replace each relational operator in P by all possible relational operators excluding the one that is being replaced.

- Assuming the set of relational operators to be: $\{<, >, \leq, \geq, ==, !=\}$, the above mutant operator will generate *a total of 5 mutants of P*

Mutation Testing Process

- Given P and a test set T
 - Generate mutants
 - Compile P and the mutants
 - Execute P and the mutants on each test case
 - Determine equivalent mutants
 - Determine mutation score
 - If mutation score is not 1 then improve the test case and repeat from Step 3

Mutation Testing Process

- In practice the above procedure is implemented **incrementally**
- One applies **a few selected mutant operators** to P and computes the mutation score with respect to the mutants generated
- Once these mutants have been distinguished or proven equivalent, **another set of mutant operators is applied**
- This procedure is repeated until either ***all the mutants*** have been exhausted or some external condition forces testing to stop

Example of Mutation Operators I

- Constant replacement
- Scalar variable replacement
- Scalar variable for constant replacement
- Constant for scalar variable replacement
- Array reference for constant replacement
- Array reference for scalar variable replacement
- Constant for array reference replacement
- Scalar variable for array reference replacement
- Array reference for array reference replacement
- Source constant replacement
- Data statement alteration
- Comparable array name replacement
- Arithmetic operator replacement
- Relational operator replacement
- Logical connector replacement
- Absolute value insertion
- Unary operator insertion
- Statement deletion
- Return statement replacement

Example of Mutation Operators II

Specific to object-oriented programming languages:

- Replacing a type with a compatible subtype (inheritance)
- Changing the access modifier of an attribute, a method
- Changing the instance creation expression (inheritance)
- Changing the order of parameters in the definition of a method
- Changing the order of parameters in a call
- Removing an overloading method
- Reducing the number of parameters
- Removing an overriding method
- Removing a hiding Field
- Adding a hiding field

Specifying Mutations Operators

- Ideally, we would like the mutation operators to be representative of (and generate) all realistic types of faults that could occur in practice.
- Mutation operators change with programming languages, design and specification paradigms, though there is much overlap.
- In general, the number of mutation operators is large as they are supposed to capture all possible *syntactic* variations in a program.
- Recent paper suggests random sampling of mutants can be used.

Important Remarks

Right-BICEP

- **Boundary** Conditions
 - See earlier slides, and also consider:
 - Garbage input values
 - Badly formatted data
 - Empty or missing values (0, null, etc.)
 - Values out of reasonable range
 - Duplicates if they're not allowed
 - Unexpected orderings

Right-BICEP

- Use this acronym, CORRECT, to remember:
 - Conformance
 - Ordering
 - Range
 - Reference
 - Does code reference anything external outside of its control?
 - Existence
 - Cardinality
 - Time (absolute and relative)
 - Are things happening in order? On time? In time?

Right-BICEP

- Check **Inverse** Relationships
 - If your method does something that has an inverse, then apply the inverse
 - E.g. square and square-root. Insertion then deletion.
 - Beware errors that are common to both your operations
 - Seek alternative means of applying inverse if possible

Right-BICEP

- **Cross-check** using other means
- Can you do something more than one way?
 - Your way, and then the other way. Match?
- Are there overall consistency factors you can check?
 - Overall agreement

Right-BICEP

- Force **Error** Conditions
- Some are easy:
 - Invalid parameters, out of range values, etc.
- Failures outside your code:
 - Out of memory, disk full, network down, etc.
 - Can simulate such failures
 - Example: use Mock Objects

Right-BICEP

- Performance
 - Perhaps absolute performance, or
 - Perhaps how performance changes as input grows.

SYSTEM TESTING

System Testing

Overview

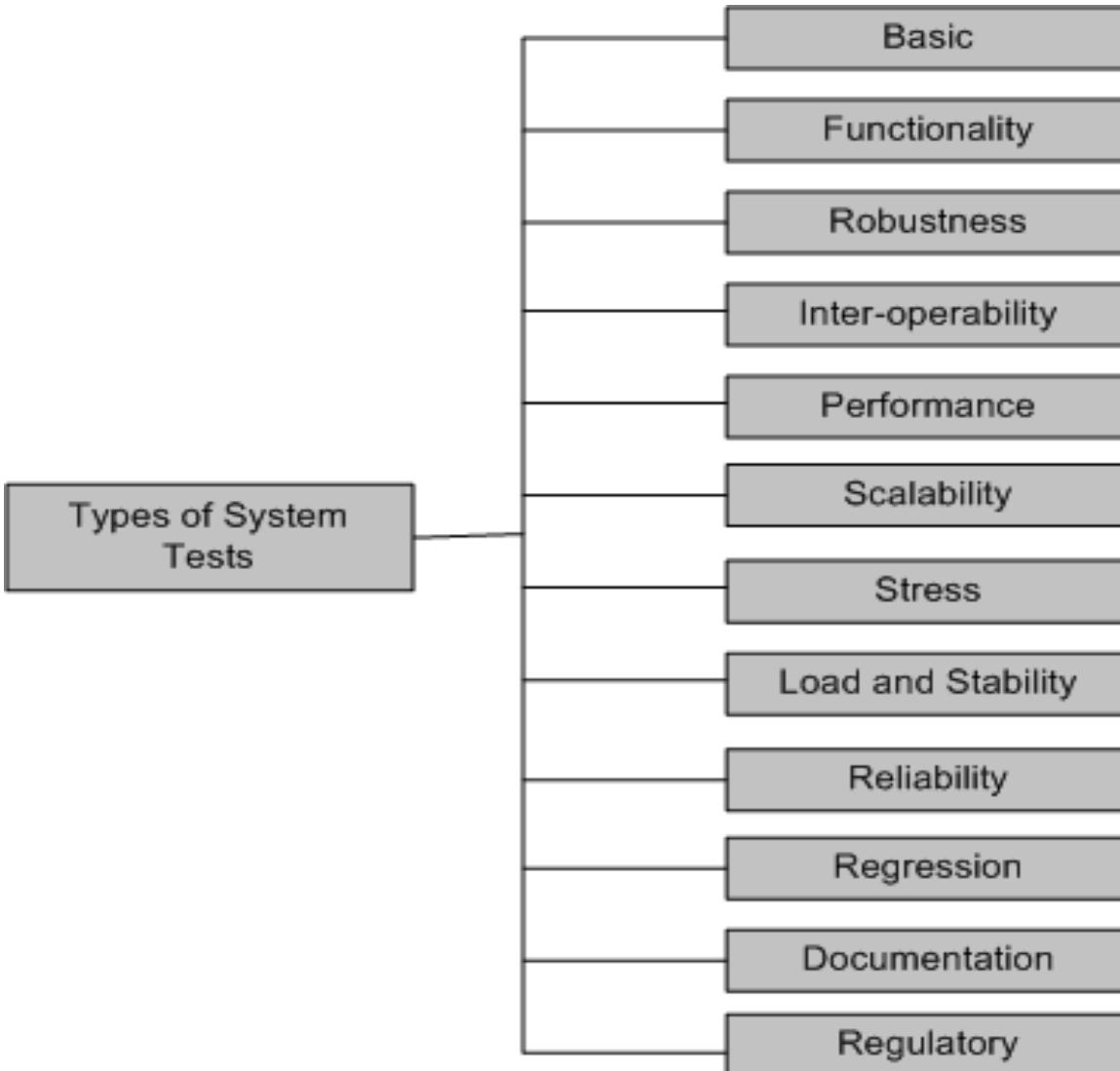
- System testing is very heterogeneous and what we include in the system test will depend on the particular application.
- The following is a list of kinds of tests we might consider applying and some assessment of their strengths and weaknesses.
- System testing can be very expensive and time consuming and can involve the construction of physical components and software to provide the test environment that may exceed the cost of the primary software development.

Different Types of System Tests

Topics covered under System Tests

- Taxonomy of System Tests
- Basic Tests
- Functionality Tests
- Robustness Tests
- Interoperability Tests
- Performance Tests
- Scalability Tests
- Stress Tests
- Load and Stability Tests
- Regression Tests
- Documentation Tests
- Regulatory Tests
 - Software Safety
 - Safety Assurance

Taxonomy of System Tests



Types of system tests

Taxonomy of System Tests

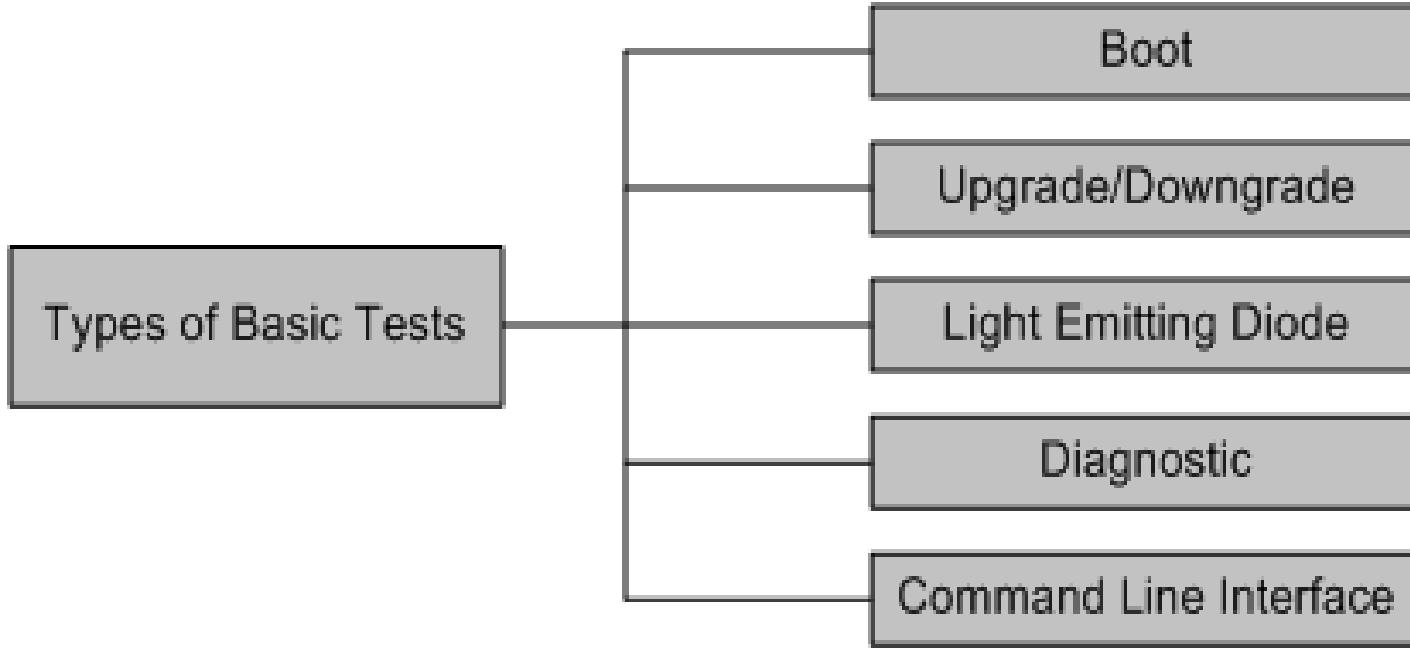
- **Basic tests** provide an evidence that the system can be installed, configured and be brought to an operational state
- **Functionality tests** provide comprehensive testing over the full range of the requirements, within the capabilities of the system
- **Robustness tests** determine how well the system recovers from various input errors and other failure situations
- **Inter-operability tests** determine whether the system can inter-operate with other third party products
- **Performance tests** measure the performance characteristics of the system, e.g., throughput and response time, under various conditions

Taxonomy of System Tests

- **Scalability tests** determine the scaling limits of the system, in terms of user scaling, geographic scaling, and resource scaling
- **Stress tests** put a system under stress in order to determine the limitations of a system and, when it fails, to determine the manner in which the failure occurs
- **Load and Stability** tests provide evidence that the system remains stable for a long period of time under full load
- **Reliability tests** measure the ability of the system to keep operating for a long time without developing failures
- **Regression tests** determine that the system remains stable as it cycles through the integration of other subsystems and through maintenance tasks
- **Documentation tests** ensure that the system's user guides are accurate and usable

What are the Basic Tests

Basic Tests



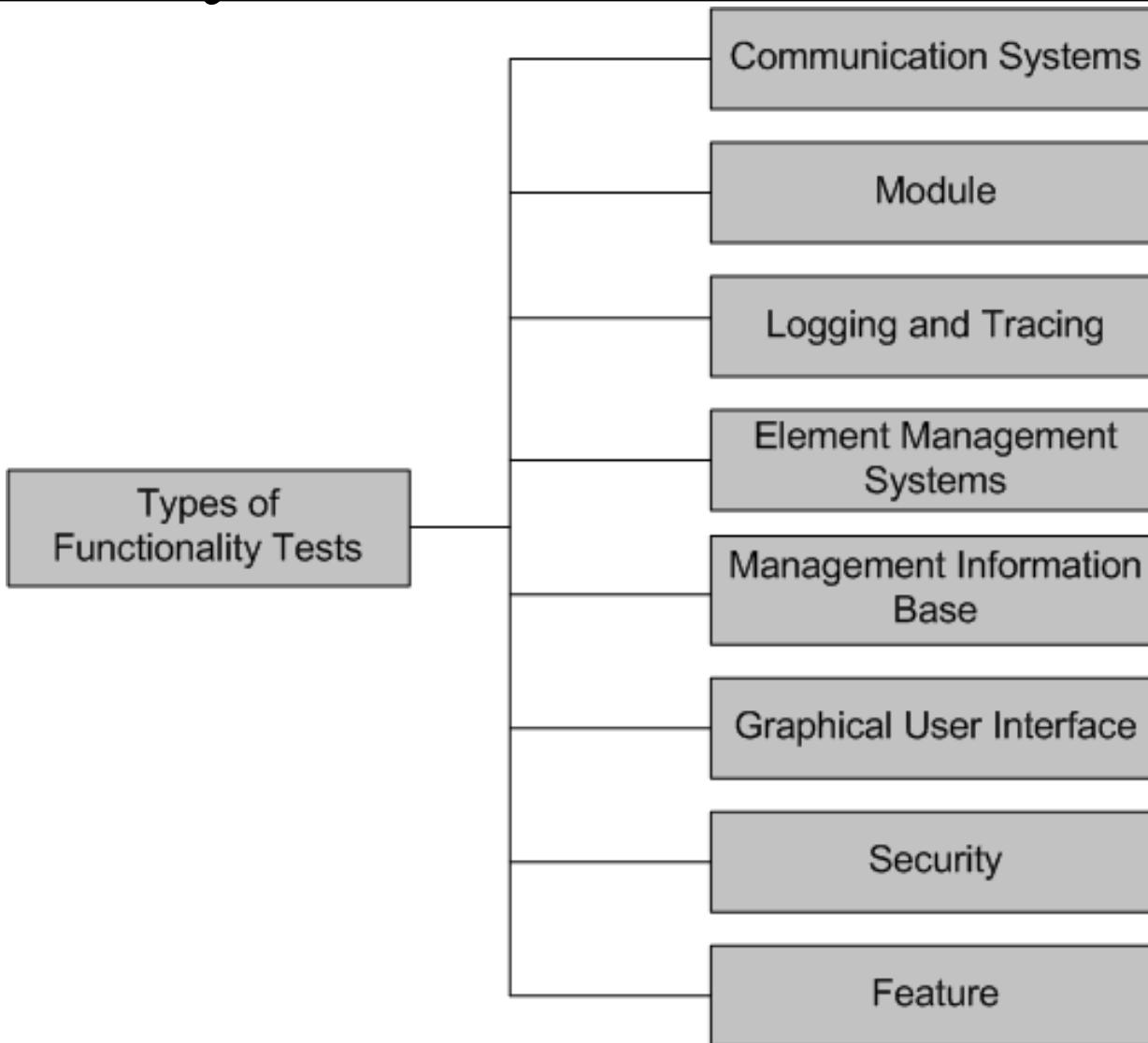
Types of basic tests

- **Boot:** Boot tests are designed to verify that the system can boot up its software image (or, build) from the supported boot options
- **Upgrade/Downgrade:** Upgrade/downgrade tests are designed to verify that the system software can be upgraded or downgraded (rollback) in a graceful manner

Basic Tests

- **Light Emitting Diode:** The LED (Light Emitting Diode) tests are designed to verify that the system LED status indicators functioning as desired
- **Diagnostic:** Diagnostic tests are designed to verify that the hardware components (or, modules) of the system are functioning as desired
 - **Power-On Self Test**
- **Command line Interface:** Command Line Interface (CLI) tests are designed to verify that the system can be configured

Functionality Tests



Types of functionality tests

Functionality Tests

- Communication Systems Tests
 - These tests are designed to verify the implementation of the communication systems as specified in the customer requirements specification
 - Four types of communication systems tests are recommended
 - Basis interconnection tests
 - Capability tests
 - Behavior tests
 - System resolution tests
- Module Tests
 - Module Tests are designed to verify that all the modules function individually as desired within the systems
 - The idea here is to ensure that individual modules function correctly within the whole system.
 - For example, an Internet router contains modules such as line cards, system controller, power supply, and fan tray. Tests are designed to verify each of the functionalities

Functionality Tests

- Logging and Tracing Tests
 - Logging and Tracing Tests are designed to verify the configurations and operations of logging and tracing
- Element Management Systems (EMS) Tests
 - EMS tests verifies the main functionalities, which are to manage, monitor and upgrade the communication systems network elements
 - It includes both EMS client and EMS servers functionalities

Functionality Tests

- Graphical User Interface Tests
 - Tests are designed to look-and-feel the interface to the users of an application system
 - Tests are designed to verify different components such as icons, menu bars, dialog boxes, scroll bars, list boxes, and radio buttons
 - The GUI can be utilized to test the functionality behind the interface, such as accurate response to database queries
 - Tests the usefulness of the on-line help, error messages, tutorials, and user manuals
 - The usability characteristics of the GUI is tested, which includes the following
 - ***Accessibility:*** Can users enter, navigate, and exit with relative ease?
 - ***Responsiveness:*** Can users do what they want and when they want in a way that is clear?
 - ***Efficiency:*** Can users do what they want to with minimum number of steps and time?
 - ***Comprehensibility:*** Do users understand the product structure with a minimum amount of effort?

Functionality Tests

- Security Tests
 - Security tests are designed to verify that the system meets the security requirements
 - Confidentiality
 - It is the requirement that data and the processes be protected from unauthorized disclosure
 - Integrity
 - It is the requirement that data and process be protected from unauthorized modification
 - Availability
 - It is the requirement that data and processes be protected form the denial of service to authorized users
 - Security test scenarios should include negative scenarios such as misuse and abuse of the software system

Functionality Tests

- Security Tests (continued) : useful types of security tests includes the following:
 - Verify that only authorized accesses to the system are permitted
 - Verify the correctness of both encryption and decryption algorithms for systems where data/messages are encoded.
 - Verify that illegal reading of files, to which the perpetrator is not authorized, is not allowed
 - Ensure that virus checkers prevent or curtail entry of viruses into the system
 - Ensure that the system is available to authorized users when a zero-day attack occurs
 - Try to identify any “backdoors” in the system usually left open by the software developers

Functionality Tests

- Feature Tests
 - These tests are designed to verify any additional functionalities which are defined in requirement specification but not covered in the functional category discussed
- Examples
 - Data conversion testing
 - Cross-functionality testing

Functional Testing

Essentially the same as black box testing

- Goal: Test functionality of system
- **Test cases are designed from the requirements analysis document (better: user manual) and centered around requirements and key functions (*use cases*)**
- The system is treated as black box.
- Unit test cases can be reused, but in end user oriented new test cases have to be developed as well.

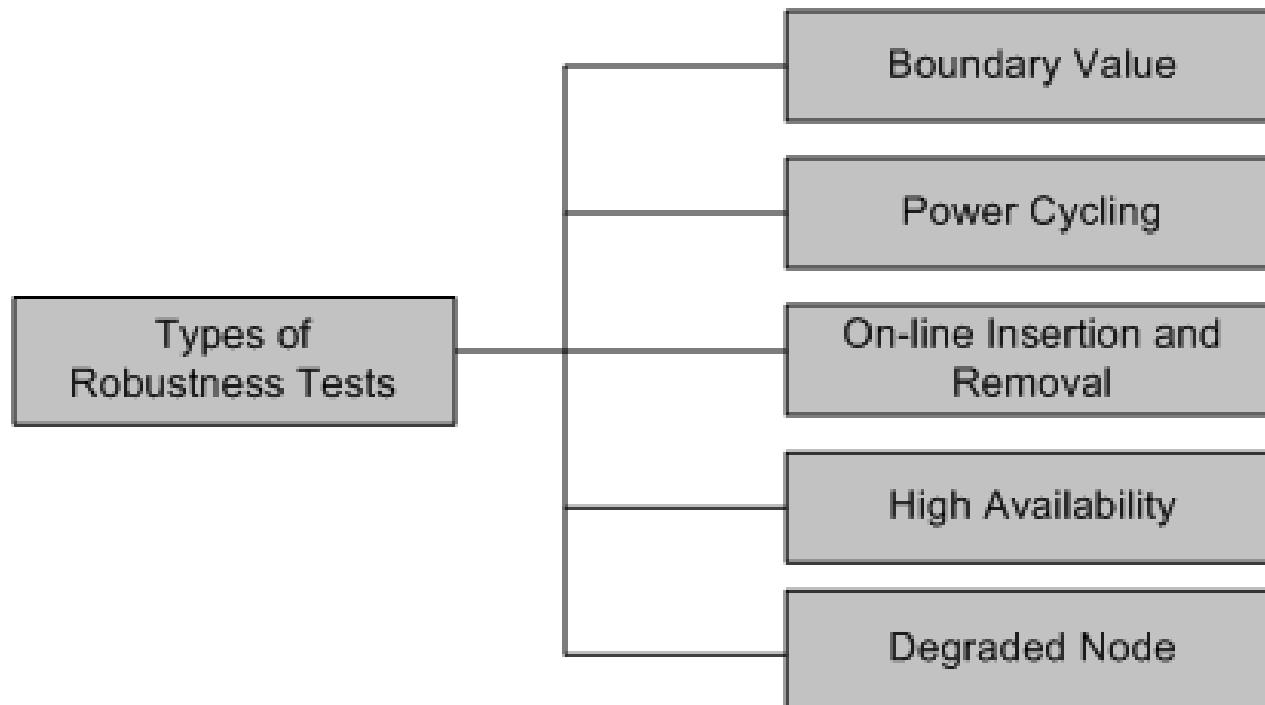
Structure Testing

- *Essentially the same as white box testing.*
- Goal: Cover all paths in the system design
 - Exercise all input and output parameters of each component.
 - Exercise all components and all calls (each component is called at least once and every component is called by all possible callers.)
 - Use conditional and iteration testing as in unit testing.

Robustness Tests

Robustness means how much sensitive a system is to erroneous input and changes its operational environment

Tests in this category are designed to verify how gracefully the system behaves in error situations and in a changed operational environment



Types of robustness tests

Robustness Tests

- Boundary value
 - Boundary value tests are designed to cover boundary conditions, special values, and system defaults
 - The tests include providing invalid input data to the system and observing how the system reacts to the invalid input.
- Power cycling
 - Power cycling tests are executed to ensure that, when there is a power glitch in a deployment environment, the system can recover from the glitch to be back in normal operation after power is restored
- On-line insertion and removal
 - On-line Insertion and Removal (OIR) tests are designed to ensure that on-line insertion and removal of modules, incurred during both idle and heavy load operations, are gracefully handled and recovered

Robustness Tests

- High Availability
 - The concept of high availability is also known as **fault tolerance**
 - High availability tests are designed to verify the redundancy of individual modules, including the software that controls these modules.
 - The goal is to verify that the system gracefully and quickly recovers from hardware and software failures without adversely impacting the operation of the system
 - High availability is realized by means of proactive methods to maximize service up-time, and to minimize the downtime
- Degraded Node
 - Degraded node (also known as failure containment) tests verify the operation of a system after a portion of the system becomes non-operational
 - It is a useful test for all mission-critical applications.

Interoperability Tests

- Tests are designed to verify the ability of the system to inter-operate with third party products
- The re-configuration activities during interoperability tests is known as configuration testing
- Another kind of inter-operability tests is called (backward) compatibility tests
 - Compatibility tests verify that the system works the same way across different platforms, operating systems, data base management systems
 - Backward compatibility tests verify that the current software build flawlessly works with older version of platforms

Performance Tests

- Tests are designed to determine the performance of the actual system compared to the expected one
- Tests are designed to verify response time, execution time, throughput, resource utilization and traffic rate
- One needs to be clear about the specific data to be captured in order to evaluate performance metrics.
- For example, if the objective is to evaluate the response time, then one needs to capture
 - End-to-end response time (as seen by external user)
 - CPU time
 - Network connection time
 - Database access time
 - Network connection time
 - Waiting time

Performance Testing

- **Stress Testing**
 - Stress limits of system (maximum # of users, peak demands, extended operation)
- **Volume testing**
 - Test what happens if large amounts of data are handled
- **Configuration testing**
 - Test the various software and hardware configurations
- **Compatibility test**
 - Test backward compatibility with existing systems
- **Security testing**
 - Try to violate security requirements
- **Timing testing**
 - Evaluate response times and time to perform a function
- **Environmental test**
 - Test tolerances for heat, humidity, motion, portability
- **Quality testing**
 - Test reliability, maintainability & availability of the system
- **Recovery testing**
 - Tests system's response to presence of errors or loss of data.
- **Human factors testing**
 - Tests user interface with user

Test Cases for Performance Testing

- Push the (integrated) system to its limits.
- **Goal: Try to break the subsystem**
- Test how the system behaves when overloaded.
 - Can bottlenecks be identified?
- Try unusual orders of execution
 - Call a receive() before send()
- Check the system's response to large volumes of data
 - If the system is supposed to handle 1000 items, try it with 1001 items.
- What is the amount of time spent in different use cases?
 - Are typical cases executed in a timely fashion?

Scalability Tests

- Tests are designed to verify that the system can scale up to its engineering limits
- Scaling tests are conducted to ensure that the system response time remains the same, or increases by a small amount, as the number of users are increased.
- There are three major causes of these limitations:
 - data storage limitations
 - network bandwidth limitations
 - speed limit
- Extrapolation is often used to predict the limit of scalability

Stress Tests

- The goal of stress testing is to evaluate and determine the behavior of a software component while the offered load is in excess of its designed capacity
- The system is deliberately stressed by pushing it to and beyond its specified limits
- It ensures that the system can perform acceptably under worst-case conditions, under an expected peak load. If the limit is exceeded and the system does fail, then the recovery mechanism should be invoked
- Stress tests are targeted to bring out the problems associated with one or more of the following:
 - Memory leak
 - Buffer allocation and memory carving

Load and Stability Tests

- Tests are designed to ensure that the system remains stable for a long period of time under full load
- When a large number of users are introduced and applications that run for months without restarting, a number of problems are likely to occur:
 - the system slows down
 - the system encounters functionality problems
 - the system crashes altogether
- Load and stability testing typically involves exercising the system with virtual users and measuring the performance to verify whether the system can support the anticipated load
- This kind of testing help one to understand the ways the system will fare in real-life situations

Reliability Tests

- Reliability tests are designed to measure the ability of the system to remain operational for long periods of time.
- The reliability of a system is typically expressed in terms of mean time to failure (MTTF)
- The average of all the time intervals between successive failures is called the MTTF
- After a failure is observed, the developers analyze and fix the defects, which consumes some time – let us call this interval the repair time.
- The average of all the repair times is known as the mean time to repair (MTTR)
- Now we can calculate a value called mean time between failure (MTBF) as $MTBF = MTTF + MTTR$
- The random testing technique is used for reliability measurement

Regression Tests

- In this category, new tests are not designed, instead, test cases are selected from the existing pool and executed
- The main idea in regression testing is to verify that no defect has been introduced into the unchanged portion of a system due to changes made elsewhere in the system
- During system testing, many defects are revealed and the code is modified to fix those defects
- One of four different scenarios can occur for each fix:
 - The reported defect is fixed
 - The reported defect could not be fixed inspite of making an effort
 - The reported defect has been fixed, but something that used to work before has been failing
 - The reported defect could not be fixed inspite of an effort, and something that used to work before has been failing

Regression Tests

- One possibility is to re-execute every test case from version $n - 1$ to version n before testing anything new
- A full test of a system may be prohibitively expensive.
- A subset of the test cases is carefully selected from the existing test suite to
 - maximize the likelihood of uncovering new defects
 - reduce the cost of testing

Documentation Tests

- Documentation testing means verifying the technical accuracy and readability of the user manuals, tutorials and the on-line help
- Documentation testing is performed at three levels:
 - ***Read test:*** In this test a documentation is reviewed for clarity, organization, flow, and accuracy without executing the documented instructions on the system
 - ***Hands-on test:*** Exercise the on-line help and verify the error messages to evaluate their accuracy and usefulness.
 - ***Functional test:*** Follow the instructions embodied in the documentation to verify that the system works as it has been documented.

Regulatory Tests

- In this category, the final system is shipped to the regulatory bodies in those countries where the product is expected to be marketed
- The idea is to obtain compliance marks on the product from various countries
- Most of these regulatory bodies issue safety and EMC (electromagnetic compatibility)/ EMI (electromagnetic interference) compliance certificates (emission and immunity)
- The regulatory agencies are interested in identifying flaws in software that have potential safety consequences
- The safety requirements are primarily based on their own published standards

Software Safety

- A *hazard* is a state of a system or a physical situation which when combined with certain environmental conditions, could lead to an *accident* or *mishap*
- An *accident* or *mishap* is an unintended event or series of events that results in death, injury, illness, damage or loss of property, or harm to the environment
- Software *safety* is defined in terms of hazards

Software Safety

Examples:

- A software module in a database application is not hazardous by itself, but when it is embedded in a missile navigation system, it could be hazardous
- If a missile takes a U-turn because of a software error in the navigation system, and destroys the submarine that launched it, then it is not a safe software

Safety Assurance

- There are two basic tasks performed by a **safety assurance** engineering team:
 - Provide methods for identifying, tracking, evaluating, and eliminating hazards associated with a system
 - Ensure that safety is embedded into the design and implementation in a timely and cost effective manner, such that the risk created by the user/operator error is minimized

Acceptance Testing

- **Goal: Demonstrate system is ready for operational use**
 - Choice of tests is made by *client*/sponsor
 - Many tests can be taken from integration testing
 - Acceptance test is performed by the client, not by the developer.
- Majority of all bugs in software is typically found by the client after the system is in use, not by the developers or testers. Therefore two kinds of additional tests:
 - ***Alpha* test:**
 - Sponsor uses the software at the *developer's site*.
 - Software used in a controlled setting, with the developer always ready to fix bugs.
 - ***Beta* test:**
 - Conducted at *sponsor's site* (developer is not present)
 - Software gets a realistic workout in target environment
 - Potential customer might get discouraged

Testing has its own Life Cycle

Establish the test objectives

Design the test cases

Write the test cases

Test the test cases

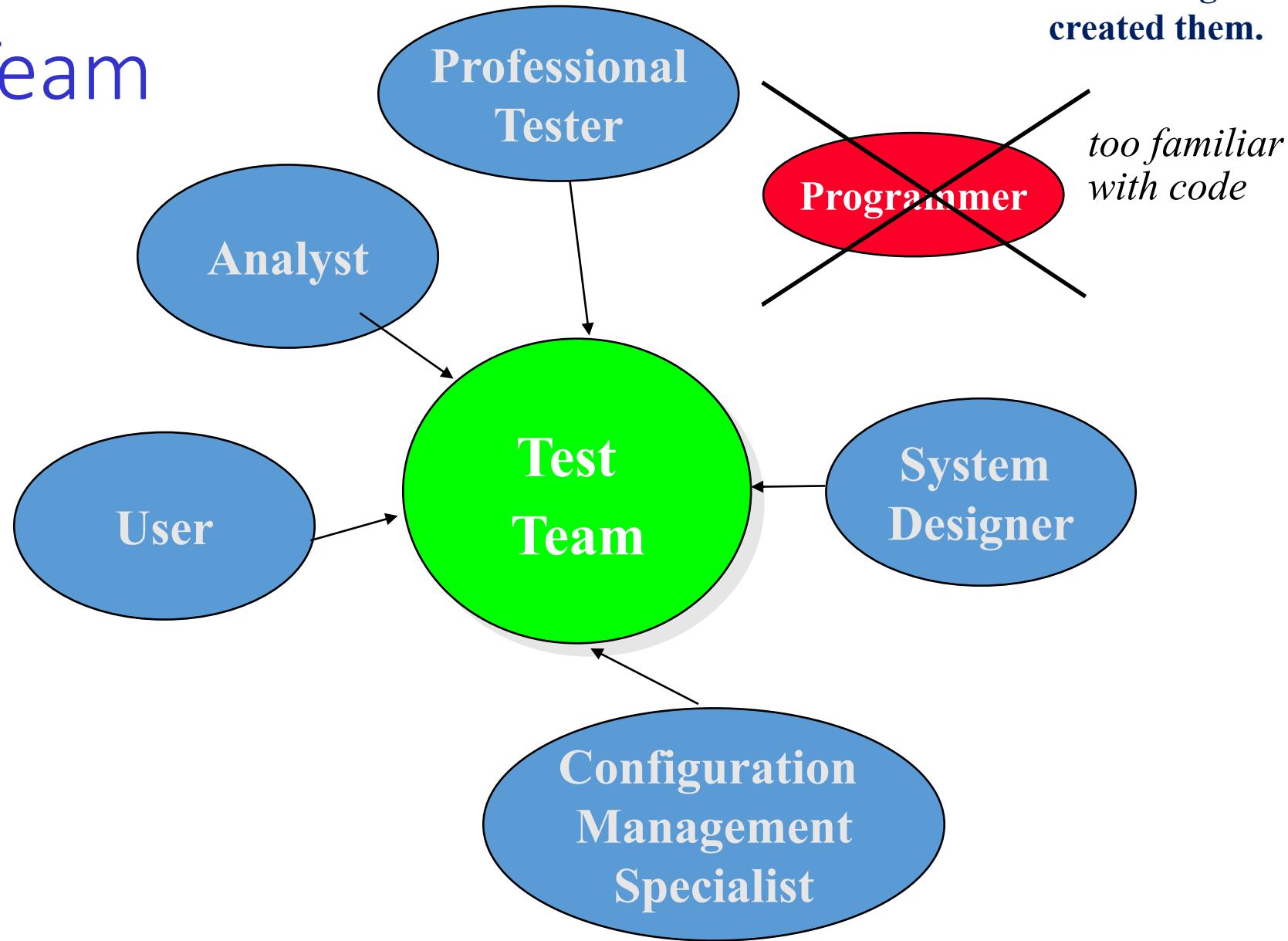
Execute the tests

Evaluate the test results

Change the system

Do regression testing

Test Team



Albert Einstein - We can't solve problems by using
the same kind of thinking we used when we
created them.

Test Plan

Test Plans – A Critical Document.

- **Document contains a complete set of test cases for a system**
 - Along with other information about the testing process.
 - Who does what; when; how; anticipated results;
- The **Test Plan** is one of the standard forms of documentation.
- Test Plan - **Written long before** the testing starts.
- **Test Plan Development:**
 - once you have developed the requirements, planning for testing should start to occur at ‘that’ time.
 - Many types of tests can be developed directly from Use Cases.
- **If a project does not have a test plan:**
 - Testing will inevitably be done in an ad-hoc manner.
 - This will lead to **poor quality software**.

How to prepare a Test Plan

The Roles of People Involved in Testing

The Roles of People Involved in Testing

- The first pass of unit and integration testing is called *developer testing*.
 - Preliminary testing performed by the software developers who do the design and programming
- *Independent testing* is performed (often) by a separate group.
 - They do not have vested interest in seeing as many test cases pass as possible.
 - They develop specific expertise in how to do good testing, and how to use testing tools.
 - Done after ‘unit’ testing. Perhaps for System Testing < Deployment
 - Can bet there is an independent test group in medium to large organizations.

Testing Performed by Users and Clients

Testing Performed by Users and Clients

- *Alpha testing*
 - Performed by the user or client, but under the supervision of the software development team.
 - Generally software is clearly not ready for release; maybe partial.
 - Maybe testing a particular major feature...
- *Beta testing*
 - Performed by the user or client in a normal work environment.
 - May be run at client shop...
 - Recruited from the potential user population.
 - An *open beta release* is the release of low-quality software to the general population. (not too low now...)
- *Acceptance testing*
 - Performed by users and customers.
 - However, the customers exercise the application on their own initiative often in **real live situations**, loading, etc..

Levels of test cases

All Test Cases are Not Created Equal...

- **Level 1:**
 - Critical test cases.**
 - Designed to verify the system runs and is safe.
- **Level 2:**
 - General test cases.
 - Verify that **day-to-day functions** correctly.
 - Still permit testing of other aspects of the system.
- **Level 3:**
 - Detailed test cases.
 - Test requirements that are of **lesser importance**.
- Somewhere in these levels may very well be the testing of **non-functional requirements** such as testing under **peak loading** (numbers of simultaneous users, files being accessed, etc.), **reliability**, **performance**, **recovery**, **simultaneous / concurrent database access**, remote object interfacing, ... and more.

Elements of a test plan 1

- Title
- Identification of software (incl. version/release #s)
- Revision history of document (incl. authors, dates)
- Table of Contents
- Purpose of document, intended audience
- Objective of testing effort
- Software product overview
- Relevant related document list, such as requirements, design documents, other test plans, etc.
- Relevant standards or legal requirements
- Traceability requirements

Elements of a test plan 2

- Relevant naming conventions and identifier conventions
- Overall software project organization and personnel/contact-info/responsibilities
- Test organization and personnel/contact-info/responsibilities
- Assumptions and dependencies
- Project risk analysis
- Testing priorities and focus
- Scope and limitations of testing
- Test outline - a decomposition of the test approach by test type, feature, functionality, process, system, module, etc. as applicable
- Outline of data input equivalence classes, boundary value analysis, error classes

Elements of a test plan 3

- Test environment - hardware, operating systems, other required software, data configurations, interfaces to other systems
- Test environment validity analysis - differences between the test and production systems and their impact on test validity.
- Test environment setup and configuration issues
- Software migration processes
- Software CM processes
- Test data setup requirements
- Database setup requirements
- Outline of system-logging/error-logging/other capabilities, and tools such as screen capture software, that will be used to help describe and report bugs

Elements of a test plan 4

- Discussion of any specialized software or hardware tools that will be used by testers to help track the cause or source of bugs
- Test automation - justification and overview
- Test tools to be used, including versions, patches, etc.
- Test script/test code maintenance processes and version control
- Problem tracking and resolution - tools and processes
- Project test metrics to be used
- Reporting requirements and testing deliverables
- Software entrance and exit criteria
- Initial sanity testing period and criteria
- Test suspension and restart criteria

Elements of a test plan 5

- Personnel allocation
- Personnel pre-training needs
- Test site/location
- Outside test organizations to be utilized and their purpose, responsibilities, deliverables, contact persons, and coordination issues
- Relevant proprietary, classified, security, and licensing issues.
- Open issues
- Appendix - glossary, acronyms, etc.

The “Test-Fix-Test” Cycle

The “Test-Fix-Test” Cycle

- When a failure occurs during testing:

- Each failure report should enter into a failure tracking system. These are likely formalized.
- Screened and assigned a Priority.
- Known Bugs List: Low-priority failures might be put here.
 - May be included with the software’s release notes.
- Somebody/some office is assigned to investigate a failure.
- That person/office tracks down the defect and fixes it.
- Finally a new version of the system is created, ready to be tested again.

The Ripple Effect – Regression Errors

The Ripple Effect – Regression Errors

- **High probability efforts to remove defects adds new defects**
- **How so?**
 - The maintainer tries to fix problems without fully understanding the consequences of the changes.
 - The maintainer makes ordinary human errors
 - The system *regresses* into a more and more failure-prone state
 - **Maintenance programmer given insufficient time to properly analyze, repair, and adequately repair due to pressures.**
- **Regression Testing**
 - Based on storing / documentation of previous tests.

Regression Testing

Regression Testing

- Is far too expensive to re-run every single test case every time a change is made to software.
- Only a subset of the previously-successful test cases is re-run.
- This process is called *regression testing*.
 - The tests that are re-run are called **regression tests**.
- Regression test cases - carefully selected to cover as much of the system as possible that can in any way have been affected by a maintenance change or any kind of change.

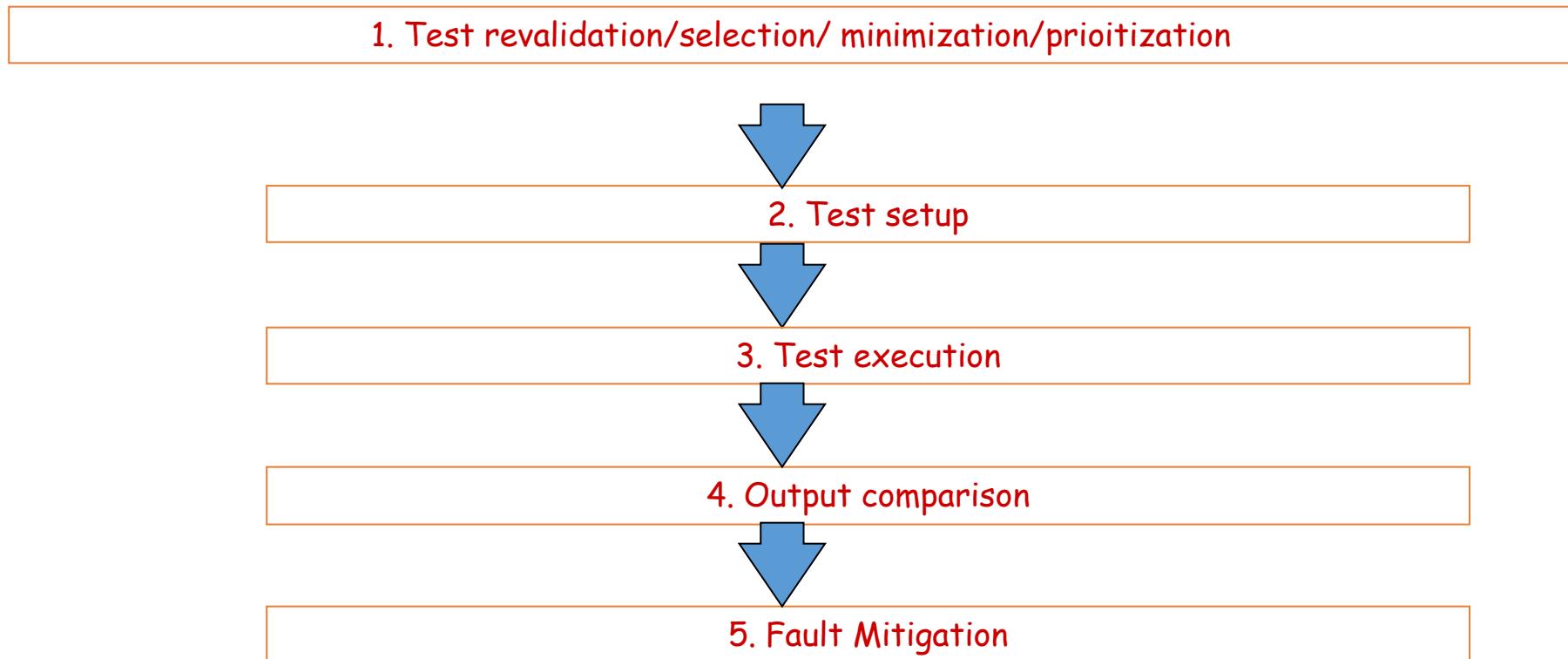
What is it?

- ❑ **Regression testing** refers to the portion of the test cycle in which a program is tested to ensure that changes do not affect features that are not supposed to be affected.
- ❑ **Corrective** regression testing is triggered by corrections made to the previous version; **progressive** regression testing is triggered by new features added to the previous version.

Develop-Test-Release Cycle

Version 1	Version 2
<ol style="list-style-type: none">1. Develop P2. Test P3. Release P	<ol style="list-style-type: none">1. Modify P to P'2. Test P' for new functionality3. Perform regression testing on P' to ensure that the code carried over from P behaves correctly4. Release P'

Regression-Test Process



Major Tasks

- **Test revalidation** refers to the task of checking which tests for P remain valid for P'.
- **Test selection** refers to the identification of tests that traverse the modified portions in P'.
- **Test minimization** refers to the removal of tests that are seemingly redundant with respect to some criteria.
- **Test prioritization** refers to the task of prioritizing tests based on certain criteria.