Software defines behavior:

network routers, finance, switching networks, other infrastructure

Software Faults, Errors & Failures:

□ Fault: A bug in the code (defect).
□ Error: Incorrect internal state due to a fault.
□ Failure: Observed incorrect external behavior when the software runs.
Analogy: Like a doctor diagnosing symptoms (failures), ailments (faults), and internal anomalies (errors).

Real-World Impacts of Poor Testing

Famous failures: Mars Polar Lander, THERAC-25, Boeing 737 Max, Amazon BOGO error, etc.

Validation vs. Verification

□ Validation: "Are we building the right product?" (meets user needs)
□ Verification: "Are we building the product right?" (meets design specs)
□ IV&V: Independent Verification and Validation—done by external teams.

Testing Goals Based on Test Process Maturity

Level	Purpose	Notes
0	No distinction between testing & debugging	Basic mindset; no reliability focus
1	Show correctness	But correctness is impossible to prove fully
2	Find failures	Negative view, can be adversarial
3	Reduce risk	More collaborative; realistic
4	Improve quality through discipline	Strategic mindset; test engineers become leaders

Testing Strategy & Cost Justification

□ Must plan tests early—align with functional requirements.
□ Coverage and test objectives must be defined.
□ Cost of not testing is always higher than the cost of testing.

Complexity of Testing Software

To manage complexity, testers use abstraction, especially through Model-Driven Test Design (MDTD).

Note! Testing only proves the presence of failures, not their absence.

Testing & Debugging

□ Testing = Running software to observe behavior.
□ Test Failure (positive result in testing) = Fault is activated.
□ Debugging = Locating and fixing the fault.
Not all inputs trigger faults—even with bugs, failure might not occur unless activated.

Fault & Failure Model (RIPR)

- For a failure to be observed, four conditions (RIPR) must occur:
    1. Reachability – The location or locations in the program that contain the fault must be reached (fault is executed.)
    2. Infection – internal state becomes incorrect.
    3. Propagation – incorrect state affects output.
    4. Revealing – incorrect output must be visible to tester.

 Visual of RIPR process: a test must reach the fault, cause infection, propagate to final state, and be revealed by test oracle.

 Important for understanding why not all bugs result in visible failures.

Software Testing Activities

 Test Engineer handles technical test work (designs, runs, analyzes).

 Test Manager handles resource/budget/coordination, not actual test design.

 Engineers do not fix bugs, only report/analyze.

Traditional Testing Levels

- Levels of testing:
    - Unit (individual methods)
    - Module (single classes/files)
    - Integration (modules together)
    - System (entire system)
    - Acceptance (end-user approval)

Object-Oriented Testing Levels

- Tailored for OOP:
    - Intra-method (each method)
    - Inter-method (method pairs)
    - Intra-class (sequence of calls in a class)
    - Inter-class (multiple classes together)

Coverage criteria

help testers pick fewer inputs that expose most bugs.

- Advantages of Coverage Criteria
    - Benefits:
        - Efficient testing.
        - Easier regression testing.
        - Traceability from requirements to tests.
        - Clear stopping rule.
        - Good tool support.

Test Requirements and Criteria

- Test Requirements (TR): Statements that must be tested.
- Test Criteria: Rules that guide creation of TRs.

Old View: Colored Boxes

- Black-box testing: Derive tests from external descriptions of the software, including specifications, requirements, and design
- White-box testing: Derive tests from the source code internals of the software, specifically including branches, individual conditions, and statements
- Model-based testing: Based on diagrams/models.

Model-Driven Test Design

- Test design is about creating effective input values.
- It's one of the most mathematical and technical testing activities.

Types of Test Activities

- Four test types:
    1. Test Design (criteria or human-based)
    2. Test Automation
    3. Test Execution
    4. Test Evaluation

Test Design—(a) Criteria-Based

- Based on formal rules/coverage.
- Requires knowledge of:
    - Programming
    - Discrete math
    - Testing
- Very technical and essential for automated test generation.

Test Design—(b) Human-Based

- Based on domain and human knowledge.
- Useful for UI/UX, edge cases, and non-obvious failures.
- Doesn't require CS background but does require strong domain expertise.

Test Automation

- Embeds test inputs into scripts.
- Requires:
    - Less theory
    - Some programming
    - Dealing with control/observation difficulties
- Test evaluators must validate expected outputs; designers may not know them.

Test Execution

- Definition: Running tests on the software and recording the results.
- Key Points:
  - It's easy and can be automated, requiring only basic computer skills.
  - Executing tests doesn't require the test designer — using them here is wasteful.
  - Manual execution (especially GUI) is labor-intensive.
  - Usability testing is often run as a dry run using standards like ISO 9241.
  - Requires care and detailed bookkeeping.

Test Evaluation

- Definition: Evaluate results and report findings to developers.
- Key Points:
  - Often harder than expected; not just pass/fail checking.
  - Requires knowledge in testing, domain context, psychology, UI.
  - Doesn't usually require a traditional CS degree.
  - Microsoft approach: Report, Replicate, Repair.
  - Intellectually rewarding but may not appeal to traditional CS students.

Other Activities

- Test Management: Policy-making, planning, coordination, and tool/criteria selection.
- Test Maintenance: Ensuring reusability, trimming unneeded tests, config control.
- Test Documentation:
  - Captures the why behind each test.
  - Ensures traceability.
  - Involves all roles (designers, automators, etc.).

Using MDTD in Practice

- One test designer does the mathematical model work.
- Then:
  - Others find values, automate, run, and evaluate tests.
- Parallels traditional engineering roles.
- Test designers = technical experts.

Observability and Controllability

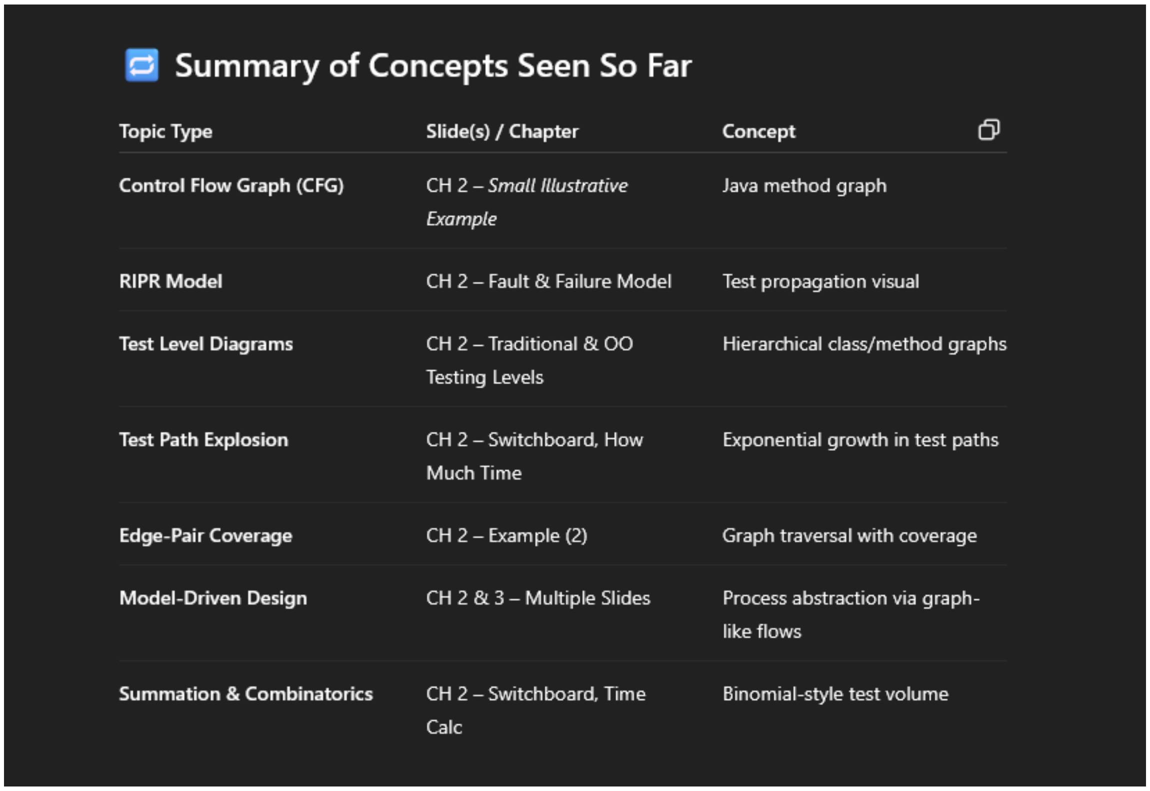
- Observability: Ease of viewing program behavior (outputs, environment, hardware interaction).
- Controllability: Ease of providing correct inputs to software.
- Note: Data abstraction and complex environments (sensors, databases) reduce both.

Affecting Controllability and Observability

- Prefix values: Set up the software state for testing.
- Postfix values: Restore or verify software state after test.
- Examples:
    - Verification values to check outputs.
    - Reset values to bring software back to stable state.

Roles

- Four main roles in testing:
    1. Test Design (criteria-based, human-based)
    2. Test Automation
    3. Test Execution
    4. Test Evaluation

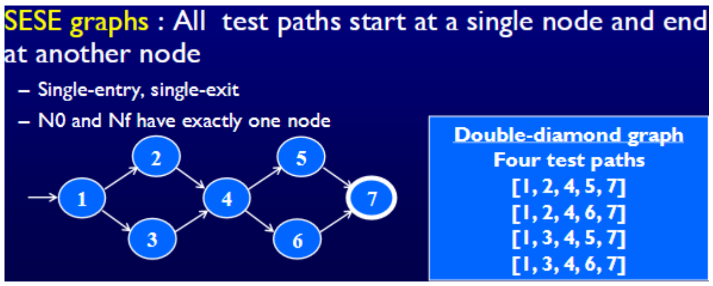


Summary of Concepts Seen So Far

Topic Type	Slide(s) / Chapter	Concept
Control Flow Graph (CFG)	CH 2 – *Small Illustrative Example*	Java method graph
RIPR Model	CH 2 – Fault & Failure Model	Test propagation visual
Test Level Diagrams	CH 2 – Traditional & OO Testing Levels	Hierarchical class/method graphs
Test Path Explosion	CH 2 – Switchboard, How Much Time	Exponential growth in test paths
Edge-Pair Coverage	CH 2 – Example (2)	Graph traversal with coverage
Model-Driven Design	CH 2 & 3 – Multiple Slides	Process abstraction via graph-like flows
Summation & Combinatorics	CH 2 – Switchboard, Time Calc	Binomial-style test volume

Definition of a Graph

- Formal graph definition includes:
    - Nodes NNN
    - Initial nodes N0N_0N0
    - Final nodes NfN_fNf
    - Edges EEE as ordered pairs (ni,nj)(n_i, n_j)(ni,nj)


SESE graphs : All test paths start at a single node and end at another node

– Single-entry, single-exit

– N0 and Nf have exactly one node

Double-diamond graph

Four test paths

[1, 2, 4, 5, 7]

[1, 2, 4, 6, 7]

[1, 3, 4, 5, 7]

[1, 3, 4, 6, 7]

– Syntactic reach : A subpath exists in the graph

– Semantic reach : A test exists that can execute that subpath

Testing and Covering Graphs

• Test Criterion : Rules that define test requirements

• Test Requirements (TR) : Describe properties of test paths

• Satisfaction : Given a set TR of test requirements for a criterion C, a set of tests T satisfies C on a graph if and only if for every test requirement tr in TR, there is a test path in path(T) that meets the test requirement tr

• Structural Coverage Criteria : Defined on a graph just in terms of nodes and edges

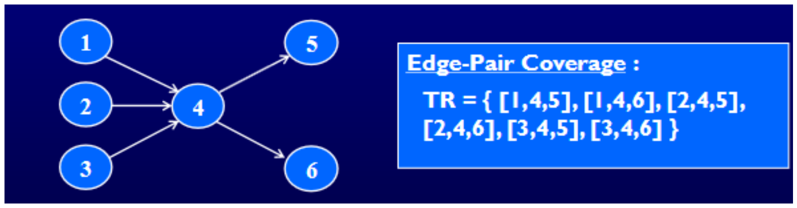
• Data Flow Coverage Criteria : Requires a graph to be annotated with references to variables

Node and Edge Coverage

- Node Coverage (NC): each reachable node must be visited.
- Edge Coverage (EC): each edge (length-1 path) must be covered.
- EC is stronger than NC.

Edge-Pair Coverage

- EPC: All subpaths of length ≤ 2 must be covered.


Edge-Pair Coverage :

TR = { [1,4,5], [1,4,6], [2,4,5], [2,4,6], [3,4,5], [3,4,6] }

Complete Path Coverage

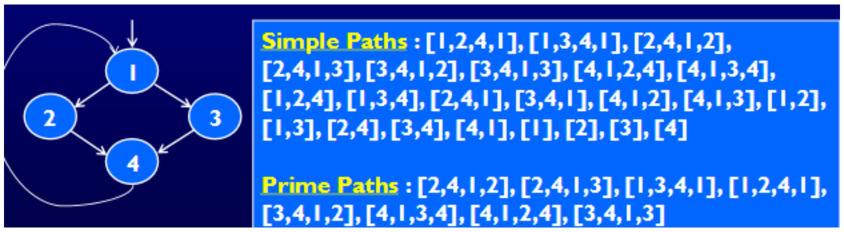
- CPC: All paths must be covered—impractical if graph has loops.
- Introduces SPC (Specified Path Coverage): only a given set of paths is required.

Handling Loops in Graphs

- Loops → infinite paths → CPC not feasible.

Simple & Prime Paths

- Simple Path: no node repetition (except maybe start/end).
- Prime Path: Simple path not a subpath of another.

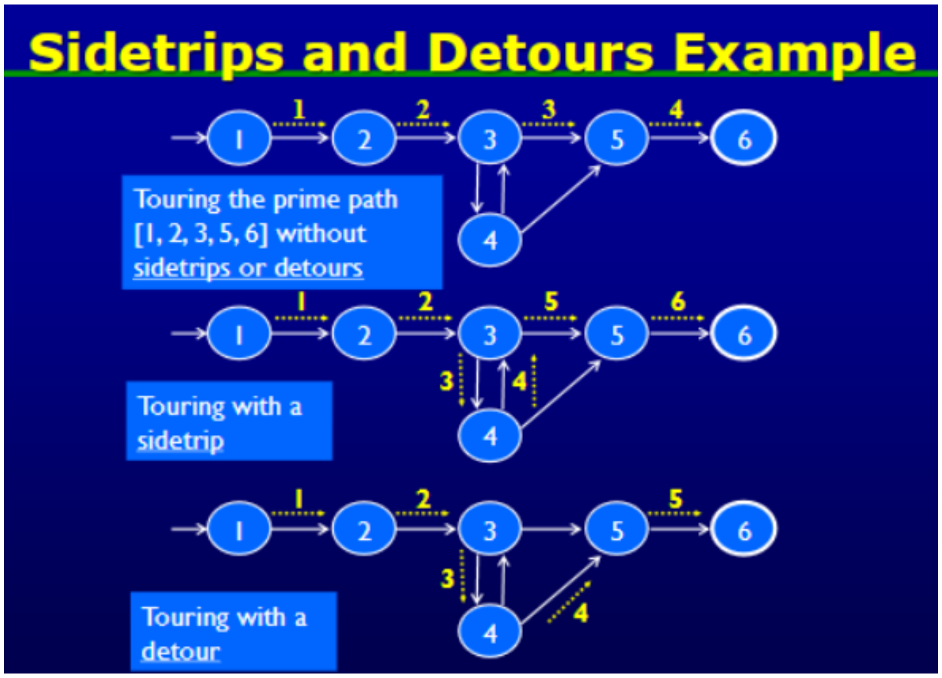


Simple Paths : [1,2,4,1], [1,3,4,1], [2,4,1,2], [2,4,1,3], [3,4,1,2], [3,4,1,3], [4,1,2,4], [4,1,3,4], [1,2,4], [1,3,4], [2,4,1], [3,4,1], [4,1,2], [4,1,3], [1,2], [1,3], [2,4], [3,4], [4,1], [1], [2], [3], [4]

Prime Paths : [2,4,1,2], [2,4,1,3], [1,3,4,1], [1,2,4,1], [3,4,1,2], [4,1,3,4], [4,1,2,4], [3,4,1,3]

Prime Path Coverage

- PPC: each prime path must be covered.
- Subsumes node and edge coverage.
- Almost covers EPC.

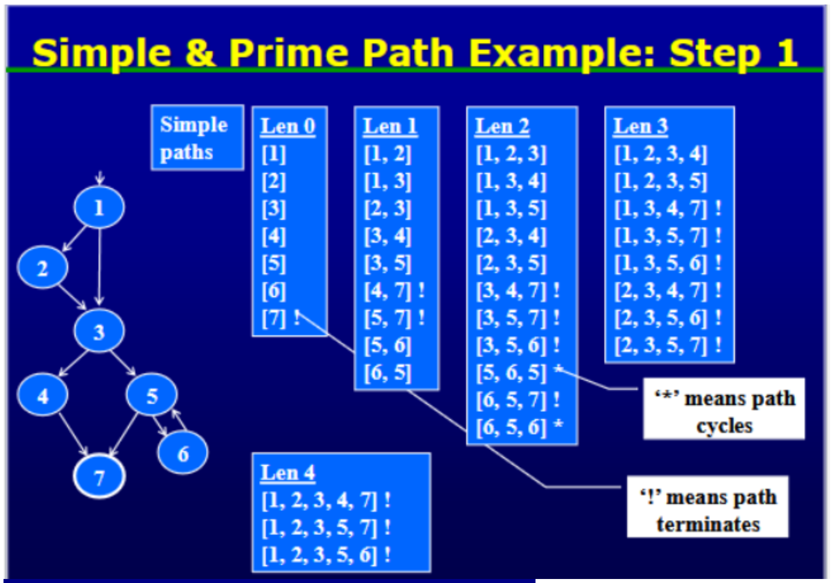


Sidetrips and Detours Example

Touring the prime path [1, 2, 3, 5, 6] without sidetrips or detours

Touring with a sidetrip

Touring with a detour



Simple & Prime Path Example: Step 1

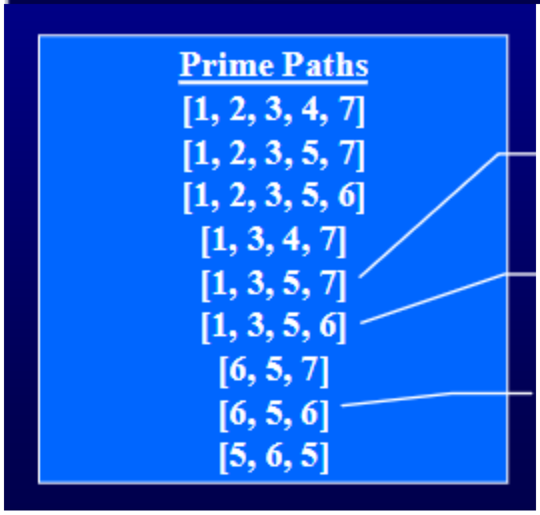
Simple paths

Len 0	Len 1	Len 2	Len 3
[1]	[1, 2]	[1, 2, 3]	[1, 2, 3, 4]
[2]	[1, 3]	[1, 3, 4]	[1, 2, 3, 5]
[3]	[2, 3]	[1, 3, 5]	[1, 3, 4, 7] !
[4]	[3, 4]	[2, 3, 4]	[1, 3, 5, 7] !
[5]	[3, 5]	[2, 3, 5]	[1, 3, 5, 6] !
[6]	[4, 7] !	[3, 4, 7] !	[2, 3, 4, 7] !
[7] !	[5, 7] !	[3, 5, 7] !	[2, 3, 5, 6] !
	[5, 6]	[3, 5, 6] !	[2, 3, 5, 7] !
	[6, 5]	[5, 6, 5] *	
		[6, 5, 7] !	
		[6, 5, 6] *	

Len 4
[1, 2, 3, 4, 7] !
[1, 2, 3, 5, 7] !
[1, 2, 3, 5, 6] !

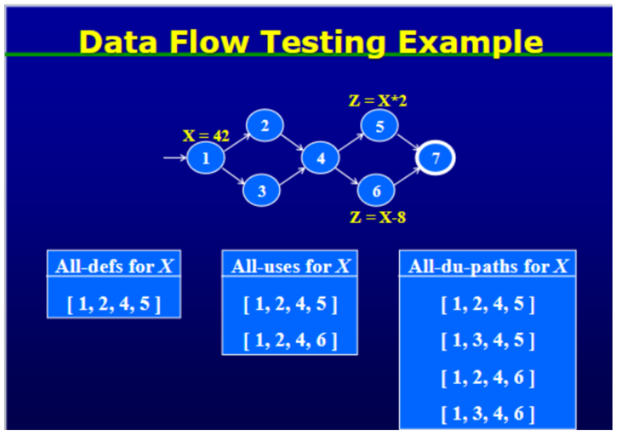
'*' means path cycles

'!' means path terminates



Prime Paths

[1, 2, 3, 4, 7]
[1, 2, 3, 5, 7]
[1, 2, 3, 5, 6]
[1, 3, 4, 7]
[1, 3, 5, 7]
[1, 3, 5, 6]
[6, 5, 7]
[6, 5, 6]
[5, 6, 5]



Data Flow Testing Example

All-defs for *X*
[1, 2, 4, 5]

All-uses for *X*
[1, 2, 4, 5]
[1, 2, 4, 6]

All-du-paths for *X*
[1, 2, 4, 5]
[1, 3, 4, 5]
[1, 2, 4, 6]
[1, 3, 4, 6]

SE317 in a Nutshell

- Key Concepts
  - Criteria-Based Testing: Based on defined coverage requirements.
  - Human-Based Testing: Based on experience and intuition.
  - Test Case: Input, prefix, assertion, expected output, postfix.
  - Automation: Scripting and continuous integration (CI/CD).
  - Parametrization: Reusing test logic with different inputs.
- System Types
  - Embedded, distributed, UI-intensive, and cloud-based systems all have unique testing challenges.

What are Neural Networks?

- □ A new way to program computers inspired by the human brain.
- □ Strengths:
  - Excellent at pattern recognition.
  - Handle problems hard to solve with traditional logic.
  - Learn, test, and correct themselves.
  - Adapt to unseen conditions (robust).
  - Exhibit graceful degradation (partial function under failure).

Background

- Artificial Neural Networks (ANNs):
  - Modeled after the brain: many interconnected neurons.
  - Learning occurs via sensing, testing, adjusting connections.
  - Modeled effectively as a graph.
  - Used in pattern recognition, data classification, etc.

Why Neural Networks?

- Comparison:
  - Neuron → Biological cell.
  - Perceptron → Mathematical model with weighted inputs and threshold-based output.
- Input (x_1, x_2...) → Multiply by weights (w_1, w_2...) → Summed → Activation.

A Neuron Model

- Fires when input > threshold.
- Learning = adjusting connection strength (weight).
- Simulation uses graphs + math to model neuron behavior.

How It Works (Math Behind NN)

- Function breakdown:
  - $g(x)=\sum xi(x)$ → summing inputs.
  - $f(g)=1$ if $g(x)\geq b$ (neuron fires), else 0.
- Neuron outputs are binary (0 or 1) based on weighted input.

Pattern Recognition

- Key NN Application:
  - Learns by training with input-output pairs.
  - Can generalize to new patterns by recognizing closest match.
- Uses feedforward architecture to associate patterns.
  - Input → Hidden Layers → Output

Interpolation vs. Extrapolation

- Interpolation: Input lies within the training data range.
- Extrapolation: Input lies outside the known data — riskier.
- Networks perform better at interpolation than extrapolation.

The Perceptron

- A refined neuron model used in ML.
- Adds:
  - Weights (w) to each input.
  - Summation + Bias.
  - Activation function to determine firing.
- Output depends on the combined weighted sum exceeding a threshold.