CS Exit Ethiopia

Revision Outlines

Computer Science Exit Examination

1 selected course | 18 November 2026

Selected courses

01. Software Engineering

https://csexitethiopia.com

abelasnake2003@gmail.com | Telegram: @Abechu7

Course contents

Each course begins on a new page. PDF bookmarks provide question, study-note section and revision-topic navigation.

Software Engineering 3

Software Engineering

Revision Outlines

Engineering processes and work products | Exam focus

- Software engineering and process
  - Systematic development, operation, maintenance
  - Product: executable code plus artifacts
  - Requirements, designs, test assets
  - Configuration and user documentation
  - Process: activities, ordering, roles, products
  - Entry/exit conditions
  - Specification → design/implementation → validation → evolution
- Decomposition and programming approaches
  - Procedural: functions
  - Object oriented: state/behavior objects
  - Either iterative or sequential
  - OOP doesn't guarantee shorter projects
  - Reuse, encapsulation, consistent models
  - Reduced duplication, easier change
  - Poor object design still complex
  - Functional: function evaluation, controlled side effects
  - Event-driven: reacts to events

Lifecycle and process selection | Exam focus

- Lifecycle phases and design levels
  - Requirements → design → implementation → testing
  - Deployment → maintenance
  - Analysis: clarify problem
  - System design: architecture
  - Object/detail design: interfaces, algorithms
  - Implementation: code
  - Testing: behavior evaluation

- Deployment: into service
- Activities overlap and repeat

• Contextual process selection
- Requirement stability and technical risk
- Delivery urgency, stakeholder availability
- Team skills and assurance needs
- Inexperience alone insufficient
- Risk-driven cycle, prototype, planned stages
- Choice depends on actual uncertainty
- Testing/documentation required in every process

Waterfall and V-model

• Classical and modified waterfall
- Planned phases, reviewed deliverables
- Stable understood requirements, familiar technology
- Late feedback raises change cost
- Controlled revisiting remains possible
- Modified waterfall: explicit feedback
- Milestone visibility
- Risk of frozen incorrect assumptions

• V-model stage/test pairing
- Requirements → acceptance tests
- System design → system tests
- Component/detail design → integration/unit tests
- Coding at bottom
- Early test planning
- Stable requirements, strong assurance fit
- Shape/process guarantees neither safety nor correctness

Iteration and incremental delivery

• Iteration versus increment
- Iteration: revisit and improve
- Increment: add usable functionality

- Processes may combine both
- Registration → payments → reporting: incremental
- Registration-prototype feedback revisions: iterative
- Each increment: design, implement, test

• Incremental-delivery benefits and limits
- Prioritized partial capability delivered early
- Smaller changes: easier debugging, feedback
- Architecture/integration still planned
- Iteration needn't rebuild whole system
- Increments needn't be independent modules
- Early version: prototype/iterative/Agile possibilities
- Purpose determines classification

Agile values and Scrum

• Agile values and work descriptions
- People/collaboration over rigid procedures
- Working results over exhaustive documents
- Customer participation over contracts
- Adaptation over fixed plans
- Right-hand values still matter
- Planning/documentation still needed
- User story: small need and value
- Acceptance criteria support story
- Epic: broader work grouping

• Scrum timeboxes and assurance
- Sprints: at most one month
- Sprint Goal, selected work, usable Increment
- Increment meets Definition of Done
- Scope discussion before/during planning allowed
- Fixed timebox, adapted content
- Sprint Goal protected
- Agile with assurance/documentation possible
- Safety-critical alone doesn't prove unsuitability

Spiral development and risk

- Risk-driven spiral cycle
  - Objectives and alternatives
  - Assess/reduce important risks
  - Develop/verify appropriate next result
  - Plan further commitment
  - Prototype risky technology before large investment
  - Risk determines work amount/type
- Spiral selection and limitations
  - Risk emphasis distinguishes spiral
  - Changing requirements alone insufficient
  - Incremental/prototyping/sequential activities incorporable
  - Uniquely most-general: textbook convention
  - Not universal theorem
  - Risk analysis needs skill
  - Low-risk small projects: excessive overhead possible

Prototypes, RAD and reuse

- Prototyping purpose and lifecycle
  - Requirements, usability, technical feasibility
  - Throwaway: discard after learning
  - Evolutionary: deliberately refine toward production
  - Quick versions may omit robustness/security/architecture
  - Evaluate before unchanged deployment
  - Prototyping may include coding
  - Construction label doesn't exclude coding
- RAD and reuse-oriented development
  - RAD: rapid construction, timeboxing
  - User participation, reusable components
  - Available users, skilled designers/developers
  - Reuse: search components → evaluate fit
  - Possible requirements adjustment
  - Design around interfaces, build gaps

- Product-line requirements/architecture/components/tests reusable

Feasibility and requirement levels | Exam focus

- Feasibility versus system analysis
  - Worth doing and deliverable?
  - Technical, economic, operational, schedule constraints
  - Existing limitations, organizational impacts
  - Benefits, cost, available skills
  - Integration risk
  - System analysis: detailed problem/behavior
  - Activities inform each other
- Requirement levels and SRS
  - Business: purpose and goals
  - User: high-level stakeholder needs
  - System: detailed services and constraints
  - Software Requirements Specification: agreed requirements
  - Module natural language: client communication
  - Analysis models: technical formalization
  - Real specifications may include diagrams

Functional rules and prohibitions | Exam focus

- Functional obligations and prohibitions
  - Observable service or behavior
  - Input/error reactions
  - Forbidden actions included
  - Intrusion sensor → alert
  - Manager-only refund approval: functional
  - Negative statement ≠ automatically nonfunctional
- Precise requirement formulation
  - Actor/trigger, action, observable result
  - Generate PDF report: functional
  - No other-customer balance disclosure: functional security
  - Broad security quality versus implementing functions

- Must/should/not-work wording: classification may remain ambiguous

Nonfunctional qualities and constraints | Exam focus

- Nonfunctional attributes and constraints
  - Performance: response time/throughput/resource use
  - Reliability: failure-free service
  - Availability: ready for use
  - Usability: task success/ease of learning
  - Security: protection
  - Compatibility: cooperating environments
  - Portability: moving environments
  - Maintainability/supportability: change, diagnosis, evolution
- Measurable qualities and classification traps
  - Search ≤2 seconds, 100 concurrent users
  - Performance requirement
  - New-user checkout ≤1 minute, unaided
  - Usability requirement
  - Use Java: technology/pseudo-requirement
  - Often separately classified system constraint
  - Dependability: reliability, safety, robustness
  - Portability not usual central dependability category
  - Peak-load degradation: primarily performance
  - Not automatically reliability

Requirement quality and traceability | Exam focus

- Requirement quality criteria
  - Correct: actual stakeholder needs
  - Complete: agreed scope, normal/exceptional behavior
  - Consistent: no contradictions
  - Clear/unambiguous: one intended interpretation
  - Feasible/realistic: achievable within constraints
  - Verifiable: objective fulfillment evidence
  - Replace vague fast/most users

- Defined metric, workload, threshold

• Traceability and conflict negotiation
  - Backward: source/rationale
  - Forward: design, code, tests
  - Requirement dependencies recorded
  - Traceability matrix links artifacts
  - Impact analysis support
  - Not merely cost/schedule ledger
  - Negotiate incompatible stakeholder demands
  - Independent incompatible approvals insufficient
  - Elicitation difficulties: scope, understanding, volatility

Elicitation, FAST and QFD | Exam focus

• Elicitation techniques
  - Interviews: stakeholder needs
  - Document analysis: manuals, contracts, records
  - Observation/ethnography: actual daily work
  - Prototypes: tacit expectations
  - Workshops/JAD: users, managers, developers
  - Facilitator and recorder
  - Consensus-building
  - Multiple sources, observed-work assumption checks

• FAST, QFD and brainstorming
  - FAST: environment elements, outputs, inputs lists
  - Consolidate lists → smaller specifications
  - QFD: customer-value prioritization
  - Normal needs: explicit
  - Expected needs: assumed
  - Exciting needs: unanticipated benefit
  - Brainstorm ideas before evaluation/prioritization
  - Not every idea becomes committed requirement

Scenarios and analysis objects | Exam focus

- Scenarios, models and analysis objects
  - Scenario: one concrete execution
  - Use case: related success/alternative scenarios
  - Functional model: use cases/scenarios
  - Structural/object model: classes
  - Dynamic model: states/interactions
  - Entity: domain information, Incident
  - Boundary: interaction mediation, ReportForm
  - Control: use-case coordination
  - UI form normally boundary, not entity
- Object identification and CRC
  - Nouns → candidate objects
  - Possessives → candidate attributes
  - Verbs manages/reports-to → candidate associations
  - Heuristics, not automatic parsing rules
  - CRC: Class, Responsibilities, Collaborators
  - Control per use-case/actor: starting heuristic
  - Not mandatory UML rule/fixed lifetime

Actors, goals and system scope | Exam focus

- Actors and external roles
  - Person, device or another system
  - One person: multiple roles
  - Multiple people: one role
  - Primary actor: seeks goal
  - Supporting actor: provides service
  - Identify main-function user groups
  - Administration/maintenance roles
  - External information-exchanging systems
- Use-case diagram scope
  - Actors outside boundary rectangle
  - Use-case ovals inside

- Goal name: verb plus noun
- Roles, goals, relationships
- No full algorithm/UI layout
- Internal component versus external actor
- Chosen system boundary determines role

Use-case flows and guarantees | Exam focus

- Complete use-case specification
  - Name/goal, actors, trigger
  - Preconditions
  - Main success flow
  - Alternatives/exceptions
  - Postconditions
  - Active subject-verb steps
  - Consistent abstraction level
  - Errors separated from normal flow
- Submit Incident scenario
  - Precondition: authenticated officer
  - Officer supplies location/description
  - System validates → records → returns ID
  - Missing location: problem and correction request
  - Failure: save nothing
  - Success: incident and ID persist
  - Failure guarantee: no partial record
  - Alternate flow: conditional variation
  - Exception flow: failure
  - Preconditions before entry, trigger starts execution

Include, extend and generalization | Exam focus

- Include versus extend
  - Include: required reused behavior
  - Submit Order includes Validate Order
  - Dashed include arrow: base → included

- Extend: conditional extension-point behavior
- Apply Discount extends Checkout
- Extend arrow: extension → base
- Base meaningful independently
- Withdrawal extends Print Receipt: reversed example

• Generalization and abstract use cases

- Specialized actor/use case → general
- Hollow triangle toward general
- Abstract: behavior for specialization
- Not directly instantiated complete use case
- Reuse alone insufficient definition
- No direct actor line insufficient definition
- Included behavior executes when inclusion reached
- Base conditional branches not necessarily all executed

UML classes and multiplicity | Exam focus

• Class diagrams and UML

- Class name, attributes, operations
- + public, - private, # protected
- ~ package visibility
- Class diagram: static types/relationships
- Object diagram: particular instances/links
- OMG maintains/standardizes UML
- Modeling language, not lifecycle process
- Structural things: conceptual/physical structural elements

• Multiplicity interpretation

- One end: objects per opposite-end object
- 1: exactly one
- 0..1: optional
- 0..*: any nonnegative number
- 1..*: at least one
- Read each end separately
- One-to-many doesn't imply composition

Association, aggregation and composition

- Association, dependency and aggregation
  - Association: related instances
  - Dashed dependency: client uses supplier
  - Generalization: solid line, hollow triangle
  - Triangle toward superclass/general element
  - Shared aggregation: hollow diamond
  - Diamond at whole end
  - Weak whole-part meaning
  - No detailed lifecycle policy implied
- Composition ownership and diamond placement
  - Filled diamond at whole end
  - At most one composite per part
  - Owned parts destroyed with composite
  - Defined detachment/transfer possible
  - Never-independent claim oversimplified
  - Car composing Engine: diamond beside Car
  - Points-to-Engine wording imprecise
  - Bank intended relationship: composition

Sequence diagrams and emergency reporting | Exam focus

- Sequence diagram semantics
  - Interactions across lifelines
  - Order/time generally downward
  - Messages between participants
  - Activation rectangles: execution
  - Creation message starts lifeline
  - X: modeled destruction
  - No garbage-collector timing guarantee
  - Synchronous call: wait
  - Asynchronous message: sender proceeds
- Emergency-report interaction example
  - Module heuristic: actor → boundary → control

- Entities follow
- Not mandatory UML column order
- FieldOfficer → ReportEmergencyButton
- EmergencyReportForm: enter details
- AcknowledgmentNotice: later response
- DispatcherStation: dispatcher side
- Initiating element: ReportEmergencyButton

Activity, state and data-flow models | Exam focus

- Activity and state-machine notation
  - Activity: workflow
  - Filled circle: start
  - Rounded actions
  - Guarded diamond: decision/alternative merge
  - Thick bars: fork/join
  - Partitions/swimlanes: responsibility
  - Join synchronizes concurrent flows
  - State machine: object lifecycle
  - Event-triggered transitions
  - Optional guards/actions
- Diagram selection
  - Sequence: participant message order
  - State machine: event-dependent state changes
  - Activity: workflow
  - Sequence/state both temporal behavior
  - Significant-state objects: entity or control
  - DFD: data stores, flows, transformations
  - Inputs → outputs
  - DFD isn't UML

System design and subsystem decomposition | Exam focus

- System versus detail design
  - Analysis model → architecture

- Design goals, subsystem responsibilities/dependencies
- Hardware mapping, data management
- Access control, control flow
- Startup/shutdown/error policies
- Detail design: APIs, parameter/return types
- Exceptions, components, algorithms
- Subsystem: manageable logical collection
- Services and interface
- Not necessarily physical replaceable artifact

• Subsystem decomposition and architecture documentation
- Group related objects/functionality
- Minimize cross-boundary associations
- Isolate communication infrastructure where useful
- Workflows inform, don't dictate internals
- Analysis: domain needs
- Subsystem decomposition: principally design result
- Architecture documents: communication, major decisions
- Not every source execution step

Cohesion, coupling and refactoring

• Cohesion, coupling and hiding
- Cohesion: related purpose within module
- Coupling: dependencies between modules
- Seek high cohesion, low coupling
- Shared mutable data increases coupling
- Internal-representation knowledge increases coupling
- Stable operations hide changeable decisions

• Cohesion types and modularity trade-offs
- Logical: similar-category operations, often flag-selected
- Temporal: same-time operations
- Functional: one task
- Split unrelated three-purpose component
- Modularity improvement

- Behavior-preserving restructuring: refactoring
- Answers needn't be mutually exclusive
- Smaller modules: understanding, independent work
- Reuse and maintenance benefits
- Too many interfaces increase complexity

Layers, tiers and architecture views

• Layers versus tiers
- Layer: logical abstraction level
- Tier: deployment separation
- Closed: only immediately lower calls
- Open: approved layer skipping
- A,B,C,D top-to-bottom
- A→B closed-compliant, A→C skips
- Lower→upper violates simple dependency rule
- Explicitly defined mediation exception

• Structures, views and architectural concerns
- Structure: elements/relations themselves
- View: selected-aspect representation
- Module: code organization
- Component-and-connector: runtime interaction
- Allocation: software→hardware/teams/environment
- Concurrency views: processes/threads
- Quality attributes/scenarios evaluate architecture
- Processor allocation, decomposition, style selection

Architecture styles and integration

• Architecture styles and trade-offs
- Repository: shared central data
- Common access versus bottleneck/schema dependence
- MVC: model, view, controller
- Domain, display, interaction separation
- Client-server: requester/provider roles

- Peer-to-peer: both roles
- Pipe-and-filter: successive data-stream transformations
- Three-tier: presentation/application logic/storage
- Four-tier variant: split presentation client/server

• Integration boundaries and architecture diagrams
- API gateway: single entry
- Routes/combines service calls
- Hides service boundaries from clients
- Central policies versus bottleneck risk
- Component diagram: components/interfaces/dependencies
- Deployment diagram: artifacts on nodes
- Required interactions/qualities guide style
- No universally best architecture

Design principles and operation contracts | Exam focus

• Design principles
- Abstraction/decomposition manage complexity
- Consistency, traceability, clear responsibilities
- Top-down: high-level refinement
- Bottom-up: existing-component assembly
- Hybrid: combine both
- SOLID: Single Responsibility
- Open-Closed: compatible extensions
- Liskov substitution
- Interface Segregation: avoid unused operations
- Dependency Inversion: depend on abstractions
- DRY: limit duplicated knowledge
- KISS: avoid unnecessary complexity
- YAGNI: avoid speculative work

• Operation contracts and full features
- Preconditions, postconditions, invariants
- withdraw requires amount>0
- Also amount<=balance

- Ensures balance=oldBalance-amount
- Invariant: balance>=0
- Validation: input checks
- Implementation: enforce transition
- UI/API plus persisted behavior required
- Diagram alone insufficient

Analysis-to-implementation example | Exam focus

- Submit Incident implementation responsibilities
  - Boundary: input
  - Service: validation/coordination
  - Repository: save Incident entity
  - require authenticated
  - missing(location) → validationError
  - new Incident(input) → repository.save
  - Return incident.id
  - Save and ID assignment atomic
  - Failure requires rollback
- Model-derived tests and traceability
  - Missing location: saves nothing
  - Valid input: saved ID returned
  - Storage failure: error, no partial record
  - Unauthorized input rejected
  - Requirement → use-case step → service operation
  - Repository effect → test
  - Analysis/design made actionable
  - Implementation-practice gap closed

Patterns, variability and testability

- Design patterns
  - Factory Method: subclass-selectable creation operation
  - Reduced concrete-constructor coupling
  - Simple factory: centralized creation

- Not identical to Factory Method
- Singleton: controlled one-instance access
- Observer: notify subscribers
- Memento: store/restore state, representation hidden
- Iterator: traversal
- Decorator: behavior through wrapping
- Bank Momento means Memento

• Variability and testability
- Selectable features, omission/inclusion, extension points
- Overloading: alternate code signatures
- Not every variability mechanism
- Controllable inputs, observable outputs
- Replaceable dependencies, limited structural complexity
- Sandbox: execution isolation
- Self-referential A-and-B option B
- No unique answer established

Planning, WBS, estimation and tools

• Continuous planning and WBS
- Scope/deliverables identification
- Work decomposition
- Effort/duration/cost estimation
- Resources assigned, dependencies sequenced
- Revise using new evidence
- WBS: hierarchical project-scope coverage
- Exclude unrelated work
- Tools coordinate ownership/dependencies/progress
- Judgment not replaced
- No automatic bug repair

• Functional size and development tools
- Function points: inputs, outputs, inquiries
- Internal logical files
- External interface files

- Weights depend on chosen method
- Not merely lines of code
- Not exact direct complexity measure
- Git: version tracking
- Jira: work/defect tracking
- Jenkins: automated CI checks
- Docker: application/runtime dependency containers

Critical-path forward and backward passes

- Critical-path network calculations
  - ES=max(predecessor EF)
  - EF=ES+duration
  - Start ES=0 unless constrained
  - Earliest project finish=max(terminal EF)
  - Backward LF=min(successor LS)
  - LS=LF-duration
  - Backward pass starts required finish
  - Float=LS-ES=LF-EF
  - Critical path: longest total duration
  - Normally zero float, unconstrained schedule
- Parallel tasks and multiple critical paths
  - A=5; B=3,C=4 follow A
  - Duration=5+max(3,4)=9 days
  - B finishes day8, C day9
  - Project waits for C
  - Paths B5+C6 and D8+E3
  - Shared zero-duration start/end
  - Both 11 days: two critical paths

Slack assumptions and PERT

- Slack requires network context
  - ES/EF alone insufficient
  - Successor relationships required

- Project completion constraint required
- C3→D6 parallel A4→B5
- Both complete paths finish day9
- Then C float=0
- Four unrelated rows don't establish network

• CPM versus PERT

- CPM: point durations
- PERT: uncertain durations
- a optimistic, m most-likely, b pessimistic
- TE=(a+4m+b)/6
- Variance=((b-a)/6)^2
- a=3,m=6,b=15 days
- TE=42/6=7 days
- Variance=4 days²
- Weighted estimate, not simple average
- Approximation, not guaranteed completion

ROI, risks and resource optimization

• ROI and risk management

- Common-date discounted benefits/costs assumed
- ROI=(benefits-costs)/costs*100%
- Birr120000 benefits, Birr100000 costs
- ROI=20%, not 120%
- Project risks: resources/schedule
- Product risks: quality
- Business risks: organization
- Likelihood/impact analysis
- Plan responses and monitor

• Risk responses and resource optimization

- Avoidance: remove risk through plan change
- Mitigation: reduce likelihood/impact
- Transfer: contractual consequence assignment
- Retention/acceptance: keep risk

- Passive acceptance: possibly no action
- Leveling: dates adjusted for resource availability
- May alter project finish
- Smoothing: float within schedule constraints
- Neither creates extra staff

Teams, organizations and process maturity

- Organization, teams and development stages
  - Functional organization: discipline specialists
  - Project organization: project grouping
  - Specialization: staffing/knowledge continuity
  - Possible handoff barriers
  - Democratic team: shared decisions
  - Chief-programmer team: centralized technical coordination
  - Different classification dimensions
  - Tuckman: forming→storming→norming→performing→adjourning
  - Orientation → conflict → norms
  - Effective work → closure
- Historical Software CMM maturity
  - 1 Initial, 2 Repeatable
  - 3 Defined, 4 Managed
  - 5 Optimizing
  - Level2: basic project discipline
  - Requirements management and project planning
  - Quality assurance and configuration management
  - Historical labels ≠ modern CMMI terminology
  - Process capability focus
  - No defect-free-product guarantee

Configuration, change and version control | Exam focus

- Configuration management and controlled items
  - Items, versions/baselines
  - Change procedures, status, audits

- Source, requirements, manuals
- Test plans, environments, build configurations
- Ordinary team browser: usually tool
- Controlled test-browser version: possible environment CI

• Change workflow and distributed Git
- Request → impact assessment → decision/prioritization
- Implementation → validation
- Configuration management: consistent affected versions
- Each clone: local history
- Independent branches then merging
- Conflicts/approvals remain possible
- Plan test configuration management early
- Apply throughout testing
- No single exclusive phase

Quality, verification and validation | Exam focus

• Assurance, control, verification and validation
- QA: processes/standards established and checked
- QC: work-product evaluation
- SQA plan: standards/responsibilities/evaluations
- Reviews, audits, assurance records
- Conformance: follow design/specification
- Poor specification can still be followed
- Verification: correct specified-obligation implementation
- Validation: intended stakeholder needs met
- Both: reviews, analysis, execution
- Human-only/computer-only distinction false

• Defect, error, failure and evidence
- Fault/defect: underlying problem
- Error: possibly incorrect internal state
- Failure: externally observed service departure
- Reliability: product quality
- Low cost/short development: project outcomes

- Tests reveal defects, increase confidence
- No general all-input correctness guarantee

Testing levels and acceptance | Exam focus

- Unit, integration and system testing
  - Unit: function/class/small component isolation
  - Mocks/stubs replace dependencies
  - Driver invokes unit
  - Stub replaces called dependency
  - Integration: combined-part interfaces/interactions
  - System: complete integrated system
  - Functional/nonfunctional requirements
  - Representative environment
- Acceptance, alpha, beta and techniques
  - Acceptance: customer/user decision support
  - Agreed needs/criteria
  - Alpha: controlled developer setting, selected users
  - Beta: external users, own environments
  - No universally longest level
  - Unit: level; white-box: technique
  - Black-box unit cases possible

Test types, regression and exploration

- Test types and workload context
  - Functional: SRS/acceptance behavior
  - Performance: speed/throughput/resources under workload
  - Load: expected demand
  - Stress: beyond normal limits
  - Usability/security/compatibility/reliability: respective qualities
  - Normal conditions alone: insufficient classification
  - Functional versus performance still unresolved
- Techniques, regression and exploration
  - Black-box: specified behavior

- White-box: internal structure
- Grey-box: partial internal knowledge
- Regression: unintended change effects
- Confirmation: specific defect repair
- Exploratory: learning/design/execution combined
- Experience-based testing
- Manual/automated: execution support
- No compulsory black-/white-box classification

Equivalence, boundaries and decision tables

• Equivalence partitions and boundaries
  - Expected-similar behavior groups
  - Sample valid and invalid groups
  - Allowed age18..60
  - Partitions: below18,18..60,above60
  - Edges and neighbors
  - 17,18,19,59,60,61
  - Exact boundary set: stated coverage scheme
• Decision tables and state-transition cases
  - Conditions/combinations → actions
  - validCard, enoughFunds
  - TT,TF,FT,FF
  - Dispense only TT
  - Proper rejection otherwise
  - Cover feasible rules
  - Impossible combinations needn't be blindly covered
  - Specification-based techniques
  - Already-closed incident: reject dispatch

Statement, decision and branch coverage

• Structural coverage and limitations
  - Statement coverage=distinct exercised statements/total executable statements
  - Declared test scope determines denominator

- Unreachable statements can prevent 100%
- Document justified exclusions/adjusted scope
- Decision coverage: decision outcomes
- Branch coverage: control-flow edges
- Unconditional edges under stated convention
- Nested decisions may need >2 tests
- Exercised structure ≠ behavioral proof

• Nested-decision coverage example

- Outer true: inner true and false
- Outer false: inner true and false
- Four tests required
- (w,l,h)=(3,2,4),(3,2,1)
- Also (2,3,4),(2,3,1)
- One run can't cover both outer branches
- All paths may be infeasible
- Loops may create infinite paths
- State chosen coverage target

Test process, plans and defect reports

• Test process and monitoring

- Plan objectives/scope/resources/schedule/risks
- Entry/exit criteria
- Analyze basis: testable features, defects
- Design cases and expected results
- Implement data/scripts/environment
- Execute, compare, report defects
- Evaluate completion
- Analysis: what/testability
- Design: how
- Monitoring/control throughout

• Test plans and defect reports

- Plan: intended work
- Incident/defect reports: normally execution outputs

- Report build/environment
- Reproduction steps
- Expected versus actual behavior
- Evidence and severity
- Anyone finding fault should record
- Testers often coordinate, no exclusive permission
- Configuration control: reproducible tested version

Maintenance and software evolution

- Maintenance categories and cost
  - Corrective: fix faults
  - Adaptive: environment changes
  - Perfective: useful capability/performance improvement
  - Preventive: reduce future risk
  - Fragile-code restructuring example
  - Motive determines category
  - Refactoring: preventive or perfective support
  - Years of legacy-code understanding
  - Regression effort
  - Maintenance may dominate lifecycle cost
- Evolution workflow and restructuring
  - Request → impact analysis → release planning
  - Implementation/testing → release
  - Request absent: impact analysis earliest listed
  - Major restructuring: reengineering
  - Architecture changed: architectural transformation more specific
  - Markets/technology/stakeholders/funding change scope/priorities

Numerical and option defects

- Scheduling question defects
  - Earliest finish: 9 days
  - Correct answer absent
  - Slack: latest time − earliest time

- Zero slack requires supplied completion-path assumption
- Equal critical paths: 11 days
- Both corresponding options true
- Network rules before supplied key

• Multiple-answer and invalid-choice defects
- Architectural structures and views
- Evaluate each statement independently
- Reusable requirements, tests, and architecture
- Requirements/tests/architecture: reusable product-line assets
- Correct options or multiple-answer instruction needed
- Defective options don't justify false rules

Context-dependent claims and model limitations

• Missing context and classification limits
- No always-most-time-consuming test level
- Team inexperience doesn't determine process
- Waterfall doesn't make later changes impossible
- Requirements include required/prohibited behavior
- Interface functions versus usability qualities
- Exact interface obligation needed
- Functional/nonfunctional classification depends on obligation

• Model and process qualifications
- Sequence/state both dynamic behavior
- Composition: diamond placement plus ownership
- Safety-critical government Agile unsuitability claim
- Assumes lightweight process without required assurance
- Not universally true
- Guide conventions: scenario hints
- Stronger engineering rules explicit

Design trade-offs and economical improvement

• Goal-based design trade-offs
- Fine-grained self-contained components

- Producer/consumer separation
- Limited shared mutable data
- Often improve maintainability
- Interfaces may add latency
- Isolation may complicate transactions
- Global state: local simplicity, wider coupling
- Measurable quality scenarios
- Component counts alone insufficient

• Pareto and economical improvement

- 80/20: prioritization heuristic
- Few causes may dominate observed impact
- No exact 80%-defects/20%-effort guarantee
- Refactoring: behavior preserved, structure improved
- Optimize against measured needs
- Record rationale
- Update traceability/regression evidence after change