CS Exit Ethiopia

Complete Study Notes

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 6

How to use these study notes

START01 A practical study cycle

Start with a small set of questions before reading.

Mark the exact thing that stopped you: an unknown term, a rule, a calculation, a trace or a choice between alternatives.

Use the question lookup to open the relevant section.

Read one section, close it, and explain the rule in your own words.

Solve the question again and say why each other choice fails.

Then change one assumption or input and solve the variation.

This checks understanding beyond recognition.

A useful 60-minute session: 10 minutes of cold questions, 20 minutes of focused reading, 20 minutes of closed-book solving and 10 minutes updating an error log.

Adjust the timing to your needs.

Revisit mistakes after a gap, for example the next day, several days later and the following week.

These intervals are a practical starting plan, not a fixed learning law.

Record the section ID, your mistaken rule, the corrected rule and one boundary case.

Keep the log short enough to review.

Mix related courses during revision: programming with data structures; databases with security; networking with administration.

Explain how the same word changes meaning across contexts.

START02 What to emphasize

Emphasize definitions, key differences, algorithm costs, SDLC models, databases, networking and security.

Use short oral questions to check understanding and correct common misconceptions.

Combine those habits with the tracing, calculation, design and diagnosis demanded by the Ministry learning outcomes.

Allocate revision by both exam share and your error log.

A course with fewer items can still deserve more time if your foundations are weak.

The blueprint is a planning baseline, not a prediction of exact questions.

Definitions: distinguish nearby concepts, not just their names.

Calculations: write units, formulas, assumptions and intermediate steps.

Code and SQL: check validity before tracing the text; state any changes before analyzing a corrected version.

Algorithms and protocols: know the input, decision rule, state change, termination and failure cases.

Design choices: state the criterion.

A choice can change when cost, reliability, consistency or latency matters more.

Course group	Items in the 100-item blueprint	Revision implication
Web Programming	9	Reserve extra practice for HTML, CSS, JavaScript, PHP, client/server behavior and safe data handling.
Data Structures and Algorithms	7	Trace operations and compare their assumptions and costs.
Each of the other fourteen courses	6 each	Study all sixteen; do not abandon a whole course based on a short note or an alleged repeated-question list.

START03 How to handle exam traps

Underline NOT, FALSE, BEST and EXCEPT before comparing choices.

Check whether the stem supplies enough facts to select one answer.

A long scenario is not automatically an analysis question, and a familiar answer letter is not evidence.

When code is invalid, a diagram is absent, or two choices express the same rule, identify the clarification needed before answering.

Learn the correct rule and the smallest clarification needed.

Do not memorize an arbitrary key as a universal fact.

Distinguish worst case from average case, and an upper bound from a tight bound.

Check language version, SQL dialect, addressing assumptions, graph weights and tie-breaking rules.

For subnetting and scheduling, calculate rather than guess from a familiar number.

Use timed mixed papers only after you can explain your errors.

Revisit the reading sections behind recurring mistakes.

Software Engineering

Complete Study Notes

SE01 | Software engineering, processes and work products

Software engineering applies systematic methods to development, operation and maintenance.

A software product includes executable code and relevant requirements, designs, test assets, configuration and user documentation.

A process defines activities, ordering, roles, products and entry/exit conditions.

Its four broad activities are specification, design/implementation, validation and evolution.

Procedural decomposition organizes functions; object oriented decomposition organizes objects with state and behavior.

Either can use iterative or sequential development.

OOP does not automatically shorten every project.

Reuse, encapsulation and consistent models can reduce duplication and ease change; poor object design can still be complex.

Functional programming emphasizes function evaluation and controlled side effects; event-driven programs react to events.

Software Engineering module

SE02 | Lifecycle phases and model selection

A common lifecycle names requirements, design, implementation, testing, deployment and maintenance.

Analysis clarifies the problem; system design chooses architecture; object/detail design specifies interfaces and algorithms; implementation writes code; testing evaluates behavior; deployment puts it into service.

These activities may overlap and repeat.

Select a process using requirement stability, technical risk, delivery urgency, stakeholder availability, team skills and assurance needs. “Team has little experience” alone does not uniquely identify a process.

A risk-driven cycle, prototype or carefully planned staged process may each help, depending on the actual uncertainty.

No process eliminates the need for testing or documentation.

Software Engineering module

SE03 | Waterfall and the V-model

Classical waterfall separates planned phases with reviewed deliverables.

It fits well-understood, stable requirements and familiar technology; late feedback makes changes costly.

It does not make change physically impossible: real projects revisit work under control, and modified waterfall explicitly includes feedback.

Rigidity offers milestone visibility but can freeze incorrect assumptions.

The V-model pairs specification/design stages with later test levels.

Requirements inform acceptance tests; system design informs system tests; component/detail design informs integration/unit tests.

Coding lies at the bottom.

Test planning starts early.

Stable requirements and strong assurance may favor this organization, but neither a V shape nor waterfall guarantees safety or correctness.

Software Engineering module

SE04 | Iteration, increments and early delivery

Iteration revisits and improves a solution; an increment adds usable functionality.

A process may do both.

Delivering registration first, then payments, then reporting is incremental; revising a registration prototype after feedback is iterative.

Each increment still needs design, implementation and testing.

Choose incremental delivery when prioritized partial capability must reach users early.

Smaller changes can be easier to debug and permit feedback, though architecture and integration still require planning.

Iteration need not rebuild the whole system every time, and increments need not be perfectly independent modules.

Broad descriptions of early working software can fit prototypes, iterative or Agile processes; identify the purpose of that early version.

Software Engineering module

SE05 | Agile values, stories and Scrum

Agile favors people collaborating, useful working results, customer participation and adaptation over rigid adherence to procedures, exhaustive documents, contracts or fixed plans.

The latter still have value.

Agile does not mean no planning or no documentation.

A user story states a small user need and value, supported by acceptance criteria; an epic groups broader work.

Scrum uses timeboxed Sprints of at most one month, with a Sprint Goal, selected work and a usable Increment meeting a Definition of Done.

Scope is discussed before and during planning; it is not forbidden to agree scope.

Timeboxes tend to stay fixed while content is adapted without endangering the goal.

Agile can be combined with required assurance/documentation; safety-critical context alone cannot prove it is unsuitable.

Software Engineering module

Agile Manifesto

Scrum Guide

SE06 | Spiral development and risk

Spiral development is risk driven.

Each cycle identifies objectives and alternatives, assesses and reduces important risks, develops/verifies an appropriate next result, and plans further commitment.

A prototype may test a risky technology before a large investment.

The amount and type of work depend on the risks.

Use risk emphasis to distinguish spiral from a generic changing-requirements scenario.

Spiral can incorporate incremental, prototyping or sequential activities, but calling it the uniquely “most general” process is a textbook convention rather than a universal theorem.

Risk analysis requires skill and may impose excessive overhead on a small low-risk project.

Software Engineering module

SE07 | Prototypes, RAD and reuse-oriented development

A prototype explores requirements, usability or technical feasibility.

A throwaway prototype is discarded after learning; an evolutionary prototype is deliberately refined toward production.

A quick prototype may omit robustness, security or architecture, so do not deploy it unchanged without evaluation.

Coding may occur during prototyping even if a diagram labels that activity construction rather than coding.

RAD uses rapid construction, timeboxing, user participation and often reusable components; it requires available users and skilled designers/developers.

Reuse-oriented development searches components, evaluates fit, may adjust requirements, designs around interfaces and builds missing parts.

Product-line assets may include requirements, architecture, components and tests; all are potentially reusable.

Software Engineering module

SE08 | Feasibility and levels of requirements

Feasibility asks whether a proposal is worth doing and can be delivered within technical, economic, operational and schedule constraints.

Examine existing limitations, organizational impacts, benefits, cost, available skills and integration risk.

System analysis studies the problem and needed behavior in more detail; the activities inform each other.

Business requirements explain why the project exists and its goals.

User requirements describe stakeholder needs at a useful high level; system requirements specify detailed services and constraints.

The Software Requirements Specification (SRS) organizes the agreed requirements.

A natural-language specification communicates with clients while analysis models formalize it for technical discussion; real specifications can also contain diagrams.

Software Engineering module

SE09 | Functional requirements include prohibited behavior

A functional requirement states an observable service or behavior, including reactions to inputs, errors and forbidden actions. “When the sensor detects intrusion, send an alert” is functional. “Only a manager may approve refunds” also specifies permitted/prohibited behavior.

A negative statement is not automatically nonfunctional.

Write one clear obligation with an actor/trigger, action and observable result. “The system shall generate a PDF report” is functional. “The system shall not reveal another customer’s balance” can be a security-related functional rule.

Security as a broad quality goal and the functions implementing it must be distinguished.

Words such as must, should and not work do not by themselves determine a requirement’s classification.

Software Engineering module

SE10 | Nonfunctional qualities and measurable constraints

Nonfunctional requirements include quality attributes and constraints.

Performance concerns response time, throughput and resource use; reliability concerns failure-free service; availability concerns being ready for use; usability concerns task success and ease of learning.

Security concerns protection; compatibility concerns cooperating environments; portability concerns moving environments; maintainability/supportability concern change, diagnosis and evolution.

“Search responds within two seconds under 100 concurrent users” is measurable performance. “New users finish checkout within one minute without help” is usability. “Use Java” is a technology/pseudo-requirement, often separately classified as a system constraint.

Dependability includes reliability, safety and robustness; portability is not its usual central category.

Performance degradation at peak load primarily concerns performance, not automatically reliability.

Software Engineering module

SE11 | Validate requirements and maintain traceability

Correct requirements reflect actual stakeholder needs.

Complete requirements address the agreed scope, normal and exceptional behavior.

Consistency excludes contradictions.

Clarity/unambiguity gives one intended interpretation; feasibility/realism means achievable within constraints.

Verifiability means objective evidence can check fulfillment.

Replace “fast for most users” with a defined metric, workload and threshold.

Trace each requirement backward to its source/rationale and forward to design, code and tests.

Record dependencies among requirements.

A traceability matrix links these artifacts and helps impact analysis; it is not just a cost/schedule ledger.

Stakeholders must negotiate conflicting requirements rather than independently approving incompatible demands.

Scope, understanding and volatility are distinct elicitation difficulties.

Software Engineering module

SE12 | Elicitation techniques, FAST and QFD

Interviews ask stakeholders about needs; document analysis examines manuals, contracts and current records; observation/ethnography studies actual daily work.

Prototypes reveal tacit expectations.

Workshops/JAD bring users, managers and developers together with a facilitator and recorder to build consensus.

Use several sources and check assumptions against observed work.

In the taught FAST technique, participants list environment elements, outputs and inputs, consolidate lists and develop smaller specifications.

QFD prioritizes customer value: normal needs are explicitly stated, expected needs are assumed, and exciting needs provide unanticipated benefit.

Brainstorm first to generate ideas, then evaluate and prioritize; do not confuse that with letting every idea become a committed requirement.

Software Engineering module

SE13 | From scenarios to analysis objects

A scenario is one concrete execution; a use case abstracts the related successful and alternative scenarios.

Analysis integrates functional (use cases/scenarios), structural/object (classes) and dynamic (states/interactions) models.

Entity objects represent domain information such as Incident; boundary objects mediate interaction, such as ReportForm; control objects coordinate a use case.

A UI form is normally boundary, not entity.

Use nouns as candidate objects, possessive descriptions as candidate attributes, and verbs such as manages/reports-to as candidate associations.

These are heuristics, not automatic parsing rules.

CRC means Class, Responsibilities, Collaborators.

A control per use case or participating actor can be a useful starting point, not a mandatory UML rule or fixed object lifetime.

Software Engineering module

SE14 | Actors, use cases and scope

An actor is an external role interacting with the system; it can be a person, device or another system.

A person may play several roles, and several people may play one role.

A primary actor seeks a goal; a supporting actor provides a service.

Ask which user groups perform main functions, administer or maintain the system, and which external systems exchange information.

Draw actors outside the system boundary rectangle and use-case ovals inside.

Name a goal with a verb and noun.

The diagram shows roles, goals and relationships, not the full detailed algorithm or UI layout.

Being an internal component is different from being an external actor; the chosen boundary determines which role applies.

Software Engineering module

SE15 | Write a complete use case with alternatives

Record name/goal, actors, trigger, preconditions, main success flow, alternatives/exceptions and postconditions.

Use active subject-verb steps at one abstraction level; separate errors from the normal flow.

Example: Submit Incident.

Precondition: officer authenticated. 1 Officer supplies location and description. 2 System validates details. 3 System records the incident. 4 System returns its ID.

Alternative 2a: location missing → report the problem and request correction; save nothing.

Success postcondition: incident and ID persist; failure guarantee: no partial record.

An alternate flow is a conditional variation; an exception flow handles a failure.

Preconditions hold before entry, while a trigger starts the use case.

Software Engineering module

SE16 | Include, extend, generalization and abstract use cases

Include factors required reused behavior: Submit Order includes Validate Order, with the dashed arrow from the including/base use case to the included one.

Extend adds conditional behavior at extension points: Apply Discount extends Checkout, with the arrow from the extension to the base.

The base is meaningful independently.

For an extension, draw the arrow from the extending use case to the base use case.

Generalization relates a specialized actor/use case to a more general one, with a hollow triangle toward the general element.

An abstract use case specifies behavior for specialization and is not directly instantiated as a complete use case.

It is not defined merely by reuse or lack of a direct actor line.

Included behavior is incorporated where the inclusion is reached; do not infer every conditional branch in the base always executes.

Software Engineering module

SE17 | Class diagrams, visibility and multiplicity

A class box lists name, attributes and operations; + public, - private, # protected and ~ package indicate visibility.

A class diagram shows static types and relationships, while an object diagram shows particular instances/links.

UML is maintained and standardized by OMG; it is a modeling language, not a lifecycle process.

Structural things describe conceptual or physical structural elements.

Multiplicity at one end counts how many objects there may correspond to one object at the opposite end: 1 exactly one, 0..1 optional, 0..* any nonnegative number, 1..* at least one.

A one-to-many association does not automatically mean composition.

Read each end separately before choosing cardinality.

Software Engineering module

SE18 | Association, aggregation, composition and generalization

Association connects related instances.

A dashed dependency arrow means a client uses a supplier.

Generalization uses a solid line and hollow triangle pointing to the superclass/general element.

Shared aggregation uses a hollow diamond at the whole end and weak whole-part meaning; it does not by itself define a detailed lifecycle policy.

Composition uses a filled diamond at the whole end and strong ownership: a part is in at most one composite at a time, and parts still owned are destroyed with that composite.

Parts may be detached/transferred under a defined model.

Thus “cannot ever exist independently” is an oversimplification.

For Car composing Engine, the diamond belongs beside Car; saying it points to Engine is imprecise.

The Car-Engine example describes composition.

Software Engineering module

OMG UML2.5.1

SE19 | Sequence diagrams and the emergency-report example

Sequence diagrams show interactions across lifelines, with order/time generally downward.

Messages cross between participants; activation rectangles show execution.

A creation message starts a lifeline; an X marks modeled destruction, not a guarantee about when a garbage collector runs.

Synchronous calls wait for completion; asynchronous messages permit the sender to proceed.

One interaction-diagram layout heuristic puts the initiating actor first, boundary second, control third, then entities.

This is not a required UML column order.

In ReportEmergency, FieldOfficer activates ReportEmergencyButton, enters details through EmergencyReportForm, and later sees AcknowledgmentNotice; DispatcherStation belongs to the dispatcher side.

Choose the button for initiating this particular scenario.

Software Engineering module

SE20 | Activity, state and data-flow models

An activity diagram models workflow: filled circle start, rounded actions, guarded decision diamond, thick fork/join bars and partitions/swimlanes for responsibility.

A diamond can also merge alternative flows; a join synchronizes concurrent flows.

A state machine models states and transitions triggered by events, often with guards/actions, for one object's lifecycle.

Choose a sequence diagram for message order between participants, a state machine for event-dependent state changes, and activity for workflow.

Both sequence and state diagrams describe behavior over time.

State machines are useful for objects with significant states, whether entity or control.

A DFD shows data stores/flows and transformations from inputs to outputs; it is not a UML diagram.

Software Engineering module

SE21 | Design and subsystem decomposition

System design turns the analysis model into architecture: design goals, subsystem responsibilities, dependencies, hardware mapping, data management, access control, control flow and startup/shutdown/error policies.

Detail/object design refines APIs, parameter/return types, exceptions, component choices and algorithms.

A subsystem is a manageable logical collection with services and an interface; it need not be a physical replaceable artifact.

Group closely related objects and functionality, minimize associations crossing boundaries, and isolate communication infrastructure where useful.

User workflows inform decomposition but do not alone dictate the internal structure.

Analysis describes domain needs; subsystem decomposition is principally a design result.

Architecture documentation supports communication and records major decisions, not every source-level execution step.

Software Engineering module

SE22 | Cohesion, coupling, modularity and refactoring

Cohesion measures how well elements inside a module serve a related purpose; coupling measures dependency between modules.

Seek high cohesion and low coupling.

Shared mutable data and knowledge of another module's internal representation increase coupling.

Information hiding exposes stable operations while concealing changeable decisions.

Logical cohesion groups similar-category operations, often selected by a flag; temporal cohesion groups operations done at the same time; functional cohesion focuses on one task.

Splitting an unrelated three-purpose component improves modularity.

If behavior is preserved, the restructuring is refactoring; the words are not mutually exclusive answers.

Smaller modules can aid understanding, independent work, reuse and maintenance, but too many interfaces also create complexity.

Software Engineering module

SE23 | Layers, tiers and architecture views

A layer is a logical abstraction level; a tier is a deployment separation.

A closed layered architecture permits calls only to the immediately lower layer; an open one permits approved skipping.

For top-to-bottom A,B,C,D, A→B obeys closed layering, A→C skips, and lower-to-upper calls violate the simple dependency rule unless explicitly mediated by a defined mechanism.

A structure is the elements/relations themselves; a view represents selected aspects.

Module structures show code organization, component-and-connector structures runtime interaction, and allocation structures map software to hardware, teams or environment.

Processes/threads belong in concurrency views.

Evaluate architectures against quality attributes and scenarios; processor allocation, decomposition and style selection can all be architectural concerns.

Software Engineering module

Rose-Hulman: What is software architecture

SE24 | Architecture styles and integration boundaries

Repository style shares central data, simplifying common access but creating bottleneck/schema dependencies.

MVC separates domain model, display view and interaction controller.

Client-server separates requesters from service providers; peer-to-peer permits both roles.

Pipe-and-filter applies successive transformations to data streams.

Three-tier separates presentation, application logic and storage; some four-tier variants split presentation client/server.

An API gateway presents one entry point and routes/combines calls to services, hiding service boundaries from clients.

It can centralize policies but also become a bottleneck.

A component diagram shows components/interfaces/dependencies; a deployment diagram places software artifacts on nodes.

Choose the style for the required interactions and qualities rather than selecting a universally best architecture.

Software Engineering module

SE25 | Design principles and operation contracts

Use abstraction and decomposition to manage complexity; preserve consistency, traceability and clear responsibilities.

Top-down design refines high-level structure; bottom-up design assembles existing components.

A hybrid combines both.

SOLID includes Single Responsibility, Open-Closed, Liskov substitution, Interface Segregation and Dependency Inversion.

Depend on abstractions; add compatible extensions; avoid forcing unused operations.

DRY limits duplicated knowledge, KISS avoids unnecessary complexity, and YAGNI avoids speculative work.

A contract states preconditions, postconditions and invariants.

For withdraw(amount): require amount>0 and amount<=balance; ensure balance=oldBalance-amount; preserve balance>=0.

Validation checks inputs; implementation enforces the transition.

A complete feature needs both UI/API and persisted behavior, not only a diagram.

Software Engineering module

University of Washington: Software Design and Style

SE26 | A small analysis-to-implementation example

For Submit Incident, let a boundary read input, a service validate and coordinate, and a repository save an Incident entity.

Pseudocode: submit(input){require authenticated; if missing(location) return validationError; incident=new Incident(input); repository.save(incident); return incident.id;}.

Save and ID assignment must succeed together, or roll back.

Derive tests from the model: missing location saves nothing; valid input returns a saved ID; a storage failure yields an error without a partial record; unauthorized input is rejected.

Trace requirement→use-case step→service operation→repository effect→test.

Use the implementation exercise to connect analysis and design tools to working code.

Software Engineering module

SE27 | Patterns, variability and testability

Factory Method lets subclasses choose a concrete product through a creation operation, reducing coupling to concrete constructors.

A simple factory centralizes creation but is not identical to Factory Method.

Singleton controls access to one instance; Observer notifies subscribers; Memento stores/restores state while hiding representation; Iterator traverses; Decorator adds behavior through wrapping. The pattern name is Memento, not Momento.

Variability can use selectable features, omission/inclusion and extension points; overloading supplies alternate signatures in code but is not every variability mechanism.

Testability improves with controllable inputs, observable outputs, replaceable dependencies and limited structural complexity.

A sandbox can isolate execution.

Assess each design claim independently before evaluating a combined statement.

Software Engineering module

Design Patterns introduction, university-hosted author text

SE28 | Planning, WBS, estimation and tools

Planning is continuous: identify scope/deliverables, decompose work, estimate effort/duration/cost, assign resources, sequence dependencies and revise as evidence changes.

A WBS is hierarchical and covers project scope without adding unrelated work.

Tools help coordinate ownership, dependencies and progress; they do not replace judgment or automatically fix bugs.

Function points estimate functional size using inputs, outputs, inquiries, internal logical files and external interface files, weighted under the chosen method.

They are not just lines of code and are not a direct exact measure of complexity.

Git tracks versions; issue trackers such as Jira track work/defects; CI tools such as Jenkins automate checks; Docker packages application/runtime dependencies into containers.

Software Engineering module

SE29 | Critical path: forward and backward passes

In a dependency network, ES=max predecessor EF and EF=ES+duration; start tasks have ES=0 unless another constraint applies.

Project earliest finish is the maximum terminal EF.

Backward, LF=min successor LS and LS=LF-duration, starting from the required finish.

Total float/slack=LS-ES=LF-EF.

A critical path has the longest total duration and normally zero float in an unconstrained schedule.

If A takes 5 days and B=3,C=4 both follow A, project duration is 5+max(3,4)=9 days.

B finishes at 8, but the project waits for C at 9.

If paths B5+C6 and D8+E3 share a zero-duration start/end, both total 11; there are two critical paths, not one.

Software Engineering module

SE30 | Slack assumptions and PERT calculations

Early start/finish values alone do not determine slack without successor relationships and the project completion constraint.

If C3→D6 is a complete path parallel to A4→B5 and both finish at day9, C has zero float.

That network cannot be silently inferred from four unrelated ES/EF rows.

Traditional CPM uses point durations; PERT models uncertainty with optimistic a, most-likely m and pessimistic b. The classic estimate is TE=(a+4m+b)/6 and variance=((b-a)/6)^2.

For 3,6,15 days, TE=42/6=7 days and variance=4 days².

Do not replace the weighted estimate by a simple average.

These formulas are approximations, not guaranteed actual completion times.

Software Engineering module

PMI: PERT estimation

SE31 | Business value, risks and resource optimization

With benefits and costs already discounted to a common date, ROI=(benefits-costs)/costs*100%.

Birr120000 benefits and Birr100000 costs give 20%, not 120%.

Identify project risks to resources/schedule, product risks to quality, and business risks to the organization.

Analyze likelihood and impact, plan responses and monitor.

Avoidance changes the plan to remove a risk, mitigation reduces likelihood/impact, transfer assigns consequences contractually, and retention/acceptance keeps the risk; passive acceptance may take no action.

Resource leveling adjusts dates to respect resource availability and can change project finish; smoothing uses available float within schedule constraints.

Neither magically creates extra staff.

Software Engineering module

PMI Lexicon of Project Management Terms

SE32 | Team organization, development and maturity

Functional organizations group specialists by discipline; project organizations group people around projects.

Functional specialization helps staffing and knowledge continuity but may create handoff barriers.

Democratic teams share decisions; chief-programmer teams centralize technical coordination.

These are different classification dimensions.

Tuckman’s sequence is forming→storming→norming→performing→adjourning: orientation, conflict, norms, effective work and closure.

Historical Software CMM levels are 1 Initial, 2 Repeatable, 3 Defined, 4 Managed, 5 Optimizing.

Level2 establishes basic project discipline including requirements management, project planning, quality assurance and configuration management.

These historical labels must not be silently replaced by modern CMMI terminology.

A maturity model concerns process capability, not a guarantee that every product is defect free.

Software Engineering module

SEI: Software CMM v1.1

UC Berkeley Team Building Toolkit

SE33 | Configuration, change control and version control

Configuration management identifies controlled items, versions/baselines, change procedures, status and audits.

Items may include source code, requirements, manuals, test plans, environments and build configurations.

A team's ordinary browser is usually a tool rather than product CI, but a controlled test-browser version may be an environment CI.

Change management handles request, impact assessment, decision/prioritization, implementation and validation; configuration management keeps all affected versions consistent.

Distributed Git gives each clone local history and supports independent branches followed by merging; it does not eliminate merge conflicts or approvals.

Plan test configuration management early and apply it throughout testing, not at just one exclusive phase.

Software Engineering module

SE34 | Quality assurance, verification and validation

Quality assurance establishes and checks processes/standards; quality control evaluates work products.

An SQA plan identifies standards, responsibilities, evaluations, reviews, audits and assurance records.

Quality of conformance means following the design/specification; a product can conform to a poor specification.

Verification asks whether specified obligations are implemented correctly; validation asks whether the product meets intended stakeholder needs.

Both can use reviews, analysis or execution, so "human-only versus computer-only" is false.

A fault/defect is an underlying problem; an error may be an incorrect internal state; a failure is an externally observed departure from required service.

Reliability is a product quality attribute; low cost or short development time are project outcomes.

Tests increase confidence and reveal defects but cannot generally guarantee correctness over every possible input.

Software Engineering module

SE35 | Testing levels and acceptance

Unit testing isolates a function, class or small component, often replacing dependencies with stubs/mocks; a driver invokes the unit and a stub substitutes a called dependency.

Integration testing exercises interfaces and interactions among combined parts.

System testing checks the complete integrated system against functional and nonfunctional requirements in a representative environment.

Acceptance testing supports the customer/user's acceptance decision against agreed needs and criteria.

Alpha typically happens in a controlled developer setting with selected users; beta uses external users in their environments.

None of these levels is universally the longest.

Unit is a level, while white-box is a technique; a unit can be tested using black-box cases too.

Software Engineering module

SE36 | Test types, regression and exploratory work

Functional tests check required behavior against the SRS and acceptance criteria.

Performance tests assess speed, throughput and resources under a specified workload; load uses expected demands and stress exceeds normal limits.

Usability, security, compatibility and reliability tests target their respective qualities. "Normal operating conditions" alone does not distinguish functional from performance testing.

Black-box techniques derive cases from specified behavior; white-box techniques derive them from internal structure; grey-box uses partial internal knowledge.

Regression testing checks for unintended effects after change; confirmation testing checks the particular defect repair.

Exploratory testing combines learning, design and execution and is experience based.

Manual/automated describe execution support, not a compulsory black-/white-box classification.

Software Engineering module

SE37 | Equivalence, boundaries and decision tables

Equivalence partitioning groups inputs expected to behave alike, then samples valid and invalid groups.

For allowed age18..60, useful partitions are below18,18..60,above60.

Boundary tests target edges and neighbors, such as 17,18,19,59,60,61, with the exact set chosen under the stated coverage scheme.

A decision table lists condition combinations and resulting actions.

For validCard and enoughFunds, enumerate TT,TF,FT,FF; specify dispense only for TT and the proper rejection for others.

Cover feasible rules rather than blindly using every impossible combination.

These are specification-based techniques.

Test requirements can also define state-transition cases, such as rejecting dispatch from an already closed incident.

Software Engineering module

ISTQB CTFL4.0.1 syllabus

SE38 | Statement, decision and branch coverage

Statement coverage is the number of distinct executable statements exercised divided by the total executable statements in the declared test scope.

Unreachable statements can prevent100%; any justified exclusions must be documented with the adjusted scope.

Decision coverage counts decision outcomes; branch coverage counts control-flow edges, including unconditional edges under the stated convention.

Testing both true and false of each decision may need more than two overall test cases when branches nest.

Coverage is evidence of exercised structure, not proof of all behavior.

In the width/length/height program, the outer if has separate inner decisions in each branch.

Need outer true with inner true and false, and outer false with its inner true and false: four tests, for example (w,l,h)=(3,2,4),(3,2,1),(2,3,4),(2,3,1).

One run cannot exercise both outer branches.

All paths may be infeasible or infinite with loops; specify the chosen coverage target.

Software Engineering module

ISTQB CTFL syllabus v4.0.1, sections 4.3.1-4.3.2

SE39 | Test process, plans and defect reports

Plan objectives, scope, resources, schedule, risks and entry/exit criteria.

Analyze the test basis for testable features and defects; design cases/expected results; implement data, scripts and environment; execute and compare; report defects; evaluate completion.

Analysis asks what to test and assesses testability; design asks how.

Monitoring/control run across these activities.

A test plan describes intended work; incident/defect reports are normally execution outputs.

A defect report should identify build/environment, steps, expected versus actual behavior, evidence and severity.

Anyone finding a fault should record it; testers often coordinate reporting, but it is not their exclusive permission.

Configuration control makes the tested version reproducible.

Software Engineering module

ISTQB CTFL4.0.1 syllabus

SE40 | Maintenance and software evolution

Corrective maintenance fixes faults; adaptive maintenance responds to environment changes; perfective maintenance improves useful capability/performance; preventive maintenance reduces future risk, such as restructuring fragile code.

Motive determines category; refactoring can support preventive or perfective work.

Maintenance often lasts years and includes understanding legacy code and regression effort, so it may dominate lifecycle cost.

A typical evolution sequence is change request→impact analysis→release planning→implementation/testing→release.

Among a list lacking the request step, impact analysis may be the earliest listed technical activity.

Major restructuring is reengineering; if the architecture itself changes, architectural transformation is the more specific description.

Markets, technology, stakeholders and funding can all change scope or priorities.

Software Engineering module

SE41 | Apply network calculations and compare design claims

The earliest-finish example yields 9 days.

Zero slack follows only from an explicit completion-path assumption when latest times or the full network are unavailable.

In the path-duration example, two paths each total 11 days.

Use the network rules to justify the calculation before comparing it with an answer key.

Evaluate descriptions of architectural structures and views separately; both can be true.

Evaluate the individual claims before considering a combined option, especially if the option refers to itself.

Requirements, tests and architecture can all be reusable product-line assets.

When several choices satisfy the rules, identify the needed distinction rather than selecting a false rule.

Software Engineering module

SE42 | Context-dependent claims and model limitations

Context matters: no test level is always most time consuming; no process is determined by team inexperience alone; no waterfall rule makes every later change impossible.

Requirements may describe required and prohibited behavior, including interface functions as well as usability qualities.

Identify an interface requirement's exact obligation before classifying it as functional or nonfunctional.

Sequence/state diagrams both show dynamic behavior.

Composition needs the diamond's placement and ownership semantics. "Agile least suitable for safety-critical government work" assumes a particular lightweight process without required assurance; it is not universally true.

Treat the guide's conventions as scenario hints and keep the stronger engineering rules explicit.

Software Engineering module

SE43 | Design choices, evidence and economical improvement

Choose designs by their effect on stated goals.

Fine-grained self-contained components, separation of producers/consumers and limited shared mutable data often improve maintainability.

Trade-offs remain: interfaces can add latency, isolation can complicate transactions, and global state can simplify one operation while coupling many others.

Prefer measurable quality scenarios to counting components alone.

Pareto's 80/20 rule is a prioritization heuristic: a small subset of causes may account for much of the observed impact.

It does not guarantee 80% of defects can be fixed with exactly 20% effort.

Refactor to preserve behavior while improving structure; optimize only against measured needs; record rationale and update traceability and regression evidence after change.

Software Engineering module

SE-S01 | Scope, understanding and changing requirements

Elicitation difficulties have several distinct sources. Scope problems concern agreeing on system boundaries and the appropriate level of detail. Understanding problems arise when stakeholders and analysts have different domain knowledge, omit tacit needs or communicate conflicting interpretations. Volatility means that requirements change as circumstances and understanding develop.

These difficulties call for repeated communication, explicit boundaries, negotiated meanings and controlled revision. They can occur together; resolving one does not automatically remove the others.

CMU SEI - Issues in Requirements Elicitation

SE-S02 | Objects, shared models and specialization

An object-oriented approach can use a consistent vocabulary of objects, classes and relationships across analysis, design and implementation. Models are refined for different purposes rather than being identical artifacts. This continuity can reduce translation errors and effort; reusable classes can shorten suitable projects, but neither shorter duration nor good design is guaranteed.

Generalization places common concepts in a more general superclass and specialized concepts in subclasses. Available inherited attributes and operations are shared by subclasses, which may extend or override behavior. Defining common features once can reduce duplicated maintenance. The top of the hierarchy is more general, not more specialized.

Encapsulation groups state with operations and hides representation behind an interface. Polymorphism permits different object types to respond differently through a common operation or interface. Abstraction exposes the relevant contract while omitting unnecessary detail. Reusability means using suitable existing classes or components in new contexts rather than writing them again. These ideas support each other but name different properties.

Software Engineering module · page 31

Software Engineering module · page 32

Software Engineering module · page 38

Software Engineering module · page 104

Software Engineering module · page 141

SE-S03 | Modeling categories and data diagrams

In the course's UML building-block terminology, structural things describe static conceptual or physical elements. Behavioral things describe dynamic interactions and state behavior. Grouping things organize model elements into packages. Annotational things attach explanatory remarks or constraints, usually as notes.

An entity-relationship diagram models entity types, their attributes and the relationships between them, commonly for data design. Its focus differs from a use-case diagram of external

roles and goals, a class diagram of classes and their features, or a deployment diagram mapping artifacts to execution nodes.

Software Engineering module · page 45

Software Engineering module · page 46

IBM - Entity relationship diagrams

SE-S04 | Plan-driven and adaptive planning

In a plan-driven process, activities are planned in advance and progress is assessed against that plan. Agile planning is incremental and adapts to changing needs. This distinction concerns planning strategy; a process can combine advance planning with staged deliveries and feedback.

Software Engineering module · page 13

SE-S05 | Exploration artifacts and collaborative implementation

A storyboard sketches a sequence of moments in a proposed interaction, helping people explore how a concept would be used. A wireframe sketches interface structure and the arrangement of content or controls. Both can inform requirements, but neither automatically supplies a complete use-case description with actors, preconditions, flows and postconditions.

Pair programming is a way for two programmers to work together on the same implementation task, commonly sharing a workstation with complementary writing and reviewing roles. It is an implementation collaboration practice rather than a replacement for modeling a user goal.

IDEO Design Kit - Storyboard

IBM Enterprise Design Thinking Toolkit - Wireframes

Agile Alliance - Pair programming

Software Engineering module · page 70

SE-S06 | Distinguish a requirements specification from neighboring terms

A Software Requirements Specification records agreed software services and constraints. A Capability Development Document is a defense capability-requirements artifact describing needed capability characteristics and performance parameters; its specialized acquisition context differs from the ordinary software SRS. Domain-driven design is an approach that organizes software models around the business domain, not the name of an agreed requirements specification.

Software Engineering module · page 62

U.S. Army - Getting Requirements Right

Microsoft - Designing a DDD-oriented microservice

SE-S07 | Scalability and robustness are different qualities

Scalability is the ability to accommodate additional workload while maintaining the required service, often by increasing available resources. It concerns growth in demand. Compatibility concerns supported cooperating environments, such as browsers and devices; usability concerns successful and learnable user tasks.

Robustness concerns reliable operation across varied or difficult operating conditions and controlled failure outside the supported range. It is different from usability, which concerns people successfully learning and carrying out tasks. Improving an interface's task success does not by itself establish robustness under adverse conditions.

Microsoft - Scaling cloud-native applications

NIST CSRC Glossary - Robustness