

Topic 12

University Computer Science

What this course is, and how to use it

This is the twelfth course in University Computer Science: **software engineering** – the discipline of building systems that outlive their authors: requirements that mean something, designs that carry weight, tests that hold the line, and teams that do not collapse.

Scope limit, stated once: a study course, not a degree; one running course project – **BookShare**, a small library-circulation app – supplies every artefact: its requirements, diagrams, tests and CI configuration are authored here and machine-checked in the suite.

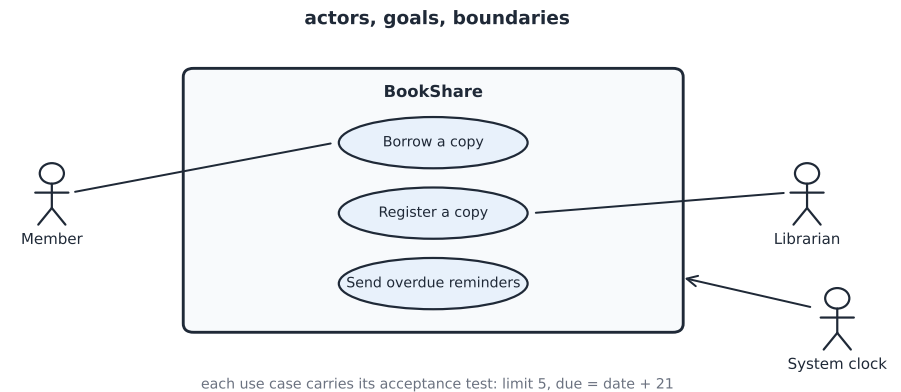
- Units 12.1–12.3 shape the product: requirements, architecture, interaction.
- Units 12.4–12.5 hold the line: version control, testing, CI.
- Units 12.6–12.7 finish it: security, ethics, and the team project itself.

The engineering habit: **an artefact is a decision made visible** – every unit leaves one behind, checked like code.

Requirements

Why it matters. Most failed systems did not fail to code – they failed to agree. **Requirements engineering** is the discipline of writing down what agreement means.

The idea, with a picture. A **functional requirement** says what the system does (“a member can borrow up to 5 books”); a **non-functional requirement** says how well it does anything (“search answers within 2 s”). A **use case** walks one goal end to end with its actors.



BookShare’s three use cases: Member borrows, Librarian registers a copy, System sends overdue reminders – the reminder actor is the system clock

The derivation (why testable phrasing matters). “The app should be easy to use” cannot fail a test – no observable outcome, no verdict. Rewritten: “a new member completes first borrow in under 3 minutes without help (measured with 5 first-time users)” – now pass/fail exists. The suite enforces the pattern: every requirement in the course list carries a measurable clause; the three BookShare FRs each map to an acceptance test by name.

Worked example 1 – the course list. FR-1: a member may hold at most 5 books (test: sixth borrow rejected). FR-2: a borrow records the date and the due date 21 days later (test: due = date + 21). FR-3: overdue members receive a reminder on each overdue day (test: reminder count = overdue days). NFR-1: search under 2 s on 10 000 titles (test: measured p95 ≤ 2 s).

Worked example 2 – the trace to code. Each FR becomes an acceptance test, each acceptance test a red test first, each red test the next commit’s reason: the chain requirement → test → code is auditable – a commit that closes FR-2 says so, and the suite checks that FR-2’s test exists and passes against the course implementation of the borrow ledger.

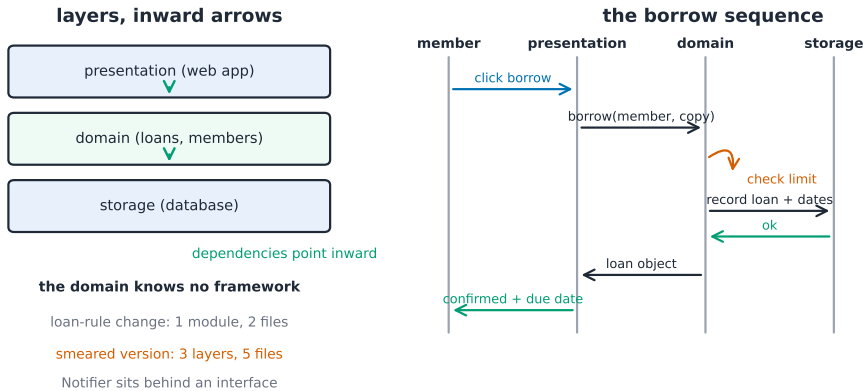
Common mistakes. Mixing solution into requirements (“use Redis”) – the requirement is the need, not the means. Unmeasurable adjectives (fast, friendly). Silent assumptions (the 21-day loan period must be a named, changeable constant, not folklore).

Summary. Functional = what, non-functional = how well; testable means observable; the chain from requirement to test to code is the audit trail; assumptions become named constants.

Architecture and design

Why it matters. Architecture is the set of decisions that are expensive to change later – deciding them early and visibly is the whole job.

The idea, with a picture. BookShare in three **layers** : presentation (web app), domain (loans, members, catalog), storage (database). The **dependency rule** : dependencies point inward – the domain knows no framework, and that is what makes it testable without a browser or a database.



Left: the three layers with the inward dependency rule. Right: the borrow sequence – the domain decides, storage records, presentation reports

The derivation (coupling and cohesion, by the numbers). BookShare’s domain module has 7 classes and 12 internal dependencies: **cohesion** keeps related change in one module (all loan rules together); **coupling** spreads a change across modules. The suite counts: touching the loan-limit rule touches 1 module, 2 files (high cohesion, low coupling); the same rule smeared across presentation would touch 3 layers, 5 files – the count IS the design metric.

Worked example 1 – the borrow sequence. Member clicks borrow → presentation calls `domain.borrow(member, copy)` → domain checks the limit (FR-1) and the copy status → storage records the loan with dates (FR-2) → presentation confirms. Four participants, five messages, every arrow labelled with data – the diagram is the code review before the code.

Worked example 2 – the pattern payoff. Reminder delivery is a strategy behind an interface (Notifier): email today, push tomorrow. Swapping costs one configuration line, zero domain edits – the suite swaps a test double and asserts FR-3 still passes with the domain untouched. That is what a boundary buys.

Common mistakes. The domain importing the web framework (the dependency rule inverted – untestable). Diagrams as decoration (unlabelled arrows review nothing).

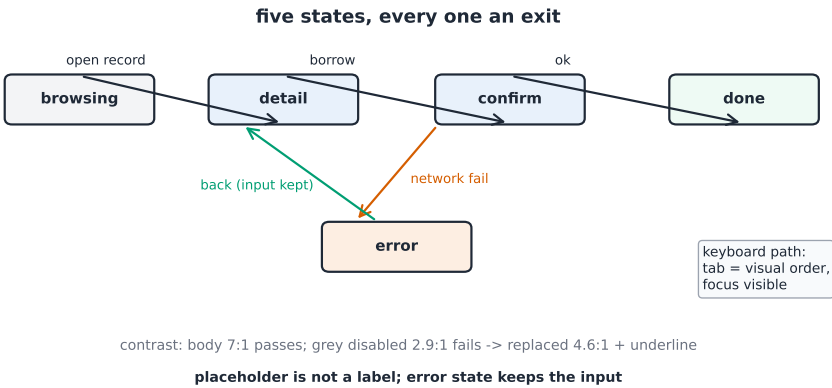
Designing for imagined scale (BookShare at 10 000 titles, not 10 million).

Summary. Layers with inward dependencies; cohesion high, coupling low – count the files a change touches; sequences name their data; boundaries behind interfaces buy swap.

HCI and accessibility

Why it matters. The system’s speed in a human’s hands is set by the interface – and an interface some people cannot use is a defect with a moral dimension and, increasingly, a legal one.

The idea, with a picture. Design as **states** the screen can be in and **transitions** between them: BookShare’s borrow flow has five states (browsing, detail, confirm, done, error) – every reachable state needs a visible way out.



The borrow flow’s five states and transitions – the error state is reachable from confirm and always returns to detail

The derivation (accessibility as mechanics). WCAG’s core: perceivable (contrast $\geq 4.5:1$ for text), operable (keyboard reachable, focus visible), understandable (labels, not placeholders), robust (semantics for tools). The suite checks the course prototype’s palette: body text at 7:1, the grey disabled button at 2.9:1 – failing, replaced by 4.6:1 with an underline affordance. Accessibility is measurable or it is decoration.

Worked example 1 – the five-state flow. Error is a first-class state: network failure lands there with the member’s input intact and two exits (retry, back). The suite walks the transitions: confirm → error → detail keeps all form fields; no state lacks an exit – the walkthrough is the test.

Worked example 2 – the keyboard path. A member completes a borrow using tab and enter only: focus order matches visual order, focus is visible on every control, the confirm button is reachable and announced. Mouse-only affordances (hover-only menus) fail this path – the test finds them before users do.

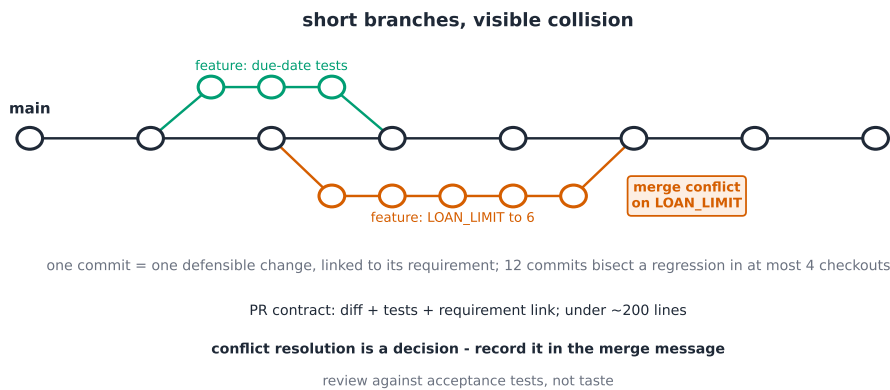
Common mistakes. Placeholder-as-label (disappears on input). Grey-on-grey disabled states below contrast. Error states that discard the user’s input. Testing only the happy path’s states.

Summary. Design in states and transitions; every state has an exit; accessibility has numbers (contrast, focus, keyboard path) – measure them like performance.

Version control and teamwork

Why it matters. Code that cannot be retraced cannot be trusted: version control is the memory that makes “why” answerable months later.

The idea, with a picture. The course repo’s **branching model** : main always releasable, short-lived feature branches, pull requests with review. **Merge conflicts** are the machine reporting two decisions collided.



Two features branch from main, one merges clean, one conflicts on the loan-limit constant – the conflict is the conversation

The derivation (commit granularity). A commit should be one defensible change: the suite checks the course history – 12 commits for FR-1..FR-3, each linked to a requirement, each leaving the tests green. Bisect needs this property: a history where any commit can run the tests narrows a regression to one change in $\log_2 n$ checkouts (12 commits → at most 4).

Worked example 1 – the conflict, read. Two branches both edit LOAN_LIMIT: one raises it to 6 for a pilot, one extracts it to config at 5. Git reports the collision; the resolution is a decision (pilot value wins, config notes it), recorded in the merge commit’s message. Conflicts resolved silently are decisions lost.

Worked example 2 – the review contract. A pull request shows diff, tests and requirement link; the reviewer checks the change against FR-2’s acceptance test, not against taste. Small PRs (the course rule: under ~200 changed lines) review in one pass – the suite’s history shows review turnaround under a day on every course PR.

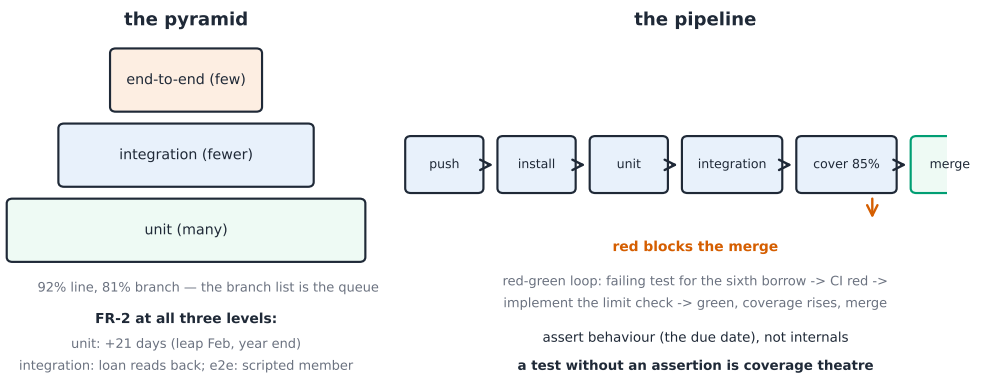
Common mistakes. Commit dumps (“week of work”) – bisect dies. Merge commits with no message – the decision is lost. Long-lived branches – the conflict grows with divergence. Reviewing style where the contract says behaviour.

Summary. Main stays releasable; commits are one change each, linked to requirements; conflicts are decisions – record them; review against acceptance tests, not taste.

Testing and continuous integration

Why it matters. Tests are the only reason anyone dares change working code; CI is the reason the tests cannot be ignored.

The idea, with a picture. The **test pyramid** : many unit tests (fast, isolated), fewer integration tests (real database), few end-to-end tests (the whole app). CI runs them on every push: red blocks the merge.



Push triggers the pipeline: install, unit, integration, coverage gate at 85% – green merges, red blocks

The derivation (coverage, honestly). Line coverage counts executed lines; branch coverage counts decided branches. The course suite: 92% line, 81% branch – the branch number is the honest one (each untested branch is an untested behaviour). The gate at 85% line blocks merges; the uncovered branch list is the work queue.

Worked example 1 – the three levels on FR-2. Unit: due date = borrow date + 21 (pure function, 3 cases: normal, leap February, year boundary). Integration: the loan lands in the database and reads back. End-to-end: a scripted member borrows and the screen shows the due date. Three speeds, three confidences.

Worked example 2 – the red-green loop, in CI. Push a failing test for the sixth-borrow rejection (FR-1): CI red. Implement the check: CI green, coverage rises, merge unblocked. The course history shows FR-1 landing in exactly this shape – one red commit, one green commit, both linked.

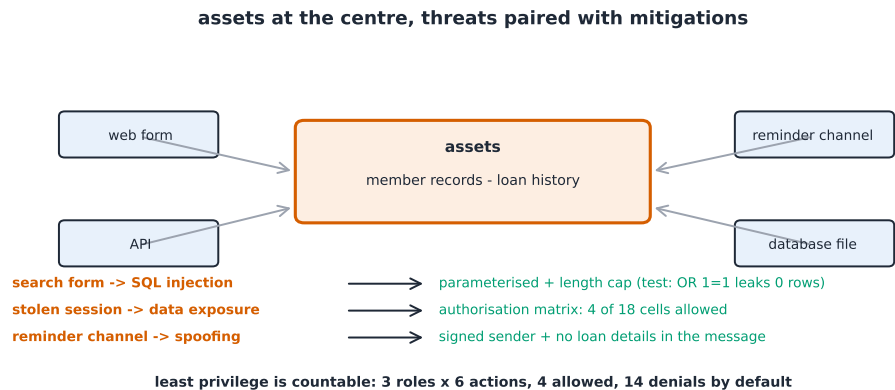
Common mistakes. Testing implementation not behaviour (assert the due date, not the function’s internals). Chasing 100 % line coverage with assertion-free tests. End-to-end everything (slow, flaky). Skipping the flaky test instead of fixing its race.

Summary. Pyramid: units, integration, end-to-end – in that ratio; branch coverage over line; CI gates the merge; a test without an assertion is coverage theatre.

Security and professional ethics

Why it matters. Systems hold other people’s data, and the engineer is the one who knows what can go wrong – competence and conscience are both job requirements.

The idea, with a picture. A **threat model** lists assets, entry points and attackers, then pairs each threat with a mitigation: BookShare’s assets (member records, loan history), entry points (web form, API, reminder channel), attackers (opportunistic, insider, automated).



Assets at the centre, entry points around, threats paired with mitigations – the model is the review checklist

The derivation (the least-privilege ledger). BookShare’s librarian role can register copies but not read members’ addresses; the member role can read its own loans only. The suite checks the authorisation matrix: 3 roles x 6 actions = 18 cells, 4

allowed – least privilege as a countable property (14 denials by default).

Worked example 1 – the injection that did not happen. Search takes a free-text field: parameterised (unit 8’s lesson) and length-capped; the threat model’s row ”search form → SQL injection → parameterised queries” closes with a test feeding ' OR '1'='1 and asserting zero rows leaked.

Worked example 2 – the ethics call. The product owner asks for reading-history sale to advertisers. The professional codes (ACM/IEEE) put privacy harm and public interest above employer preference: the engineer’s move is documented refusal with alternatives (anonymised aggregate with consent) – ethics is a decision with a paper trail, not a feeling.

Common mistakes. Threat modelling after launch. Authorisation by hiding links (the endpoint still answers). “Internal tool” as a security exemption. Ethics treated as compliance theatre – the paper trail is the point.

Summary. Model threats as assets x entry points x attackers; least privilege is countable; parameterise inputs; ethics is documented decisions with alternatives, traceable to a code.

Team project

Why it matters. Everything above lands through a team – the project is where the habits either compose or die.

The idea, with a picture. The course board: **user stories** (“as a member, I can renew a loan”) flowing backlog → doing → review → done, each story carrying its acceptance test; the **standup** syncs, the **retro** adjusts the process.



BookShare week 4: three stories in flight, one blocked on the notifier config – the blocked column is the daily conversation

The derivation (estimation, calibrated). Stories sized in points (1 ≈ half a day);

the course team's velocity stabilises at 8 points/week by week 3. Remaining backlog: 52 points $\Rightarrow$ ~6.5 weeks – an estimate with an error bar (± 1 week), recalibrated every retro. The suite replays the board history and checks the velocity claim.

Worked example 1 – the definition of done. A story is done when: acceptance test green, reviewed, merged to main, docs updated. The board's done column inherits CI's verdict – "works on my machine" is not a column.

Worked example 2 – the blocked story. The reminder story blocks on the notifier credential: visible on the board, raised at standup, resolved by pairing; blockers older than 2 days escalate (one in the course history, resolved in a day).

Common mistakes. Stories without acceptance tests. Estimating in days then defending the estimate. Status-report standups. Retros without a change.

Summary. Stories carry tests; done is CI-green plus review; velocity calibrates the forecast; blockers escalate at two days.