Ch. 2 — Software Architecture Fundamentals

Summary

The terminology backbone of the CPSA-F curriculum (iSAQB learning goals LG 1-1 … 1-10). Defines software-intensive systems, argues that every system inherently has an architecture ("a framework for change"), builds the core vocabulary — building blocks, interfaces, views, architectural levels — and closes with a bird's-eye view of the design process (Twin Peaks) and the architect's role.

Key points

Why every system has an architecture

- **The magic rectangle**: functionality, quality, effort, time — the four axes every project is judged on. Requirements engineering and architecture design are the highest-leverage disciplines because both force far-reaching decisions at the moment of least knowledge.
- **Software-intensive system**: a system whose essential tasks are carried out by software building blocks. Three categories, each with a typical instinct: *information systems* (data-heavy, many users → layered architectures, data/transaction problems), *embedded systems* (physical, resource-constrained, safety-critical → loosely coupled processes over buses, scheduling/comms problems), *mobile systems* (autonomous, personal, intermittently connected → shared-memory processes, UI-vs-hardware tension). Real systems blur categories, but each points to a toolbox.
- **Every system has an architecture** — inherent, not optional; the only choice is explicit design vs. accident. Rausch: "*Software architecture is a framework for change.*" Load-bearing wall analogy: architecture decides which parts are load-bearing (expensive to change) and which are decorative (cheap to change) — and that split is **relative to which future changes actually happen**, not a fixed property of the code.
- **Definition (ISO/IEC/IEEE 42010:2011)**: "fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution." Architecture objectives are **long-term**, often amortizing only after the project ends — unlike short-term project objectives (LG 1-7). Implicit

assumptions and constraints must be made explicit (LG 1-8).

Building blocks & interfaces

- **Building block** (deliberately not "component"): any abstraction of code, from function to subsystem. Three defining characteristics:
  1. *Provided and required interfaces* — provided interfaces are a contract to the outside world, but only honored when the block's own required interfaces are satisfied.
  2. *Encapsulation and interchangeability* — implementation is hidden behind interfaces; anything offering the same provided/required interfaces should be swappable without callers noticing.
  3. *Configuration and hierarchical (de)composition* — a building block can itself be a configuration of smaller building blocks wired together.
- **Interface**: a well-defined access point (syntax, behavior, errors, non-functional characteristics, protocols, semantics…). Interfaces can **never be fully specified** — Java's `Collection` documents everything except insert performance, which is exactly what decides `ArrayList` vs `LinkedList`. The architect decides which unstated aspects matter enough to pin down.
- **View depth**: black box (provided/required interfaces only — caller's view), gray box (+ technical/runtime interfaces), white box (internal configuration — implementer's view).
- **Who defines an interface**: standard (third party) / provided (exporter — most common) / required (importer — plugin style) / independent (neither side owns it + an adapter connects them). Independent maximizes decoupling but costs effort; if an adapter is used as a shortcut without ever generalizing the interface, "temporary" quietly becomes permanent.

Describing architecture: views & levels

- **IEEE 42010 description model**: stakeholders → concerns → viewpoints (conventions) → views, plus documented **rationale**. Views are **projections** — the same 3D object casts a circle from below and a triangle from the side; neither view is wrong or complete on its own.
- **Four architectural levels, two dimensions** (abstraction × perspective): architectural style (high/functional, e.g. "3-layer web + MVC"), technical infrastructure (high/technical, e.g. "thin client + app container + relational DB"), functional **A-architecture** (domain building blocks), technical **T-architecture** (cross-cutting concerns: persistence, transactions, logging). Siedersleben: **"A and T are blood groups — don't mix them."**
- **Environment**: four surrounding areas, each a two-way street — project management, product management/requirements engineering, execution platform/operations (the most neglected interface), tools/dev environment.
- **Quality of an architecture** is relative to objectives and lifecycle — good architecture keeps the magic rectangle achievable. ISO 25010 gives a quality-attribute checklist to start from.

Design process & the architect's role

- **Twin Peaks model**: requirements and architecture descend in parallel iterative spirals — effort estimates only become real once a draft architecture exists. Four equally weighted, **non-sequential** activities: analyze requirements/constraints; design views and technical concepts; evaluate architecture and decisions; support/review implementation.
- The architect is both a communication platform and the owner of the design/implementation blueprint, interfacing with nearly every other role on the project.

One example, all the vocabulary — `DocumentStore`

One running scenario to hold the terms together, built from the two real incidents discussed below: a `DocumentStore` building block that saves files to a cloud provider.

Concept	In `DocumentStore`
Building block	`DocumentStore` itself — an abstraction from "save a file" down to whatever actually implements it
Provided interface	`save(file) -> id`, `fetch(id) -> file` — the promise made to every caller
Required interface	A `CloudClient` (network + auth) — the promise only holds if this dependency is satisfied
Encapsulation & interchangeability	Swap `GoogleDriveClient` for `OneDriveClient` behind the same `DocumentStore` interface; callers shouldn't notice
Configuration & decomposition	`DocumentStore` = `RetryPolicy` + `Cache` + a `CloudClient` adapter, wired together internally
Interface incompleteness (the `Collection` lesson)	`save()` doesn't document max file size or latency — same kind of gap as `Collection` omitting insert performance; someone still has to decide if the gap matters
Interface definer type	Independent interface + adapter: `DocumentStore` belongs to neither Google nor Microsoft; `GoogleDriveAdapter` / `OneDriveAdapter` implement it
Black / gray / white box	Caller sees black box (`save`, `fetch`); ops sees gray box (retry/timeout config); adapter author sees white box (raw Drive API calls)
A/T blood groups	`DocumentStore` and its adapters are pure T-architecture; the mistake would be naming it `ClaimAttachmentStore` and hardcoding claim logic inside — A leaking into T

Concept	In `DocumentStore`
The bug actually hit	No independent interface existed — code called `SaveToGoogleDrive()` directly, a *provided* interface named after its *required* dependency. Migrating providers meant hunting every call site instead of writing one new adapter.

The one habit that would have prevented both real incidents below: name interfaces after what they *promise* (the capability), never after who currently *provides* them or which domain concept happens to *call* them.

Discussion notes

Three probes, all resolved with real examples from Mehdi's own work:

1. **Load-bearing walls / framework for change.** Worked through a 40-page static HTML site: nav duplication and content-in-markup are decorative under "keep the site current," but become load-bearing the moment a requirement like i18n arrives — because now every hardcoded string is something a translation process must touch. Mehdi's own framing: "*the whole HTML content became a load bearing [wall] that was not before.*" Fix while cheap = separate content from structure (externalize strings) before the requirement lands, not after.
2. **A/T mixing, case 1 — ORM naming.** Mehdi had custom ORM functions named after domain concepts (`claim()`, `filing()`, `process()`) instead of technical ones. Domain vocabulary had leaked into the T-architecture's *provided interface names*, so a later business-vocabulary change forced a rename across every consuming service — a technical migration that should have been invisible to callers instead broke all of them. Fixed by renaming the technical side back to technical terms; the rename itself was costly precisely because the interface names had become a de facto contract.
3. **A/T mixing, case 2 — `SaveToGoogleDrive()`.** A different but related failure: the interface was named after a *required* dependency (Google Drive) rather than the *capability* it provides. No adapter layer existed, so when the provider needed to change to OneDrive, the fix was hunting down and manually verifying every call site — the cost an independent-interface-plus-adapter design would have avoided by confining the change to one new adapter.

Cross-link: LG 1-8 (implicit assumptions → explicit statements) is the same phenomenon as the METR paper's "AI lacks tacit repo context" (digests/metr-early-2025-ai-developer-productivity-rct) — knowledge that lives only in maintainers' heads, or in a function name nobody questioned, is invisible to any newcomer, human or AI.

Concepts

- concepts/software-architecture — architecture as inherent, framework for change, load-bearing walls relative to anticipated change
- concepts/building-blocks-and-interfaces — the three characteristics, interface completeness, interface definer types, the `DocumentStore` example
- concepts/architectural-views-and-levels — IEEE 42010 description model, four levels, A/T blood groups
- concepts/twin-peaks-model — requirements/architecture co-evolution, four non-sequential design activities (light for now — grows with ch. 3)

Open questions

- "Test your knowledge" section (LG 1-1 … 1-10) not yet used — good material for a future review session.
- How do the four architectural levels map onto the Twin Peaks design activities? Revisit in ch. 3 (Designing Software Architectures).
- When is it worth paying for an independent interface + adapter *upfront* versus accepting the risk and refactoring later? (effort vs. risk tradeoff — ties into Twin Peaks' point that estimates only firm up once a draft architecture exists)

Revision #2
Created 20 July 2026 12:51:58 by EMB
Updated 20 July 2026 12:53:16 by EMB