iSAQB CPSA

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
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)

Ch. 2 Glossary — Software Architecture Fundamentals

Exhaustive term list for ch. 2, alphabetical. Companion to digests/software-architecture-fundamentals/02-software-architecture-fundamentals — read that first for the narrative and worked example; use this to look up a precise definition. Quoted boxes are the book's own formal definitions (with source); unmarked entries paraphrase the surrounding text.

A

Adapter — A mediating building block placed between an importer and an exporter when both sides define their own (independent) interfaces, so the two can be connected despite not sharing an interface definition. The cost of decoupling; if never removed/generalized, a "temporary" adapter becomes permanent.

Analysis of requirements and constraints — One of the four architecture design activities: analyzing objectives, constraints, and functional/ non-functional requirements from the surrounding areas, including gaps, flexibility, and susceptibility to change. All parties need a shared initial understanding of the architecture style and technical infrastructure before this can proceed.

Application software — One of three software categories in the IEEE Glossary (with support software and system software); classification is observer-relative — the same software can be application software to one stakeholder and system software to another (e.g., a web browser: application software to the surfer, system software to the plug-in developer).

A-architecture — See **Functional architecture level**.

Architect's role / Software architect — See dedicated entries under **Communication and discussion platform** and **Design and implementation plan**.

Architectural description — See **Architecture Description**.

Architectural level — A layer of abstraction/perspective at which an architecture is described (e.g. architectural style, technical infrastructure, functional architecture level, technical architecture level). Per the book's conceptual model (fig. 2-11): an Architectural description consists of 1.. *Architectural levels, each combining 1..* Views, which consist of Diagrams and/or Free text, with Diagrams stored in Models.

Architectural style — The central architectural metaphor of a system at the highest, most abstract, functional-perspective level — e.g. "three-layer web system with relational database," "service-oriented architecture with object-oriented data access layer," "pipe and filter batch system."

Architecture Description (AD) — Per IEEE 42010:2011: a collection of artifacts that together document a software architecture (not a single document). An AD identifies stakeholders and their concerns, expresses the architecture, and is organized into architecture views per governing viewpoints, with rationale.

Architecture Model — Governed by a Model Kind; the concrete modeling artifact(s) that make up part of an architecture view (per IEEE 42010's conceptual model, fig. 2-9).

Architecture Rationale — The documented reasoning behind architecture decisions; part of the Architecture Description, used to justify decisions against stakeholders' concerns.

Architecture View —

> "An architecture view in an AD expresses the architecture of the system of interest from the perspective of one or more stakeholders to address specific concerns, using the conventions established by its viewpoint. An architecture view consists of one or more architecture models." [ISO/IEC/IEEE 42010]

Views are **projections**: like a 3D object casting different 2D shadows from different angles, no single view is the whole architecture, and none is "wrong" for showing only part of it (fig. 2-10).

Architecture Viewpoint —

> "An architecture viewpoint is a set of conventions for constructing, interpreting, using and analyzing one type of architecture view. A viewpoint includes model kinds, viewpoint languages and notations, modeling methods and analytical techniques to frame a specific set of concerns. Examples of viewpoints are: operational, systems, technical, logical, deployment, process, information." [ISO/IEC/IEEE 42010]

B

Black box view — The view of a building block showing only its provided and required interfaces — the caller's / user's view. Respects the information-hiding principle by hiding internal (private) details. Describable with UML component diagrams.

Blood groups (Siedersleben) — Metaphor for keeping the A-architecture (application/functional layer) and T-architecture (technical layer) separate: "as with blood groups, mixing of the architecture layers is undesirable."

Building block — The book's deliberate replacement for "component" (which it avoids because "component" is overloaded — UML components vs. programming constructs like packages/JavaBeans). Formal definition, given via its three defining characteristics:

> "A building block provides interfaces that it guarantees in the sense of a contract. This guarantee, however, only applies when the interfaces that it requires are made available within the scope of a corresponding configuration. (**Provided and required interfaces**) Via the provided and required interfaces, the building block encapsulates the implementation of these interfaces. It can thus be replaced by other building blocks that provide and, where appropriate, also require the same interfaces. (**Encapsulation and interchangeability**) Building blocks are also the unit of hierarchical (de)composition of a software-intensive system. In other words, a building block can be implemented using an appropriate configuration of other (sub-)building blocks and their interrelationships... (**Configuration and hierarchical (de)composition**)"

A building block is the central static-structure element of a software architecture: an abstraction of source code ranging from small (functions, classes) through medium (packages, libraries) to large (subsystems, layers, frameworks) — and can itself be composed of other building blocks (fig. 2-4).

Building block instance / placeholder — Within a configuration (white-box view), the sub-elements (called "parts" in UML) that occupy slots and use an instance of a building block are placeholders, not building blocks themselves — comparable to variables: a value (=building block instance) plus a type (=building block). E.g., placeholder "b1 : Building block B1" can only hold an instance of building block B1. Loosely also just called "building blocks" when the distinction doesn't matter.

C

Communication and discussion platform — One of the two core functions of the software architect (alongside **Design and implementation plan**): using the architecture to present requirement feasibility to the requirements engineer/customer/user, correlate and prioritize requirements, identify contradictions, evaluate alternative solutions, and advise the project manager on planning and risk.

Concern — A stakeholder's interest in a system, relative to which they have a view (per IEEE 42010's conceptual model). Architecture views/ viewpoints address concerns; concerns motivate viewpoints.

Configuration and hierarchical (de)composition — One of the three defining characteristics of a building block: it can be implemented as a wiring of sub-building blocks (a configuration), which it encapsulates; external interfaces may be delegated to internal sub-blocks and vice versa.

Correspondence / Correspondence Rule — Elements of IEEE 42010's conceptual model (fig. 2-9) relating architecture description artifacts to each other; not elaborated further in this chapter beyond the diagram.

Custom software — Software built for a specific customer/use case, as opposed to **Standard software**; one common (if imperfect) categorization axis for software-intensive systems.

D

Degree of abstraction (dimension) — One of two axes (with **Perspective**) along which architecture description/refinement methods (TOGAF, RM-ODP, Zachman, and the book's own 4-level model) organize themselves: how abstract vs. detailed a given description is.

Design and implementation plan — The second core function of the software architect: defining building blocks, interfaces, and interaction patterns; driving adoption of new technologies; owning programming guidelines; supporting developers with prototypes, code reviews, and reused implementations; supporting testers by defining test conditions tied to architecture objectives; and serving as point of contact for architecture- relevant faults, operations staff, and security experts.

Development of architecture views and technical concepts — One of the four architecture design activities: the more detailed, view-based design of the functional and technical architecture levels — breaking down functional requirements into the functional level, and designing/ documenting cross-cutting technical solution building blocks at the technical level, all within the constraints of the architecture style and technical infrastructure.

E

Embedded systems — One of three software-intensive system categories: software embedded in physical objects, under significant hardware resource constraints, carrying out tasks critical to data security and functional reliability (regulation, control, communication). Examples: washing machines, machine tools, production-line controllers, mobile-network radio cells, airbag controllers, parking assistants. Typical architecture: active processes/modules, loosely coupled, interacting over networks (e.g. bus-based). Typical design problems: scheduling of active processes, network communication load.

Encapsulation and interchangeability — One of the three defining characteristics of a building block: the implementation behind provided/ required interfaces is hidden, so any building block offering the same interfaces can be substituted without callers noticing (subject to caveats about required-interface side effects).

Evaluation of architecture and design decisions — One of the four architecture design activities: quality-assuring the developed architecture via review techniques, prototypes, tests, analysis; centrally, deriving concrete scenarios from requirements to test the architecture's quality.

Execution platform and operation — One of the four surrounding environment areas of software architecture: the existing (or to-be-built) operations/hardware/infrastructure landscape that new systems should reuse where possible, and that can in turn generate new requirements from the architecture. Called out in the text as "the interface... far too often neglected."

F

Free text — Along with Diagrams, one of the two content types that make up a View in the book's conceptual model for architecture descriptions (fig. 2-11); essential for documenting understanding and design rationale that diagrams alone can't carry.

Functional architecture level (A-architecture) — The more detailed, functional-perspective architectural level: application/domain building blocks and their relationships, designed to implement the system's functional requirements (e.g. a generic insurance product model). Paired with, and must not be mixed with, the **Technical architecture level** ("blood groups").

Functional Suitability — One of the eight ISO/IEC 25010 top-level quality characteristics (with Reliability, Usability, Performance efficiency, Security, Maintainability, Compatibility, Portability), offered as a starting checklist for deriving quality characteristics for a specific architecture.

FURPS — Referenced attribute set (Functionality, Usability, Reliability, Performance, Supportability) usable to cross-check completeness of derived quality requirements; pointed forward to ch. 5.

G

Gray box view — The view of a building block showing its additional, mostly technical required interfaces — e.g. configuration interfaces or runtime-environment interfaces. Sits between black box and white box. Describable with UML deployment diagrams.

I

Implementation support and review — One of the four architecture design activities: communicating the architecture to all project participants at the right level of detail for each audience, and maintaining a two-way feedback loop during implementation so unforeseen problems get folded back into the architecture rather than silently worked around.

Independent interface — An interface definition type where neither the importer nor the exporter owns the interface — both define their own, and an **Adapter** bridges them. Maximizes decoupling and independent development/testing, at the cost of extra effort and the risk that a "temporary" adapter never gets removed.

Individual software — Bespoke, one-off software at the opposite end of the diffusion-rate/specialization axis from **Standard software** (fig. 2-2).

Information systems — One of three software-intensive system categories: focused on managing/processing information — large data volumes or complex structures, often serving many simultaneous interactive users. Examples: insurance core systems, SAP systems, CAD systems, weather- simulation systems, engineering simulations. Typical architecture: layered. Typical design problems: data management, transaction control.

Interface —

> "An interface represents a well-defined access point to the system or its building blocks. In this context, an interface describes the characteristics (for example, attributes, data, and functions) of this access point. The objective is to define these characteristics as precisely as possible with all the necessary aspects, such as syntax, data structures, functional behavior, error behavior, non-functional characteristics, the interface usage log, technologies, protocols, access modifiers, file formats, conditions/constraints, and semantics."

Interfaces can never be *completely* specified in practice (the Java `Collection` / insert-performance example: `ArrayList` vs. `LinkedList` are interchangeable per the interface, but differ exactly on the property the interface leaves undocumented). It falls to the architect to judge which unstated aspects matter enough to pin down.

IEEE Standard 610.12-1990 — *IEEE Standard Glossary of Software Engineering Terminology*; source of the base definitions of **System** and **Software** that the chapter combines to define **Software-intensive system**.

ISO/IEC/IEEE 42010:2011 — *Recommended Practice for Architectural Description for Software-Intensive Systems*; source of the chapter's adopted definition of software architecture, and of the conceptual model underlying **Architecture Description**, **Stakeholder**, **Concern**, **Architecture Viewpoint**, and **Architecture View**.

ISO/IEC 25010 — Standard defining eight top-level software quality characteristics, used here as a checklist starting point (see **Functional Suitability**).

L

Learning goals (LG 1-1 … 1-10) — The iSAQB CPSA-F curriculum's ten learning goals underlying this chapter: definitions of software architecture (1-1); benefits/objectives (1-2); architecture as part of the lifecycle (1-3); architect's tasks/responsibilities (1-4); architect's relation to other stakeholders (1-5); development approaches ↔ architecture correlation (1-6); architecture vs. project objectives (1-7); explicit statements vs. implicit assumptions (1-8); architect roles in organizational context (1-9); differences between IT system types (1-10). Detailed sub-points for each are reproduced verbatim in §2.5 ("Test your knowledge") — good material for a dedicated review session.

M

Magic rectangle — The four axes on which every software project is judged: functionality, quality, effort (cost), and time. Requirements engineering and architecture design are the two highest-leverage disciplines because both force high-impact decisions early, when knowledge is least complete (fig. 2-1).

Mobile systems — One of three software-intensive system categories: (semi-)autonomous, personal units with high interaction requirements; provide local (semi-)autonomous functions but also intermittently synchronize/coordinate with centralized stationary systems; connectivity is not continuous. Examples: smartphones, (semi-)autonomous transport robots, sensor/actuator nodes in ad-hoc networks. Typical architecture: active processes communicating mainly via shared memory (single, possibly multicore, processor). Typical design problem: balancing a resource-heavy, high-quality GUI against hardware-level sensor/actuator functions.

Model Kind — Governs an Architecture Model in IEEE 42010's conceptual model; associated with a Viewpoint.

P

Perspective (dimension) — One of two axes (with **Degree of abstraction**) along which architecture description methods organize themselves: which architectural area is being addressed (e.g. data, process, service, program organization; or, in the book's own model, functional vs. technical).

Product Management and Requirements Engineering — One of the four surrounding environment areas of software architecture: supplies functional and non-functional requirements, quality objectives, and constraints (which can change mid-project); architecture, in turn, can surface conflicts between requirements or feed back impulses for requirement changes.

Project environment and project management — One of the four surrounding environment areas of software architecture: supplies budgets, development-approach constraints, and project objectives; also encompasses the wider, continuously changing project/application landscape whose churn (new or terminated adjacent projects) can significantly impact the system's

interfaces.

Provided interface — An interface definition type where the interface is defined by the exporting (providing) building block. The most common type of interface after **Standard interface**.

Q

Quality of a software architecture — Per [BCK03]: an architecture is "good" to the extent it enables the project and system to meet their objectives within the **Magic rectangle**, across the system's lifecycle. Quality is inherently subjective/relative to the assessor and to the specific (often partly implicit) objectives, constraints, and requirements in play — there is no context-free "good architecture." Implicit quality objectives inherent to the notion of architecture itself: ease of understanding, up-to-date documentation, correct implementation, ease of development, simple/economic operation, extensibility, maintainability, and maximal reuse of existing building blocks.

R

Required interface — An interface definition type where the interface is defined by the importing (requiring) building block — common in framework configurations, where you plug specific functionality into a predefined program structure.

S

Software (IEEE 610.12-1990) —

> "Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system." [IEEE 610.12-1990, p. 66]

Software architecture — Three definitions given, in order:

1. (Rausch) "Software architecture is a framework for change."
2. (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."
3. (This book's working definition, adopted for the rest of the text) "The software architecture defines the fundamental principles and rules for the organization of a system and its

> structure into building blocks and interfaces, and their relationships to each other and to the surrounding environment. It thus defines guidelines for the entire software lifecycle, the developer, and the software's operator, from analysis via design and implementation to operation and enhancement."

Has two aspects: **constructive** (system structure: building blocks + interfaces + relationships + environment) and **procedural** (principles/rules governing the lifecycle, developer, and operator). Architecture objectives can be long-term, outlasting and outlasting-in- amortization the project's own objectives.

Software architecture design — The overall activity of producing a software architecture; framed as four non-sequential, equally-weighted, iteratively/incrementally interleaved activities (fig. 2-16): **Analysis of requirements and constraints**, **Development of architecture views and technical concepts**, **Evaluation of architecture and design decisions**, **Implementation support and review**. Operates as continuous top-down/ bottom-up flow across abstraction levels (surrounding-area constraints → architecture style/technical infrastructure → functional/technical architecture levels → program design/implementation), each direction feeding information back the other way (fig. 2-17).

Software-intensive system — The chapter's own synthesis definition, built by combining the IEEE definitions of **System** and **Software**:

> "A software-intensive system is a collection of building blocks that are organized in such a way that they together accomplish the purpose of the system. Building blocks of such a system that entirely or for the most part consist of software carry out essential tasks for achievement of the purpose of the system. The software element of the system consists of a collection of programs, procedures, data, and associated documentation."

Three categories given (not mutually exclusive; many real systems blur the lines): **Information systems**, **Embedded systems**, **Mobile systems**.

Stakeholder — A party with an interest (**Concern**) in a system; per IEEE 42010's conceptual model, stakeholders' concerns are what an architecture description must address and justify decisions against. Customers, users, and developers can all be stakeholders in architecture discussions, not just fellow architects.

Standard interface — An interface definition type where an external third party defines the interface, and both the providing and requiring building blocks comply with that external definition.

Standard software — Off-the-shelf, broadly diffused software, as opposed to **Custom software** / **Individual software**; one axis of the multidimensional classification in fig. 2-2 (diffusion rate vs. specialization).

Support software — One of three IEEE software categories (with application software and system software); classification is observer-relative (e.g., an insurance database is system software to the customer, support software to the programmer).

System (IEEE 610.12-1990) —

> "A collection of components organized to accomplish a specific function or set of functions." [IEEE 610.12-1990, p. 73]

System-of-Interest — In IEEE 42010's conceptual model: the system being architected, which exhibits an Architecture and has stakeholders with interests in it.

System software — One of three IEEE software categories (with application and support software); observer-relative (e.g., a web browser is system software to the plug-in developer, application software to the end user).

T

Technical architecture level (T-architecture) — The more detailed, technical-perspective architectural level: cross-cutting solution building blocks designed and documented from non-functional requirements (e.g. a general versioning solution that functional entities at the A-architecture level then consume). Must not be mixed with the **Functional architecture level** ("blood groups" — Siedersleben).

Technical infrastructure — The high-abstraction, technical-perspective architectural level: the network/deployment profile of the system, e.g. "thin client with a web and application container and a relational database."

Tools and development environment — One of the four surrounding environment areas of software architecture: the tooling, IDEs, and development organization needed to actually implement the architecture; architecture decisions can in turn create new tooling requirements (e.g. expanded test infrastructure for a newly selected technology).

Twin Peaks Model — [Nus01]. Model in which requirements engineering and architecture design descend together as two interleaved iterative spirals from general/independent (top) to detailed/implementation-dependent (bottom) (fig. 2-14). Rationale: effort estimates for requirements can only be realistically assessed once a draft architecture exists; presenting such estimates back to the customer often leads them to waive requirements, or reveals that the requirements weren't precisely defined.

V

View — In the book's own conceptual model for architecture descriptions (fig. 2-11, distinct from but related to IEEE 42010's **Architecture View**): the child of an Architectural level, composed of Diagrams (backed by Models) and/or Free text. See also **Architecture View** for the IEEE-standard version of the same concept, and **Views are projections** for the intuition.

Views are projections — The chapter's core intuition for why multiple views coexist without contradiction: like a single 3D object casting a circular shadow from one angle and a triangular shadow from another, different architecture views are non-contradictory projections of the same underlying architecture, each incomplete on its own (fig. 2-10).

W

White box view (glass box view) — The view of a building block showing its internal details: its decomposition into a configuration of sub-building blocks (or another implementation), including the delegation of its provided/required interfaces to that internal structure. This is the implementer's view. Describable with UML composite structure diagrams.

Z

Zachman Framework — [O'RF+03]. Cited example of an architecture description approach that differentiates architecture levels along two dimensions: roles and perspectives (analogous in spirit to this chapter's own perspective × abstraction model, fig. 2-12).

Standards and frameworks cited (non-glossary reference list)

- **IEEE 610.12-1990** — *IEEE Standard Glossary of Software Engineering Terminology* (System, Software definitions)
- **ISO/IEC/IEEE 42010:2011** — *Recommended Practice for Architectural Description for Software-Intensive Systems* (architecture definition, stakeholders/concerns/viewpoints/views conceptual model)
- **ISO/IEC 25010** — Software quality characteristics
- **[BCK03]** — Source for the "good architecture" quality definition
- **[SD00]** (Siedersleben) — "Blood Groups" (A/T architecture separation)
- **[Kru95]** (Kruchten) — 4+1 architectural view model
- **[O'RF+03]** — Zachman Framework
- **[Nus01]** — Twin Peaks Model

- **TOGAF®, RM-ODP** — Named as other architecture description/refinement frameworks sharing the perspective × abstraction structure; detailed in ch. 4