XF

Cross-Framework Architectural Model

XF Architectural Model Cross-Framework

XF: A Framework-Agnostic Architectural Reference Model for Unified Development Process

Israel Sanjurjo Cuenca
ORCID 0009-0006-9584-7977

June 9, 2026

https://xfarch.org

Document code	XF-CFAM-001:2026
Model version	1.0.0
Document status	Version 1.0.0 — Reference edition
Issue date	June 9, 2026
Autor	Israel Sanjurjo Cuenca
ORCID	0009-0006-9584-7977
Editorial framework	Cross-Framework Architectural Model (CFAM) — https://xfarch.org
Change policy	Model changes are managed through a public proposal process. Versions are identified with three-level semantic numbering (major.minor.patch): *substantive* changes to the taxonomy or the axioms increment the major version; *editorial* additions to the rule catalog or reorganizations increment the minor version; *clarifying* wording refinements, without altering the normative content, increment the patch version.

Patent rights declaration

The author of this document holds no patent right whose application is necessary for the implementation of the XF model, and is not aware of any third-party patent rights over the provisions described herein. The author draws attention to the possibility that the implementation of this document may require the use of patents not yet identified. Any party that believes it holds patent rights essential to the application of the model is invited to notify the author in writing.

Abstract

The rapid proliferation of programming languages, frameworks, and development tools has produced a highly fragmented software development ecosystem in which each technology imposes its own nomenclature, structural patterns, and organizational conventions. This fragmentation generates measurable operational costs — increased context-switching overhead, inconsistent implementations, and structural dependency on proprietary tools — despite the fact that all software artifacts implicitly implement the same three invariant stages of any formal process: interaction, processing, and access. This paper presents XF (Cross-Framework), a framework-agnostic architectural reference model that organizes the internal structure of any software artifact through two orthogonal classification dimensions: an abstraction dimension that stratifies the artifact into three layers derived from the invariant stages of formal processes, and a functional dimension that classifies components into five types with precise and non-overlapping responsibilities. The model prescribes a closed and exhaustive component taxonomy, a canonical nomenclature, and a canonical folder structure that are universal and independent of the implementation technology. The model introduces a formal conformance framework — a catalog of verifiable rules organized in nine thematic groups and a deterministic algorithm for computing the conformance level of any artifact — that enables automated static analysis tools to evaluate architectural correctness without human interpretation. Additionally, the formal ontology of the model acts as a high-precision functional specification language that reduces ambiguity in requirements description and enables the algorithmic transformation of functional specifications into architecturally correct and verifiable implementations. XF addresses the three problems that prior models — Clean Architecture, Domain-Driven Design, layered architectures — solve only partially or independently: layer structure derived from first principles, closed and exhaustive component taxonomy, and canonical nomenclature that makes visible the structural convergence that software already implements implicitly.

Contents

List of Tables

List of Figures

List of Listings

1 Introduction

The internal clauses are informative. Their purpose is to set out the motivation, the conceptual foundations, and the relationship of the XF model with existing standards and reference models. The normative content of the model is established in clauses 2, 3, 4, and 6 to 12.

This document establishes the normative guidelines of the *Cross-Framework* Architectural Model (hereinafter, **XF**), a reference model for structuring software artifacts independently of the programming language, development framework, or execution environment used.

The XF model defines a component taxonomy, a hierarchical stratification into abstraction layers and a set of conformance rules that make it possible to organize the logic of any software artifact uniformly, from mobile applications and web services to embedded systems, edge-computing devices, and industrial automation solutions. The model does not prescribe concrete implementations nor restrict the use of existing development frameworks; instead, it provides a conceptual abstraction layer that allows the structure of any artifact to be expressed in universally understandable terms, independently of the underlying technology.

The scope of this document comprises:

- The formal definition of the abstraction layers that make up the XF Architecture and the responsibilities of each.
- The exhaustive typing of the software components admitted by the model and the applicable classification criteria.
- The conformance rules that determine the degree to which an artifact adheres to the model.
- The mechanisms for instantiation, initialization, and termination of the architecture.
- The compatibility and interoperability rules with artifacts that do not implement the XF model.
- The ontology of terms and concepts of the XF Architecture domain.

The following are outside the scope of this document: the definition of software project management methodologies, source-code quality criteria beyond architectural conformance, the implementation specifications specific to each development framework, and the certification procedures for individuals or artifacts, which will be the subject of complementary documents.

1.1 Motivation

The discipline of software engineering is characterized by a constantly expanding technological diversity. The sustained appearance of new development frameworks, programming languages, and build tools has given rise to a highly heterogeneous ecosystem in which each technology imposes its own nomenclature, its own structural patterns, and its own code organization conventions.

Before describing the consequences of this heterogeneity, it is worth stating a central observation: **the fragmentation of the industry is, first and foremost, a**

terminological and structural problem.

In most software artifacts that solve a real problem with practical value, the tripartition that the XF model formalizes emerges implicitly: the artifact executes processing logic as its necessary core and, when it communicates with its environment — the typical case of management, transaction, and automation systems —, it does so through only two structural channels: it accesses external data when it needs it, and it exposes the results to its consumers when it receives an invocation.

Fowler [12] documented that the same architectural patterns recur in heterogeneous *enterprise* systems, regardless of the language or development framework used. This observation is evidence that the underlying structure of software converges toward common solutions that the industry's terminological diversity has prevented from being recognized as equivalent. For illustrative and informative purposes: Spring calls `@Service` what Angular calls `Injectable`, what Flutter calls `BLoC`, and what Android calls `ViewModel` — four different names that designate the same architectural role in each framework. Likewise, Spring calls `@Repository` what Django calls `QuerySet`, what Node.js calls `Model`, and what J2EE calls `DAO` — four different names that fulfill the same architectural role in each framework.

Documented industrial practice suggests that software, in its deep nature, conforms to the formal modeling of processes. What is not uniform is the way that conformance is expressed — each development framework, each team, and each project names it, structures it, and organizes it differently. The consequence is not that the software is poorly built — it is that it cannot be read, understood, or maintained with the same ease by developers coming from different technological backgrounds. The fragmentation is not a problem of the substance of the software but of its surface: the nomenclature, the stratification, and the vocabulary with which it is described and communicated.

This observation has a direct implication for the scope of the problem that the XF model addresses. It is not about correcting how software works — it is about making explicit and uniform what software already does implicitly. XF does not impose on the developer a new way of building software — it provides a common vocabulary and a prescribed structure to describe what they already build, so that any other developer trained in the model can recognize the same concepts under the same names, regardless of the technology used.

This heterogeneity generates measurable operational consequences in the organizations that develop and maintain software:

Knowledge fragmentation. Developers specialized in one development framework find it difficult to operate effectively in another, even when the underlying business processes are equivalent. Architecturally identical concepts receive different names depending on the technology used, while the same term may designate radically different concepts in different frameworks: the term `Service` refers to business logic in Spring, to an asynchronous task in Android, and to a point of systemic interaction in the XF model. Evans [11] introduced the concept of *ubiquitous language* as a necessary condition for precise communication among the members of a development team, and identified its absence as one of the main causes of semantic ambiguity in software projects — a diagnosis that the XF model extends to the level of the industry: the

terminological fragmentation between development frameworks prevents developers from recognizing as equivalent the concepts they already implement convergently. This semantic inconsistency constitutes a structural barrier to knowledge transfer between projects, teams, and organizations, and confines knowledge to technological silos that hinder internal mobility and raise the cost of onboarding new team members.

Increased context-switching cost. Each transition between projects with different technologies imposes a learning curve that does not reflect the complexity of the business domain, but rather the particularities of the tool. A developer with sound knowledge of software architecture must invest time and resources in learning the proprietary conventions of each framework before being able to contribute effectively, regardless of whether the problems to be solved are functionally analogous to those already solved in other technologies. This cost adds no value to the product and constitutes a structural inefficiency of the development process, replicated systematically across the entire industry.

Implementation inconsistency. The absence of a common reference model leads different teams to solve equivalent problems with divergent approaches within the same organization, and even within the same project when the teams have been trained in different technologies. Jones [25] identified inconsistency in design conventions as one of the factors with the greatest negative impact on development speed and on the defect density of the product — a direct consequence of the terminological fragmentation that prevents teams from recognizing and reusing equivalent architectural solutions. This divergence hinders code review, solution auditing, maintenance-effort estimation, and the onboarding of new developers, who must understand not only the business domain but also the architectural decisions adopted in each project.

Structural dependency on the tool. In the prevailing development model, the architecture of the artifact is subordinated to the chosen development framework. Architectural decisions do not derive from the analysis of the process being automated, but from the conventions imposed by the tool, generating as many ways of working as there are tools on the market. Booch [6] established that the architecture of a system must be understandable to all of its participants, not only to its creators — a property that the industry's terminological fragmentation systematically compromises by making the same architectural structure unrecognizable to developers trained in different technologies. This dependency has long-term implications: when a technology becomes obsolete or is replaced, the architectural knowledge accumulated around it is not directly transferable to the new tool, forcing organizations to bear a migration cost that goes beyond the source code and affects processes, training, and documentation.

Difficulty of scalability and maintenance. Software projects must evolve over time to adapt to new business requirements, to changes in third-party integrations, and to the growth of the development team. Architectural diversity hinders this evolution, since differences in dependency management, configuration, and project structure generate inconsistencies that are amplified with scale.

These consequences are not independent of one another — they reinforce each other and produce a cumulative effect that negatively impacts development speed, the quality of the software produced, and the ability of organizations to maintain and

evolve their systems over time. This document starts from the hypothesis that they share a common root cause: not that the software is poorly built, but that the same well-built software is described, named, and structured in mutually incompatible ways.

In established engineering disciplines — civil, electrical, mechanical — the existence of normative standards that regulate terminology, processes, and the interfaces between components has been a factor in the systematic advancement of the discipline and in the sustained reduction of its production costs [7]. A civil engineer does not conceive of constructing a building outside the applicable regulations; an electrical engineer does not design a distribution system without adhering to the established schematics and safety protocols. Software engineering, despite its relative maturity as a discipline, still lacks an equivalent in the field of application architecture. The XF model constitutes a step in that direction — not to change how software is built, but to make uniform and explicit what software already does.

1.2 Foundations of the model

The XF model is founded on two premises of a universal character, derived from the theory of the formal modeling of processes and from the layered-abstraction principles of the OSI model [17]. Both premises are observations about intrinsic properties of software — they are not conventions that the model imposes but realities that the model formalizes and makes explicit.

First premise: every software artifact already automates a formal process Every formal process automatable by software necessarily contains a *processing* stage — the ordered execution of the procedures that produce the expected result — and, when it communicates with its environment, it does so exclusively through two structurally distinct channels: *interaction*, when the process receives the order to execute, verifies its preconditions, or consumes data from the outside, and *access*, when the process delegates part of its execution to external systems through a defined communication protocol. These two channels are optional depending on the process — there are purely computational processes without external communication, internally triggered processes without interaction, and self-contained processes without access — but they are **exhaustive**: there is no third structural channel of communication with the environment. This property — necessary processing, exhaustive communication in its two forms — is what the XF model formalizes as the three abstraction layers of the artifact. The BPMN 2.0 standard [35] structures business processes with the same tripartition — *Activities*, *catch Events*, *throw Events* — and the calculus of communicating sequential processes CSP [16] formalizes it mathematically. The rigorous derivation and the formal traditions that support it are developed in §5.2.

Every software artifact that solves a problem with practical value implements processing as its necessary core and, in most cases of practical interest — management applications, business services, automation systems —, also the two channels of communication with the environment. A temperature-management application accesses data from the thermostat, processes it according to the domain rules, and exposes it to the user — access, processing, interaction —. An authentication microservice queries credentials in a database, verifies their validity, and returns a token — access,

processing, interaction —. An automation script reads configuration files, transforms the data, and writes the results — access, processing, interaction —. The underlying formal process follows the same structural pattern in all three cases. What varies is the name that each development framework gives to each stage and the way it organizes it in its structure.

Parnas [39] established that the structure of a system must reflect the design decisions most likely to change, and that these decisions are determined by the nature of the process the system implements. This observation anticipates the first premise of the XF model: if the nature of the process determines the structure of the system, and if the processing–interaction–access tripartition captures the only possible structural categories of a formal process, then the structure of every software artifact should converge toward a common organization — regardless of the technology. The fragmentation documented in §1.1 does not contradict this structural convergence — it confirms it: development frameworks converge toward the same solutions because the processes they model have the same underlying structure, but they do so with different nomenclatures that conceal that convergence.

Brooks [7] identified accidental complexity — that introduced by the tools and the development processes, not by the problem itself — as one of the main causes of failures in software projects. The industry's terminological fragmentation is precisely that kind of complexity: it does not derive from the problem the software solves but from the way the tools have chosen to describe it. The XF model eliminates that accidental complexity by providing a single terminology to describe what the software already does — making visible the structural convergence that development frameworks implement implicitly.

Second premise: the architecture must precede the tool If every software artifact implements processing as its necessary core and, when it communicates with the environment, it does so through the two exhaustive channels of the formal process, then the structure of the artifact must derive from that tripartition — not from the conventions of the development framework. This premise inverts the prevailing paradigm in the industry: instead of the tool imposing the architecture, the developer models the architecture according to the properties of the process being automated and uses the tool to implement it. The development framework ceases to be the origin of the architecture and becomes its instrument.

Martin [30] identifies the coupling of the architecture to the tool as one of the most harmful antipatterns in the design of software systems, arguing that an architecture that cannot be described independently of its implementation tools has lost its function as a conceptual model. The XF model addresses this coupling directly: by deriving its structure from the invariant properties of the formal process — universal and technologically neutral — it provides an architecture that can be described, understood, and communicated independently of any implementation technology.

This inversion is possible precisely because the processing–interaction–access tripartition is structurally universal: processing as the necessary core and the two channels of communication with the environment as exhaustive categories exist in every software artifact that requires them, regardless of the technology. A developer who knows the XF model can work in Spring, Angular, Flutter, or any other development framework

because they recognize in each of them the same underlying structure, expressed with a different nomenclature. The learning curve of a new technology is reduced to learning the proprietary nomenclature of the development framework — not to learning a new way of structuring software. This property — the transferability of architectural knowledge across technologies — is a central operational benefit of the model and a direct consequence of the architecture preceding the tool.

Derivation of the model The XF model starts from these two premises to establish that there exists a universal architectural structure — composed of three abstraction layers that correspond one-to-one to the three invariant stages of every formal process — and whose component taxonomy is defined independently of any implementation technology. The Access Layer models the access stage, the Business Layer models the processing stage, and the Interaction Layer models the interaction stage. This structure is not an invention of the model — it is the explicit formalization of what the software already does implicitly. The model names it, prescribes it, and makes it uniform.

The formal derivation of these layers from the invariant structural properties of formal processes is developed in §5. Their complete normative definition — guiding principles, component taxonomy, and canonical folder structure — is developed in §6.

1.3 Relationship with existing standards and reference models

The XF model does not emerge in isolation — it positions itself in continuity with an established tradition of reference models and architectural patterns that have addressed, from different perspectives and with different scopes, the problem of the structural organization of software. This clause establishes precisely the relationship of the XF model with the most relevant standards and models, distinguishing in each case what XF contributes that those models do not, and in which respects XF extends, generalizes, or complements them.

OSI model — [17] The Open Systems Interconnection model is a direct formal antecedent of the XF model and the normative reference on which its derivation is founded. The OSI model organizes network communication protocols into seven abstraction layers with clearly delimited responsibilities, where each layer provides services to the layer immediately above it and consumes the services of the layer immediately below it. Its fundamental contribution to the industry was not technical but semantic: by defining a common vocabulary to describe communication between heterogeneous systems, it enabled interoperability between technologically incompatible implementations without each implementation needing to know the internal details of the others.

The XF model extends the stratification principle of the OSI model to an orthogonal dimension: whereas OSI standardizes communication *between* artifacts, XF standardizes the *internal* structure of each artifact. The two stratifications are technically independent and conceptually complementary. The extension preserves the four formal properties of the OSI model — encapsulation of services through primitives,

strictly unidirectional dependency between layers, implementation isolation, and composition of services — applied to the new dimension. The interior of applications, a space that the OSI model explicitly leaves outside its scope and that has historically remained without an equivalent reference framework, is the domain that the XF model formalizes. The detailed technical derivation is developed in §5.5.

The analogy between XF and OSI is not only structural but also methodological. The OSI model demonstrated that defining layers with clearly delimited responsibilities and well-defined interfaces between them enables interoperability between heterogeneous implementations. The XF model applies that same principle to the interior of applications: by defining layers with clearly delimited responsibilities, it allows different teams, technologies, and development frameworks to collaborate over a common conceptual structure.

Clean Architecture — [30] Robert C. Martin's Clean Architecture establishes the dependency inversion principle as the foundation of an architecture that separates business logic from implementation details — development frameworks, databases, user interfaces. Its central contribution is the dependency rule: source-code dependencies must always point inward — toward the business policies — and never outward — toward the implementation details.

The XF model shares this principle and formalizes it in the principle of layer isolation (§6.2.2) and in the principle of precedence of the architecture over the tool (§6.2.3). XF extends the scope of Clean Architecture in two respects. The first is **universality of application**: Clean Architecture defines a structure of concentric layers — entities, use cases, adapters, frameworks — that, although agnostic in theory, has been adopted mainly in the context of object-oriented backend development. The XF model prescribes a structure that applies with equal force to frontend, backend, mobile, embedded, and any other type of artifact. The second is the **component taxonomy**: Clean Architecture does not prescribe a component nomenclature — it leaves each team to define its own within the layer structure. The XF model prescribes a closed and exhaustive taxonomy that closes the space of nominal decisions that Clean Architecture leaves to the team.

Layered architectures — [3] Layered architectures establish the principle of organizing software into horizontal layers with differentiated responsibilities, where each layer provides services to the layer above and consumes those of the layer below. Their contribution is the formalization of the concept of layered abstraction as an instrument for the separation of responsibilities in software systems.

The XF model is a specialization of layered architectures — it adopts their principle of organization into layers and instantiates it in three concrete layers derived from the invariant stages of formal processes. The fundamental difference is that layered architectures are a generic structural pattern — they do not prescribe how many layers a system must have nor what responsibilities each must have. The XF model prescribes exactly that: three layers with normative responsibilities, derived from first principles of the formal modeling of processes.

Hexagonal Architecture — [8] Alistair Cockburn's Hexagonal Architecture — also known as *Ports and Adapters* — introduces the principle of isolating the application core from its external interfaces through abstract ports that the core exposes and concrete adapters that implement them. Its contribution is the explicit formalization of the direction of dependencies between domain and infrastructure: the domain defines the contract, the adapters satisfy it.

The XF model shares with the Hexagonal Architecture the separation between business logic and the mechanisms of communication with the outside. The Business Layer fulfills the role of the application core, and the Access Layer instantiates the role of the *driven* adapters. The fundamental difference is that the Hexagonal model does not prescribe the input direction into the system — *driving adapters* — as a differentiated layer with its own identity: it concentrates it together with the rest of the adapters under a binary input/output distinction. XF explicitly separates the input direction as the Interaction Layer, deriving it from the property of formal processes whereby invocation and the presentation of results constitute a stage structurally distinct from access to external systems.

Onion Architecture — [38] Jeffrey Palermo's Onion Architecture organizes the system into concentric rings where dependencies always point toward the center — the domain — and never toward the outer rings. Its contribution is the topological visualization of the dependency rule that Clean Architecture also articulates and that, in its linear stratification, the XF model articulates as well.

The XF model and the Onion Architecture share the direction of dependencies and the centrality of the domain, but they differ in the form of the structure and in the prescription of types. The Onion does not prescribe a closed component taxonomy or a canonical nomenclature, nor does it define how the rings are materialized in a verifiable folder structure. The XF model prescribes both, while preserving the direction of dependencies that the Onion captures visually.

Domain-Driven Design — [11] Domain-Driven Design introduces a rich terminology for modeling complex business domains — Aggregates, Repositories, Domain Services, Application Services, Bounded Contexts — and establishes the concept of Ubiquitous Language as a necessary condition for precise communication between developers and domain experts. Its semantic contribution to the modeling of the business domain is considerable and has influenced the way the industry thinks about the separation between domain logic and infrastructure.

The relationship between DDD and XF is one of complementarity, not competition. DDD operates mainly in the space of the Business Layer — its concepts of `Repository`, Domain `Service`, and Application `Service` have direct correspondence with the logical access and business components of the model. The compatibility materializes specifically within `/business/logic/instance`, the region of the artifact where the model recognizes the functional domain as a legitimate subdivision criterion (§7.2.2). The organization by subdomains or by bounded contexts that DDD prescribes finds its natural place there; the Access and Interaction layers, by contrast, are organized by technical criteria orthogonal to the domain — protocol and input type respectively —, so that the replication of technical resolution that a

domain-based subdivision across all layers would introduce is avoided. Additionally, DDD does not prescribe how to organize the access and interaction layers, nor does it provide a component taxonomy applicable to frontend, mobile, or embedded artifacts. XF extends the scope of semantic standardization beyond the business domain — to the complete set of the artifact — and does so in a technologically agnostic way, without the domain-complexity assumptions that limit the adoption of DDD to enterprise systems of a certain scale.

Design patterns — [14] The Gang of Four design patterns describe recurring solutions to local design problems within components or sets of related components. Their contribution is a shared vocabulary for describing design solutions at the component level — `Factory`, `Observer`, `Strategy`, `Repository`.

The XF model is not a design pattern in the sense of Gamma et al. — it does not describe a solution to a local design problem. It is a reference model that operates at a higher level of abstraction: it prescribes how the artifact as a whole is organized, not how a specific design problem within it is solved. Design patterns are compatible with XF — they can be applied within the logical, generalization, or utility components of any layer — but they are orthogonal to it: their application does not affect conformance with the model.

The differential contribution of the XF model None of the models and standards described in this clause simultaneously addresses the three problems that the XF model solves. The OSI model standardized communication between systems but not the interior of applications. Layered architectures, the Hexagonal, the Onion, and Clean Architecture standardized the organization into layers and the direction of dependencies, but not the component taxonomy or the nomenclature. DDD standardized the semantics of the business domain, but not the organization of the artifact as a whole nor its applicability to heterogeneous technologies. Design patterns standardized local solutions but not the global structure of the artifact.

The XF model addresses the three problems simultaneously and in an integrated way: it prescribes a layer structure derived from first principles, a closed and exhaustive component taxonomy, and a technologically agnostic canonical nomenclature. This integration is what allows the model to solve the problem that none of its predecessors solved separately: to make visible and uniform the structural convergence that software already implements implicitly, regardless of the technology used.

1.4 Structure of the document

This document is organized as follows.

Clause 2 (Normative references) establishes the external normative references on which the model is founded — standards and documents whose application is indispensable for the correct interpretation of the provisions of this document.

Clause 3 (Terms and abbreviations) defines the terms and abbreviations used in the document with a specific meaning in the context of the XF model. The terms defined here constitute the normative vocabulary of the model and shall be interpreted with the precision assigned to them in this clause.

Clause 4 (Normative conventions) establishes the normative conventions used in the drafting of the document — the use of the prescriptive auxiliary verbs, the distinction between mandatory requirements and recommendations, and the interpretation criteria applicable in case of ambiguity.

Clause 5 (Theoretical foundations) sets out the theoretical foundations of the model on an informative basis. It establishes the reasoning that justifies the normative decisions of the following clauses — the principle of isomorphism between process and artifact, the derivation of the three invariant stages of every formal process, the typing of components as an instrument of comprehension, and the limitations of proprietary typing — and formally derives the three-layer model from the invariant structural properties of formal processes.

Clause 6 (The XF Architectural Model) defines the model in its normative terms: it presents its general overview and establishes the four guiding principles — technology agnosticism, layer isolation, precedence of the architecture over the tool, and closed and exhaustive typing.

Clause 7 (Component taxonomy) organizes every artifact in the two-dimensional classification matrix. It describes the matrix structure of layers and types; the *abstraction dimension*, which defines the three layers — Access, Business, and Interaction — with their responsibility, their boundaries, and the process model of their internal logic; the *functional dimension*, which exhaustively defines the five types of components recognized by the model; and the canonical folder structure that materializes the matrix in the file system.

Clause 8 (Instantiation of the architecture) specifies the mechanism for instantiating the architecture. It describes the execution start point of the artifact, the complete lifecycle with its four phases — instantiation, initialization, execution, and termination —, the initial instantiation of the logical components by the injection component (which establishes the initial state σ_0 of each logical component by construction of the singleton, not by invocation) and the two invocable operations — initialization and termination —, the initialization order derived from the dependency graph between components, and the `XF` element as the centralized control point of initialization.

Clause 9 (Data flows) describes the treatment of data in the XF architecture. It establishes that Transfers constitute a single, homogeneous domain whose transmission is carried out through the communication channel of the operations, materialized in the native syntactic constructs of the implementation language, without the model defining transmission mechanisms of its own. It formalizes the types of data structures that the model recognizes, establishes the difference with the OSI model regarding the treatment of data between layers, develops the principle of data-structure unification and the structural transformation rule, and prescribes the directionality restrictions of transfer components between layers.

Clause 10 (Compatibility and interoperability) establishes the compatibility and interoperability rules of the model. It defines the principle of bidirectional backward compatibility, formalizes the two levels of integration between artifacts — application level and artifact level —, develops the integration patterns for the four possible combinations between XF and traditional artifacts, and describes the ecosystem of XF libraries as a natural consequence of the forms of integration.

Clause 11 (Conformance of the XF model) defines the conformance model of the XF model. It introduces the concepts specific to conformance — verifiable element, violation, and conformance level —, establishes the five discrete conformance levels, presents the exhaustive catalog of verifiable rules organized in nine thematic groups, and defines the algorithm for determining the conformance level with its formal expression.

Clause 12 (Ontology of the architecture) presents the ontology of the XF architecture. It includes the conceptual map of the model and the dictionary of terms with the precise definition of each concept of the model — both the terms specific to XF and those terms in general use in software engineering to which the model gives an exact normative meaning.

Clause 13 (Discussion) contrasts the model with the state of the art, analyzes its limitations, and develops its implications for automatic code generation and *Specification-Driven Development*. **Clause 14** (Conclusions) synthesizes the contribution of the model, states the hypotheses pending empirical validation, and establishes the lines of future work.

The document is completed by six annexes, all informative.

Annex A (Implementation examples: Access Layer) illustrates, through real code, how the components prescribed for the Access Layer are materialized in a concrete technology.

Annex B (Implementation examples: Business Layer) illustrates the equivalent materialization for the Business Layer.

Annex C (Implementation examples: Interaction Layer) illustrates the equivalent materialization for the Interaction Layer, distinguishing between service components and view components.

Annex D (Projection of external vocabularies onto the XF model) formalizes the correspondence between the vocabulary of the model and that of the prior reference architectures (Clean Architecture, Domain-Driven Design, Hexagonal, and Onion) and of the most widely adopted development frameworks.

Annex E (Terminological equivalences with other vocabularies) presents the table of equivalences between the canonical terms of the XF model and the analogous terms of the state of the art in software engineering.

Annex F (Revision history) records the modifications applied to successive editions of the document.

2 Normative references

The following documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the most recent edition of the referenced document applies. The references are listed by issuing organization (ISO/IEC first, ISO/IEC/IEEE next, others last) and, within each block, by ascending number, in accordance with the ISO/IEC Directives Part 2 §10.2 convention.

ISO/IEC 7498-1:1994 [17] — *Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model*. International Organization for Standardization / International Electrotechnical Commission.

Provides the Open Systems Interconnection reference model on which the XF model is conceptually founded. XF extends the OSI stratification principle to an orthogonal dimension — the internal architecture of the artifact, not communication between artifacts — preserving its formal properties: encapsulation of services, unidirectional dependency, implementation isolation, and composition. The concepts of abstraction layer, layer responsibility, and dependencies between layers are adopted in the normative sense assigned to them by this standard.

ISO/IEC Directives, Part 2:2021 [21] — *Principles and rules for the structure and drafting of ISO and IEC documents* (9th ed.). International Organization for Standardization / International Electrotechnical Commission.

Establishes the normative drafting conventions applied in this document — the use of the prescriptive auxiliary verbs **shall / shall not**, **should / should not**, **may / cannot** and **it is recommended / it is not recommended** (§4.1), the distinction between mandatory requirements and recommendations, and the structure and presentation criteria for normative documents.

ISO/IEC/IEEE 12207:2017 [19] — *Systems and software engineering — Software life cycle processes*. International Organization for Standardization / International Electrotechnical Commission / Institute of Electrical and Electronics Engineers.

Provides the normative framework of software life cycle processes against which the XF model positions itself as an instrument for the architectural organization of the artifact, without replacing or modifying the life cycle process provisions. The instantiation, initialization, execution, and termination of the artifact developed in §8 are understood as the run-time manifestation of the life cycle activities that this standard describes.

ISO/IEC/IEEE 24765:2017 [20] — *Systems and software engineering — Vocabulary*. International Organization for Standardization / International Electrotechnical Commission / Institute of Electrical and Electronics Engineers.

Establishes the standardized vocabulary of software engineering on which the terminology specific to the XF model is built. The basic terms — *software artifact*, *component*, *module*, *interface* — are interpreted in the sense assigned by this standard unless explicitly restricted by the XF model, as developed in §3.1.

ISO/IEC/IEEE 42010:2022 [22] — *Software, systems and enterprise — Architecture description*. International Organization for Standardization / International Electrotechnical Commission / Institute of Electrical and Electronics Engineers.

Provides the conceptual framework for the description of software architectures — including the concepts of architectural reference model, viewpoints, views, concerns, and stakeholders — that the XF model adopts in its positioning as a reference model in the sense of this standard, as developed in §1.3 and §13.3.

OMG BPMN 2.0.2:2014 [35] — *Business Process Model and Notation (BPMN), Version 2.0.2*. Object Management Group. The BPMN 2.0 specification was adopted

by ISO/IEC as **ISO/IEC 19510:2013** [18] (*Information technology — Object Management Group Business Process Model and Notation*); the OMG's version 2.0.2 (2014) incorporates subsequent editorial corrections.

Provides the formal definition of formal process and its structural properties — activities, inputs, outputs, and transition rules — that the XF model adopts in §5.1 and §5.2 as the basis for the derivation of the three invariant stages of every formal process and the corresponding stratification into three abstraction layers.

3 Terms and abbreviations

3.1 Terms

For the purposes of this document, the following terms and definitions apply. For the complete definitions and the conceptual relationships between terms, see §12.2.

3.1.1 Artifact — self-contained piece of code that formalizes a process, defines an area of responsibility, and is subject to version control. In the XF model, every artifact is structured into exactly three abstraction layers — Access, Business, and Interaction — and always executes in a single execution space. *See §6.1.*

Note 1 to entry: The term *Artifact* is a specialization of the homonymous concept defined in OMG UML 2.5.1 [36] and standardized in ISO/IEC/IEEE 24765:2017 [20], restricted to the subset of **executable and self-contained** elements that automate a complete formal process. The operational difference of the XF Artifact with respect to the established vocabularies of the discipline lies in the simultaneous prescription of three conditions that none of them requires jointly: self-containment, exactly three abstraction layers derived from the invariant stages of the formal process, and a single execution space. The detailed comparison with the analogous terms of the literature is developed in Annex E.

3.1.2 Application — executable unit resulting from loading one or more artifacts into a single execution space, typically a single operating-system process. The term *application* designates the executable functional unit; the term *artifact* designates the self-contained structural unit that the application loads and composes. *See §6.1 and §10.2.*

3.1.3 Abstraction layer — horizontal division of the artifact that groups the components implementing the same stage of the formal process that the artifact automates. The XF model defines exactly three layers — Access, Business, and Interaction. *See §6.1.*

3.1.4 Component — atomic unit of classification of the XF model with its own identity, a unique name within the artifact, and a classification defined in the matrix of layers and types. *See §7.1.*

3.1.5 Functional type — classification of a component according to the function it performs within its layer. The XF model defines exactly five functional types — Logical, Generalization, Injection, Utility, and Transfer. *See §7.3.*

3.1.6 Effective logic — minimal set of operations that justify the existence of a logical component in the artifact, which cannot be abstracted into any other type of

component without losing the specificity of the concept that the component models. *See §7.3.1.*

3.1.7 Contextual logic — auxiliary logic that supports the effective logic of a logical component. It is formally subdivided into three species according to the type of component that hosts it: **structural logic** in generalization components, **utility logic** in utility components, and **transfer logic** in transfer components — the structural representation of the concept and, optionally, self-contained operations for transformation between equivalent representations, without behavioral logic over the domain. The subdivision is exhaustive: all contextual logic falls into exactly one of these three species. *See §7.3.1.*

3.1.8 State — set of data of a logical component that have direct correspondence with information of the business domain and that determine the observable behavior of the component in response to the operations that invoke it. Only logical components may maintain state. *See §7.3.1.*

3.1.9 Operation — subset of the effective logic of a logical component that executes atomically in response to an event or invocation, producing a transformation of the component's state, a result for the invoker, or both. *See §7.3.1.*

3.1.10 Condition — statement of an operation that evaluates the component's state or the input parameters before executing the effective logic. It constitutes the execution precondition of the operation. *See §7.3.1.*

3.1.11 *Singleton Gathering* — design pattern implemented by injection components that aggregates at a single point the unique instances of the logical components of a layer, guaranteeing their uniqueness and providing a named, stable point of access. The term designates specifically the concurrence of the uniqueness (*singleton*) and aggregation (*gathering*) properties that characterize the injection component. *See §7.3.3.*

3.1.12 Instantiation — creation of an object in memory through the constructor of a component. In the XF model, the instantiation of **logical components** is an operation exclusive to the injection component of the corresponding layer; the instantiation of components of other types — in particular transfers — is not subject to that restriction. It is distinguished from initialization. *See §7.3.3.*

3.1.13 Initialization — execution of the start-up logic of a logical component that prepares it to process the operations for which it was designed. It is distinguished from instantiation. *See §7.3.3.*

3.1.14 Termination — execution of the shutdown logic of a logical component that releases the resources acquired during initialization. *See §7.3.3.*

3.1.15 Primitive type — primitive data type in the sense of ISO/IEC/IEEE 24765:2017 [20]: a type from which other types are built, present as a base type in most programming languages. In the XF model, primitive types enter the artifact through the Access Layer — the boundary with the Presentation Layer of the OSI model (Level 6) —, are prior to any semantic transformation, and constitute the raw material on which all layers operate. Exchange formats of the operating system or the execution environment whose

structure is determined by the platform are treated as primitive types for the purposes of the model. *See §7.3.4.*

3.1.16 Structure — grouping of primitive types or other structures that models a concept of the artifact's domain. Synonym of transfer component in its structural sense. It unifies the terms `DTO`, `Entity`, `Model`, `POJO`, `Record`, `VO`, and `Schema` under a single concept. *See §9.2.*

3.1.17 Protocol — set of rules and formats that govern the communication between two software artifacts over a communication channel. The management of protocols is a structural responsibility of the two boundary layers of the artifact, in complementary roles: the **Interaction Layer** *defines and exposes* the protocol through which external consumers invoke the artifact; the **Access Layer** *consumes* the protocols published by the external systems to which the artifact delegates part of its execution. It includes both network protocols and local-access protocols. *See §7.2.1 and §7.2.3.*

3.1.18 Entry point — mechanism declared by the Interaction Layer through which an external consumer — human or systemic — invokes the operations that the artifact exposes. Each logical component of the Interaction Layer declares exactly one entry point. It shall not be confused with the execution start point of the artifact, which is internal to the artifact and is not invocable by consumers. *See §7.2.3.*

3.1.19 Execution start point — fragment of code that the execution environment invokes to start the artifact's process — `main`, `App.main()`, `index.ts`, or equivalent depending on the language. It is the first code of the artifact that executes and receives control from the environment; its sole responsibility is to delegate the XF initialization and, at the end, the termination. It is not a component classifiable in the matrix of layers and types and resides outside the root of the artifact. It shall not be confused with the entry point (§3.1.18), which is invocable by external consumers through the Interaction Layer. *See §8.1.*

3.1.20 Dependency — any relationship between two components A and B in which A needs to know the definition or behavior of B in order to be compiled, instantiated, or executed. Every dependency is subject to the principle of layer isolation. *See §6.2.2.*

3.1.21 Execution space — computational context in which the components of an artifact operate. Determining criterion for classifying an integration between artifacts as application-level or artifact-level integration. *See §10.2.*

3.1.22 Verifiable element — any element of the artifact against which a conformance rule can be evaluated. The XF model recognizes two disjoint types of verifiable element: the **code element** (class, module, or file of cohesive functions; it comprises the component — classified in the matrix of layers and types — and the `XF` start-point element) and the **structural element** (folder of the canonical structure). *See §11.1.2.*

3.1.23 Conformance rule — verifiable condition that operationalizes a normative provision of the XF model into a form evaluable over a verifiable element of the artifact (§3.1.22). It has a canonical identifier in kebab-case format and is characterized by its **verifiability** — structural or semantic, according to whether its evaluation can

be performed deterministically from observable syntactic properties of the source code or requires semantic interpretation of the functional responsibility. The breach of any rule of the catalog produces a violation that affects the conformance level of the artifact. The closed and exhaustive catalog is developed in §11.3. *See §11.1 and §11.3.*

3.1.24 Violation — determination that a verifiable element (§3.1.22) of the artifact does not satisfy the condition prescribed by a conformance rule (§3.1.23) of the catalog. Every violation is associated with exactly one rule and exactly one verifiable element. *See §11.1.2.*

3.1.25 Code totality — condition that an artifact satisfies when every component it contains has a defined position in the matrix of layers and types. The `XF` element, when declared, is not a component and falls outside the totality condition by construction. Necessary condition for reaching conformance levels 2, 3, or 4. *See §11.1.3.*

3.1.26 Conformance level — discrete property of an artifact that determines the extent to which it satisfies the normative provisions of the XF model. It takes values in the set {0, 1, 2, 3, 4}. *See §11.2.*

3.1.27 Compatibility — property of the XF model that prescribes how an XF artifact integrates components from other artifacts: by recognizing the functional responsibility of the integrated component, assigning it a cell of its own matrix of layers and types, and applying the mechanism prescribed for that cell and that type of component. It is bidirectional and non-invasive: no integrated artifact shall be modified in order to make its integration possible. *See §10.1.*

3.1.28 Structural logic — species of contextual logic (§3.1.7) encapsulated in a generalization component, which abstracts a structural behavior pattern common to two or more logical components of the **same layer** and transmits it to them by inheritance. It is **parametric with respect to the domain**: its logic applies regardless of the concrete concept that each logical component inheriting it models. The presence in a generalization component of logic tied to a concrete concept of the artifact's domain indicates that it has ceased to be structural logic and shall be moved to the logical component to which it belongs. *See §7.3.2.*

3.1.29 Structural verifiability — property of a rule of the conformance catalog (§3.1.23) whose evaluation can be performed deterministically from observable syntactic properties of the artifact's source code — file names, location in the folder structure, import declarations between modules, suffixes and prefixes of identifiers, dependency graph between components. Rules with structural verifiability are algorithmically evaluable by a static analysis tool. *See §11.1.4.*

3.1.30 Semantic verifiability — property of a rule of the conformance catalog (§3.1.23) whose evaluation requires understanding the functional responsibility of the code and is not decidable exclusively by structural analysis. Its evaluation demands analysis beyond that of the static analysis tool — human architectural review, data-flow analysis, or analysis of the code's intent. The existence of rules with semantic verifiability determines that the application of the conformance function is not fully automatable: some violations cannot be detected by static analysis alone and require human evaluation. *See §11.1.4.*

3.1.31 Logical component — functional type that implements the effective logic of the layer to which it belongs. It is the main functional unit of each layer — the only type that may maintain state in the formal sense of the model and the only one whose instances are managed by the injection component. Canonical nomenclature: suffix `Repository` in Access, `Business` in Business, `Service` or `View` in Interaction. *See §7.3.1.*

3.1.32 Generalization component — functional type that abstracts structural behavior common to two or more logical components of the **same layer** and transmits it by inheritance. It does not maintain domain state and is not directly accessible through the injection component. *See §7.3.2.*

3.1.33 Injection component — singleton functional type that centralizes the lifecycle and the access to the logical components of its layer. Exactly one per layer: `R` in Access, `B` in Business, `A` in Interaction. *See §7.3.3.*

3.1.34 Utility component — functional type that provides pure, stateless auxiliary operations of scope local to the layer that hosts it. Canonical nomenclature: suffix `Utils`. *See §7.3.4.*

3.1.35 Transfer component — functional type that models the data structures that circulate between components of the artifact. It does not model business processes; it may declare self-contained operations over its own data. Exceptions are a subtype of transfer component with their own nomenclature: suffix `Exception`. *See §7.3.5.*

3.1.36 Statement — elementary instruction of the body of an operation: it queries or transforms the component's state, evaluates a condition, or delegates to another component through the injection component of the corresponding layer. The body of every operation is an ordered sequence of statements. *See §7.3.1.*

3.2 Abbreviations

For the purposes of this document, the following abbreviations apply. The abbreviations are textual acronyms replaceable by their full form when the term is invoked in prose; the mathematical symbols of the formal apparatus of the model (sets, functions, relations) are introduced in the clauses where they are used. When the literal expansion of the acronym alone is not sufficient to identify the referent unambiguously, the full form includes, in parentheses, a disambiguating context — source standard, model author, or scope of application.

XF model acronyms

XF	Cross-Framework (XF Architectural Model)
CFAM	Cross-Framework Architectural Model (international public name of the XF model)

Standards, norms, and software-engineering vocabularies

OMG	Object Management Group
UML	Unified Modeling Language
BPMN	Business Process Model and Notation

OSI Open Systems Interconnection (ISO/IEC 7498-1 model)

Architectural paradigms, patterns, and methodologies

IPO Input-Process-Output (decomposition model of structured engineering)
CSP Communicating Sequential Processes (Hoare's calculus of communicating processes)
DDD Domain-Driven Design

Implementation patterns and object types

DTO Data Transfer Object
DAO Data Access Object
BLoC Business Logic Component (Flutter state-management pattern)

Ecosystem technologies and protocols

API Application Programming Interface
SDK Software Development Kit
IDE Integrated Development Environment
JPA Java Persistence API
REST Representational State Transfer
gRPC Google Remote Procedure Call
HTTP HyperText Transfer Protocol
URL Uniform Resource Locator
SQL Structured Query Language
JSON JavaScript Object Notation
IoT Internet of Things

4 Normative conventions and scope

4.1 Prescriptive auxiliary verbs

This document uses the prescriptive auxiliary verbs defined in the ISO/IEC Directives, Part 2:2021 standard [21] with the precise meaning established below. The correct interpretation of these verbs is a necessary condition for the correct application of the model's provisions.

shall / shall not The verb **shall** indicates a requirement — a provision whose fulfillment is mandatory for conformance with the model. Its breach constitutes a violation that affects the conformance level of the artifact according to the conditions defined in §11.2.

The verb **shall not** indicates a prohibition — a provision whose fulfillment is equally mandatory for conformance. Its breach likewise constitutes a violation.

The rules of the catalog in §11.3 correspond to provisions expressed with **shall** or **shall not** in the normative body of the document.

should / should not The verb **should** indicates a normative recommendation — a provision whose fulfillment is preferred but not mandatory for conformance with the model. Its breach does not constitute a violation and does not affect the conformance level of the artifact, but it indicates that the artifact departs from the preferred practice that the model prescribes. The verb **should not** indicates a negative recommendation with the same status.

it is recommended / it is not recommended This document additionally uses the forms **it is recommended** and **it is not recommended** as normative equivalents of **should** and **should not**, respectively. This choice does not extend the catalog of six canonical categories of ISO/IEC Directives Part 2:2021 [21]: it corresponds to the equivalent forms for *should / should not* explicitly admitted in Annex E of the standard itself (*"it is recommended that" / "it is not recommended that"*). They are used for natural phrasing in constructions where the impersonal form reads more fluently than the modal, without altering the normative status or the conformance impact of the *recommendation* category to which they belong.

may / cannot The verb **may** indicates an action or decision that is explicitly authorized by the model but that is neither mandatory nor recommended. Its non-application does not constitute a violation of any kind.

The verb **cannot** indicates a **structural impossibility**: a property that the model cannot express or realize by construction of its own formal apparatus. It is not a normative provision — it generates no violation — but the statement of a limit of the model. For prohibitions that do constitute a violation, **shall not** is always used.

The following table summarizes the normative status of each auxiliary verb:

Table 1. Prescriptive auxiliary verbs and their impact on conformance.

Auxiliary verb	Normative status	Conformance impact
shall	Mandatory requirement	Violation *affects the conformance level*
shall not	Mandatory prohibition	Violation *affects the conformance level*
should	Normative recommendation	No impact
should not	Negative normative recommendation	No impact
it is recommended	Equivalent form of *should* (Annex E)	No impact
it is not recommended	Equivalent form of *should not* (Annex E)	No impact
may	Authorized action, not mandatory	No impact
cannot	Structural impossibility	No impact

4.2 Notation conventions

This document uses the following notation conventions in the normative body and in the annexes.

Code notation The names of components, operations, folders, and access patterns are represented in monospaced typography — `UserBusiness`, `R.init()`, `/src/repository/logic`, `B.<business>.<operation>()`. This notation indicates that the term is to be interpreted as a code identifier, not as a natural-language term.

Mathematical notation The formal expressions of the model use the standard mathematical notation of set theory and functions, with the following typographic conventions by family:

- **Sets of components of the artifact** (blackboard bold): $\mathbb{C}$ for the finite set of components; its subsets are denoted with a subscript, such as $\mathbb{C}_T$ for the transfer components.
- **Injection components** (uppercase Latin letters): R, B, A for the three layers according to the context.
- **Complete artifact** (fraktur): $\mathfrak{A}$.
- **Structural and conformance functions** (Greek letters): ϕ for the classification function, which assigns to each component its cell in the $L \times T$ matrix ($\phi\colon \mathbb{C} \to L \times T$); Λ for the conformance-level function, whose application to an artifact yields $\Lambda(\mathfrak{A}) \in \{0, 1, 2, 3, 4\}$; and λ for its result, the conformance level reached by the artifact.
- **State of a logical component** (Greek letter with subscript): σ_0 for the initial state, subsequent to the construction of the component and prior to the invocation of `init()`.
- **Set-theoretic and function notation**: membership $\in$, union $\cup$, Cartesian product $\times$ ($L \times T$ matrix), and the function arrow $\to$. Quantifications and logical connectives are expressed in prose.

Cross-reference notation The document uses the term *clause* in a unified way to designate any hierarchical subdivision of the normative body and the annexes, regardless of its level of depth. The alternative designations *section*, *subsection*, and *subclause* are not used; the context and the subdivision number express the depth when relevant. References to clauses of the document are expressed by means of the § symbol followed by the clause number — §6.2.2, §11.3.1. References to annexes are expressed by means of the annex letter followed by the internal clause number.

Figures in the document The figures of the document are included as vector diagrams with a descriptive caption and a unique numeric label. Every figure is referenceable from the body of the text by means of the cross-reference mechanism described in the preceding paragraph. The complete list appears in the *List of Figures* at the beginning of the document.

Tables in the document The tables of the document are inserted non-floating at the argumentative place where they appear, with page splitting when they exceed the available space, so that the argumentative thread of the text is preserved. Each table carries a descriptive caption, optionally accompanied by a *short caption* in brackets for the *List of Tables* entry, and a unique label that makes it referenceable by means of the cross-reference mechanism. The complete list appears in the *List of Tables* at the beginning of the document.

Code listings The code fragments of the normative body are presented in delimited blocks with a descriptive caption and a unique referenceable label. They are not expressed in any concrete programming language: they use a neutral object-oriented pseudocode — with constructions common to the mainstream languages — chosen for readability and in coherence with the technology agnosticism of the model; they illustrate the structure and the relationships between components, they do not prescribe a syntax or an implementation language. The complete list appears in the *List of Listings* at the beginning of the document.

4.3 Distinction between normative and informative content

This document explicitly distinguishes between normative content and informative content.

The **normative content** establishes requirements, prohibitions, recommendations, and permissions that determine the conformance of an artifact with the XF model. It constitutes the prescriptive body of the document. Clauses §2, §3, §4, and §6 to §12 are normative.

The **informative content** provides context, justification, examples, and implementation guidance that facilitate the understanding and application of the normative content, but that do not establish additional requirements. §1, §5, Annexes A to F, §13 (Discussion), and §14 (Conclusions) are informative.

When the informative content appears to contradict the normative content, the normative content prevails.

4.4 Interpretation criteria

In case of ambiguity in the interpretation of any normative provision of this document, the following criteria apply in order of priority.

The first criterion is **consistency with the guiding principles**: between two possible interpretations, the one that best satisfies the four guiding principles of the model established in §6.2 — technology agnosticism, layer isolation, precedence of the architecture over the tool, and closed and exhaustive typing — prevails.

The second criterion is **verifiability**: between two interpretations equally consistent with the principles, the one that produces a deterministically verifiable condition prevails. The XF model aspires for all its provisions to be algorithmically verifiable — an interpretation that produces a non-verifiable condition is indicative of ambiguity in the provision that shall be resolved in a future version of the document.

The third criterion is **reference to the dictionary of terms**: when the ambiguity derives from the meaning of a technical term, the definition in §12 is the normative reference. If the term does not appear in §12, its meaning in the ISO/IEC/IEEE 24765:2017 standard [20] applies; failing that, its meaning in ISO/IEC/IEEE 42010:2022 [22].

4.5 Scope of the model

The XF model normalizes the **code architecture** of the artifact. Its domain of application begins at the source code of the artifact and ends at its external boundaries. The elements of the project that surround the source code — and that are necessary to build, package, and deploy the artifact — fall outside the direct scope of the model.

The following are explicitly outside the scope of the model:

- The project structure outside the source code of the artifact: the management of external dependencies, the packaging descriptors, the resources associated with deployment, the configuration parameters of the execution environment, and the files specific to the build tools are the responsibility of the development framework, not of the XF model.
- The management of the development cycle: version control, continuous integration, deployment automation, and the quality-assurance processes external to the code are instruments distinct from the model, and their standardization corresponds to other normative and methodological frameworks.
- The code generated automatically by external tools that are unaware of the XF model: the internal organization of that code depends on the tool that produces it. For the purposes of XF integration and conformance, that code is functionality not normalized in XF, and its treatment is governed by the general provisions of §10.2.1 — it is reformulated into a component of the artifact's own that encapsulates it, or it is kept outside the scope of conformance by placing it outside the root of the artifact according to the provision of §7.4.

These elements may coexist with an XF artifact, and their correct organization is relevant to the quality of the project as a whole, but their standardization is the responsibility of instruments distinct from the model. The XF model describes the organization of the code that is developed, not the process by which it is developed nor the environment in which it executes.

5 Theoretical foundations

The internal clauses are informative. Their purpose is to establish the theoretical reasoning from which the XF Architecture Model defined in the following clauses is derived. The content of this clause does not constitute a normative requirement.

This clause sets out the theoretical foundations from which the XF Architecture Model is derived. Its purpose is to establish the reasoning that justifies the normative decisions adopted in the following clauses, so that the reader may understand not only what the model prescribes, but why it prescribes it.

The argument unfolds across five internal clauses with a chained structure. Clauses §5.1 and §5.2 establish the invariant structural properties of every formal process, which constitute the foundation from which the three-layer architecture is derived. Clauses §5.3 and §5.4 analyze the typing of components as an instrument for managing complexity and document the limitations of the proprietary typing model prevailing in the industry. Clause §5.5 synthesizes both lines of argument in the formal derivation

of the three-layer model and establishes the formal properties that every XF artifact shall satisfy.

5.1 Software as the automation of formal processes

Software engineering has as its object the construction of systems that execute, autonomously and reproducibly, processes that pre-exist in the domain they model. This characterization is not new: the 1968 NATO conference on software engineering [33] marks the institutional recognition of the discipline and the formulation of the so-called *software crisis*, whose resolution requires the systematic and reproducible production of reliable executable systems from process specifications. What has changed since then is the scale and heterogeneity of the processes that software models, not their fundamental nature.

A software system does not create processes; it formalizes them. This distinction has a first-order architectural implication: if the function of software is to represent a pre-existing process, the internal structure of the artifact is isomorphic to the structure of the process it models. The architecture of the system is therefore not a discretionary decision of the developer or a convention imposed by the development framework — it is a property derived from the nature of the automated process.

This principle of isomorphism between process and artifact connects directly with the theory of the formal modeling of processes. The BPMN 2.0 standard [35, 18] characterizes a process — in its business domain — as a sequence of activities with precise semantics, well-defined inputs and outputs, and verifiable transition rules, a characterization that the XF model generalizes to the concept of formal process. From this perspective, any process that can be described with sufficient precision is, in principle, automatable by software — from order management in an organization to the control of actuators in an embedded system, passing through the synchronization of data between distributed services.

The practical implication is direct: the analysis of the structure of the process must precede the design of the artifact's architecture. Brooks [7] identified accidental complexity — that introduced by the tools and the development processes, not by the problem itself — as one of the main causes of failures in software projects. An architecture that does not derive from the structure of the process it models introduces precisely that kind of complexity: structural inconsistencies that do not originate in the business domain, but in arbitrary implementation decisions that accumulate as technical debt and manifest during the maintenance and evolution of the system.

The XF model starts from this premise to establish that there exists a set of structural properties common to every formal process, regardless of its domain, complexity, or implementation technology. The identification and formalization of these properties is the object of the following clauses.

5.2 Invariant stages of a formal process

If every software artifact models a formal process, and if the architecture of the artifact is isomorphic to the structure of that process, then the identification of the

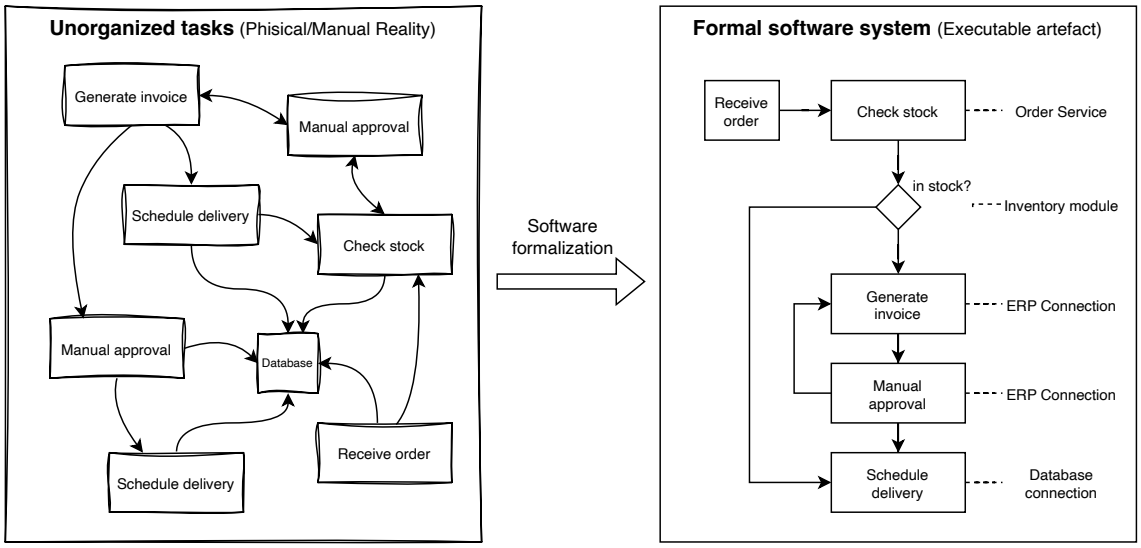

Figure 1. Software as the automation of formal processes. Every software artifact executes an ordered sequence of operations that transforms an initial state into a final state by applying deterministic rules; on that correspondence rests the possibility of deriving a universal architecture from structural properties of the modeled process.

structural properties common to every formal process constitutes, in the XF model's proposal, the step prior to the derivation of any software architecture model with a universal vocation.

The study of formal processes in the field of computation theory and systems engineering reveals a notable structural property. Regardless of the domain, the complexity, or the nature of the process considered, every formal process admits a canonical decomposition into three stages with distinct formal status — **processing**, **interaction**, and **access** — whose individual development and status formalization are developed in the following subclauses. The XF model posits this decomposition not as a design convention or a methodological choice, but as a structural property derivable from the analysis of the minimal conditions that any formal process must satisfy in order to be executable.

Parnas [39], in his seminal work on the modular decomposition of software systems, observed that the structure of a system must reflect the design decisions most likely to change, and that these decisions are determined by the nature of the process the system implements. This observation anticipates the isomorphism principle described in the preceding clause and points toward the existence of an underlying process structure that every system must recognize and represent.

The three stages that make up every formal process are the following:

- **Interaction.** Every formal process requires a mechanism by which it receives the order to be executed and verifies that the preconditions necessary for its start are satisfied. This stage is not part of the internal logic of the process — it is the interface between the process and its environment, the point at which the process becomes visible and accessible to the external agents that invoke it. Without this stage, the process cannot be started in a controlled manner, nor can it be guaranteed that the execution conditions are adequate. In terms

of Meyer's theory of contracts [31], this stage corresponds to the verification of the process's preconditions — those conditions that must be true at the moment of invocation for the process to execute correctly.

- **Processing.** Once the execution preconditions are satisfied, the process applies, in an ordered and deterministic manner, the set of operations that define it to transform its inputs into the expected result. This is the stage in which the process's own logic resides — the algorithms, the business rules, and the data transformations that constitute the value the process provides. Turing [43] demonstrated that any computable function can be decomposed into a finite sequence of elementary operations; processing is precisely that sequence, instantiated for the specific domain of the process considered.
- **Access.** Every formal process that operates in a non-trivial environment requires information or capabilities that reside outside its own boundaries. The access stage is the mechanism by which the process delegates part of its execution to other processes or external systems, with which it communicates through a defined interaction protocol. This stage is the exact reflection of the interaction stage seen from the perspective of the delegated process: what is access for the invoking process is interaction for the invoked process. This symmetry is not coincidental — it is a direct consequence of the compositionality of formal processes, the property that allows complex processes to be built through the composition of simpler processes [16].

These three stages have a structural status that should be made precise. **Processing** is the structurally necessary stage of every formal process: without logic that transforms inputs into results there is no process. The **interaction** and **access** stages are structurally optional depending on the context of the process — there are internally triggered processes that receive no external invocation, and there are purely computational processes that require no collaboration with external systems — but they are **exhaustive and mutually exclusive** as categories of communication with the environment. Every process that communicates with its environment does so through exactly one of two channels — receiving invocation or invoking the outside — with no third structural type of communication existing. This property — necessary processing, exhaustive communication in its two forms — is the formal basis of the tripartition and, when a communication stage is present, its semantics with respect to processing is invariant: interaction causally precedes processing, and access occurs as delegation during or after it.

The structural-exhaustiveness property of the two communication channels is supported by at least four independent formal traditions that reach the same conclusion. The BPMN 2.0 standard [35] structures each process into *Activities* — processing — *catch Events* — incoming interaction — and *throw Events* — outgoing access —, with *Activities* being the only structurally necessary element. The calculus of communicating sequential processes CSP [16], together with its mobile extension — Milner's π-calculus [32] — demonstrates that all concurrent computation can be expressed with three primitives: internal action (τ), reception over a channel ($c?x$), and emission over a channel ($c!v$). The *Actor Model* [1] defines every actor by the processing of the message in progress, the optional sending of messages to other actors — equivalent to access — and the optional reception in its mailbox — equivalent to interaction —. The *Input-Process-Output* (IPO) model of structured engineering

[47] formalizes the classical decomposition with a necessary *Process* and optional *Input/Output*. The four traditions converge on the same structure: an internal transformation stage as the necessary core and two — and only two — directions of communication with the environment.

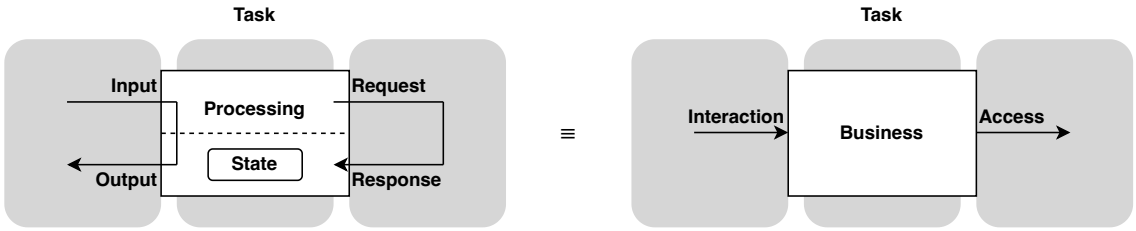

Figure 2. The three invariant stages of the formal process: Interaction, Business (processing), and Access, represented as an execution sequence with their input and output arrows.

The identification of these three stages — necessary processing, interaction and access as exhaustive channels of communication with the environment — constitutes the starting point for the derivation of the three-layer model that defines the XF Architecture. That derivation, including the formal correspondence between each stage and its corresponding abstraction layer, is developed in §5.5 once the complementary foundations on which it also depends have been established.

The artifact as the canonical scale of terminological uniformization The IPO structural pattern is not exclusive to the artifact as a unit: it appears recurrently at multiple scales in every well-built software system — at the level of the individual statement, the function, the method, the component, and the subsystem. This scalar self-similarity has been recognized in at least four independent formal traditions. Cybernetics [45] established the input–process–output–feedback loop as a universal primitive of adaptive systems. General systems theory [5] posited that all systems — biological, social, and technical — share this structure regardless of the scale at which they are observed. Structured engineering [47] formalized it as a recursive technique of functional decomposition of software. The calculus of communicating sequential processes [16] demonstrated its algebraic compositionality by means of three primitives: internal action, reception over a channel, and emission over a channel. Scalar self-similarity is therefore an emergent property of the well-built software system, recognized across the literature and derived from the very nature of the formal process — a notion that Mandelbrot [28] generalized to the concept of fractal self-similarity as a structural property of natural systems.

The scope of the XF model is neither that self-similarity nor its recursive formalization. Within the hierarchy of scales, the model identifies a specific one — that of the **artifact** — and adopts it as the canonical level of terminological uniformization. The reason for this choice is structural and not conventional: the artifact is the minimal scale at which a complete formal process is realized one-to-one as a code structure. The three invariant stages of the formal process — interaction, processing, and access — correspond to three regions of the code that constitute layers with their own structural identity. Below the artifact, the units of decomposition — components, classes, functions — are encapsulations of responsibility that serve the process but are not complete formal processes in themselves: a class has no “interaction stage” of

its own, a module has no "access stage" of its own. Above the artifact, the units of composition — distributed systems, ecosystems, organizations — coordinate multiple processes and do not reduce to a single process.

This delimitation of the canonical scale is what distinguishes the XF model as a reference model — in the sense of the ISO/IEC/IEEE 42010 standard [22] — and not as one more architecture among those that coexist in the industry. What the model contributes is not structure — which is already universally present in any software artifact by the nature of software as the automation of formal processes — but **vocabulary**. The XF model couples the nomenclature to the formal model of the process so that the structure, already existing, is recognizable and communicable across any technology. The fragmentation documented in §5.4 is not structural but terminological: the same architectural concept receives incompatible names depending on the development framework used, and that incompatibility — not the underlying structure — is what produces the inter-framework learning curve and the operational costs of fragmentation. The XF model resolves that fragmentation by normalizing the nomenclature at the scale where the structure of the formal process admits a one-to-one translation to code, and only at that scale.

5.3 Component typing as an instrument of comprehension

The development of a software artifact involves the construction and coordination of a potentially extensive set of code units — classes, functions, modules — each of which solves a specific problem within the system. As the complexity of the artifact grows, the developers' ability to understand the function of each unit and its relationship with the rest of the system becomes a determining factor in the team's productivity and in the quality of the resulting product.

The management of this complexity has been recognized as one of the central problems of software engineering since its origins. Dijkstra [10] argued that the intellectual quality of a program is inversely proportional to the conceptual distance between the problem it solves and the structure of the code that implements it. This observation points toward a fundamental principle: the structure of the code must be readable, and its readability depends to a large extent on each code unit communicating its purpose and responsibility immediately.

Component typing is the mechanism by which development teams assign to each code unit a category that communicates its function within the artifact as a whole. This categorization is usually materialized in the component's nomenclature — through suffixes, prefixes, or naming conventions — so that the component's name is, by itself, sufficiently informative about its role in the system. Thus, a component named `UserRepository` communicates unambiguously that its responsibility is access to user data; a component named `AuthenticationService` indicates that its function is to provide authentication logic; a component named `TemperatureUtils` signals that it contains utility functions related to the handling of temperature data.

Martin [29] identifies the clarity of names as one of the fundamental principles of clean code, arguing that a name that requires an additional comment to be understood is a name that has failed in its purpose. Typing extends this principle beyond the individual name: it not only communicates what the component does, but what type

of logic it implements and what place it occupies in the system's architecture.

The advantages of consistent typing are documented and measurable. First, it reduces the cognitive load on developers by allowing them to set precise expectations about the behavior of a component before reading its implementation. Second, it facilitates the onboarding of new team members, who can orient themselves in the structure of the artifact more quickly when the components are categorized coherently. Third, it improves code review, since reviewers can verify not only the correctness of the implementation but also its adequacy to the assigned type. Fourth, it favors the separation of responsibilities [47], by making the responsibility of each component explicit and facilitating the detection of violations.

The implicit convergence of the industry These advantages have not gone unnoticed by the software industry. The recurrent emergence of component taxonomies in heterogeneous architectures [41, 3] is evidence that component typing is a valuable instrument: the most widely adopted development frameworks implement it autonomously — each with its own taxonomy, but all with the same objective: to make the structure of the artifact comprehensible by categorizing its components.

Fowler [12] documented that the same typing patterns recur in heterogeneous enterprise systems — `Repository` for data access, `Service` for business logic, `Controller` for entry points. This recurrence is not coincidental — it reflects that developers converge toward the same component categories because the processes they automate have the same underlying structure. Spring types its components with `@Repository`, `@Service`, and `@Controller`. Angular types its own with `Injectable`, `Component`, and `Pipe`. Flutter types its own with `BLoC`, `Widget`, and `Repository`. Each development framework has independently reached the same conclusion: component typing is necessary to make the structure of the artifact comprehensible.

This implicit convergence of the industry, observable in the body of recurrent patterns documented by Bass et al. [3] and Fowler [12], constitutes empirical confirmation that component typing is not an arbitrary convention but a structural necessity derived from the nature of software as the automation of formal processes. If development frameworks converge toward the same component categories, it is because the processes they model have the same invariant stages. Typing does not create that structure — it makes it visible.

However, this implicit convergence has a structural limitation that no development framework can resolve on its own: each development framework types with its own proprietary taxonomy, incompatible with the taxonomies of the others. Development frameworks converge toward the same solutions but describe them with different vocabularies — vocabularies that, as documented in §5.2 and to be developed in §5.4, produce heterotechnical synonymy and interframework homonymy that fragment knowledge and raise the cost of context switching between technologies.

Component typing is, ultimately, a form of structural documentation embedded in the code itself. Unlike external documentation — comments, wikis, diagrams — typing is inseparable from the artifact and evolves with it naturally [4]. This property makes it an especially valuable instrument in agile development environments, where traditional documentation tends to become outdated quickly. However, its effectiveness as an instrument of communication — not only within a team but

between teams and between technologies — depends on a condition that current development models do not satisfy: that the type taxonomy be consistent, exhaustive, and universally shared. When this condition is not met, typing loses its value as an instrument of inter-technological communication and becomes an additional source of fragmentation. This is precisely the problem addressed by the following clause.

5.4 Limitations of proprietary typing

The preceding clause established that component typing is a necessary instrument for managing complexity in software development, and that the most widely adopted development frameworks implement it convergently — independently reaching the same component categories because the processes they model have the same underlying structure. This implicit convergence, in line with the recurrence of architectural patterns documented in heterogeneous *enterprise* systems [12], supports the view that typing is not an arbitrary convention but a structural necessity derived from the nature of software.

However, this same convergence reveals the fundamental limitation of the current state of the industry: development frameworks converge toward the same structural solutions, but describe them with proprietary taxonomies incompatible with one another. The convergence exists — but it is invisible to developers because it is hidden behind different vocabularies. A developer working with Spring and one working with Angular are typing their components with the same underlying conceptual categories, but with names so different that they cannot recognize that equivalence without an explicit translation process. The result is that the industry repeatedly pays the cost of learning what it already knows — expressed in another nomenclature.

The structural cause of this situation lies in the fact that component typing has not historically been the object of independent standardization, but has remained subordinate to the development frameworks. Each development framework defines its own type taxonomy as part of its programming model, with the consequence that typing is consistent within a concrete framework but incompatible with the taxonomies of other frameworks. The result is an ecosystem in which the real structural convergence of software is fragmented at the surface by proprietary taxonomies that prevent it from being recognized.

This phenomenon can be characterized precisely by means of the concept of **semantic coupling to the development framework**: the architecture of the artifact does not derive from the structure of the process it models — which is universal — but from the conventions of the development framework used — which are proprietary. Martin [30] identifies this coupling as one of the most harmful antipatterns in the design of software systems, arguing that an architecture that cannot be described independently of its implementation tools is an architecture that has lost its function as a conceptual model and has become a mere technological artifact.

The two patterns of inconsistency The magnitude of the problem becomes evident when examining the resulting semantic fragmentation in widely adopted development frameworks. A systematic analysis of proprietary taxonomies reveals two patterns of inconsistency that occur recurrently and simultaneously.

The first is **heterotechnical synonymy**: architecturally equivalent concepts receive different names in different frameworks. The logic for accessing persistent data is called `Repository` in Spring, `QuerySet` in Django, `DAO` in J2EE, and `Model` in Node.js. The HTTP-interaction logic is called `Controller` in Spring, `View` in Django, `Router` in Node.js, and `Handler` in development frameworks such as Gin or Echo. A developer moving between these technologies must learn not only the syntax of the new language, but a new conceptual taxonomy to describe the same architectural problems they already solved in their previous technology — even though the underlying structure is identical.

The second is **interframework homonymy**: the same term designates architecturally distinct concepts in different frameworks, generating ambiguity that can lead to design errors. The term `Service` is the most illustrative example: in Spring it designates a business-logic component; in Android it designates an asynchronous task executed in the background; in Angular it designates a general-purpose singleton object that may contain both business logic and access logic; in microservices architecture it designates an independent deployment unit. Four architecturally distinct concepts, a single term, four different frameworks. The practical consequence is that, in teams with developers trained in different technologies, the use of the term `Service` in a technical discussion communicates nothing precise without an explicit technological context — exactly the opposite of what typing is supposed to achieve.

The operational consequences These inconsistencies have operational consequences that go beyond mere terminological inconvenience. Jones [25], in his quantitative analysis of the factors that determine productivity in software projects, identifies ambiguity in requirements and inconsistencies in design conventions among the factors with the greatest negative impact on development speed and on the defect density of the product. The semantic fragmentation of proprietary taxonomies contributes directly to both factors: it generates ambiguity in design discussions and produces structural inconsistencies that translate into divergent implementation decisions within the same team.

A second far-reaching effect is the accumulated cost of technology-specific training that organizations must bear when they onboard new developers or when they migrate systems to new platforms. This cost — analogous in its nature to the technical debt described by Cunningham [9] but of an organizational rather than structural nature — accumulates invisibly in the product and manifests in onboarding costs, in the dependence on scarce specialized profiles, and in the difficulty of scaling teams quickly. In a context of high professional turnover and incompatible proprietary taxonomies, this type of debt becomes a structural burden affecting the entire industry.

The root cause The root of the problem does not lie in the development frameworks themselves — which fulfill a legitimate and valuable function by providing specialized implementation abstractions — nor in the quality of the software they produce. The XF model identifies a structural root cause: the convergence that software already implements implicitly has no formal and universal expression. The development frameworks have independently reached the same underlying conceptual categories — because the processes they model are the same — but each one has named and organized them in a proprietary way, without any mechanism that allows

that equivalence to be recognized and communicated between technologies.

In terms of Dijkstra's layered architecture [10], the problem is that the conceptual layer and the technological layer are not separated: the developer's mental model is coupled to the tool, when it should be coupled to the process that the tool implements. Proprietary typing confuses the instrument with the concept — it makes the developer learn Spring's `@Repository` instead of learning the concept of a data-access component that `@Repository` implements, and then re-learn that same concept as `QuerySet` when switching to Django, as `DAO` when working in J2EE, and as `Model` when working in Node.js.

The XF model addresses this problem by defining an independent conceptual abstraction layer whose component taxonomy derives from the invariant structural properties of formal processes — established in §5.1 and §5.2 — and not from the conventions of any particular development framework. This taxonomy does not replace the proprietary taxonomies of the development frameworks — the development frameworks will continue to call their access components `@Repository`. What it provides is the level of abstraction that allows developers to recognize that `@Repository`, `QuerySet`, `DAO`, and `Model` are the same concept expressed with different names — and to communicate about it with a common vocabulary. The formal derivation of this taxonomy is the object of §5.5.

5.5 Derivation of the three-layer model

Recapitulation of propositions The preceding clauses have established three propositions that, taken together, necessarily determine the structure of the XF Architecture Model:

1. The **first proposition** establishes that every software artifact models a formal process, and that the architecture of the artifact is isomorphic to the structure of the process it models (§5.1).
2. The **second proposition** establishes that every formal process admits a canonical decomposition into three stages with distinct formal status: *processing* as the structurally necessary stage, and *interaction* and *access* as structurally optional but exhaustive and mutually exclusive categories of communication with the environment (§5.2).
3. The **third proposition** establishes that component typing is the mechanism by which the architecture of an artifact makes its structure explicit, and that its effectiveness requires a taxonomy that is consistent, exhaustive, and independent of any proprietary development framework (§5.3 and §5.4).

From the conjunction of these three propositions a direct conclusion follows: there exists a universal architectural structure, composed of three abstraction layers that correspond one-to-one to the three invariant stages of every formal process, and whose component taxonomy is defined independently of any implementation technology. This structure is the XF Architecture Model.

Formal derivation of the layers The derivation of the three layers of the XF model from the invariant stages of formal processes follows the principle of structural correspondence stated in §5.1: each stage of the process corresponds to exactly one

abstraction layer of the artifact, and each abstraction layer of the artifact covers exactly one stage of the process.

The **interaction stage** of the formal process — the reception of the order to execute and the verification of the preconditions — corresponds to the **Interaction Layer** of the artifact. This layer contains the logic that makes the artifact accessible to its consumers, whether human users through graphical interfaces or external systems through defined communication protocols. Its responsibility is exclusively that of establishing the conditions of interaction — validating inputs, defining access protocols, presenting results — without incorporating processing logic or data-access logic.

The **processing stage** of the formal process — the ordered execution of the operations that transform the inputs into the expected result — corresponds to the **Business Layer** of the artifact. This layer contains the logic that constitutes the value of the artifact: the algorithms, the business rules, and the data transformations that define what the system does and how it does it. Its responsibility is exclusively that of processing — not interacting with external consumers or directly accessing remote data sources.

The **access stage** of the formal process — the delegation of part of the processing to external systems or processes — corresponds to the **Access Layer** of the artifact. This layer contains the logic that enables the artifact to communicate with other systems through defined protocols, abstracting the details of each protocol from the rest of the artifact. Its responsibility is exclusively that of establishing and managing communications with external systems — without processing the obtained data beyond what is necessary for its normalization, nor exposing interaction interfaces to external consumers.

Positioning within the OSI model The three-layer structure of the XF model does not emerge in isolation, but in continuity with an established and widely validated reference framework: the Open Systems Interconnection model ISO/IEC 7498-1 [17]. The OSI model organizes network communication protocols into seven abstraction layers with clearly delimited responsibilities, where each layer provides services to the layer immediately above it and consumes the services of the layer immediately below it.

The highest level of the OSI model — the Application Layer — is defined as the boundary between the communication layers and the application processes: it is the means of access to the OSI environment by the application processes, not a container of their internal logic. By design, ISO/IEC 7498-1 ends at that boundary. The internal architecture of the application processes — how the logic is organized *within* each artifact — is explicitly outside the scope of the OSI model and has historically remained without an equivalent reference framework.

The XF model occupies that space. It does not subdivide the OSI Application Layer — ISO/IEC 7498-1 does not specify the interior of applications and, therefore, there is nothing to subdivide — but rather extends the OSI stratification principle to an orthogonal dimension: whereas OSI stratifies communication *between* artifacts, XF stratifies the *internal* structure of each artifact. The two stratifications are technically independent and conceptually complementary. An XF artifact executes over the OSI

environment — typically communicating with other artifacts through the service primitives of the Application Layer — but its internal structure is organized following the XF stratification, orthogonal to the seven OSI layers.

The extension preserves the four formal properties of the OSI model, applied to the new dimension:

- **Encapsulation of services through primitives**: each layer exposes to the layer immediately above it a defined set of services through the layer's injection component, which acts as a single, named point of access. The upper layer consumes those services without knowing the internal details of their implementation.
- **Strictly unidirectional dependency**: each layer depends exclusively on the layer immediately below it, never on the one above it or on non-adjacent layers. This property is materialized in the principle of layer isolation established in §6.2.2.
- **Implementation isolation**: a layer may change internally without affecting the upper layer, provided that it preserves the service primitives it exposes. This property enables the technological independence that the model prescribes in §6.2.1.
- **Composition of services**: the total behavior of the artifact is obtained by composition of the services that each layer provides to the one immediately above it, starting from the external services consumed in the lower layer.

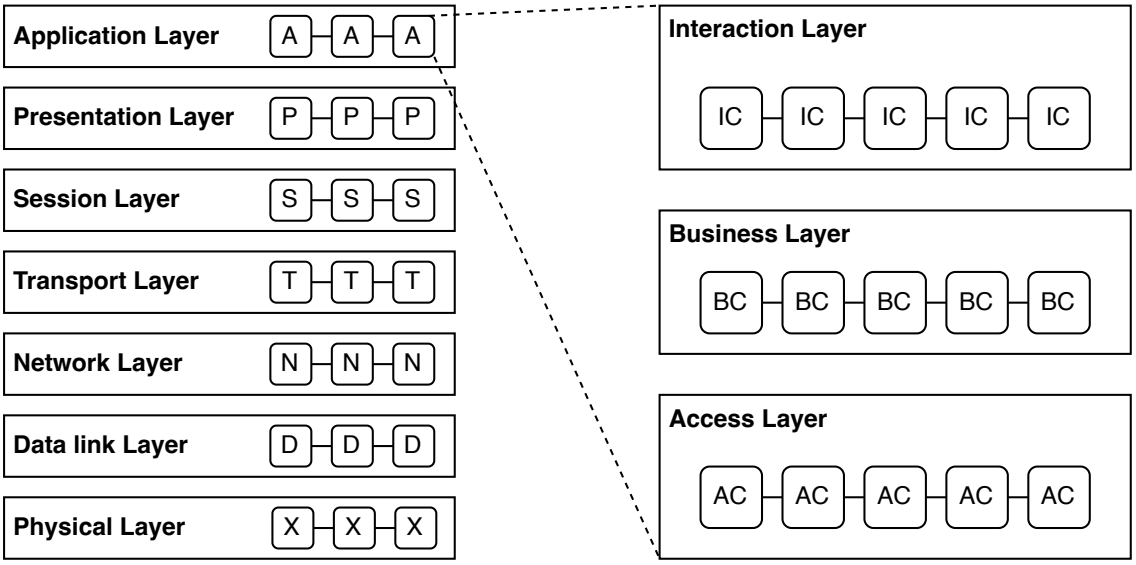

Figure 3. Orthogonality between the two stratifications: the seven layers of the OSI model organized vertically as a stack of protocols for communication between artifacts, and the three layers of the XF model organized in an orthogonal dimension as a stratification of the internal structure of the artifact. The OSI Application Layer (Level 7) is the point of contact between the two dimensions.

The continuity with the OSI model is not merely formal: it is also pragmatic. The lasting contribution of OSI to the industry was not its protocol stack — it was TCP/IP that was adopted as the effective stack — but its standardizing contribution at three levels. The first is the **common vocabulary**: OSI gave generic names —

transport layer, *session layer* — that allowed engineers from competing companies to describe the same problems with the same terminology, eliminating the semantic fragmentation between proprietary nomenclatures. The second is **independence from implementation**: the service primitives were defined without assuming any concrete protocol, allowing different implementations to be compared as realizations of the same abstraction. The third is **cross-validation and pedagogy**: the OSI stratification made it possible to identify which protocols operate in which layer, facilitating systematic teaching, standardized documentation, and architectural auditing.

The XF model pursues exactly the same standardizing contribution applied to the internal architecture of the artifact. A common vocabulary that replaces the proprietary nomenclatures of the development frameworks. Technology-independent interfaces that allow an access component to be described without assuming a concrete protocol, a persistence library, or a specific framework. Cross-validation that allows the identification of which logic of an artifact operates in each layer of the XF stratification, enabling automated static analysis and systematic teaching. The analogy between what OSI contributed to communication between systems and what XF intends to contribute to the internal architecture of the artifact is the foundation of the model's positioning in continuity with the tradition of software standardization.

Formal properties of the three-layer model The foregoing derivation determines three formal properties that the three-layer model satisfies and that constitute, ultimately, the correctness criteria of any XF artifact.

The first is **completeness**: the three layers cover, exhaustively and without overlap, the three structural categories of the formal process — necessary processing, interaction as the incoming communication channel, access as the outgoing communication channel. Every component of a software artifact can be classified into exactly one of the three layers according to the category of the process it implements — there is no program logic that does not belong to one of the three categories, nor logic that belongs simultaneously to more than one. In artifacts that do not require communication with the environment (pure processing), the Access and Interaction layers may remain empty without this affecting the conformance of the artifact with the model.

The second is **strict separation**: each layer implements exclusively the logic corresponding to its stage of the process. The presence of logic of one stage in the layer corresponding to another stage constitutes a violation of the structural correspondence between process and artifact, and therefore a violation of the model. This separation is the basis of the principle of layer isolation that is formally defined in §6.2.2.

The third is **directionality**: the dependency between layers is strictly unidirectional in the downward direction — the Interaction Layer depends on the Business Layer, and the Business Layer depends on the Access Layer, but never in the reverse direction. This property derives directly from the order of execution of the stages of the formal process: interaction precedes processing, and processing precedes access; a dependency in the reverse direction would imply that an earlier stage of the process

depends on a later stage, which contradicts the causal structure of the process.

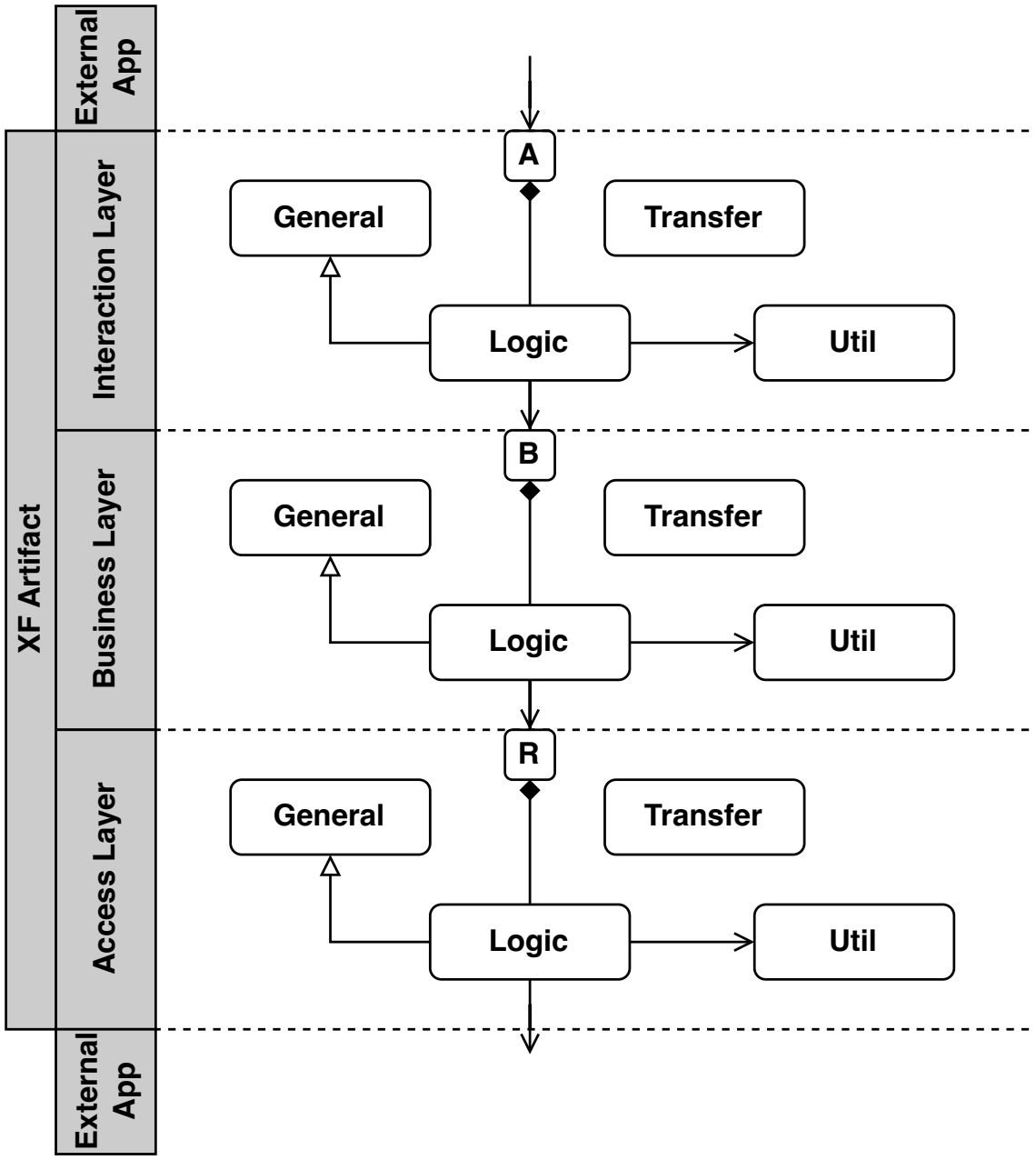

Figure 4. Stratification of the XF artifact: the three layers — Interaction, Business, and Access — with their internal inheritance trees (generalization components as the base and logical components as specializations) and the downward unidirectional dependencies between layers.

These three properties — completeness, strict separation, and directionality — constitute the formal foundations on which the normative model developed in the following clauses is built. Their derivation from the invariant structural properties of formal processes guarantees that the XF model is not an arbitrary design convention, but a necessary consequence of the nature of software as an instrument for the automation of processes.

6 The XF Architectural Model

This clause defines the XF Architectural Model in its normative terms. Unlike §5, whose purpose was to set out the theoretical reasoning that justifies the model, the content of this clause is prescriptive in nature: it establishes the principles that every XF artifact shall satisfy, the structure it shall adopt and the component taxonomy it shall employ. The provisions of this clause constitute the fundamental requirements of the model, from which all its conformance rules derive.

The XF Architectural Model is defined as an **architectural reference model** in the sense established by Bass et al. [3]: a set of design decisions applicable to a domain of recurring problems that, once adopted, establish constraints on the architecture of the systems that implement them and favour certain qualities of the resulting system. In the case of the XF model, the domain of recurring problems is the automation of formal processes through software, and the favoured qualities are comprehensibility, maintainability, interoperability between teams and technological independence.

It is important to distinguish the XF model from other architectural instruments with which it could be confused. The XF model is not a design pattern in the sense of Gamma et al. [14] — it does not describe a solution to a local design problem within a component or a set of related components. Nor is it an architectural style in the sense of Shaw and Garlan [41] — it does not describe a family of systems characterized by a particular structural organization. The XF model is a reference model: a conceptual abstraction that establishes a common vocabulary, a universal organizational structure and a set of guiding principles that are applicable to any software artifact regardless of its domain, technology or scale.

This distinction has direct practical implications. A design pattern applies or does not apply according to the local problem being solved. An architectural style is chosen among alternatives according to the qualities to be favoured in the system. The XF model, by contrast, is not an option among equivalent alternatives — it is a conceptual abstraction that can coexist with any design pattern and with any architectural style, adding upon them an organizational structure and a component taxonomy of universal validity.

The clause is organized as follows. Clause §6.1 presents the general overview of the model, describing its structure and its constituent elements in synthetic form. Clause §6.2 defines the four guiding principles of the model, which have the rank of axioms: no normative provision of this document may be interpreted in a manner contrary to them.

6.1 General overview

The XF Architectural Model organizes the internal structure of any software artifact through two orthogonal dimensions of classification: the **abstraction dimension**, which determines in which layer of the artifact a component resides according to the stage of the formal process it implements, and the **functional dimension**, which determines what type of role that component plays within its layer. The combination of these two dimensions produces a matrix structure that categorizes each component of the artifact unambiguously, eliminating the structural ambiguity that characterizes

the proprietary architectures described in §5.4.

The **abstraction dimension** divides the artifact into three hierarchical layers ordered from the lowest to the highest level of abstraction. The Access Layer constitutes the lowest level of abstraction and is responsible for all communication with systems external to the artifact. The Business Layer constitutes the intermediate level of abstraction and is responsible for the domain logic and the data transformations that define the behaviour of the artifact. The Interaction Layer constitutes the highest level of abstraction and is responsible for the entry points to the artifact, both for human users and for external systems that consume it. The correspondence between these three layers and the three invariant stages of formal processes — access, processing and interaction — has been established formally in §5.5.

The **functional dimension** divides the components of each layer into five types according to the function they perform within it. Logical components implement the effective logic of the layer. Generalization components abstract behaviours common to logical components of the same layer. Injection components manage the life cycle and centralized access to the instances of the logical components. Utility components provide auxiliary operations of scope local to the layer. Transfer components define the data structures that circulate between components. Each of these types is defined exhaustively in §7.3.

These two dimensions are **independent of each other** in the sense that the classification of a component in a layer does not determine its functional type, and its classification in a functional type does not determine its layer. A logical component may belong to any of the three layers; a utility component may equally belong to any of the three layers. What the model prescribes is that every component shall be classified simultaneously in exactly one layer and exactly one type — neither more nor less. This property of unambiguous classification is what confers on the model its capacity to eliminate structural ambiguity and provide a common basis for communication between teams.

The XF model additionally recognizes two elements that operate over the whole of the artifact and are not inscribed in any cell of the $L \times T$ matrix. The first is the **`XF` element**, named XF, which encapsulates the initialization and termination of the whole of the artifact in a single point of control. The second is the **canonical folder structure**, which physically materializes in the project's file system the matrix organization of the model, making the architecture visible and inspectable without the need for additional tools. The **injection components** — named `R`, `B` and `A` in the Access, Business and Interaction Layers respectively — are not cross-cutting: each one belongs to the *Injection* cell of its layer and manages the life cycle of the logical components of that same layer.

A fundamental property of the model is its **universality of application**. The XF model establishes no constraints on the business domain of the artifact, the programming language employed, the development framework used, the programming paradigm adopted — object-oriented, functional, reactive — nor the execution environment of the artifact — server, mobile device, browser, embedded system, container. The universality of the model derives directly from the fact that its foundations — the invariant structural properties of formal processes — are equally universal. Every artifact that automates a formal process, independently of any

other technological consideration, may and shall be organized according to the XF model.

This universality does not imply uniformity of implementation. Two artifacts developed in technologies of distinct nature and implementing independent business domains will be, if both follow the XF model, structurally equivalent in terms of layer organization, component typology and dependency flows. This structural equivalence is precisely what allows a developer trained in the XF model to orient themselves rapidly in any XF artifact independently of the technology in which it is implemented, reducing the context-switching cost documented in §5.4.

The XF model generalizes the structure underlying every software artifact; it is not constructed as a reaction to the architectures prevailing at a given moment, but as an abstraction of the structural properties that every architecture shares. This generality implies that its application is not reduced to paradigms with nominal classes nor to artifacts with mutable state: any software unit that automates a formal process admits XF organization, independently of the paradigm — object-oriented, functional, reactive, procedural — and of the syntactic primitives that the language offers to express components. The concrete form that components adopt varies with the constraints of the technical substrate: a logical component in a purely functional paradigm does not maintain internal state and its operations reduce to pure functions; a logical component in a language without nominal structure may materialize as a module or as a cohesive grouping of functions of global scope rather than as a class; an injection component may take the form of an object with immutable references, of an observable state container or of a lexical binding managed by the execution platform. Variations of form do not weaken the application of the model: they are its natural projection onto the technical substrate of the artifact. What the model prescribes is the classification and the relations between components, not the syntactic primitives with which those components are expressed.

The projection of the model onto paradigms that are not classically object-oriented — functional programming with unidirectional state flow and declarative views, reactive systems based on streams, architectures with distributed deployment — lies outside the scope of the present standard. In all those paradigms the $L \times T$ classification retains its total and exhaustive character; only the syntactic form with which each component materializes changes, not its architectural role.

6.2 Guiding principles

The guiding principles of the XF model have the rank of normative axioms. No provision of this document may be interpreted in a manner contrary to them, and in case of ambiguity in the application of any conformance rule, the interpretation that prevails is the one that best satisfies the set of principles defined in this clause.

The principles are independent of each other in their statement, but interdependent in their application: the correct implementation of any of them presupposes the correct implementation of the others. An artifact that satisfies the principle of layer isolation but not that of closed typing is not a conformant XF artifact — the principles are satisfied together or they are not satisfied.

6.2.1 Technology agnosticism

Statement: the XF model establishes no constraints on the programming language, the development framework, the programming paradigm — object-oriented, functional, reactive, procedural —, the execution platform, the **deployment infrastructure** — monolithic, distributed or of any intermediate topology —, the **state management model** of the artifact — with mutable state, immutable or stateless — nor the business domain. Every software artifact that automates a formal process may be organized according to the XF model independently of any technological consideration.

The constraints that a specific language or paradigm imposes on the syntactic form of the components — absence of nominal structures, mandatory immutability, life cycle ceded to the development framework, absence of encapsulation mechanisms — do not bound the scope of the XF model: they determine the form that its components adopt in that context, without altering their classification or their relations.

Motivation Technology agnosticism is the property that makes possible the principal function of the XF model: to provide a common conceptual abstraction layer over a heterogeneous development ecosystem. A reference model that established constraints on the implementation technology would be, at best, a proprietary standard — useful within its ecosystem, irrelevant outside it. The history of the software industry is populated with standards having this characteristic, whose adoption was limited to the technological environments for which they were conceived and whose obsolescence arrived with the obsolescence of those environments.

The XF model avoids this structural limitation by deriving its principles from the invariant properties of formal processes — established in §5.2 — and not from the conventions of any particular technology. Given that the properties of formal processes are independent of the technology employed to automate them, the model that formalizes them inherits that same independence.

This design decision connects with the principle of separation of concerns stated by Dijkstra [10]: conceptual design decisions — what the system does and how it is organized — must be separated from implementation decisions — with what tools and in what language it is built. The technology agnosticism of the XF model is the materialization of this principle at the architectural level: the model defines the conceptual structure of the artifact, and the chosen technology is the instrument with which that structure is implemented.

Normative implications The first implication is that the XF model shall be capable of being implemented in any language or development framework without requiring extensions, adaptations or exceptions to the model. If the implementation of the model in a specific technology requires violating any of its principles, the cause of that violation is a limitation of the technology, not a limitation of the model. The treatment of these situations is the subject of §10.

The second implication is that the evaluation of the conformance of an artifact with the XF model shall be capable of being performed independently of the technology in which it is implemented. The conformance rules of the model are necessarily

technologically neutral: they describe structural properties of the artifact — presence of layers, classification of components, directionality of dependencies — that can be verified independently of the language or framework employed.

The third implication affects nomenclature. The XF model prescribes canonical names for the injection components — `R`, `B`, `A` —, for the folders of each layer — `/repository`, `/business`, `/api` — and for the component types — suffixes `Repository`, `Business`, `Service`, `View`, `Utils`. This nomenclature is part of the model and shall be respected independently of the naming conventions proper to the language or development framework employed. When the conventions of the framework conflict with the nomenclature prescribed by the model, the nomenclature of the model prevails.

Scope and limits of the principle Technology agnosticism has a precise scope that it is advisable to delimit in order to avoid excessively broad interpretations. The XF model is agnostic with respect to the implementation technology — language, framework, platform — but it is not agnostic with respect to the conceptual structure of the artifact. The model prescribes with precision how an artifact shall be organized, what types of components it shall contain and how they shall relate to each other. This structural prescription is not negotiable as a function of the technological preferences of the development team.

Put another way: technology agnosticism means that the model does not say with what tool the artifact is built, but it does say how it shall be organized independently of the chosen tool. The freedom that the principle grants is freedom of implementation, not freedom of structure.

Note. The canonical projection of the model onto each specific language and paradigm — the syntactic form that the components adopt in each case — is the subject of a complementary per-language compatibility document, external to the present standard. The convergence that the model formalizes is introduced in §5.3, and its detailed projection per development framework, in §D.1.

Relation with other principles Technology agnosticism is the principle that gives meaning to the three remaining principles. Layer isolation (§6.2.2), the precedence of the architecture over the tool (§6.2.3) and closed and exhaustive typing (§6.2.4) are properties that the model prescribes precisely because they are independent of technology — if they depended on it, they could not be principles of an agnostic model. In this sense, technology agnosticism is the foundational principle from which the others derive their universality.

6.2.2 Layer isolation

Statement: every component of an XF artifact depends exclusively on components classified in its same layer or in layers of lower level of abstraction. No component may depend on a component classified in a layer of higher level of abstraction. Dependencies between layers are strictly unidirectional in the descending direction: the Interaction Layer may depend on the Business Layer; the Business Layer may depend on the Access Layer; no dependency in the inverse direction is admissible.

Motivation Layer isolation is the structural property that guarantees that the architecture of the artifact is isomorphic to the formal process it models. As was established in §5.2, when the communication stages are present in a formal process, their causal order with respect to processing is invariant: interaction causally precedes processing — the process receives the invocation before executing its logic — and access occurs as delegation during or after processing — the process accesses the exterior when it needs information or capabilities that reside outside its own boundaries. That causal order is unidirectional. A dependency of the Access Layer on the Business Layer would imply that the access stage of the process depends on the processing stage, which contradicts the causal structure of the process and, therefore, the premise on which the model is founded.

Parnas [39] established that effective modularity requires that each module hide a design decision from the others, so that changes in that decision do not propagate beyond the module that encapsulates it. Layer isolation is the materialization of this principle at the architectural level: each layer encapsulates a type of logic — access, business, interaction — so that changes in that logic do not propagate to the layers that must not know it. A change in the communication protocol with an external system — for example, the migration from REST to GraphQL — shall remain confined within the Access Layer and shall not require modifications in the Business Layer nor in the Interaction Layer. This property, known in the literature as resistance to local change [47], is one of the most valuable maintainability qualities in long-lived systems.

Normative implications The normative implication of layer isolation is that every dependency between components of distinct layers shall be strictly descending: a component may depend on those of layers of lower level of abstraction, never on those of higher level. To fix with precision the scope of this prescription, the model defines below what constitutes a dependency for its purposes.

Formal definition of dependency For the purposes of the XF model, a **dependency** between two components A and B is defined as any relation in which A requires knowing the definition or the behaviour of B in order to be compiled, instantiated or executed. This definition encompasses the following concrete relations:

- A invokes an operation of B through the injection component of the layer to which B belongs.
- A inherits from B or implements an interface defined by B.
- A declares an attribute, parameter or return value whose type is B or derives from B.
- A imports or references the module, package or namespace in which B is defined.

This definition is intentionally broad. The XF model does not distinguish between types of dependency for the purposes of the isolation principle — every relation that implies that A knows B is a dependency, and every dependency is subject to the directionality constraints of the model.

Transfer components and isolation Transfer components are governed by the same descending directionality as the rest of the dependencies of the model: a component classified in a layer may reference transfers defined in its own layer or in layers of lower level of abstraction, never in layers of higher abstraction. This directionality does not constitute an exception to the isolation principle: it satisfies it in full.

Isolation and generalization The principle of layer isolation has direct implications for the use of generalization components. Inheritance between components is restricted to components of the same layer: no logical component may inherit from a generalization component defined in a distinct layer, and no generalization component may abstract behaviour from components belonging to distinct layers.

This restriction has a consequence that may turn out to be counterintuitive: when two or more layers require implementing the same structural pattern — for example, the pattern of observation of mutable state or the periodic execution of tasks — each layer **shall** implement its own generalization component independently, even though the implementation is structurally identical.

The XF model prescribes this duplication explicitly because the alternative — a generalization component cross-cutting several layers — would violate the isolation principle and would introduce a structural dependency between layers that would compromise the resistance to local change of the artifact. Controlled duplication between layers is a cost deliberately assumed by the model for the benefit of the structural independence of each layer.

Isolation and non-trivial shared behaviour The isolation principle does not prevent components of distinct layers from using common logic — it prevents that common logic from being implemented as generalization between layers or from creating dependencies in the ascending direction. When several components of distinct layers need to access non-trivial shared behaviour — recording of debug traces, metrics, data encryption, reading of persisted configuration — the XF model prescribes that that behaviour be encapsulated in a logical component classified in the **lowest layer** that requires it, so that it may be invoked by the upper layers respecting the descending directionality.

Scope and limits of the principle The isolation principle has a precise scope that it is advisable to delimit. It restricts the direction of dependencies *between layers* — which component may know, reference or inherit from which —, admissible only in the descending direction. It does not restrict dependencies between components of a same layer, which may relate to each other without restriction of direction, nor the direction in which information circulates at run time.

Put another way: the principle determines who may depend on whom, not how data travel during execution. This distinction — between the direction of the dependency and that of the data flow — is what allows information to ascend between layers without inverting the dependencies, as developed below.

Ascending communication — event-oriented programming The isolation principle prescribes that the direction of dependencies is descending, but it does not prevent information from flowing in the ascending direction. The distinction is fundamental: the **direction of the dependency** determines who knows whom; the **direction of the data flow** determines in which direction information travels at run time.

When a component of a lower layer needs to communicate a change of state to components of upper layers — for example, a loss of connectivity detected by a business component, or a real-time data update received through a WebSocket channel — the XF model prescribes that that communication ascend **without introducing ascending structural dependency**: the coupling remains descending even though the information flows towards the upper layers. This property is typically realized through event-oriented programming and the observation pattern — the components of upper layers register as observers of the state and react to its mutations without the lower-layer component invoking them directly —, but the choice of the concrete mechanism corresponds to the implementer.

This mechanism preserves the isolation principle because the dependency continues to be descending while the notification flows in the ascending direction without the lower layer knowing its observers. The business component mutates its state and notifies; the registered observers react. The Business Layer never invokes directly any component of the Interaction Layer. The following figure illustrates this mechanism of ascending communication.

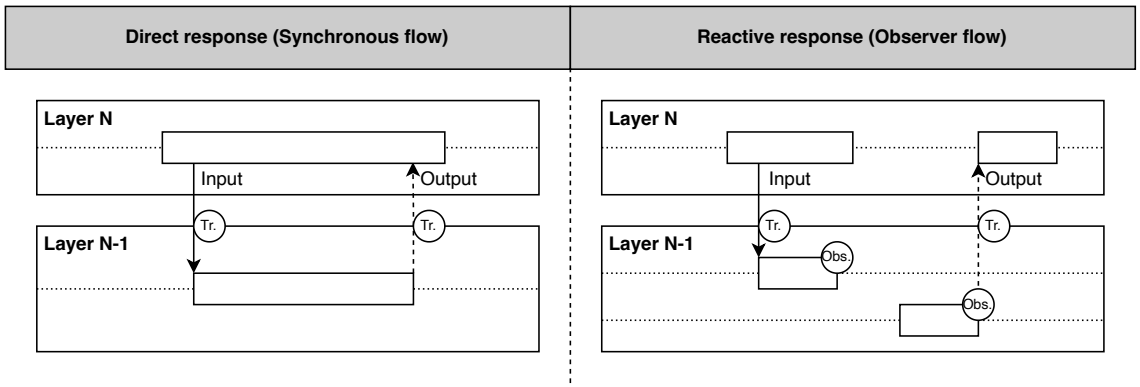

Figure 5. Mechanism of ascending communication through the observer pattern: a business component mutates its state and notifies the registered observers; an interaction component reacts to the change without establishing a direct ascending dependency on the business. This inversion preserves the principle of layer isolation.

Relation with other principles Layer isolation is the principle that gives operational content to technology agnosticism (§6.2.1): without isolation, technological changes in one layer would propagate to the others, creating technological coupling between layers that would compromise the independence of the artifact with respect to its implementation. Likewise, layer isolation is the necessary condition for closed and exhaustive typing (§6.2.4) to be meaningful: classifying a component in a layer has normative value only if that classification implies real constraints on the dependencies that the component may establish.

6.2.3 Precedence of the architecture over the tool

Statement: the architectural structure of an XF artifact derives from the properties of the formal process it automates, and not from the conventions, patterns or constraints imposed by the development framework, the programming language or the execution platform employed. The development framework is the instrument by means of which the architecture is implemented; it is neither its origin nor its determinant.

Motivation The development model predominant in the software industry implicitly establishes a relation of subordination between the architecture and the tool: the developer chooses a development framework and adopts the architecture that that framework prescribes or favours. Spring prescribes an architecture based on controllers, services and repositories managed by an inversion-of-control container. Angular prescribes an architecture based on modules, components and services with dependency injection. Flutter prescribes an architecture based on widgets, providers and blocs. In all these cases, the tool precedes the architecture — the developer does not design the architecture of the artifact, but rather inherits it from the framework.

This subordination has consequences that go beyond mere stylistic preference. When the architecture of the artifact is determined by the development framework, design decisions become coupled to technological decisions. A change of framework — motivated by technological obsolescence, by changes in performance requirements or by evolution of the ecosystem — implies not only a migration of code, but a complete architectural restructuring. The architectural knowledge accumulated by the team during the development of the artifact is not transferable to the new technology because that knowledge is expressed in terms proprietary to the previous framework.

Brooks [7] identified **accidental complexity** — that introduced by the tools and the development processes, not by the problem itself — as one of the principal causes of failures in software projects. The subordination of the architecture to the tool is a systemic source of accidental complexity: it introduces into the artifact structural decisions that do not derive from the business domain but from the conventions of the framework, increasing the conceptual distance between the process that the artifact models and the code that implements it.

The XF model inverts this relation explicitly and normatively. The architecture of the artifact — its organization into layers, the classification of its components, the dependency rules between them — derives from the properties of the formal process it automates, properties that are independent of any technology. The development framework is the instrument with which that architecture is implemented, not its origin. This inversion is not a methodological preference — it is a direct consequence of the principle of isomorphism between process and artifact established in §5.1.

Normative implications The first implication is that the adoption of the XF model in a specific development framework requires a work of **technological mapping**: identifying the mechanisms that the framework provides and determining how they may be used to implement each element of the model — layers, component types, injection components, folder structure — without adopting the proprietary architecture of the framework as a substitute for the XF architecture. This mapping

is specific to each framework and is the subject of the complementary per-language compatibility document, external to the present standard.

The second implication affects the treatment of the proprietary constructs of the framework. Every development framework provides proprietary constructs — annotations, decorators, base classes, interfaces — that implement framework-specific mechanisms. The XF model does not prohibit the use of these constructs, but it prescribes that their use remain **encapsulated** within the corresponding XF component, so that the proprietary construct is not visible outside the component that uses it. A component of the Access Layer may use the `@Repository` annotation of Spring to benefit from the automatic transaction management that Spring provides, but that annotation shall remain confined in the corresponding access component and shall not propagate to components of other layers. The artifact uses the framework; the framework does not determine the architecture of the artifact.

The third implication is about the **order of design decisions**. In an XF artifact, the analysis of the formal process to be automated — its decomposition into stages, the identification of the domain concepts, the definition of the business operations — precedes any decision on the implementation technology. The choice of the development framework is a subsequent decision, subordinate to the architectural requirements of the artifact and not the other way round. This order of decisions is what guarantees that the architecture of the artifact derives from the process and not from the tool.

The framework as implementer of the architecture The relation that the XF model establishes between architecture and tool may be characterized with precision through an analogy with consolidated engineering disciplines. A building architect designs the structure, the distribution and the properties of the building independently of the materials with which it is to be built — the architectural design precedes the choice of materials and determines them, not the other way round. Likewise, in the XF model the architectural design of the artifact precedes the choice of the development framework and determines it: the framework is the material with which the designed architecture is built, not the origin of that architecture.

This analogy is not merely illustrative. Parnas [39] established that the design decisions with the greatest impact on the maintainability of a system are those that are most likely to change during the life cycle of the system. In software development, development frameworks change more frequently than the business domains that the artifacts model — the business processes of an organization evolve gradually, whereas the technological ecosystem may change radically in short periods. An artifact whose architecture derives from the business process and not from the framework is, therefore, structurally more resistant to technological changes than one whose architecture is determined by the framework.

Scope and limits of the principle The precedence of the architecture over the tool does not imply ignorance of the capabilities and constraints of the framework. A good architectural design in the XF model takes advantage of the capabilities of the chosen framework to implement efficiently the elements that the model prescribes — the connection management mechanisms of the framework for the access components, the state management mechanisms for the business components,

the routing mechanisms for the interaction components. What the principle prescribes is that these capabilities be used in the service of the XF architecture, and not that they determine its structure.

Likewise, when a framework imposes constraints that hinder the implementation of some element of the model — for example, the automatic management of the life cycle in development frameworks with automatic dependency injection, which may enter into tension with the initialization mechanism prescribed by the injection components — the principle of precedence of the architecture establishes that the solution shall be sought in the way of implementing the spirit of the model within the constraints of the framework, not in the relaxation of the requirements of the model. The specific treatment of these situations lies outside the scope of the present standard. The following figure contrasts the traditional paradigm with that of the XF model.

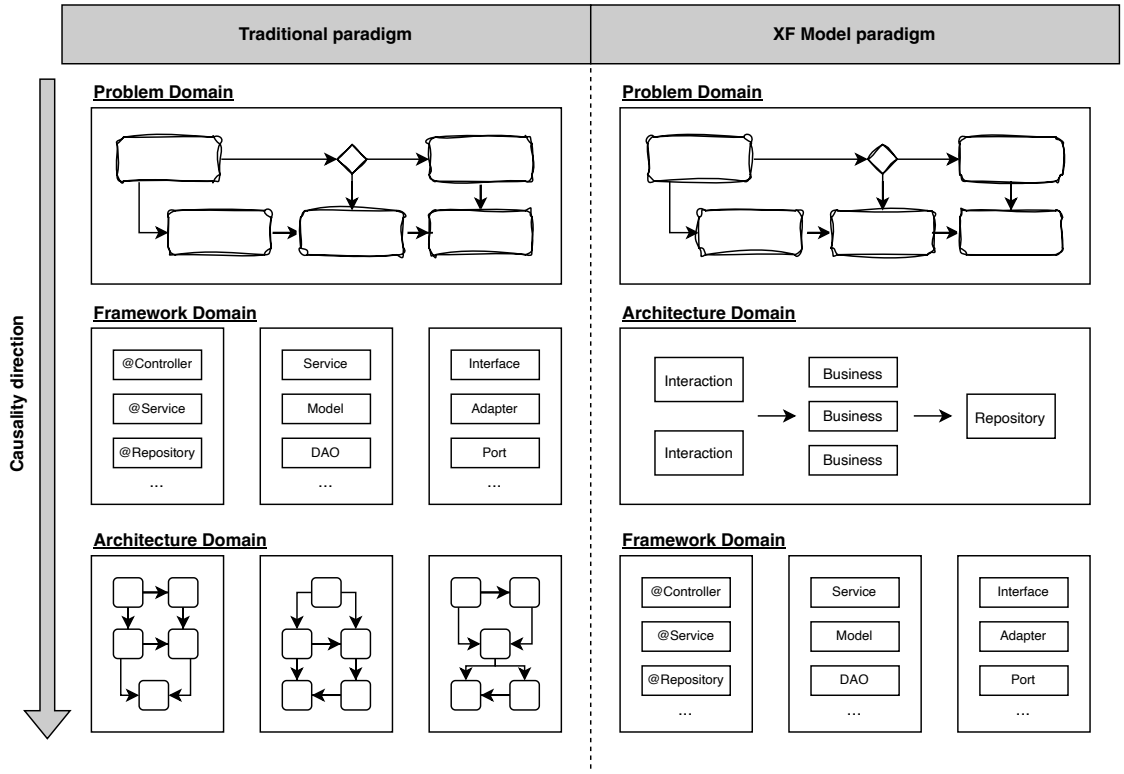

Figure 6. Inversion of the causality between architecture and development framework. In the traditional paradigm, the framework precedes and determines the structure of the artifact (Problem → Framework → Architecture); in the XF model, the architecture precedes and determines the use of the framework (Problem → Architecture → Framework).

Relation with other principles The precedence of the architecture over the tool is the condition that makes technology agnosticism possible (§6.2.1): if the architecture depended on the tool, the model could not be agnostic with respect to it. Likewise, this principle reinforces layer isolation (§6.2.2): when the architecture precedes the tool, the layers are defined in terms of the process — access, business, interaction — and not in terms of the constructs of the framework, which eliminates the risk that the proprietary conventions of the framework distort the layer structure of the artifact.

6.2.4 Closed and exhaustive typing

Statement: the XF model defines a component classification system that is simultaneously **closed** and **exhaustive**. It is closed because the set of component types recognized by the model is finite and non-extensible: five functional types distributed across three abstraction layers produce a classification space of fifteen possible categories, outside which the model recognizes no additional category. It is exhaustive because every component of a software artifact admits classification in exactly one of those fifteen categories: there is no component that lacks a valid classification in the model, and there is no component that requires belonging simultaneously to more than one category. Exhaustiveness is a structural property of the classification system — not an assertion of triviality. The classification of some components is not immediate and requires contextual analysis of their principal responsibility and of their mode of operation; the model provides canonical patterns for the non-trivial cases later in this same clause.

Motivation Closed and exhaustive typing is the property that confers on the XF model its capacity to eliminate the structural ambiguity documented in §5.4. An open classification system — in which development teams may define additional types according to their needs — reproduces exactly the problem that the model intends to solve: the proliferation of proprietary taxonomies incompatible between projects and teams. A non-exhaustive classification system — in which there exist components that cannot be classified — introduces structural grey zones that are a source of inconsistencies and of technical debt.

The property of closure and exhaustiveness of a type system is a classical foundation of type theory in software engineering. By analogy with the principle of behavioural subtyping — formalized by Liskov and Wing [27] in the domain of type systems — the correctness of a typed system depends on every instance of a type preserving the structural and behavioural properties declared by its type. The analogous extension to an architectural classification system is direct: a closed and exhaustive taxonomy makes it possible to verify deterministically that every component respects the properties of its category, whereas an open or non-exhaustive taxonomy introduces grey zones where the contract of the type is not binding. The XF model applies this principle to the component classification system: a classification system that cannot classify all the elements of its domain is an incomplete system, and an incomplete system cannot be the basis of a normative standard. The XF model establishes that its domain — the set of all the components of any software artifact — is completely covered by its classification system, and that this coverage is a property of the model that shall be preserved in any future evolution.

From a practical perspective, closed and exhaustive typing has a corollary of great operational value: it **obliges the developer to take an explicit design decision before implementing any component**. In the traditional development model, a developer may begin to implement a component without having decided with precision what type of logic it is going to contain — that decision is taken implicitly during implementation, with the risk that the result is a component with mixed responsibilities that violates the principle of separation of concerns. In the XF model, the classification of the component in a layer and a type shall precede its implementation, because that classification determines what logic it may contain and

on what other components it may depend. Typing is, in this sense, an instrument of design discipline as much as an instrument of structural organization.

Normative implications The normative implication of closed and exhaustive typing is that every component of the artifact shall be classified in exactly one of the foreseen categories, without residues or overlaps: no valid component falls outside the taxonomy. This property of closure distinguishes the model from the previous taxonomies of the discipline, as contrasted below.

Differentiation with respect to previous closed taxonomies The property of closure of the classification system has precedents in the discipline of Architecture Description Languages. Formal systems such as ACME [2] provide syntax to define closed sets of component types and connector types per project, leaving the choice of the concrete types to the criterion of the architect. The standard [22] establishes a closed metamodel of abstract concepts — view, viewpoint, concern — on which every architectural description must be built, without prescribing concrete component types.

The XF model is distinguished from these precedents in a decisive dimension: it prescribes a closed, exhaustive and language-agnostic set of *concrete types* of component — not a syntax to define them. The operational consequence is that an XF artifact has a valid and verifiable classification without dependence on the project, on the team or on the description framework adopted. The closed taxonomy of universal concrete types is the structural novelty of the model with respect to the state of the art.

The classification system The classification space of the XF model is defined by the Cartesian product of two finite sets:

- The set of layers: $\mathbf{L} = \{\text{Access, Business, Interaction}\}$, ordered by increasing level of abstraction.
- The set of functional types: $\mathbf{T} = \{\text{Logical, Generalization, Injection, Utility, Transfer}\}$.

The Cartesian product $L \times T$ produces fifteen possible classification categories. Every component of an XF artifact belongs to exactly one element of $L \times T$. The classification function $\phi : \mathbb{C} \to L \times T$, where $\mathbb{C}$ is the set of components of the artifact, is total — defined for every element of $\mathbb{C}$, such that no component belongs to more than one category.

Classification as a design decision prior to implementation The XF model prescribes that the classification of a component in a category of $L \times T$ is a decision that **shall** be taken before beginning its implementation. This prescription is not merely organizational — it has direct normative consequences on what the component may contain.

The classification in a layer determines what types of logic the component may implement: a component classified in the Access Layer implements exclusively logic of access to external systems; a component classified in the Business Layer implements

exclusively domain logic; a component classified in the Interaction Layer implements exclusively interaction logic. The presence in a component of logic corresponding to a layer distinct from the one in which it is classified constitutes a violation of the model, regardless of whether that logic is functionally correct.

The classification in a functional type determines what function the component performs within its layer: a component of logical type implements the effective logic of the layer; a component of generalization type abstracts common behaviour between logical components of the same layer; a component of injection type manages the life cycle of the logical components of the layer; a component of utility type provides auxiliary operations; a component of transfer type defines data structures. A component that does not satisfy the function corresponding to its classified type likewise constitutes a violation of the model.

Scope and limits of the principle

Treatment of non-classifiable components The XF model establishes that every component of a software artifact admits classification in some category of $L \times T$. However, in development practice teams may encounter components whose classification is not immediate. The model prescribes that in these cases the team **shall** apply one of the following three analyses before proceeding to the implementation.

The first is the **single-responsibility analysis**: verifying whether the component that cannot be classified is assuming responsibilities that correspond to more than one category of $L \times T$. If so, the component shall be decomposed into as many classifiable components as the distinct responsibilities it contains. This situation is the most frequent and its resolution through decomposition is the practice that the model favours.

The second is the **logic-completeness analysis**: verifying whether the component that cannot be classified is implementing incomplete logic that should be complemented in order to fit into one of the categories of the model. A component that performs data transformations but does not implement any business rule may not fit into the logical type, but could be completed in order to become a utility component or a transfer component, according to the nature of the transformations it performs.

The third is the **operational-context analysis**: when the responsibility of the component is single and its logic is complete but its classification continues not to be immediate, the model prescribes identifying the applicable canonical pattern according to the context in which the component operates. This clause documents the canonical patterns for the cases that most frequently generate doubt in practice.

Execution triggers An execution trigger is the logic that initiates the execution of an operation without that operation having been invoked explicitly by another component of the artifact at that instant. The model distinguishes two subtypes according to the origin of the trigger condition:

- **Autonomous** triggers: the trigger condition is internal to the component — elapsing of a time interval measured by the component itself, reaching of an internal state, completion of a previous internal process. The trigger logic

belongs to the same layer as the logical component that requires it. The autonomous trigger does **not** convert the component into an entry point of the Interaction Layer, because no interaction with an external consumer mediates: the Interaction Layer formalizes the protocol by which an external consumer invokes operations of the artifact, not any internal mechanism that starts them.

- Triggers based on **external signals**: the trigger condition is the reception of a signal from an external system through a communication protocol. The subscription to the external system and the reception of the signal are communication with the exterior; these components belong to the **Access Layer** and are classified as logical access components.

The distinction is relevant because the origin of the trigger condition determines the classification: if the condition is internal to the artifact, the trigger logic lives in the layer of the triggered component; if the condition comes from an external system, the trigger logic is access logic.

Contextual logic to multiple logical components Contextual logic — that which gives support to the effective logic without forming part of the principal purpose of any logical component — does not require any additional component type: it is classified within the five canonical types of the $L \times T$ matrix like any other logic of the artifact, according to its nature and its layer. The taxonomy remains closed and exhaustive, without additional types.

If, after the application of the three analyses — single responsibility, logic completeness and operational context — the component still cannot be classified in any category of $L \times T$, the model prescribes that its existence shall be submitted to architectural review, since the impossibility of classification indicates that the component does not conform to the principles of the model and constitutes a violation of exhaustive typing.

Relation with other principles Closed and exhaustive typing is the principle that makes the fulfilment of the others verifiable. Technology agnosticism (§6.2.1) is verifiable because the classification in $L \times T$ is independent of technology. Layer isolation (§6.2.2) is verifiable because the dependency rules are applied over precise classification categories. The precedence of the architecture over the tool (§6.2.3) is verifiable because the classification of all the components in categories derived from the process — and not from the framework — is a verifiable condition. Without closed and exhaustive typing, the three preceding principles would be declarative principles without a mechanism of verification.

This property of verifiability is what makes possible the definition of static conformance analysis tools — code analyzers, IDE plugins, conformance report generators — that can evaluate automatically the degree of fit of an artifact to the XF model. The formal and closed nature of the classification system is a necessary condition for these tools to be able to operate with precision and without ambiguity. The conformance rules of the model are founded directly on the properties of the classification system established in this principle.

7 Component taxonomy

The XF model's component taxonomy organizes every artifact into a two-dimensional matrix: the *abstraction dimension* — the layers — and the *functional dimension* — the component types —, whose intersection classifies each component and which is physically materialized in the project's folder structure. This clause first defines the matrix structure that articulates both dimensions, then each dimension separately and, finally, its canonical materialization in folders.

7.1 Matrix structure: layers and component types

The simultaneous application of the two classification dimensions defined in §6.1 — the abstraction dimension and the functional dimension — produces the fundamental organizational structure of the XF model: a matrix of three rows and five columns in which each row corresponds to an abstraction layer and each column corresponds to a functional component type.

Each cell of this matrix represents a valid component category in the model. Every component of an XF artifact occupies exactly one cell of the matrix — no more and no less. The matrix is therefore the canonical representation of the classification space $L \times T$ defined in §6.2.4, and constitutes the structural reference on which all of the model's conformance rules are founded.

The matrix structure is presented below in tabular form (Table 2).

Tabular representation The following table presents the complete matrix structure of the XF model, with the canonical nomenclature prescribed for each component category:

Table 2. Canonical naming pattern by component category. The mark `<N>` denotes the name of the concept modeled by a logical, utility or transfer component; `<P>` denotes the name of the structural pattern abstracted by a generalization. Injection components carry single-letter nomenclature — `A`, `B`, `R` — common to all XF artifacts, independent of the concept they house.

Layer \ Type	Logical	Generalization	Injection	Utility	Transfer
Interaction	`<N>Service` `<N>View`	`<P>Service` `<P>View`	`A`	`<N>Utils`	`<N>`
Business	`<N>Business`	`<P>Business`	`B`	`<N>Utils`	`<N>`
Access	`<N>Repository`	`<P>Repository`	`R`	`<N>Utils`	`<N>`

Notation: `<N>` represents the name of the concept that the component models — typically a domain name for logical, utility and transfer components. `<P>` represents the name of the structural pattern that the generalization component abstracts. Injection components are the only ones whose nomenclature is a single-letter constant — `R`, `B`, `A` — common to all XF artifacts.

Properties of the matrix structure The matrix structure has three formal properties that derive directly from the governing principles established in §6.2 and that shall be satisfied by every XF artifact.

The first property is **completeness**: every cell of the matrix may be populated in any XF artifact. There is no empty cell by definition — the absence of components in a particular cell within a particular artifact is a valid design decision, not a constraint of the model. A purely backend artifact may have no `View`-type components in the Interaction Layer; a simple artifact may not require generalization components in any layer. The matrix defines the space of the possible, not the set of the mandatory.

The second property is **classification uniqueness**: no component may belong simultaneously to more than one cell. The classification of a component is a total function from the set of components of the artifact into the classification space $L \times T$ — each component has exactly one image in the matrix.

The third property is the **per-row dependency constraint**: dependencies between logical components — Logical, Generalization and Injection types — are constrained by the row to which they belong, following the principle of isolation between layers established in §6.2.2. A component in the Interaction row may depend on components in the Business and Access rows, but never the other way around. Within a single row, all access between logical components is performed exclusively through the injection component of that row.

Relationships between columns In addition to the per-row constraints, the matrix structure establishes functional relationships between the columns that determine how the different component types interact within each layer.

Logical-type components are the functional core of each layer — they implement the effective logic that justifies the layer's existence. They are the only components that may hold mutable state in the artifact, and they are the only ones whose instances are managed by the **Injection** component of their layer. All access to a logical component from outside its layer is performed exclusively through the corresponding injection component.

Generalization-type components exist as a function of the logical components of their own layer — they abstract common behavior among them and have no usefulness independent of the logical components that inherit them. Their scope is strictly limited to the layer in which they are defined: a generalization component of the Business Layer shall not be inherited by a logical component of the Access Layer or of the Interaction Layer.

Injection-type components have a singular position in the matrix structure: they are the only point of access to the logical components of their layer from any other component of the artifact, including components of the same layer. There is exactly one injection component per layer — `R` for the Access Layer, `B` for the Business Layer, `A` for the Interaction Layer — and its definition determines which logical components are available in the artifact and in what order they are initialized.

Utility-type components provide stateless auxiliary operations without side effects on the data structures of their layer. They are not instantiable and do not hold state. Their scope is local to the layer in which they are defined, with the exception of the Access Layer utility components that operate on primitive types — strings, dates, numbers, decimals — formally inherited from the Presentation Layer of the OSI model (Level 6). These utility components over primitive types may be referenced

by components of any layer without this constituting a violation of the isolation principle, given that primitive types are prior to any semantic transformation of the artifact.

Transfer-type components define the data structures that flow between components. They contain no behavioral logic — their function is exclusively to model the form of the data at a specific point in the processing flow.

Canonical nomenclature The XF model prescribes a canonical nomenclature for each category of the matrix that shall be respected independently of the naming conventions of the programming language or development framework employed. This nomenclature is part of the model and its consistency is a necessary condition for the semantic interoperability between teams and projects that is one of the model's central objectives.

The logical components of the Access Layer shall carry the suffix `Repository`. The logical components of the Business Layer shall carry the suffix `Business`. The logical components of the Interaction Layer shall carry the suffix `Service` or `View` according to whether they are systemic or graphical interactions respectively. Utility components shall carry the suffix `Utils`. Transfer components carry no additional suffix — their name describes the concept of the artifact's domain that they model (business, access or interaction). Injection components receive the canonical names `R`, `B` and `A` respectively. Generalization components carry the prefix or the name of the pattern they abstract followed by the layer name — for example, `StatefulBusiness`, `RestRepository`, `FormView`.

The following table illustrates the $L \times T$ matrix classification on a concrete artifact (a thermostat).

Table 3. Concrete example of the $L \times T$ matrix structure applied to the thermostat artifact. Each cell contains the actual components of the artifact classified into their canonical category.

Layer \ Type	Logical	Generalization	Injection	Utility	Transfer

Interaction	`TemperatureService`
	`ThermometerView`
	`LoginView`
		`RestService`
		`FormView`
			`A`	`HttpUtils`	`Response`
					`User`

Business	`SessionBusiness` `TemperatureBusiness`	`StatefulBusiness`	`B`	`TemperatureUtils`	`Session` `Temperature`

Access	`IdentityRepository`
	`DatabaseRepository`
		`RestRepository`
		`SqlRepository`
			`R`	`StringUtils`
				`DateUtils`
					`User`
					`NetworkException`
					`AuthResponse`

7.2 Abstraction dimension: the layers

This clause defines, in normative terms, the three abstraction layers that make up the XF model — Access, Business and Interaction. It generalizes the process of each individual layer, defining its specific responsibility, the boundaries that separate it from the adjacent layers, the process model it prescribes for its internal logic and the structural organization it shall adopt in the project file system.

The stratification of the XF model is not an organizational convention — it is a direct consequence of the principle of isomorphism between process and artifact established in §5.1 and formalized in §5.5. Each layer corresponds biunivocally to an invariant stage of every formal process: the Access Layer models the access stage, the Business Layer models the processing stage and the Interaction Layer models the interaction stage. This correspondence determines precisely which logic belongs to each layer and, by exclusion, which logic does not belong to it — a classification criterion that is independent of the technology, the business domain and the scale of the artifact.

The three layers share the same internal structure of component types — defined in §7.3 — and the same dependency rules — defined in §6.2.2. What differentiates them is the nature of the logic they implement and the process model that this logic shall satisfy. The definition of that process model for each layer is the central object of this clause.

The clause is organized into three parallel and symmetric internal clauses, one per layer, each with the same internal structure: responsibility and boundaries, layer process model, subdivisions of logical components and internal organization. This symmetry is intentional — it reflects the structural symmetry of the model and facilitates navigation of the document by developers who seek information on a specific layer without needing to read the complete clause.

7.2.1 Access Layer

The Access Layer constitutes the lowest abstraction level of the XF Architecture and is the only layer of the model that has direct contact with the Presentation Layer of the OSI model (Level 6). Its position in the stratification is not arbitrary — it is a direct consequence of the formal derivation established in §5.5: the access stage of every formal process is that in which the process delegates part of its execution to external systems, and that delegation necessarily requires resolving the communication protocols before any business logic can operate on the obtained data.

Every datum that the artifact needs to obtain from an external system — regardless of whether that system is a relational database, a REST API, an operating-system file, a device connected via serial port or an operating-system service — enters the artifact through the Access Layer. Likewise, every datum that the artifact needs to send to an external system leaves the artifact through this layer. The Access Layer is, in this sense, the boundary between the artifact and its environment — the point at which the artifact ceases to be autonomous and needs to communicate with other systems to fulfil its function.

The logical components of the Access Layer — called Repositories — encapsulate the communication protocols so that the upper layers never know the details of how that communication is established. This protocol-abstraction property is what makes possible the principle of technological agnosticism of the XF model at the data level: a change in the communication protocol with an external system — for example, the migration from a REST API to GraphQL — remains confined to the corresponding Repository and does not require modifications in the Business Layer or in the Interaction Layer.

Responsibility and boundaries This clause delimits the responsibility of the Access Layer and its boundaries: the lower boundary with the data environment of the artifact and the upper boundary with the Business Layer.

Responsibility The responsibility of the Access Layer is to resolve, completely and autonomously, the communication protocols with the systems external to the artifact, to abstract the details of those protocols from the upper layers, and to provide the Business Layer with normalized data in transfer components that are independent of the protocol used to obtain them.

This responsibility has three dimensions that the XF model prescribes:

The first dimension is **protocol abstraction**: every detail relating to the communication protocol — connection strings, HTTP headers, SQL statements, serialization formats, authentication mechanisms with external systems — shall remain confined to the logical components of the Access Layer. No component of the Business Layer or of the Interaction Layer shall know the protocol used to obtain a datum. If a business component knows that the user data is obtained by means of a REST call to a specific endpoint, the protocol abstraction has been violated.

The second dimension is **data normalization**: the data obtained from external systems arrive at the artifact in the format that each external system defines — byte streams, JSON strings, database cursors, proprietary-protocol frames. The Access Layer is responsible for transforming those data into transfer components that are structured and interpretable by the Business Layer, without applying any semantic transformation to them. Normalization is a syntactic transformation — from protocol format to data structure — not a semantic transformation. Applying business rules to the data during normalization constitutes a violation of the responsibility of the layer.

The third dimension is **communication error management**: the Access Layer is responsible for detecting and propagating towards the upper layers the errors that indicate unexpected execution conditions in the communication with external systems. The concrete management of each type of error is an implementation decision of the developer — the XF model prescribes the responsibility, not the implementation.

Lower boundary — OSI Presentation Layer (Level 6) The lower boundary of the Access Layer is the Presentation Layer of the OSI model (Level 6), which provides the artifact with the primitive types (§3.1.15) that constitute the raw material on which the Access Layer operates. These primitive types are prior to any semantic transformation of the artifact and, as established in §7.3.4, the utility components that operate on them are defined in the Access Layer and may be referenced by any layer of the artifact without restriction.

Upper boundary — Business Layer The upper boundary of the Access Layer is the Business Layer, to which it provides normalized data in transfer components. The Access Layer does not know what the Business Layer does with those data — its responsibility ends at the moment it delivers a correctly normalized transfer component or propagates an exception indicating that the communication could not

be completed satisfactorily. All logic that decides what to do with the obtained data belongs to the Business Layer, not to the Access Layer.

Process model The access process model defines what functionally characterizes the logic that belongs to the Access Layer, so that every developer can determine unambiguously whether a specific piece of logic belongs to this layer or to another. The prescriptions of this clause are independent of the communication protocol employed, of the type of external system accessed and of the language or development framework used for the implementation.

The access process model does not prescribe how the communication with external systems shall be implemented — that decision corresponds to the developer as a function of the concrete protocol, the available libraries and the characteristics of the external system. The model prescribes which responsibilities every access component shall satisfy and which responsibilities are explicitly forbidden to it. Implementation guidance for the most common protocols is collected in Annex A of this document.

Prescribed responsibilities Every logical component of the Access Layer **shall** satisfy the following responsibilities independently of the protocol it implements:

The first responsibility is the **complete encapsulation of the protocol**. Every detail relating to the communication protocol — connection parameters, serialization formats, authentication mechanisms with external systems, management of the communication channel — shall remain confined to the access component. No component of the Business Layer or of the Interaction Layer can know the protocol used to obtain or send a datum. The violation of this responsibility is verifiable directly: if a business component imports a protocol library, references a URL, constructs an SQL statement or manipulates HTTP headers, the encapsulation has been violated.

The second responsibility is the **normalization of the received data**. The data obtained from external systems arrive at the artifact in the format that each external system defines. The access component is responsible for transforming those data into transfer components that are structured and interpretable by the Business Layer. This transformation is strictly syntactic — it converts the protocol format into a data structure — and shall not incorporate any semantic transformation of the data. An access component that filters, aggregates, enriches or semantically transforms the data during normalization is assuming responsibilities that belong to the Business Layer.

The third responsibility is the **propagation of usage-context errors**. The errors that indicate unexpected execution conditions in the communication with the external system — system unavailable, response with an unexpected format, data syntactically incoherent with respect to the protocol contract — shall be propagated towards the upper layers by means of exception transfer components. The access component shall not take business decisions based on them. The Business Layer is responsible for deciding how to react to a communication error — the access component shall limit itself to communicating it precisely.

Excluded responsibilities The XF model explicitly prescribes that the following types of logic do not belong to the Access Layer. Their presence in an access component constitutes a violation of the isolation principle established in §6.2.2:

An access component **shall not** apply business rules to the obtained data. No semantic transformation, domain validation, derived calculation or decision based on the meaning of a datum belongs to this layer. If an access component takes a decision based on the value of a business datum — and not on the response of the communication protocol — that logic belongs to the Business Layer.

An access component **shall not** manage the domain state. Keeping in memory the current state of a business concept, coordinating data among multiple sources or notifying observers of state changes are responsibilities of the Business Layer.

An access component **shall not** define interaction interfaces. No access component can define entry points to the artifact, service contracts or presentation logic of any kind.

An access component **shall not** invoke components of upper layers. The Access Layer shall not invoke operations of the Business Layer or of the Interaction Layer through their injection components. All upward communication — notification of state changes, system events — is resolved without direct upward invocation, typically by means of the observer pattern (§6.2.2).

Classification criterion The XF model prescribes the following operational criterion for determining whether a specific piece of logic belongs to the Access Layer. A piece of logic belongs to this layer if and only if it simultaneously satisfies the following three conditions:

The first condition is that it **knows a communication protocol with an external system** — it knows how to establish the communication with that system, regardless of whether it is remote or local to the device.

The second condition is that it **knows no business rule of the domain** — it does not take decisions based on the meaning of the data, only on the response of the communication protocol.

The third condition is that **its result is a normalized transfer component or a communication exception** — it does not produce results that imply semantic interpretation of the data or domain decisions.

If a piece of logic does not satisfy the first condition, it does not belong to the Access Layer. If it satisfies the first but does not satisfy the second or the third, it shall be decomposed: the part that resolves the protocol belongs to the Access Layer, and the part that applies semantics or takes domain decisions belongs to the Business Layer.

Reference terminology The XF model defines the following terms to describe precisely the elements that make up the access logic of an artifact. This terminology constitutes a common vocabulary that facilitates communication among developers, analysts and reviewers about the access logic of any XF artifact independently of the technology employed.

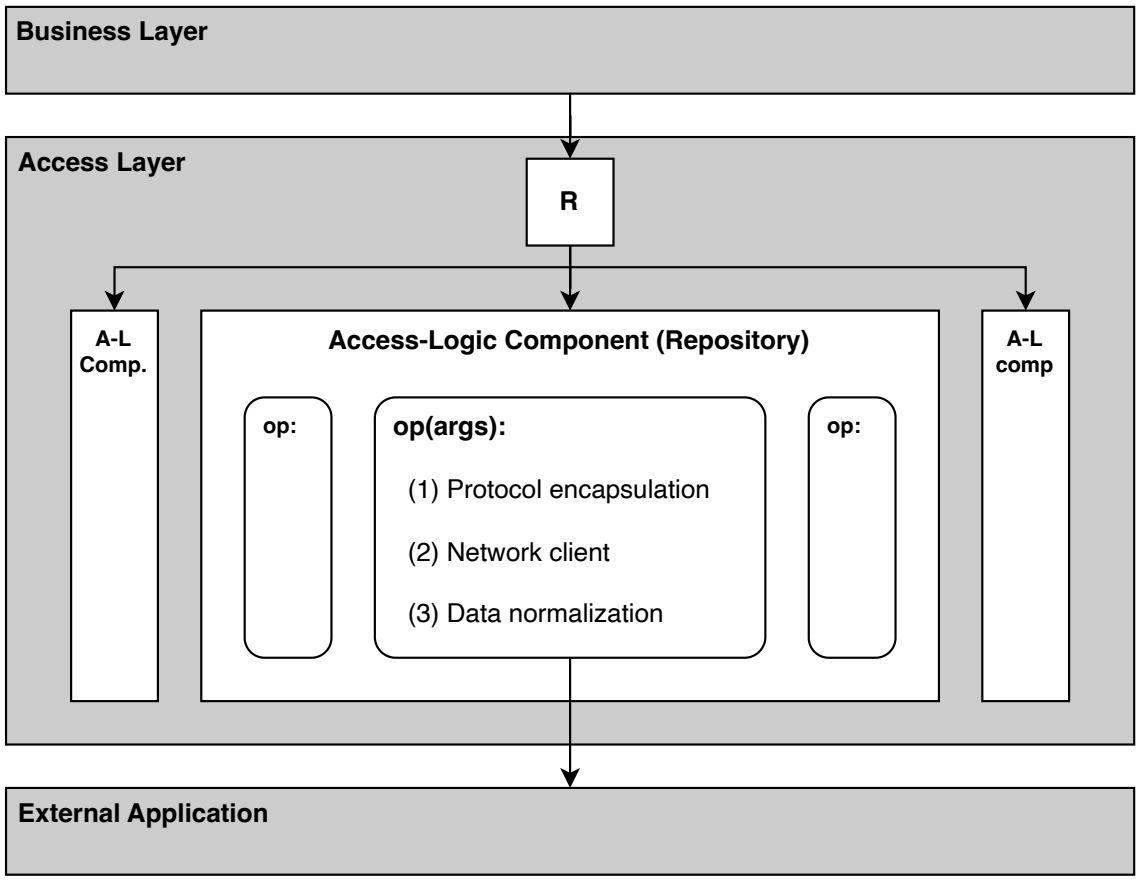

Figure 7. Prescribed content of a logical component of the Access Layer. The `Repository` encapsulates the communication protocol with an external system, receives invocations through the injection component `R`, syntactically normalizes the protocol data and delivers transfer components or communication exceptions to the Business Layer. No domain logic resides in this component; no upper-layer component is accessed from it.

Protocol abstraction is the term for the property of the logical components of the Access Layer by which the details of the communication protocol with an external system remain confined to the component and are invisible to the upper layers. Protocol abstraction is the structural property that makes possible the principle of technological agnosticism at the data level: a protocol change remains confined to the corresponding repository without requiring modifications in the Business Layer or in the Interaction Layer.

Data normalization is the term for the transformation that an access component applies to the data received from an external system in order to convert them into transfer components interpretable by the Business Layer. Normalization is a strictly syntactic transformation — from protocol format to data structure — without semantic transformation or application of domain rules.

Communication error propagation is the term for the transmission towards the upper layers, by means of exception transfer components, of the unexpected execution conditions detected during the communication with an external system. The Access Layer propagates the error without taking business decisions about it; the interpretation and response correspond to the Business Layer.

Communication exception is the term for the transfer component originated in the Access Layer that models an error condition in the communication with an external system — network failure, protocol error, system unavailability, invalid response. It is a transfer component $C_T \in \mathbb{C}_T$ like any other, transmitted typically as a language exception (§9.5).

Subdivisions of logical components The `/logic` folder of the Access Layer contains all the logical components — Repositories — of the artifact. In small-scale artifacts, this folder may contain a reduced number of components that do not require additional organization. However, as the artifact grows in functionality and the number of external systems with which it communicates increases, the `/logic` folder may come to contain a number of components sufficiently high to hinder its navigation and comprehension.

For these cases, the XF model permits — and recommends — the creation of internal subdivisions within the `/logic` folder that group the logical components according to the criterion that the development team considers most appropriate for its concrete artifact. These subdivisions are a design decision of the team — not a normative prescription of the model — and their implementation does not affect the conformance of the artifact.

Recommended subdivision The XF model proposes as a reference subdivision the division between local accesses and remote accesses, materialized in two subfolders within `/logic`:

- `/repository/logic/remote` — logical components that access systems located outside the device that executes the artifact, whose communication requires a communication peripheral.
- `/repository/logic/local` — logical components that access systems or resources located on the same device that executes the artifact, whose communication does not require an additional communication peripheral.

This subdivision is intuitive for most artifacts because it reflects a relevant functional difference between the components: remote accesses and local accesses have different communication characteristics, different types of errors and potentially different initialization patterns. Grouping them separately facilitates the location of components and orients the developer on the implications of each type before beginning the implementation.

However, the concrete criterion for classifying a component as local or remote in ambiguous cases — an operating-system service with a network interface, a database in a local container — is a decision of the implementer. The XF model does not prescribe how to resolve these cases.

Subdivision criterion The legitimate subdivision criterion in the Access Layer is the one derived from its technical responsibility: the resolution of communication protocols with external systems. Every internal subdivision of `/repository/logic` shall group the logical components according to properties of the communication they resolve — location (local or remote), type of protocol, characteristics of the channel —

and not according to the functional domain of the artifact. An artifact with multiple communication protocols may subdivide by protocol — `/repository/logic/rest`, `/repository/logic/database`, `/repository/logic/filesystem`. An artifact with heterogeneous accesses according to the nature of the channel may subdivide by characteristics of the communication — synchronous/asynchronous, connective/non-connective.

Subdivision by domain: non-conformant The subdivision of `/repository/logic` by functional domain of the artifact — for example, grouping logical components in subfolders whose name coincides with concepts of the domain modeled by the Business Layer — is not conformant to the XF model. The reason is structural: the access logic resolves communication protocol, not domain. Several domains of the artifact may share the same protocol, and subdividing by domain would imply replicating the same protocol resolution in several subfolders with the risk of redundancy and of mixing responsibilities. The domain is a legitimate subdivision criterion exclusively in the Business Layer (§7.2.2).

Formal conditions Every subdivision is valid provided that three conditions are respected. The first is that all the logical components of the Access Layer are located within the `/repository/logic` folder, directly or in one of its subfolders. The second is that no subfolder contains components of a type other than the logical one — the generalization, utility, transfer and injection components shall remain in their corresponding canonical folders regardless of the subdivision adopted for `/logic`. The third is that the adopted subdivision criterion belongs to the technical vocabulary of the layer — properties of the communication — and not to the vocabulary of the domain modeled by the Business Layer.

Absence of subdivision An artifact that does not implement any subdivision within `/logic` is fully conformant to the XF model. The absence of subdivision is the simplest organization and is appropriate for artifacts with a reduced number of logical components where the subdivision does not provide organizational value.

```
// Without subdivision - valid for small-scale artifacts
/repository/logic
├── DatabaseRepository
├── IdentityRepository
└── FileRepository

// With recommended subdivision - valid for larger-scale artifacts
/repository/logic
├── /remote
│   ├── DatabaseRepository
│   └── IdentityRepository
└── /local
    └── FileRepository

// With subdivision by protocol - equally valid
/repository/logic
├── /rest
│   └── IdentityRepository
├── /database
│   └── DatabaseRepository
└── /filesystem
    └── FileRepository
```

Listing 1. Absence of subdivision in the Access Layer

Internal organization The Access Layer is materialized in the project file system under the canonical folder `/repository`, whose internal structure follows the organization prescribed in §7.4. This clause consolidates that organization applied to the specific context of the Access Layer, describing the function of each subfolder and its relation to the prescriptions of the process model defined in §7.2.1.

```
/repository
├── /general (Generalization components)
├── /logic (Logical components)
│   ├── /local (Local accesses -- recommended)
│   └── /remote (Remote accesses -- recommended)
├── /transfers (Transfer components)
├── /utils (Utility components)
└── R (Injection component)
```

Listing 2. Internal organization of the Access Layer

`/general` — Generalization components The `/general` folder contains the generalization components of the Access Layer — the abstractions of structural behavior common among repositories. As established in §7.3.2, these components do not implement effective logic of any concrete access — they abstract contextual protocol logic that is shared by several repositories of the same layer, freeing the concrete logical components to concentrate exclusively on their effective logic: the establishment of the connection with the concrete external system and the obtaining or updating of its specific data.

The protocol generalization forms an **inheritance tree**: under the base generalization `Repository` hang, as siblings, the generalizations of the different protocols that the artifact uses — for example `RestRepository` and `FileRepository` —; under each protocol generalization hang, also as siblings, the generalizations of the different clients that share that protocol; and under each client hang the concrete logical components. A generalization is justified only if it abstracts behavior *shared by at least two descendants*: adding an intermediate level with a single child introduces a linear chain without reuse and does not constitute a legitimate generalization. A typical tree could be the following:

```
Repository (base behavior of any access)
├── RestRepository (generic HTTP/REST protocol)
│   ├── MyRestRepository (client of the MyApi server:
│   │       authentication, request structure,
│   │       response structure { code, description, data })
│   │   ├── UserRepository (logical -- User endpoints and structures)
│   │   └── PetRepository  (logical -- Pet endpoints and structures)
│   └── AuthRepository (logical -- client of the authentication server)
└── FileRepository (local-file access protocol)
    └── SessionRepository (logical -- persists the session on disk)
```

Listing 3. Inheritance tree of the Access Layer

In this tree, `UserRepository`, `PetRepository`, `AuthRepository` and `SessionRepository` are the concrete logical components — they inherit all the contextual protocol logic from their generalizations and implement only their effective logic: the specific endpoints, the concrete schemas and the data structures proper to each resource. The more shared generalization exists in the upper levels, the purer the effective logic of the concrete logical components.

The shape of the tree is a decision of the implementer and depends on the diversity of protocols and of clients that the artifact uses. An artifact that accesses a single external system with a simple protocol may dispense entirely with protocol generalizations; an artifact that combines several protocols and several clients per protocol builds a richer tree. Both options are conformant to the XF model provided that the isolation principle is respected — all the components of the tree belong to the Access Layer and no node can contain logic that belongs to another layer — and the legitimate-generalization principle — no intermediate node with a single child.

`/logic` — Logical components The `/logic` folder contains the Repositories of the artifact — the logical components of the Access Layer. Each repository encapsulates the communication with a concrete external system or with a logical set of data of an external system, satisfying the responsibilities prescribed in §7.2.1: complete encapsulation of the protocol, normalization of the received data and propagation of usage-context errors.

The effective logic of each repository — the establishment of the connection and the obtaining or updating of the data of the concrete external system — is what distinguishes a logical component from a generalization component in this layer. A component that only defines protocol behavior without implementing any concrete access to any specific resource is a generalization component, not a logical component.

The internal organization of `/logic` — with or without subdivisions — follows the prescriptions of §7.2.1. The concrete content of this folder is specific to each artifact and depends on the external systems with which the artifact communicates.

`/transfers` — Transfer components The `/transfers` folder contains the transfer components defined in the Access Layer — the data structures that model the information in its form closest to the external system that originates it, before any semantic transformation by the Business Layer.

In the context of the Access Layer, the typical transfer components model the information as the external system defines it: the entities that a database persists, the structures that an external API publishes in its responses, the data formats that a connected device transmits. It is equally usual to find in this folder auxiliary protocol structures — response wrappers, message headers, error structures specific to the external system — that are necessary for the implementation of the protocol but have no direct correspondence with concepts of the business domain.

As prescribed in §7.3.5, these structures may be referenced directly by components of the Business Layer provided that they are not modified. If the Business Layer needs to transform a structure defined in this folder, it shall define its own transfer component in `/business/transfers`.

`/utils` — Utility components The `/utils` folder contains the utility components of the Access Layer — stateless auxiliary operations that serve as support to the repositories for the achievement of their access logic.

In the context of the Access Layer, the typical utility components provide operations of data serialization and deserialization — `JsonUtils`, `XmlUtils` —, operations

of query-parameter construction — `QueryUtils` — and operations on primitive types — `StringUtils`, `DateUtils`, `NumberUtils`. As established in §7.3.4, the utility components on primitive types defined in this folder may be referenced by components of any layer of the artifact without restriction, given that the primitive types are prior to any semantic transformation of the artifact.

`R` — Injection component The injection component `R` is located at the root of the `/repository` folder and constitutes the only access point to the repositories of the artifact from any other component. Its definition, properties and life cycle are developed in §7.3.3. In the context of the Access Layer, R is the point through which the Business Layer accesses all the external data that the artifact needs — every invocation of a repository from a business component follows the pattern `R.<repository>.<operation>()`.

The location of `R` at the root of `/repository` — at the same level as the subfolders of component types — reflects its transversal nature within the layer: it aggregates and gives access to all the logical components of the layer without belonging to any subfolder of a concrete type.

7.2.2 Business Layer

The Business Layer constitutes the intermediate abstraction level of the XF Architecture and is the layer that defines the functional identity of the artifact. If the Access Layer resolves how the artifact communicates with the exterior, and the Interaction Layer resolves how the artifact is consumed by its users and external systems, the Business Layer resolves what the artifact does — what its purpose is, which domain concepts it manages and which operations on those concepts are possible.

It is in this layer that the ontology of the solution resides: the structured set of concepts, relationships and operations that describe the domain that the artifact automates. A change in the functional requirements of the artifact — a new business rule, a new use case, a new domain constraint — always manifests as a change in the Business Layer. The other layers may change for technological reasons — a new access protocol, a new interaction channel — but the Business Layer changes for domain reasons.

The logical components of the Business Layer — termed Businesses — are the functional core of the artifact. Although every logical component, regardless of its layer, may hold state in the formal sense defined in §7.3.1, the state of Access logical components models conditions of the communication with external systems and that of Interaction logical components models conditions of the protocol or of the interface; only the state of Businesses establishes a correspondence with the concepts, relationships and operations of the domain that the artifact automates. This ontological exclusivity of the Business Layer state is what constitutes it as the semantic memory of the artifact during its execution.

Responsibility and boundaries This clause delimits the responsibility of the Business Layer and its boundaries: the lower boundary with the Access Layer and the upper boundary with the Interaction Layer.

Responsibility The responsibility of the Business Layer is to implement the effective logic of the domain of the artifact — the operations, transformations and rules that constitute the functional value of the system — abstracting both the details of communication with external systems and the details of presentation and interaction. The Business Layer receives normalized data from the Access Layer, applies the domain logic to it and provides results to the Interaction Layer through well-defined operations.

This responsibility has three dimensions that the XF model prescribes:

The first dimension is the **implementation of the domain ontology**: the Business Layer defines the set of concepts that the artifact manages and the operations that are possible on them. Each business logical component models exactly one domain concept — a user, a session, a temperature, an order — and defines the operations that transform, validate or query that concept. The set of all business logical components constitutes the complete ontology of the solution.

The second dimension is the **management of the domain state**: although every logical component may hold state in the formal sense of the model, the Business Layer is the only one in which that state establishes a correspondence with the domain ontology of the artifact. The state of Access logical components models conditions of the communication with external systems and that of Interaction logical components models conditions of the protocol or of the interface; neither of them models domain concepts. The state of a Business represents the dynamic context of the domain concept that it models during the execution of the artifact and is accessible exclusively through the injection component `B`.

The third dimension is the **abstraction of the data source**: the Business Layer does not know the protocol used to obtain the data that it processes. All communication with external systems is abstracted in the Access Layer and is accessible through `R`. This property guarantees that a change in the access protocol does not require modifications in the Business Layer — the business component invokes `R.<repository>.<operation>()` regardless of whether that repository accesses a relational database, a REST API or a file of the system.

Lower boundary — Access Layer The lower boundary of the Business Layer is the Access Layer, from which it consumes data normalized into transfer components through the injection component `R`. The Business Layer does not know the details of how that data is obtained — its responsibility begins at the moment it receives a normalized transfer component or a communication exception coming from the Access Layer.

Upper boundary — Interaction Layer The upper boundary of the Business Layer is the Interaction Layer, to which it provides well-defined domain operations through the injection component `B`. The Business Layer does not know how its operations are invoked nor how its results are presented — its responsibility ends at the moment it delivers a result or propagates a business exception.

Process model The business process model defines what functionally characterizes the logic that belongs to the Business Layer, so that every developer can determine

without ambiguity whether a concrete piece of logic belongs to this layer or to another. As in §7.2.1, the model prescribes responsibilities and constraints — not concrete implementations. Implementation guidance is collected in Annex B.

In this clause the XF model introduces a reference terminology to describe the concepts of the Business Layer with precision. This terminology is not a normative prescription on the internal nomenclature of teams — it is the vocabulary that the model uses to describe the elements of the layer and that allows teams to communicate about the business logic of an artifact with precision and without ambiguity, regardless of the technology employed.

Prescribed responsibilities Every logical component of the Business Layer **shall** satisfy the following responsibilities:

The first responsibility is the **implementation of the effective logic of the domain**. The effective logic of a business component is the minimal set of operations that justify its existence in the artifact — those that directly resolve the use cases of the domain concept that the component models and that cannot be abstracted into any other component without losing the specificity of that concept. A business component that manages user authentication defines the operations “verify credentials”, “refresh session” and “expire session” — these are effective operations because they directly resolve the authentication problem. The management of the access protocol to the identity server or the storage of the session on disk are contextual logic that belongs to other types of components.

The second responsibility is the **management of the execution preconditions**. Before executing the part of the effective logic that depends on them, every business component verifies the conditions necessary for the operation to be able to complete correctly. These conditions — termed **business conditions** in the terminology of the model — check the state of the component and the input parameters of the operation. When a condition is not satisfied, the operation usually propagates an exception to the invoker; other forms of management — default value, retry, defined fallback logic — are admissible when the semantics of the process justify it. The XF model recommends that conditions be evaluated as soon as possible in the operation, ideally at the start when the nature of the process permits, so that the anomalous condition is transmitted without executing unnecessary logic.

The third responsibility is the **delegation toward the Access Layer**. When the effective logic of a business component requires data from external systems, that retrieval shall be delegated to the logical components of the Access Layer through the injection component `R`. A business component never accesses external systems directly — it invokes `R.<repository>.<operation>()` and receives the result as a normalized transfer component or a communication exception.

The fourth responsibility is the **management of business exceptions**. When a business operation cannot complete because a business condition is not satisfied, or because a dependency of the Access Layer has propagated a communication exception, the business component decides the appropriate response according to the semantics of the process: it typically propagates toward the Interaction Layer a business exception that describes with precision the cause of the failure, but other responses — default value, retry, local management of the failure — are admissible

when the process requires it. The XF model recommends explicit propagation as the preferred practice to preserve the traceability of the failure, but does not impose it as a structural obligation.

Excluded responsibilities The XF model explicitly prescribes that the following types of logic do not belong to the Business Layer:

A business component **shall not** contain communication protocol logic with external systems. All communication with external systems belongs to the Access Layer and shall be accessed through `R`. If a business component imports a protocol library, builds an SQL statement or manipulates HTTP headers, the responsibility of the layer has been violated.

A business component **shall not** contain presentation logic nor definition of interaction interfaces. The composition of graphical elements, the definition of service contracts or the syntactic validation of input parameters belong to the Interaction Layer.

Classification criterion The XF model prescribes the following operational criterion for determining whether a concrete piece of logic belongs to the Business Layer. A piece of logic belongs to this layer if and only if it simultaneously satisfies the following three conditions:

The first condition is that it **knows at least one domain concept of the artifact** — it operates on data that has a direct correspondence with the ontology of the solution.

The second condition is that it **knows no communication protocol with external systems** — it does not know how the data on which it operates is obtained, it only operates on it.

The third condition is that it **does not know how its results are presented or exposed** — it does not know whether its results will be shown in a graphical interface, returned in an HTTP response or persisted in a file.

If a piece of logic does not satisfy the first condition, it does not belong to the Business Layer. If it satisfies the first but does not satisfy the second or the third, it shall be decomposed: the part that knows the protocol belongs to the Access Layer, and the part that knows the presentation belongs to the Interaction Layer.

Reference terminology The XF model defines the following terms to describe with precision the elements that compose the business logic of an artifact. This terminology constitutes a common vocabulary that facilitates communication between developers, analysts and reviewers about the business logic of any XF artifact regardless of the technology employed.

The term **business ontology** denotes the conceptual model that defines the structured set of key concepts handled by the artifact, their relationships, the operations permitted on them and the rules that govern them. The business ontology is the formal description of the domain that the artifact automates.

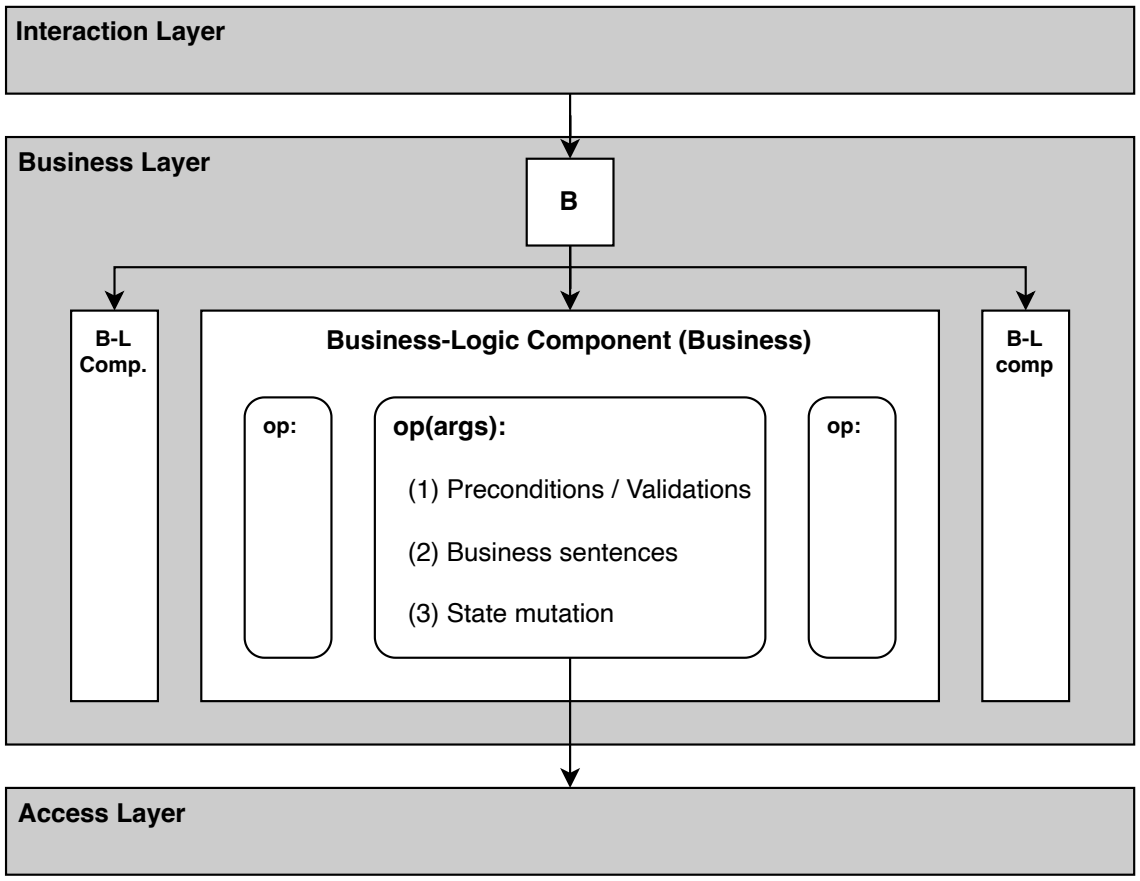

Figure 8. Prescribed contents of a logical component of the Business Layer. The `Business` implements the effective logic of the domain concept that it models: it receives invocations through the injection component `B`, evaluates business conditions, executes domain operations, maintains the state of the concept and propagates results or business exceptions. It delegates operations of access to external systems to logical components of the Access Layer through `R`. No communication protocol nor presentation logic resides in this component.

The term **business component** denotes each of the key concepts of the ontology that has representation as a logical component in the artifact. A business component models exactly one domain concept and defines the operations that are possible on it.

The term **business structure** denotes the set of structured data that represents the information managed by a business component and that has a direct correspondence with the characteristics, facts or values of the concept that it models. Business structures are implemented as transfer components of the Business Layer.

The term **business operation** denotes each of the subsets of the effective logic of a business component that is executed atomically upon an event or invocation, producing a transformation of the state of the component, a result for the invoker, or both.

The term **business statement** denotes each of the logical steps that compose a business operation. Statements are executed in an ordered manner and their set constitutes the complete implementation of the operation.

The term **business condition** denotes the specific statements of an operation that evaluate the state of the component or the input parameters before executing the effective logic. Conditions constitute the execution preconditions of the operation.

The term **business exception** denotes the transfer component that the Business Layer typically transmits as a language exception when a business condition is not satisfied or a dependency of the Access Layer cannot be resolved. The distinction is of the syntactic construct of transmission (§9.5), not of domain: the business exception is a transfer component $C_T \in \mathbb{C}_T$ like any other.

Subdivisions of logical components The `/logic` folder of the Business Layer contains all the business logical components — the Businesses — of the artifact. In small-scale artifacts, this folder may contain a reduced number of components that do not require additional organization. However, as the ontology of the artifact grows in complexity and the number of managed domain concepts increases, the `/logic` folder may come to contain a number of components sufficiently high to hinder its navigation and comprehension.

For these cases, the XF model permits — and recommends — the creation of internal subdivisions within the `/logic` folder that group the logical components according to the criterion that the development team considers most appropriate for its concrete artifact. These subdivisions are a design decision of the team — not a normative prescription of the model — and their implementation does not affect the conformance of the artifact.

Recommended subdivision The XF model proposes as a reference subdivision the division between instance businesses and device businesses, materialized in two subfolders within `/logic`:

- `/business/logic/instance` — logical components that model the specific ontology of the solution that the artifact automates. They are the businesses that define the proper domain of the artifact — the concepts, the rules and the operations that constitute the functional value of the system. Typical examples: `TemperatureBusiness`, `UserBusiness`, `SessionBusiness`, `ScheduleBusiness`.
- `/business/logic/device` — logical components that model processes relative to the device that executes the artifact or to the peripherals connected to it. They are the businesses that manage the capabilities of the execution environment — connectivity, location, language, encryption, operating system permissions. Typical examples: `NetworkBusiness`, `LocationBusiness`, `KeychainBusiness`, `LanguageBusiness`.

The usefulness of this subdivision is twofold. On the one hand, it facilitates orientation in artifacts with a high number of logical components, separating the businesses that model the domain of the solution from those that manage the execution environment. On the other hand, it favors reuse: instance businesses may be replicated in other artifacts that model the same domain ontology, while device businesses may be replicated in artifacts executed on devices with the same characteristics, regardless of the business domain.

The concrete criterion for classifying a component as instance or device in ambiguous cases is a decision of the implementer. A component that manages the configuration of the artifact may be considered instance — if the configuration is proper to the domain — or device — if it depends on the capabilities of the execution environment. The XF model does not prescribe how to resolve these cases.

Subdivision criterion The Business Layer is the **only layer of the XF model in which the functional domain is a legitimate subdivision criterion**. The reason is structural: the business logic is the only logic of the artifact that has as its object the modeled domain — the other two layers resolve technical problems (protocol and input type) that are orthogonal to the domain. An artifact with an extensive domain ontology may subdivide `/logic/instance` by subdomains — `/business/logic/instance/user`, `/business/logic/instance/temperature`, `/business/logic/instance/schedule`. An artifact with multiple device integrations may subdivide `/logic/device` by capability type — `/business/logic/device/connectivity`, `/business/logic/device/security`. The compatibility of the XF model with domain modeling methodologies — in particular with organization by subdomains or bounded contexts — materializes precisely at this level: within `/business/logic/instance`, not in the top-level structure of the artifact nor in the internal subdivision of the other layers.

Formal conditions Every subdivision is valid provided that three conditions are respected. The first is that all the logical components of the Business Layer are located within the `/business/logic` folder, directly or in one of its subfolders. The second is that no subfolder contains components of a type other than logical — the generalization, utility, transfer and injection components shall remain in their corresponding canonical folders. The third is that the adopted subdivision criterion belongs to the vocabulary of the domain modeled by the Business Layer — the only layer of the XF model where the functional domain is a legitimate subdivision criterion.

Absence of subdivision An artifact that does not implement any subdivision within `/logic` is fully conformant to the XF model. The absence of subdivision is the simplest organization and is appropriate for artifacts with a reduced number of logical components where the subdivision does not provide organizational value.

```
// Without subdivision - valid for small-scale artifacts
/business/logic
├── TemperatureBusiness
├── UserBusiness
├── SessionBusiness
└── NetworkBusiness

// With recommended subdivision - valid for larger-scale artifacts
/business/logic
├── /instance
│   ├── TemperatureBusiness
│   ├── UserBusiness
│   └── SessionBusiness
└── /device
    └── NetworkBusiness

// With subdivision by subdomain - equally valid
/business/logic
├── /instance
```

```
│   ├── /user
│   │   ├── UserBusiness
│   │   └── SessionBusiness
│   └── /temperature
│       └── TemperatureBusiness
└── /device
    └── NetworkBusiness
```

Listing 4. Absence of subdivision in the Business Layer

Internal organization The Business Layer materializes in the file system of the project under the canonical folder `/business`, whose internal structure follows the organization prescribed in §7.4. This clause consolidates that organization applied to the specific context of the Business Layer, describing the function of each subfolder and its relationship with the prescriptions of the process model defined in §7.2.2.

```
/business
├── /general (Generalization components)
├── /logic (Logical components)
│   ├── /instance (Instance businesses -- recommended)
│   └── /device (Device businesses -- recommended)
├── /transfers (Transfer components)
├── /utils (Utility components)
└── B (Injection component)
```

Listing 5. Internal organization of the Business Layer

`/general` — Generalization components The `/general` folder contains the generalization components of the Business Layer — the abstractions of structural behavior common between business logical components. As established in §7.3.2, these components do not implement effective logic of any concrete domain concept — they abstract structural contextual logic that is shared by several logical components of the same layer, freeing the concrete logical components to concentrate exclusively on their effective logic.

As in the Access Layer, generalization in the Business Layer may operate at multiple levels of specificity, forming inheritance chains where each level abstracts a more concrete aspect of the structural behavior. The most usual structural patterns in this layer are the following:

`Business` — base generalization component that abstracts the most general structural behavior of any business logical component. It may contain logic common to all the businesses of the artifact — trace logging, common error management, initialization patterns.

`StatefulBusiness<T>` — generalization component that abstracts the pattern of observation and notification of mutable state changes. The logical components that inherit it — `TemperatureBusiness`, `PetBusiness` — have observable data whose mutation automatically notifies the registered observers. This pattern is independent of `ScheduledBusiness` — a logical component may inherit from one, from both or from neither according to its needs.

`ScheduledBusiness` — generalization component that abstracts the pattern of periodic execution without necessarily implying observable state. The logical compo-

nents that inherit it execute logic automatically at defined time intervals — data synchronization, cache updating, periodic state checks.

`BroadcastBusiness` — generalization component that abstracts the pattern of data coordination between multiple sources — for example, persisting a datum simultaneously in local cache and in remote server, choosing the most up-to-date source on each read.

A typical inheritance chain in the Business Layer could be the following:

```
Business (base behavior of any business logical component)
├── StatefulBusiness<T> (management of observable state and notification to
    observers)
│   └── TemperatureBusiness (logical -- management of the Temperature concept)
├── ScheduledBusiness    (automatic periodic execution)
└── BroadcastBusiness    (coordination of data between multiple sources)
```

Listing 6. Canonical generalization patterns in the Business Layer

These patterns are illustrative of the typical content of `/general` in the Business Layer, not a prescriptive list. The concrete content of this folder depends on the structural patterns that the artifact needs to abstract and on the degree of reuse that the team considers appropriate.

It is important to recall that the principle of layer isolation applies with the same force to the business generalization components as to the logical components. The structural patterns that are necessary both in the Business Layer and in the Access Layer — such as the state observation pattern or the periodic task pattern — shall be implemented independently in each layer. `StatefulBusiness` and `StatefulRepository` are independent implementations of the same pattern — they do not share code between layers.

`/logic` — Logical components The `/logic` folder contains the Businesses of the artifact — the logical components of the Business Layer. Each business models exactly one domain concept of the artifact and defines the operations that are possible on that concept, satisfying the responsibilities prescribed in §7.2.2: implementation of the effective logic of the domain, management of preconditions, delegation toward the Access Layer and propagation of business exceptions.

The effective logic of a business logical component — the operations that justify its existence in the artifact — is what distinguishes it from a generalization component in this layer. A component that only defines generic structural behavior without modeling any concrete domain concept is a generalization component, not a logical component.

The granularity of the business logical components — how many domain concepts are grouped into a single component or separated into several — is a decision of the implementer that the XF model does not prescribe. As established in §7.2.2, both a single `UserBusiness` that manages all aspects of the user and the separation into `UserBusiness`, `UserProfileBusiness` and `UserSessionBusiness` are equally valid. What the model prescribes is that all the business logical components be accessible exclusively through the injection component `B` and that each one model a domain concept with a well-defined responsibility.

The internal organization of `/logic` — with or without subdivisions — follows the prescriptions of §7.2.2.

`/transfers` — Transfer components The `/transfers` folder contains the transfer components defined in the Business Layer — the data structures that model the information in its semantic domain form, after the structural transformations applied by the Business Layer on the data received from the Access Layer.

In the context of the Business Layer, the typical transfer components model the domain concepts as they are managed by the business logic of the artifact. When the Business Layer needs a structure with a definition different from the one provided by the Access Layer — removing sensitive attributes, adding calculated attributes or renaming attributes to adjust them to the semantics of the domain — it defines that new structure in this folder.

As prescribed in §7.3.5, the obligation to define a new structure is activated only when the structural definition is modified — adding or removing attributes. The modification of values on an existing structure does not require the definition of a new structure. The structures defined in this folder may be referenced directly by components of the Interaction Layer provided that they are not structurally modified.

It is equally usual to find in this folder the business exceptions — transfer components that are typically transmitted as language exceptions and model error conditions proper to the domain of the artifact: `TemperatureOutOfRangeException`, `SessionExpiredException`, `NetworkUnavailableException`.

`/utils` — Utility components The `/utils` folder contains the utility components of the Business Layer — stateless auxiliary operations that serve as support to the business logical components for the achievement of their effective logic without belonging directly to the domain of the artifact.

In the context of the Business Layer, the typical utility components provide transformation and conversion operations on the domain concepts — `TemperatureUtils` for the conversion between units of measure, `LocationUtils` for the calculation of distances, `CipherUtils` for the encryption of data, `ScheduleUtils` for the management of time slots. These operations are auxiliary in the sense that they do not model any domain concept on their own, but they are necessary for the logical components to be able to implement their effective logic.

Unlike the utility components of the Access Layer that operate on primitive types — `StringUtils`, `DateUtils` — the utility components of the Business Layer operate on domain concepts and have scope local to this layer. They shall not be referenced by components of the Access Layer.

`B` — Injection component The injection component `B` is located at the root of the `/business` folder and constitutes the only point of access to the business logical components from any other component of the artifact — including other logical components of the same layer. Its definition, properties and life cycle are developed in §7.3.3.

In the context of the Business Layer, B is the point through which the Interaction Layer invokes all the domain operations of the artifact — every invocation from an interaction component follows the pattern `B.<business>.<operation>()`. Likewise, all access between business logical components is performed through `B` — a `UserBusiness` that needs to invoke an operation of `SessionBusiness` always does so through `B.session.<operation>()`, never by means of direct reference between components.

7.2.3 Interaction Layer

The Interaction Layer constitutes the highest level of abstraction of the XF Architecture and is the only layer of the model that has direct contact with the consumers of the artifact — human users through graphical interfaces, or external systems through defined communication protocols. Its position in the stratification is likewise a direct consequence of the formal derivation established in §5.5: the interaction stage of every formal process is the one in which the process receives the order to be executed and verifies that the preconditions necessary for its initiation are satisfied.

The Interaction Layer exists in every XF artifact as a consequence of the formal modelling of processes — every formal process has an interaction stage. However, the presence of logical components in this layer is not mandatory in every artifact. An artifact that does not allow external interaction — an autonomous periodic task, a notification service, a batch process — has an Interaction Layer that is structurally present but empty of logical components. The absence of logical components in the Interaction Layer does not constitute a violation of the model — it is the correct architectural expression of an artifact that does not expose external entry points.

The XF model transcends the traditional distinction between backend and frontend artifacts by recognising that both have an Interaction Layer — simply with different component types. A backend artifact exposes its Interaction Layer through services — endpoints, API contracts, console commands. A frontend artifact exposes its Interaction Layer through views — screens, forms, interactive graphical elements. An artifact that combines both types of interaction — a thermostat with an integrated screen and a remote-control API — has components of both types in its Interaction Layer. In all cases, the responsibility of the layer is the same: to define the entry points to the artifact and to establish the conditions under which the business operations may be invoked.

Unlike the Access and Business layers — where generalization forms a single tree under a single abstract root —, the Interaction Layer exhibits **two parallel sub-trees** under two sibling abstract bases: `Service` for systemic consumers and `View` for human consumers. Both sub-trees are independent and coexist with their own identity: no node of one inherits from the other. The concrete form of each sub-tree is a decision of the implementer and depends on the nature of the consumers of the artifact. An artifact that only exposes systemic endpoints may have only the `Service` sub-tree; an artifact that only presents a graphical interface may have only the `View` one; an artifact that combines both develops them in parallel. The invariant prescriptions are the same as in the previous layers: strictly intra-layer inheritance, no intermediate node with a single descendant and no component with logic of another layer. The canonical materialization of both sub-trees is developed

later in this same clause.

Responsibility and boundaries This clause delimits the responsibility of the Interaction Layer and its boundaries: the lower boundary with the Business Layer and the upper boundary with the consumers of the artifact.

Responsibility The responsibility of the Interaction Layer is to define the entry points to the artifact — how its consumers may invoke the business operations it implements — and to establish the formal conditions under which that invocation is valid. This responsibility has three dimensions that the XF model prescribes:

The first dimension is the **definition of the interaction protocol**: every logical component of the Interaction Layer defines the mechanism through which a consumer may invoke the business operations of the artifact. For services, that mechanism is a systemic protocol — HTTP, WebSocket, console commands, queue messages. For views, that mechanism is a graphical interface — screens, forms, interactive elements. In both cases, the interaction component is responsible for defining the protocol with precision and for guaranteeing that the invocations that satisfy it can be processed correctly by the Business Layer.

The second dimension is the **validation of the interaction conditions**: before delegating execution to the Business Layer, every interaction component shall verify that the data received satisfies the formal preconditions of the point of interaction. This validation covers everything that does not belong to the domain model of the artifact — type, format, presence-of-mandatory-field and formal-protocol-constraint checks. The verification that a value has a correspondence with a domain concept — that an identifier belongs to an existing entity, that a value satisfies a business rule — belongs to the Business Layer, not to the Interaction Layer.

The third dimension is the **representation of the results**: every interaction component is responsible for transforming the results of the business operations into the form that the consumer of the artifact expects to receive. For services, that form is the protocol response — status codes, response structures, error formats. For views, that form is the visual representation — which graphical elements are shown, in what state and with what data.

Lower boundary — Business Layer The lower boundary of the Interaction Layer is the Business Layer, whose operations it invokes through the injection component `B`. The Interaction Layer does not know how the Business Layer obtains or processes the data — its responsibility begins at the moment it receives a request from a consumer and ends at the moment it delivers the corresponding response. Everything that occurs between those two moments — the domain logic, the access to external data — is the responsibility of the lower layers.

Upper boundary — Consumers of the artifact The upper boundary of the Interaction Layer is the consumers of the artifact — human users or external systems. The Interaction Layer is the boundary between the artifact and its consumption environment, in the same way that the Access Layer is the boundary between the artifact and its data environment. This symmetry is intentional — the XF model

treats interaction with consumers and access to external systems as two instances of the same principle of boundary abstraction, applied in opposite directions.

Process model The interaction process model defines what functionally characterises the logic that belongs to the Interaction Layer, so that every developer can determine without ambiguity whether a concrete piece of logic belongs to this layer or to another. As in §7.2.1 and §7.2.2, the model prescribes responsibilities and constraints — not concrete implementations. Implementation guidance is collected in Annex C.

The Interaction Layer recognises two types of logical components with distinct process models: **services** — components of systemic interaction — and **views** — components of graphical interaction. Although both types satisfy the same prescribed responsibilities of the layer, their internal process model has its own characteristics that the XF model describes with precision. In this clause, the XF model introduces a reference terminology to describe the elements of each type, with the same function that the business terminology fulfils in §7.2.2 — to provide a common vocabulary that allows teams to communicate about interaction logic with precision and without ambiguity, independently of the technology employed.

Prescribed responsibilities Every logical component of the Interaction Layer — both services and views — **shall** satisfy the following responsibilities:

The first responsibility is the **definition of the interaction protocol**. Every interaction component defines with precision the mechanism through which a consumer may invoke the business operations of the artifact. This mechanism shall be sufficiently precise that any consumer satisfying it can be served correctly, and sufficiently restrictive that the invocations that do not satisfy it are rejected before reaching the Business Layer.

The second responsibility is the **validation of the interaction conditions**. Every interaction component shall verify that the data received satisfies the formal preconditions of the point of interaction before delegating execution to the Business Layer. This validation covers exclusively what does not belong to the domain model: type, format, presence-of-mandatory-field and formal-protocol-constraint checks. The verification that a value has a correspondence with a domain concept belongs to the Business Layer.

The third responsibility is the **delegation to the Business Layer**. All domain logic shall be delegated to the logical business components through the injection component `B`. An interaction component does not implement domain logic — it invokes `B.<business>.<operation>()` and receives the result as a transfer component or a business exception.

The fourth responsibility is the **representation of the result**. Every interaction component is responsible for transforming the result of the business operations into the form that the consumer expects to receive — the protocol response in services, the visual representation in views. This transformation may entail the definition of new transfer components in the Interaction Layer when the structure of the result differs from the structure provided by the Business Layer.

Excluded responsibilities An interaction component **shall not** contain domain logic. Business rules, semantic transformations of data and operations on domain concepts belong to the Business Layer and shall be invoked through `B`.

An interaction component **shall not** contain logic for accessing external systems. All communication with external systems belongs to the Access Layer and shall be accessed through `R`. An interaction component that directly invokes a repository violates the isolation principle established in §6.2.2.

An interaction component **shall not** verify conditions that belong to the domain of the artifact. The verification that a value has a correspondence with a domain concept — that an identifier belongs to an existing entity, that a value satisfies a business rule — belongs to the Business Layer.

Classification criterion A piece of logic belongs to the Interaction Layer if and only if it simultaneously satisfies the following three conditions:

The first condition is that it **defines or manages an entry point to the artifact** — it establishes how a consumer may invoke a business operation or receive information from the artifact.

The second condition is that it **knows no business rule of the domain** — it does not make decisions based on the meaning of the data, only on the formal preconditions of the interaction protocol.

The third condition is that **its result is a representation for the consumer** — a protocol response, a presentation structure or a visual state — not a processed domain datum.

Reference terminology The XF model defines the following terms to describe with precision the elements that make up the interaction logic of an artifact. This terminology constitutes a common vocabulary that facilitates communication between developers, analysts and reviewers about the interaction logic of any XF artifact independently of the technology employed. The terms are grouped into two sets corresponding to the two types of logical components of the layer: services (systemic consumer) and views (human consumer).

The **service protocol** is the formal definition of the interaction mechanism of a service with its systemic consumer — the access path to the resource, the invocation method, the transport protocol and the response format. The service protocol is the complete description of how an external system may communicate with the service.

The **input validation** is the set of checks that a service performs on the data received before delegating to the Business Layer. Input validation covers the formal constraints of the protocol — presence of mandatory parameters, correct data types, valid formats — without entering into domain validations.

The **business invocation** is the delegation of execution to the logical business components through `B`. The business invocation is the central operation of the service — everything that precedes it is preparation, and everything that follows it is representation of the result.

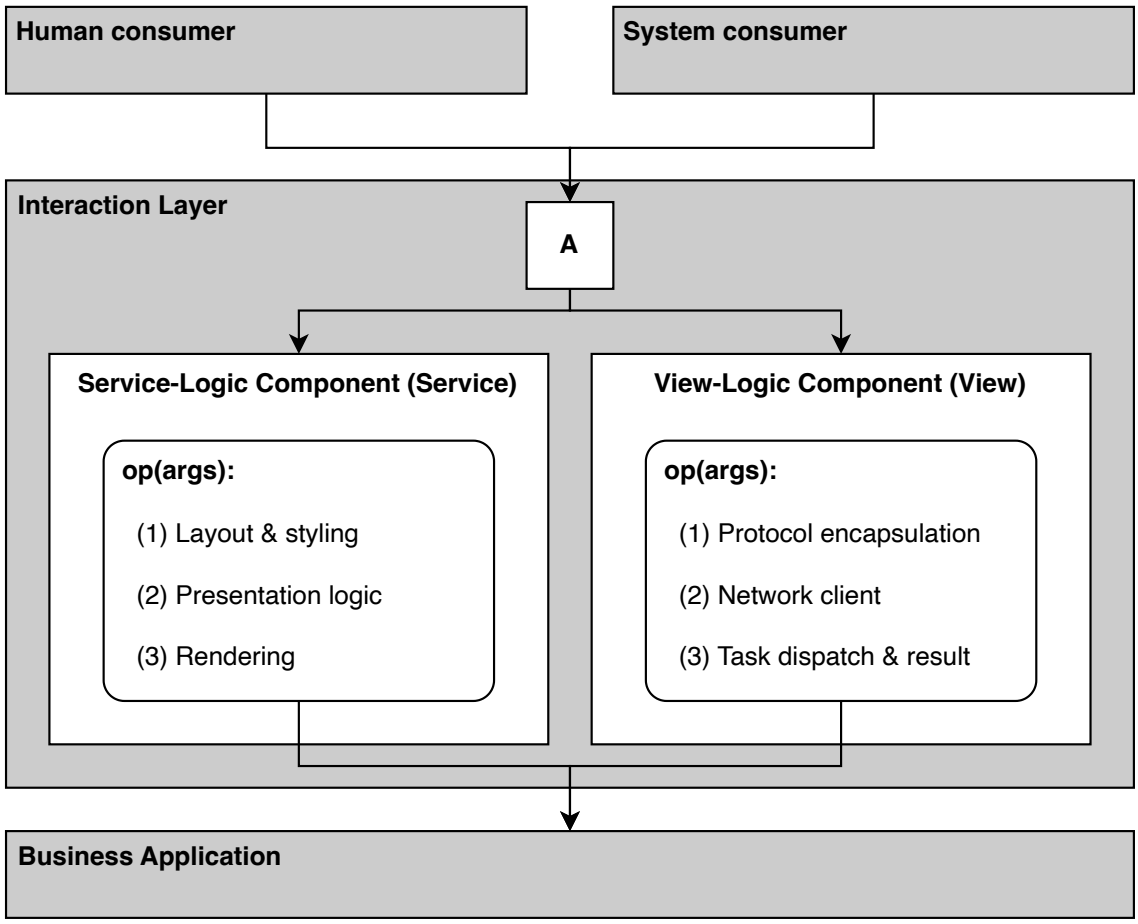

Figure 9. Prescribed content of a logical component of the Interaction Layer. It receives invocations from the external consumer — an external system in the case of the service, a human user in the case of the view —, validates the formal preconditions of the interaction protocol, delegates the domain logic to the Business Layer through the injection component `B` and transforms the result into the representation that the consumer expects. No protocol for accessing external systems nor business rule resides in this component.

The **service response** is the transformation of the result of the business invocation into the form that the transport protocol requires — status codes, response structures, error formats. The service response includes both the success case and the handling of the business exceptions received from the Business Layer, which shall be transformed into appropriate protocol error responses.

The **request tracing** is the recording of the invocation of a service and its result for the purposes of auditing, debugging or monitoring. Tracing is an optional responsibility of the service that does not affect the main flow of the invocation.

The **composition** is the definition of the hierarchy of graphical elements that form a view — which elements compose it, how they are organised among themselves and what hierarchical relationships exist between them. Composition is the structure of the view — the equivalent of HTML in web development or of the widget hierarchy in Flutter.

The **style** is the definition of the visual appearance with which a view is represented — margins, colours, typography, spatial distribution of the elements. Style is inde-

pendent of composition in its conceptual definition, although in many development frameworks both are expressed jointly.

The **presentation logic** is the set of algorithms and checks that determine how a view should behave in the face of the data it receives and the events that occur on it. Presentation logic is the functional core of the view — it defines what should be shown, when and in what state — and it comprises the decisions about the representation of pending and empty results, business invocations in response to user events and reactions to observed state changes.

The **presentation state** is the set of data that determines the observable visual behaviour of a view at a given moment — which data is being shown, whether an operation is in progress, whether the form is in edit mode. The presentation state is state in the formal sense defined in §7.3.1 — it determines the observable behaviour of the component in the face of the operations that invoke it — but its scope is limited to the Interaction Layer and it has no direct correspondence with business domain information: it models the dynamic context of the graphical interaction.

Process model of services A service defines a point of systemic interaction — a mechanism through which another system may invoke operations of the artifact by means of a defined communication protocol. Its process model follows a fixed sequence: the external system invokes an endpoint defined by the service protocol; the service applies input validation to the data received; when the validation is satisfied, it performs the business invocation through `B`; the result — success or business exception — is transformed into the service response according to the format of the transport protocol. Request tracing operates transversally to the sequence, without affecting the main flow.

Process model of views A view defines a point of graphical interaction — a visual element through which a human user may invoke operations of the artifact or receive information from it. Its process model combines three simultaneous dimensions that the XF model strongly recommends encapsulating in a single logical component, although it recognises that some development frameworks may require their distribution into separate files: composition defines the static hierarchy of graphical elements; style determines the visual appearance; presentation logic governs the dynamic behaviour. The view maintains a presentation state that captures the observable context of the interaction and is updated in the face of domain data observed through `B`.

By way of guidance, the following are typical examples of presentation logic:

- The logic that determines what should be represented while the result of a business operation is awaited — a progress indicator, a content skeleton, a waiting message (`loading`).
- The logic that determines what should be represented when the business operation returns an empty result or one without relevant data (`emptyState`).
- The logic that determines which business operation should be invoked through `B` when the user interacts with an element of the view — pressing a button, entering data, gestures (`onClick`, `onInput`, `onSwipe`).

- The logic that determines which operations should be executed at each phase of the view's life cycle — which data to load on creation, which state to persist on pause, which resources to release on destruction (`onCreate`, `onPause`, `onDestroy`).
- The logic that determines how the visual representation of the view should be updated when the state of an observable business component changes, by means of the observation pattern defined in §6.2.2 (`onStateChange`).

These examples are illustrative — they do not constitute an exhaustive or prescriptive list. The presentation logic of a concrete view will be determined by the interaction needs of the artifact. The criterion for identifying it is the following: all logic that determines what is shown or what is executed in response to a datum or an event, without applying domain rules or accessing external systems, is presentation logic and belongs to the Interaction Layer.

An interaction component **shall not** derive its presentation state directly from the domain state of a business component without mediation — it shall obtain it through `B` and transform it into presentation state if necessary. The XF model recommends managing the presentation state by means of the generalization component `StatefulView<T>`, which abstracts the mechanism of state update and re-rendering of the view. The autonomous implementation of the presentation state in the logical component itself is equally valid.

Criterion for distinguishing between service and view The criterion for classifying a logical interaction component as a service or as a view is the type of consumer it is directed at. A component is a service if its consumer is an external system that communicates with the artifact by means of a defined systemic protocol. A component is a view if its consumer is a human user who interacts with the artifact by means of a graphical interface. This distinction is recommended — it facilitates the internal organisation of the Interaction Layer — but not prescriptive. Both types of components are equally conformant to the XF model independently of how they are organised in the folder structure.

Subdivisions of logical components The `/logic` folder of the Interaction Layer contains all the logical interaction components — services and views — of the artifact. In small-scale artifacts, this folder may contain a small number of components that do not require additional organisation. However, as the artifact grows in functionality and the number of entry points increases, the `/logic` folder may come to contain a number of components high enough to hinder its navigation and comprehension.

For these cases, the XF model permits — and recommends — the creation of internal subdivisions within the `/logic` folder that group the logical components according to the criterion that the development team considers most appropriate for its concrete artifact. These subdivisions are a design decision of the team — not a normative prescription of the model — and their implementation does not affect the conformance of the artifact.

Recommended subdivision The XF model proposes, as a reference subdivision, the division between graphical and systemic interactions, materialized in two

subfolders within `/logic`:

- `/api/logic/view` — logical components that define points of graphical interaction — views — through which a human user may invoke operations of the artifact or receive information from it by means of a visual interface. Typical examples: `MainView`, `LoginView`, `TemperatureView`, `ScheduleView`.
- `/api/logic/service` — logical components that define points of systemic interaction — services — through which an external system may invoke operations of the artifact by means of a defined communication protocol. Typical examples: `TemperatureService`, `AuthService`, `ScheduleService`.

The usefulness of this subdivision is clear in artifacts that combine both types of interaction — for example a thermostat with an integrated screen and a remote-control API — where the separation between views and services facilitates orientation in the structure of the artifact. In purely backend or purely frontend artifacts, the subdivision may prove redundant — one of the two folders will be empty — and the team may choose not to implement it.

The concrete criterion for classifying a component as graphical or systemic follows the distinction criterion established in §7.2.3: the type of consumer it is directed at. In ambiguous cases — for example, a component that manages a push notification that has both a visual representation and a systemic protocol — the classification is a decision of the implementer.

Subdivision criterion The legitimate subdivision criterion in the Interaction Layer is the one derived from its technical responsibility: the resolution of the type of input that the artifact exposes to its consumers. Every internal subdivision of `/api/logic` shall group the logical components according to properties of the type of input they resolve — nature of the consumer (human or systemic), type of invocation protocol, type of interface — and not according to the functional domain of the artifact. An artifact with multiple service protocols may subdivide by protocol — `/api/logic/service/rest`, `/api/logic/service/websocket`. An artifact with several types of graphical interface may subdivide by the nature of the interface — for example, separating main views from reusable components — provided that the criterion is technical, not domain-based.

Subdivision by domain: not conformant The subdivision of `/api/logic` by the functional domain of the artifact — for example, grouping views or services into subfolders whose name coincides with concepts of the domain modelled by the Business Layer — is not conformant to the XF model. The reason is structural: interaction logic resolves type of input, not domain. Several domains of the artifact may share the same type of graphical interface or the same service protocol, and subdividing by domain would entail replicating the same input resolution across several subfolders with the risk of redundancy and of mixing responsibilities. The domain is a legitimate subdivision criterion exclusively in the Business Layer (§7.2.2).

Formal conditions Every subdivision is valid provided that three conditions are respected. The first is that all the logical components of the Interaction Layer are located within the `/api/logic` folder, directly or in one of its subfolders. The

second is that no subfolder contains components of a type other than logical — the generalization, utility, transfer and injection components shall remain in their corresponding canonical folders. The third is that the subdivision criterion adopted belongs to the technical vocabulary of the layer — properties of the type of input — and not to the vocabulary of the domain modelled by the Business Layer.

Absence of subdivision An artifact that does not implement any subdivision within `/logic` is fully conformant to the XF model. The absence of subdivision is the simplest organisation and is appropriate for artifacts with a small number of logical components or for artifacts that only implement one type of interaction — only views or only services.

```
// Without subdivision - valid for small-scale artifacts
/api/logic
├── MainView
├── LoginView
└── TemperatureService

// With recommended subdivision - valid for artifacts with both types
/api/logic
├── /view
│   ├── MainView
│   └── LoginView
└── /service
    └── TemperatureService

// With subdivision by service protocol - equally valid
/api/logic
├── /view
│   ├── MainView
│   └── LoginView
└── /service
    ├── /rest
    │   └── TemperatureService
    └── /websocket
        └── LiveUpdateService
```

Listing 7. Absence of subdivision in the Interaction Layer

Internal organisation The Interaction Layer is materialized in the project's file system under the canonical folder `/api`, whose internal structure follows the organisation prescribed in §7.4. This clause consolidates that organisation applied to the specific context of the Interaction Layer, describing the function of each subfolder and its relation to the prescriptions of the process model defined in §7.2.3.

```
/api
├── /general (Generalization components)
├── /logic (Logical components)
│   ├── /view (Graphical interactions -- recommended)
│   └── /service (Systemic interactions -- recommended)
├── /transfers (Transfer components)
├── /utils (Utility components)
└── A (Injection component)
```

Listing 8. Internal organisation of the Interaction Layer

`/general` — Generalization components The `/general` folder contains the generalization components of the Interaction Layer — the abstractions of structural behaviour common between logical interaction components. As established in §7.3.2, these components do not implement the effective logic of any concrete point of

interaction — they abstract shared structural contextual logic across several logical components of the same layer, freeing the concrete logical components to concentrate exclusively on their effective logic.

As in the previous layers, generalization in the Interaction Layer can operate at multiple levels of specificity. The most common structural patterns in this layer are the following:

For view components, the typical generalization patterns abstract structural behaviours common between views. `StatefulView<T>` abstracts the mechanism of presentation state management and the re-rendering of the view when that state changes — it is the equivalent of `StatefulBusiness<T>` in the Business Layer, applied to the presentation state. `FormView` abstracts the common logic of forms — field validation, management of the edit state, data submission. `FullscreenView` abstracts the common structure of full screens — navigation management, full-screen life cycle.

For service components, the typical generalization patterns abstract interaction protocols common between services. `RestService` abstracts the common logic of all the REST endpoints of the artifact — deserialization of the request, serialization of the response, handling of protocol errors. `FormalService` abstracts the response structure proper to the artifact — for example, a triple { code, description, data } that all the services of the artifact use as a response wrapper. `WebSocketService` abstracts the management of the WebSocket channel and the life cycle of the connection.

The typical inheritance chains in the Interaction Layer materialize the two parallel sub-trees presented in §7.2.3:

```
Service (base behaviour of any systemic entry point)
└── RestService (generic HTTP/REST protocol)
    └── FormalService (response structure proper to the artifact)
        ├── TemperatureService (logical -- temperature endpoints)
        └── AuthService        (logical -- authentication endpoints)

View (base behaviour of any graphical entry point)
├── ScreenView (full-screen pattern)
│   └── TemperatureView (logical -- temperature display screen)
└── AtomicView (reusable atomic graphical element pattern)
    ├── ButtonView    (logical -- action element)
    └── TextInputView (logical -- text capture element)
```

Listing 9. Canonical generalization patterns in the Interaction Layer

As in the previous layers, the concrete logical components — `TemperatureService`, `AuthService`, `TemperatureView`, `ButtonView`, `TextInputView` — inherit all the contextual logic of the protocol or of the graphical pattern from their generalizations and implement only their effective logic: the concrete endpoints or elements, the validations specific to each resource and the invocation of the corresponding business operations.

The presence and the content of this folder depend on the complexity of the interaction protocols of the artifact and on the degree of reuse that the team considers appropriate. An artifact with a single point of interaction may not require any generalization component.

`/logic` — Logical components The `/logic` folder contains the services and views of the artifact — the logical components of the Interaction Layer. Each logical component defines exactly one entry point to the artifact and the interaction logic associated with that point, satisfying the responsibilities prescribed in §7.2.3: definition of the interaction protocol, validation of the interaction conditions, delegation to the Business Layer and representation of the result.

The effective logic of a logical interaction component is what distinguishes it from a generalization component in this layer. A component that only defines generic structural protocol behaviour without implementing any concrete entry point is a generalization component, not a logical component. A component that defines the /temperature endpoint with its specific validations and its invocation to `B.temperature.update()` is a logical component — that is its effective logic.

The internal organisation of `/logic` — with or without subdivisions — follows the prescriptions of §7.2.3.

`/transfers` — Transfer components The `/transfers` folder contains the transfer components defined in the Interaction Layer — the data structures that model the information in the form that the consumers of the artifact receive or send.

In the context of the Interaction Layer, the typical transfer components model the input and output data of the points of interaction of the artifact. For services, structures that model the bodies of the requests received and the responses sent are common — with the attributes that the consumer needs to see, removing those that are internal to the domain. For views, structures that model the presentation state of each view are common — which data is shown, which options are available, which errors should be represented.

As prescribed in §7.3.5, the obligation to define a new structure in this layer is triggered only when the structural definition of a structure coming from the Business Layer is modified. If the business structure can be presented to the consumer without structural modification, it may be referenced directly without the need to define a new structure in this folder.

It is equally common to find in this folder the interaction exceptions — transfer components that are typically transmitted as language exceptions and model the errors proper to the interaction protocol: `InvalidParameterException`, `UnauthorizedException`, `ResourceNotFoundException`.

`/utils` — Utility components The `/utils` folder contains the utility components of the Interaction Layer — stateless auxiliary operations that serve as support to the logical interaction components for the achievement of their effective logic without belonging directly to the domain of the artifact or to the access protocol.

In the context of the Interaction Layer, the typical utility components provide operations for the formal validation of input parameters — `ValidationUtils` for the verification of formats, `FormUtils` for the validation of forms —, presentation transformation operations — `ColorUtils` for the computation of derived colours, `AnimationUtils` for the management of transitions —, and operations for the construction of protocol responses — `ResponseUtils` for the construction of standardised

HTTP responses.

These utility components have a scope local to the Interaction Layer. They shall not be referenced by components of the lower layers.

`A` — Injection component The injection component `A` is located at the root of the `/api` folder and constitutes the only point of access to the logical interaction components from any other component of the same layer. Its definition, properties and life cycle are developed in §7.3.3.

In the context of the Interaction Layer, `A` aggregates all the services and views of the artifact under a centralized point of access. Unlike `R` and `B` — which are accessed frequently from upper layers — A is typically the least invoked injection component from other layers, given that the Interaction Layer is the highest level of the architecture and its components are not invoked by upper layers. Its main function is the management of the life cycle of the logical interaction components — their initialization in the correct order and their controlled termination — and the access between logical components of the same layer when necessary.

7.3 Functional dimension: the component types

The XF model defines five functional component types that, combined with the three abstraction layers, produce the model's complete classification space. The definitions in this clause are normative: they establish precisely what each type is, what it may contain, what restrictions apply to it and how it shall relate to the other components of the artifact.

For each type the following are established: its definition, its normative properties, its restrictions, its canonical nomenclature and its organization within the artifact's folder structure.

7.3.1 Logical components

Definition A logical component is the principal functional unit of a layer. It implements the **effective logic** of the layer to which it belongs — understood as the minimal set of operations that justify its existence in the artifact: those that cannot be abstracted into any other component type without losing the specificity of the concept the component models.

Effective logic is distinguished from contextual logic in that it is inherent to the concept the component models and cannot exist in any other component without losing its identity. Everything that can be shared with other components, reused or expressed in a generic form is contextual logic — and belongs to generalization, utility or transfer components according to its nature. The other component types of the layer provide that contextual logic as support for the achievement of the effective logic, but they neither replace it nor contain it.

Pragmatically, a logical component is the answer to the question: *what does this artifact do with this domain concept?* A business logical component that manages user authentication defines the operations "verify credentials", "refresh session" and "expire session" — these are effective operations because they directly solve the

authentication problem. The in-transit encryption of the credentials, the access protocol to the identity server or the storage of the session on disk are contextual logic that belongs to other component types of the corresponding layer.

Internal structure of a logical component A logical component is internally structured into three elements that the XF model defines precisely: its state, its operations and the statements that make up each operation.

The **state** of a logical component is the set of data that has a direct correspondence with information from the business domain and that determines the observable behavior of the component in response to the operations that invoke it. State shall not be confused with the internal operational attributes of the component — a reference to a database connection, an initialization flag or a retry counter are attributes, not state. State describes the dynamic context of the concept the component models: the current temperature recorded in `TemperatureBusiness`, the active session data in `SessionBusiness`, or the list of pets loaded in `PetBusiness` are examples of state. Logical components are the only components of the artifact that may maintain state in this formal sense.

An **operation** is a subset of the effective logic of the component that executes atomically in response to an event or invocation, producing a transformation of the component's state, a result for the invoker, or both. Each operation of a logical component corresponds to exactly one well-defined action on the concept the component models — "update temperature", "refresh session", "get pets by category". An operation that cannot be described as an action on the domain concept is an indication that the logic it implements is contextual and belongs to another component type.

A **statement** is each of the logical steps that make up an operation. The statements of an operation are executed in an ordered manner and their set constitutes the complete implementation of the action the operation models. A statement may be a data transformation, an invocation of a logical component of a lower layer through the corresponding injection component, an evaluation of a business condition, or a mutation of the component's state. The XF model recommends that the number of statements per operation be the minimum necessary and sufficient for the achievement of the operation's objective — an operation with an excessive number of statements is an indication that it is assuming responsibilities that should be delegated to other components, other operations, or to lower layers.

A **condition** is a specific statement that evaluates the component's state or the input parameters of an operation before executing the logic the operation models. Conditions constitute the execution preconditions of the operation in the sense of Meyer [31]: when a condition is not satisfied, the operation typically propagates an exception transfer component to the invoker, although other forms of management — default value, retry, defined fallback logic — are admissible according to the semantics of the process. The XF model recommends that conditions be evaluated as early as possible in the operation, ideally at the start when the nature of the process permits it, so that the anomalous condition is transmitted without executing unnecessary logic. Processes whose preconditions depend on data obtained by prior delegation are a legitimate exception to this recommendation.

The following code fragment illustrates this internal structure in a business logical component:

```
// Business logical component - example implementation
class TemperatureBusiness extends StatefulBusiness<Temperature> {

    // State: the current temperature managed by the component
    Temperature current;

    // Operation: update the temperature
    void update(Temperature temperature) {

        // Conditions - preconditions of the operation
        if (TemperatureUtils.toCelsius(temperature) < 14)
            throw new TemperatureOutOfRangeException();

        if (TemperatureUtils.toCelsius(temperature) > 32)
            throw new TemperatureOutOfRangeException();

        if (B.network.signal < Network.GOOD)
            throw new NetworkUnavailableException();

        // Statements - steps of the operation
        R.server.saveTemperature(temperature); // delegates to lower layer
        this.current = temperature; // mutates the state
        super.update(temperature); // notifies observers
    }
}
```

Listing 10. Internal structure of a logical component

Normative properties The first property is **single responsibility**: each logical component models exactly one business domain concept, exactly one access protocol to an external system, or exactly one interaction point of the artifact. A component that models more than one independent concept shall be decomposed into as many logical components as independent concepts it contains. This property is a necessary condition for the correct application of the closed and exhaustive typing established in §6.2.4 — a component with mixed responsibilities cannot be classified unambiguously in the space $L \times T$.

The second property is **state capability**: logical components are the only components of the artifact that may maintain state in the formal sense defined in this clause. This property does not imply that every logical component shall maintain state — a stateless logical component is perfectly valid in the model — but rather that no other component type may maintain it.

The third property is **formal derivability**: the operations a logical component implements are typically derived from the formal specification of the process the artifact automates — sequence diagrams, state diagrams or use case specifications. The model recommends this traceability as a design practice, but does not prescribe it normatively: artifacts without a prior formal specification remain conformant. An operation that cannot be traced back to the process specification, when one exists, is an indication that the logic it implements is contextual and belongs to another component type.

The fourth property is **delegation toward lower layers**: when an operation of a logical component requires information or capabilities that reside in systems external to the artifact, that logic shall be delegated to logical components of layers of a lower abstraction level through the corresponding injection component. A business logical

component does not directly access databases, external APIs or system files — it delegates that access to the corresponding access logical component through `R`. An interaction logical component does not implement domain logic — it delegates that logic to the corresponding business logical component through `B`. This property is the operational expression of the layer isolation principle established in §6.2.2.

Restrictions A logical component **shall** be classified in exactly one layer. The logic it implements shall correspond exclusively to the responsibility of that layer. The presence of logic corresponding to another layer constitutes a violation of the isolation principle established in §6.2.2.

A logical component **shall** implement its operations in terms of statements over its state, conditions over its preconditions and delegations to lower layers through the injection component. It shall not implement logic that can be classified as contextual logic of another component type.

A logical component **may** inherit from a generalization component of its same layer. It **shall not** inherit from another logical component nor from a generalization component of another layer.

A logical component **may** reference transfer components of its same layer or of layers of a lower abstraction level. It shall not reference transfer components of layers of a higher abstraction level. The rule applies uniformly to the transfers of the artifact itself and to those inherited from integrated external artifacts: all are transfer components of the single domain $\mathbb{C}_T$.

A logical component **may** invoke operations of logical components of layers of a lower abstraction level, always through the injection component of the corresponding layer. It shall not invoke operations of logical components of layers of a higher abstraction level.

Life cycle of the logical component A logical component **shall** expose an invocable operation `init()` and an invocable operation `terminate()`, which the injection component of its layer orchestrates at the startup and termination of the artifact (§8.2). Its constructor is restricted to the trivial initialization of the component's own attributes: all startup logic that depends on environment resources or that invokes other components belongs to `init()`, not to the constructor.

Canonical nomenclature The logical components of the Access Layer **shall** bear the suffix `Repository` — for example, `DatabaseRepository`, `IdentityRepository`, `FileRepository`.

The logical components of the Business Layer **shall** bear the suffix `Business` — for example, `UserBusiness`, `SessionBusiness`, `TemperatureBusiness`.

The logical components of the Interaction Layer **shall** bear the suffix `Service` when they implement systemic interactions — for example, `TemperatureService`, `AuthService` — or the suffix `View` when they implement graphical interactions — for example, `LoginView`, `MainView`, `ThermometerView`.

Organization within the folder structure Logical components are grouped under the subfolder `/logic` within the folder of their corresponding layer:

```
/src
├── /repository/logic (Access logical components)
├── /business/logic (Business logical components)
└── /api/logic (Interaction logical components)
```

Listing 11. Folder organization: logical component

7.3.2 Generalization components

Definition A generalization component is a component that abstracts structural behavior common to two or more logical components of the **same layer**, allowing that behavior to be inherited by the logical components that require it without the need to duplicate it in each of them. Unlike logical components, a generalization component does not by itself solve any effective logic of the artifact — it models no domain concept, implements no business operation and has no utility independent of the logical components that inherit it. Its function is exclusively to provide a reusable structure on which logical components build their effective logic.

Pragmatically, a generalization component is the answer to the question: *what structural behavior do several logical components of this layer share?* When several logical components of the Business Layer need to observe and notify changes of mutable state, that observable behavior is structurally identical in all of them regardless of the business concept each one models. The generalization component `StatefulBusiness<T>` encapsulates that behavior, and each logical component that inherits it — `TemperatureBusiness`, `UserBusiness`, `SessionBusiness` — specializes it with the concrete state it manages. Likewise, when several logical components of the Access Layer implement the REST protocol, the generalization component `RestRepository` encapsulates the common logic of the protocol and each concrete repository inherits from it to specialize it with the resource it accesses.

Distinction with respect to logical components: parametric with respect to the domain The distinction between a generalization component and a logical component is not always immediate, and its correct application is one of the most relevant design criteria of the XF model. The defining criterion is the following: a component is a generalization component if its logic can be applied to more than one domain concept without modification — that is, if it is **parametric with respect to the domain**. A component is a logical component if its logic is **bound to a specific domain concept** and cannot be applied to another without modification.

`StatefulBusiness<T>` is a generalization component because its logic — maintaining an observable state and notifying observers when it changes — is independent of the type of state T it manages. `TemperatureBusiness` is a logical component because its logic — validating temperature thresholds, synchronizing with the server, converting units — is bound to the temperature concept and cannot be applied to any other concept without modification.

When a component that has been classified as a generalization component begins to contain references to concrete domain concepts — for example, a base class that makes explicit reference to `Temperature` or to `User` — it is an indication that the

logic it contains has ceased to be structural and has become effective, and therefore shall be moved to the concrete logical component that models that concept.

Mutable attributes and state Generalization components may have mutable attributes insofar as they are reproducible logical subsets of the logical components that inherit them. However, there is an important normative distinction between the attributes of a generalization component and the state of a logical component.

A generalization component may define mutable attributes that support the structural behavior it abstracts — for example, the list of registered observers in `StatefulBusiness<T>`, or the reference to the HTTP client in `RestRepository`. These attributes are operational: they serve the internal functioning of the structural pattern, but they have no direct correspondence with information from the business domain.

State in the formal sense of the model — the set of data with a direct correspondence with information from the domain — always belongs to the concrete logical component that inherits from the generalization component, never to the generalization component itself. A generalization component shall not define attributes that model business domain information — that responsibility is exclusive to logical components.

```
// Generalization component - operational attributes, no domain state
abstract class StatefulBusiness<T> {

    // Operational attribute: list of registered observers
    List<Observer<T>> observers = new ArrayList<>();

    // Operational attribute: reference to the current state (generic, no
        domain semantics)
    T state;

    void observe(Observer<T> observer) {
        observers.add(observer);
    }

    void update(Partial<T> partial) {
        this.state = partial.applyOn(this.state);
        observers.forEach(obs -> obs.notify(this.state));
    }
}

// Logical component - state with concrete domain semantics
class TemperatureBusiness extends StatefulBusiness<Temperature> {

    // The domain state is Temperature, defined in the logical component
    void update(Temperature temperature) {

        if (TemperatureUtils.toCelsius(temperature) < 14)
          throw new TemperatureOutOfRangeException();

        R.server.saveTemperature(temperature);

        // delegates the notification to the generalization component
        super.update(temperature);
    }
}
```

Listing 12. Mutable attributes and state

The isolation principle applied to generalization The layer isolation principle established in §6.2.2 applies to generalization components with the same force as to

logical components. A generalization component is classified in a specific layer and its scope is strictly limited to that layer: no logical component may inherit from a generalization component of a layer different from its own.

This restriction has a consequence that the XF model prescribes explicitly: when the same structural pattern is needed in more than one layer, each layer shall implement its own generalization component independently. The model prescribes this controlled duplication because the alternative — a generalization component transversal to several layers — would introduce a structural dependency between layers that would compromise the isolation and the resistance to local change of the artifact.

The structural patterns that most frequently require independent implementation per layer are the following:

The **mutable state observation pattern** — needed when components of upper layers need to react to state changes in components of lower layers without the latter invoking them directly — is implemented through a generalization of its own in each layer that requires it, for example, `StatefulRepository`, `StatefulBusiness` and `StatefulView` in the Access, Business and Interaction layers respectively (the latter applied to the presentation state). The three implementations embody the same pattern, but they are independent generalization components and do not share code across layers.

The **periodic task pattern** — needed when a component needs to execute logic automatically at defined time intervals — is implemented through a generalization of its own in each layer that requires it, for example, `ScheduledRepository`, `ScheduledBusiness` and `ScheduledView` respectively. Likewise, the three implementations are independent of one another.

The **protocol communication pattern** materializes in two complementary generalizations, one in each boundary layer of the artifact. In the Access Layer — for example, `RestRepository`, `WebSocketRepository` or `PostgresRepository` — it abstracts the common logic of *consuming* protocols published by external systems. In the Interaction Layer — for example, `RestService`, `WebSocketService` or `FormalService` — it abstracts the common logic of *exposing* the protocols of the artifact itself to its consumers. Both are distinct generalizations despite sharing a protocol name: one consumes and the other exposes, and each respects the isolation of its layer.

Non-trivial shared behavior between layers Generalization between layers is not admissible: behavior shared by components of different layers is encapsulated as a logical component of the **lowest layer** that requires it, invocable by the upper ones in a descending direction.

Normative properties The first property is **domain neutrality**: a generalization component shall not contain references to concrete concepts of the artifact's business domain. Its logic shall be applicable to any logical component of its layer that satisfies the structural preconditions of the pattern it abstracts. The presence of references to concrete domain concepts in a generalization component is an indication that part of its logic shall be moved to the concrete logical component.

The second property is **dependence on logical components**: in the context of a final artifact (not a library), a generalization component has no independent utility — it exists as a function of the logical components that inherit it. This property is stated as a design criterion and not as a normative rule: utility libraries may export generalization components intended to be consumed by third-party artifacts, without this constituting a conformance violation of the library-artifact.

The third property is **layer isolation**: the scope of a generalization component is strictly limited to the layer in which it is defined. It shall not be inherited by components — logical or generalization — of other layers, nor shall it reference injection components of any layer — neither its own nor any other — since access to the instances of logical components through injection components is reserved for other logical components (§7.3.3).

Restrictions A generalization component **shall** be classified in exactly one layer and its scope is limited to that layer.

A generalization component **shall** be abstract — it cannot be instantiated directly. Its instantiation always occurs through the concrete logical components that inherit it.

A generalization component **may** have mutable attributes that support the structural behavior it abstracts, but it shall not define attributes that model business domain information.

A generalization component **shall not** contain references to concrete concepts of the artifact's business domain.

A generalization component **shall not** be inherited by components — logical or generalization — of a layer different from its own, nor by components of a type other than logical or generalization.

A generalization component **shall not** invoke operations of logical components through injection components, in any layer — neither its own nor any other —. Access to the instances of logical components through injections is reserved for other logical components (§7.3.3). If a generalization needs to access logic encapsulated in a logical component, that need is an indication that the logic it contains is effective and belongs to the concrete logical component, not to the generalization component.

Canonical nomenclature Generalization components **shall** bear the name of the structural pattern they abstract followed by the canonical suffix of the layer to which they belong (`Repository`, `Business`, `Service` or `View` as appropriate) — for example, `StatefulBusiness`, `ScheduledBusiness`, `BroadcastBusiness` in the Business Layer; `RestRepository`, `PostgresRepository`, `StatefulRepository`, `ScheduledRepository` in the Access Layer; `StatefulView`, `ScheduledView`, `FormView`, `RestService` in the Interaction Layer.

Life cycle of the generalization component A generalization component **shall** satisfy the same life cycle obligations as the logical component that inherits it: declare (or inherit) an invocable operation `init()` and an invocable operation `terminate()` (§8.2), and restrict its constructor to the trivial initialization of structural attributes

proper to the abstraction. All startup logic that depends on environment resources or that invokes other components belongs to `init()`, not to the constructor.

Organization within the folder structure Generalization components are grouped under the subfolder `/general` within the folder of their corresponding layer. The membership of a component in the category "generalization in layer l" is determined by its location under `/src/`l`/general/`; the classification is total by construction (§7.4).

```
/src
├── /repository/general (Access generalization components)
├── /business/general (Business generalization components)
└── /api/general (Interaction generalization components)
```

Listing 13. Folder organization: generalization component

7.3.3 Injection components

Definition An injection component is the sole point of access to the logical components of its layer from any other component of the artifact. It aggregates the unique instances of all the logical components of its layer under a centralized point of access, guaranteeing that each logical component has exactly one instance throughout the entire execution of the artifact, and that this instance is accessible directly and predictably from any point of the artifact that requires it.

The injection component implements the **Singleton Gathering** pattern — an extension of the Singleton pattern [14] that aggregates at a single point the unique instances of a set of related components, instead of managing the uniqueness of a single component. It is this extension that confers upon the injection component its dual function: on the one hand, it guarantees the uniqueness of instance of each logical component of the layer; on the other, it provides a named and stable point of access to each of those instances, making the structure of logical components of the layer visible and inspectable without the need for additional tools.

Pragmatically, the injection component is the answer to the question: *which logical components exist in this layer, how are they initialized, and in what order?* A developer who consults the injection component of the Business Layer — `B` — immediately obtains the complete list of business logical components available in the artifact, their canonical names, and the order in which they are initialized. This self-documentation property is one of the most practical benefits of the model for development teams.

The XF model defines exactly **three injection components** in every artifact, one per layer:

- `R` — injection component of the Access Layer, which aggregates the unique instances of the logical components of type `Repository`.
- `B` — injection component of the Business Layer, which aggregates the unique instances of the logical components of type `Business`.
- `A` — injection component of the Interaction Layer, which aggregates the unique instances of the logical components of type `Service` and `View`.

Nature of the injection component The injection component is a **non-instantiable** class — it cannot be the object of instantiation by any component of the artifact. No one creates a `new B()` or a `new R()`. This property is intentional and has two direct normative consequences.

The first is that access to the logical components through the injection component is always **static** — the references to the instances of the logical components are public static attributes of the injection component, accessible directly via `B.session`, `R.database`, `A.temperatureService` without the need to instantiate the injection component.

The second is that the injection component **maintains no state** — it has no instance attributes, has no state-mutation operations, and cannot be observed. Its sole function is to aggregate static references to instances of logical components and to manage their lifecycle through the initialization and termination operations.

The static references that the injection component maintains are **post-initialization immutable**: they are assigned exactly once during the `init()` operation of the injection component and shall not be reassigned during the execution of the artifact. Before `init()` has been invoked, the references may be in an uninitialized state — the XF model prescribes that no component of the artifact access the references of an injection component before its `init()` operation has completed. Non-compliance with this prescription may produce race conditions or accesses to uninitialized references whose behavior is undefined.

The lifecycle managed by the injection component The injection component manages the complete lifecycle of the logical components of its layer by means of an **initial instantiation** — which establishes the initial state σ_0 of each logical component — and **two invocable operations** — initialization and termination — which the XF model prescribes with precision:

The **initial instantiation** is the declaration, by the injection component, of the unique instances of the logical components of the layer as public static attributes. It is not an invocable operation: it is the effect, on the state of the artifact, of the language's initialization of static fields upon loading the singleton class — before any operation of the injection component is invoked —, the result of invoking the constructor of each logical component. This instantiation fulfills two functions: it establishes which logical components exist in the layer and under what canonical name they are accessible and, simultaneously, it establishes the **initial state σ_0 of each logical component** — the state in which each logical component finds itself immediately after its construction, before invoking `init()` —. The injection component does not possess mutable state of its own: σ_0 is the property of the instantiated logical component, not of the injection component that instantiates it. It is important to distinguish that **instantiation** — the creation of the object in memory by means of the constructor — is a programming-language operation that may execute initialization logic for the logical component's internal attributes, but **shall not** execute logic that depends on external resources nor invoke operations of other components. Any startup logic that depends on external resources shall be delegated to the `init()` operation of the logical component.

```
1 // Injection component B - initial instantiation of the logical components
```

```
final abstract class B {

    // Initial state sigma_0 - instantiation of the logical
    // components of the layer upon loading the singleton class

    static readonly SessionBusiness session = new SessionBusiness();

    static readonly UserBusiness user = new UserBusiness();

    static readonly TemperatureBusiness temperature = new
        TemperatureBusiness();

    static readonly NetworkBusiness network = new NetworkBusiness();

}
```

Listing 14. Lifecycle (I): initial instantiation of the logical components

The **initialization** is the execution of the startup logic of each logical component of the layer, delegated through the `init()` operation of each logical component. The initialization is the responsibility of the XF model — it is the operation that prepares each logical component to process the operations for which it was designed, establishing connections with external systems, starting timers for scheduled tasks, loading configuration from external sources, or establishing the initial state of the component.

```
// Injection component B - complete composition with init() operation
final abstract class B {

    static readonly SessionBusiness session = new SessionBusiness();

    static readonly UserBusiness user = new UserBusiness();

    static readonly TemperatureBusiness temperature = new
        TemperatureBusiness();

    static readonly NetworkBusiness network = new NetworkBusiness();

    // Initialization - in order of dependencies between components
    static void init() {
        B.network.init(); // NetworkBusiness does not depend on other
            components
        B.session.init(); // SessionBusiness may depend on NetworkBusiness
        B.user.init(); // UserBusiness may depend on SessionBusiness
        B.temperature.init(); // TemperatureBusiness may depend on both
    }
}
```

Listing 15. Lifecycle (II): complete composition with the init() operation

The **termination** is the execution of the shutdown logic of each logical component of the layer, delegated through the `terminate()` operation of each logical component in the reverse order of initialization. The termination releases the resources acquired during initialization — it closes connections with external systems, stops timers for scheduled tasks, persists the state that must be preserved between executions, and leaves the system in a defined resting state for the next execution of the artifact. As a best practice, the artifact invokes `terminate()` before the completion of its process whenever the execution environment permits it — both under controlled-shutdown conditions and upon the detection of unhandled exceptions —; the effective invocation depends on the environment, and the model's policy regarding this situation is developed in §8.3.

```
// Injection component B - terminate() operation in reverse order
final abstract class B {

    // ... definition and initialization ...

    // Termination - in the reverse order of initialization
    static void terminate() {
        B.temperature.terminate();
        B.user.terminate();
        B.session.terminate();
        B.network.terminate();
    }
}
```

Listing 16. Lifecycle (III): termination in reverse order

The access pattern to logical components Every access to a logical component from any point of the artifact — including components of the same layer — is performed exclusively through the injection component of the layer to which the invoked logical component belongs, following the pattern:

```
<injection_component>.<logical_component>.<operation>(<parameters>)
```

Listing 17. Canonical access pattern: general form

This pattern applies without exception. A business logical component that needs to invoke an operation of another business logical component does so through `B`, not by means of a direct reference between components:

```
// Use of the injection component - canonical access pattern from a logical
    component
class UserBusiness {

    User getUser(String token) {

        // Access to SessionBusiness through the injection component B
        Session session = B.session.decode(token);
        return session.user;
    }
}
```

Listing 18. Canonical access pattern: usage example

The prohibition of direct instantiation of logical components outside the injection component is absolute. The creation of an instance of a logical component outside the injection component — by means of `new UserBusiness()` at any point of the artifact other than the injection component `B` — constitutes a violation of the model regardless of the context in which it occurs. This restriction guarantees that the uniqueness of instance prescribed by the model is statically verifiable by the conformance analysis tools.

Concurrency and uniqueness of instance The XF model prescribes logical uniqueness of instance within the execution space of the artifact: there exists exactly one instance of each logical component per execution context. In concurrent execution environments — multiple threads accessing the artifact simultaneously — the concrete coordination between the initialization of the static references and the rest of the artifact's logic is the implementer's decision and depends on the concurrency model of the implementation language; the XF model does not prescribe mechanisms of

mutual exclusion, atomicity, or order of visibility between threads, by remaining agnostic with respect to the concurrency model.

When the artifact is executed simultaneously in multiple instances with disjoint memory spaces, each deployed instance constitutes an **independent XF artifact**, with its own set of singleton instances managed by its injection components. The internal architecture of each of those artifacts is fully conformant to the model and is subject to all of its prescriptions. The coherence between distinct instances — state synchronization, event propagation, consensus — pertains to the communication infrastructure between artifacts, not to the internal structure of any of them. It is that inter-artifact coherence, and not the distributed-execution case in itself, that falls outside the scope of the XF model, in direct application of the principle of agnosticism with respect to the deployment infrastructure (§6.2.1).

Normative properties The first property is **uniqueness**: there exists exactly one injection component per layer in each XF artifact root. There shall not exist multiple injection components for one and the same layer, nor shall there exist an injection component that aggregates logical components of more than one layer.

The second property is **functional exhaustiveness**: the injection component aggregates the instances of **all** the logical components of its layer that must be accessible from other components of the artifact. A logical component that is not registered in the injection component of its layer is not accessible from any other component of the artifact. This property is stated as a design criterion and not as a normative rule: there may exist logical components that are declared but deliberately not exposed through the injection (for example, internal logical components of a library that are used only within their own package).

The third property is **centrality of access**: the injection component is the sole point of access to the logical components of its layer, and this access is reserved to other logical components. The prescription follows by direct inference from two properties established in §7.3.1: (P1) the logical components are the only components of the artifact that encapsulate the effective domain logic, and (P2) the effective logic can only be executed by the logical components that encapsulate it. Given that access to a logical component through its injection component — the canonical pattern `<injection>.<component>.<operation>()` — is by definition an invocation of the effective logic encapsulated in that logical component, (C) such references can only originate in logical components. Generalization, utility, transfer, or other injection components shall not reference slot members of any injection. There exists no alternative mechanism of access to a logical component that is compatible with the XF model.

The fourth property is **post-initialization immutability**: the references to the logical components are assigned during the definition and initialization of the injection component and shall not be modified during the execution of the artifact. The structural constancy of the artifact at the global level during the execution phase is additionally guaranteed as a formal property of the model, without mechanical operationalization: the prescription is the observable property, not a concrete mechanism.

Restrictions The restrictions of the injection component are grouped into two blocks: the structural ones, which prescribe the composition of the class and the declarable members, and the lifecycle ones, which prescribe the invocable operations and the conditions of their use.

Structural restrictions An injection component **shall not** be instantiable. Its implementation shall explicitly prevent the creation of instances — by means of a final abstract class, a private constructor, or another equivalent mechanism provided by the programming language.

The static references to logical components declared in the injection component **shall** have public visibility, so that they are accessible from any point of the artifact by means of the canonical pattern `<injection>.<component>.<operation>()`. The applicability of the rule is trivial in languages that do not differentiate the visibility of static members.

An injection component **shall not** declare members other than typed static references to logical components of its layer and the static operations `init()` and `terminate()` that orchestrate the lifecycle of those logical components. Any other member — an attribute not typed to a logical component of the layer, a static operation other than `init()` or `terminate()`, domain logic embedded anywhere in the class — constitutes a violation of the model.

An injection component **shall not** be accessed from a layer of a lower level of abstraction than the layer to which it belongs. The injection component `A` of the Interaction Layer shall not be accessed from the Business Layer nor from the Access Layer. The injection component `B` of the Business Layer shall not be accessed from the Access Layer. The injection component `R` of the Access Layer may be accessed from the Business Layer and from the Interaction Layer.

An injection component **shall not** inherit from any component, internal or external to the artifact. The injection is a unique singleton per layer whose contract is fixed by the model — a closed set of members, non-instantiable, purely static, with two operations `init()` and `terminate()` prescribed by the model —, and inheritance would introduce additional inadmissible members or would conflict with the structural uniqueness of the injection component. The injection does not participate in inheritance hierarchies.

Lifecycle restrictions An injection component **shall** declare the static operation `init()`. This operation orchestrates the ordered initialization of the logical components of its layer; its absence prevents the artifact from invoking the initialization of the layer.

An injection component **shall** declare the static operation `terminate()`. This operation releases in a controlled manner the resources acquired during `init()`; its absence prevents the artifact from invoking the controlled termination of the layer, regardless of whether the execution environment guarantees its invocation.

The body of the `init()` operation of an injection component **shall not** contain statements other than invocations of the `init()` operation of the logical components aggregated as slots in that same injection. The injection is a pure orchestrator

of the initialization of its layer: any other statement — invocations of operations other than `init()`, invocations of other injections, processing logic, I/O against the environment, access to external resources — exceeds the formal role of the Singleton Gathering and constitutes a violation.

The body of the `terminate()` operation of an injection component **shall not** contain statements other than invocations of the `terminate()` operation of the logical components aggregated as slots in that same injection. It constitutes the symmetric restriction, over the termination phase, of the one prescribed for the `init()` operation.

The `init()` and `terminate()` operations of an injection component **shall** be symmetric in their delegation: every logical component whose `init()` is invoked by the injection's `init()` **shall** have its `terminate()` invoked by the `terminate()` of that same injection, in reverse order. The asymmetry — components initialized without a corresponding termination, or terminated without having been initialized — constitutes a violation of the formal property of resource symmetry of the lifecycle (§8.2).

The invocation of the `init()` and `terminate()` operations of a logical component **is reserved** to the injection component of its layer. No other component of the artifact — neither logical components of the same or another layer, nor generalizations, nor utilities, nor transfers, nor other injections — may invoke `init()` or `terminate()` on a logical component. The orchestration of the logical component's lifecycle is confined to the injection component of its layer.

The invocation of the `init()` and `terminate()` operations of an injection component **is reserved** to the execution start point of the artifact. No classifiable component of the artifact — logical, generalization, utility, transfer, or another injection — may invoke `R.init()`, `B.init()`, `A.init()` nor their symmetric `terminate()` counterparts.

The existence of at least one valid invocation order among the injected logical components is a property derived from the model: as a consequence of applying the principle of layer isolation (§6.2.2), the inter-layer dependency graph turns out to be acyclic (a property derived from the model). The concrete choice of the order, among the topologically valid ones, is left to the discretion of the implementer. The invocation order **between** the injections `R`, `B`, and `A` respects the descending dependency between layers: `R.init()` precedes `B.init()`, which precedes `A.init()`, and `terminate()` is invoked in reverse order.

Canonical nomenclature The three injection components of the XF model have fixed canonical names that admit no variation:

- `R` — injection component of the Access Layer.
- `B` — injection component of the Business Layer.
- `A` — injection component of the Interaction Layer.

Organization in the folder structure Each injection component is located in the root of the folder of its corresponding layer, at the same level as the subfolders `/logic`, `/general`, `/utils`, and `/transfers`:

```
/src
├── /repository
│   └── R (Injection component of the Access Layer)
├── /business
│   └── B (Injection component of the Business Layer)
└── /api
    └── A (Injection component of the Interaction Layer)
```

Listing 19. Organization in folders: injection component

7.3.4 Utility components

Definition A utility component is a component that provides auxiliary operations of local scope to its layer, which serve as contextual support to the logical components of that layer for the achievement of their effective logic, but which do not by themselves resolve any problem directly related to the domain of the artifact. Its function is to encapsulate general-purpose algorithms and transformations that, were they not to exist, would have to be duplicated across multiple logical components of the same layer.

Pragmatically, a utility component is the answer to the question: *which auxiliary operations do the logical components of this layer need in order to execute their effective logic, that do not belong to the domain of the artifact?* A business logical component that manages temperature needs to convert values between units of measurement — Celsius, Fahrenheit, Kelvin — before applying its business rules. That conversion is not business logic — it models no domain concept, it does not modify the state of any logical component, and it does not depend on any specific business rule. It is general-purpose auxiliary logic that can be encapsulated in `TemperatureUtils` and be reused by any business logical component that needs it.

Distinction with respect to logical components: absence of modeling and of side effects The distinction between a utility component and a logical component is operational and verifiable by means of two complementary criteria.

The first is the criterion of **absence of domain modeling**: a utility component may operate on concepts of the domain of the artifact — filter, map, group, structurally transform their instances, read their attributes — but shall not implement the rules that define their transitions, their semantic transformations, or the conceptual modeling of the domain. `TemperatureUtils` may convert numeric values between scales (Celsius, Fahrenheit, Kelvin) because the conversion is a mathematical operation applicable to any thermal magnitude, not a rule of the domain of the artifact. If, on the other hand, it applied thresholds defined by the business, decided whether a temperature is “critical” or “safe”, or executed the thermostat’s response logic to a variation, those operations would be rules of the domain of the artifact and would belong to a logical component — not to a utility.

The second is the criterion of **absence of side effects**: a utility component does not modify the state of any component of the artifact, does not invoke operations of logical components through the injection components, does not perform direct I/O against the system or the network, does not mutate its arguments by reference, and does not depend on mutable external state. The operations of a utility component

are pure functions in the sense of functional programming — given the same input, they always produce the same output. If an operation produces any of the above effects, it belongs to a logical component.

Structural properties Utility components have a set of structural properties that distinguish them from the other types of components and that the XF model prescribes explicitly.

Utility components **are not instantiable**. They have no public constructor and cannot be the object of instantiation by any component of the artifact. Their operations are static — accessible directly by means of the name of the utility component without the need to instantiate it: `TemperatureUtils.toCelsius(temperature)`, `JsonUtils.serialize(object)`, `DateUtils.format(date, pattern)`.

Utility components **are extensible**. A utility may be the base of another utility that adds static operations to its catalog. Extensibility is orthogonal to non-instantiability: the non-instantiability of a utility derives from the static character of its operations, not from an `abstract` modifier; neither the base utility nor the derived one needs to be declared abstract for the inheritance to be legitimate. The only thing the extension requires is that the declaration of the base utility permit it (non-final, non-sealed, or equivalent according to the language) and that the derived utility respect the same contract of the type — static operations, without mutable state, non-instantiable. This extensibility is what allows the ecosystem of XF libraries to publish base utilities (§10.3) that consuming artifacts extend with their own operations without duplicating the imported catalog.

Utility components **do not maintain state**. They have no instance or class attributes that are mutated during the execution of the artifact. All of their attributes, if they have any, are constants — immutable values that parameterize the behavior of their operations without constituting state in the formal sense.

These three properties together place utility components in an imperative or functional programming paradigm, unlike the object-oriented paradigm that characterizes the other types of components of the XF model. This distinction is not incidental — it reflects the nature of the logic they encapsulate: stateless operations, without side effects and without dependencies among them.

Local scope and the exception of primitive types The scope of a utility component is restricted to the layer in which it is defined. A logical component of the Business Layer may use `BusinessUtils` or any utility component of the Business Layer, but shall not directly use a utility component of the Interaction Layer — that dependency would violate the principle of layer isolation established in §6.2.2.

However, the XF model recognizes an explicit and well-delimited exception to this restriction of local scope: the utility components of the Access Layer that operate on **primitive types**.

Primitive types are defined in §3.1.15 in the sense of ISO/IEC/IEEE 24765:2017 [20]. The utilities that operate on primitive types — `StringUtils`, `DateUtils`, `NumberUtils`, `ArrayUtils` — do not depend on any concept of the domain of the artifact, do not assume any business semantics, and are applicable in any layer

without their use implying knowledge of the access logic that characterizes the Access Layer.

For these reasons, the XF model prescribes that utility components on primitive types be defined in the Access Layer and may be referenced by components of any layer without this constituting a violation of the isolation principle. This exception is justified because primitive types are, in the sense of the OSI model, prior to XF stratification — they do not belong to any layer in particular but rather are the raw material on which all layers operate.

```
// Valid use from any layer: a utility on a primitive type
// defined in the Access Layer may be referenced from Business
class TemperatureBusiness {

    void update(Temperature temperature) {
        // StringUtils is defined in the Access Layer.
        String formatted = StringUtils.format(temperature.value, 2);
        ...
    }

}
```

Listing 20. Local scope and the exception of primitive types

The utility components that do not operate on primitive types — `TemperatureUtils`, `QueryUtils`, `ColorUtils`, `ValidationUtils` — are subject to the restriction of local scope and may only be referenced by components of their own layer.

Normative properties The first property is the **absence of domain modeling**: a utility component may operate on concepts of the domain of the artifact but shall not implement the rules that govern them — domain state transitions, semantic transformations, decisions based on business thresholds, conceptual modeling logic —. That responsibility belongs exclusively to the logical components.

The second property is **functional purity**: the operations of a utility component are pure functions — they have no side effects, they do not modify the state of any component, they do not invoke injections, they do not perform direct I/O, they do not mutate their arguments by reference, and they always produce the same result for the same input.

The third property is **layer locality**: the scope of a utility component is restricted to the layer in which it is defined, with the exception of the utility components on primitive types of the Access Layer, which may be referenced from any layer.

Restrictions A utility component **shall not** be instantiable. Its implementation shall explicitly prevent instantiation by means of the mechanisms that the programming language provides — inaccessible constructor that throws when invoked, static class, singleton object, functional module, or equivalent. This prescription controls exclusively non-instantiability — a property derived from the static character of the operations of the type — and not its abstractness or its inheritability. Extension by inheritance is admissible when the declaration of the base utility permits it, without this requiring the base to be declared abstract; the derived utility is subject to the same contract of the type.

A utility component **shall not** declare instance members — non-static attributes or methods —. The utility is purely static: it operates in an imperative or functional paradigm, not an object-oriented one. Its only admissible members are static operations and immutable static constants.

A utility component **shall not** maintain mutable state. All of its attributes, if it has any, shall be immutable constants.

A utility component **shall not** implement logic that exceeds its auxiliary role: neither rules of the modeling of the domain of the artifact, nor invocations of logical components through the injection components, nor operations that produce observable side effects — direct I/O against the system or the network, mutation of arguments by reference, dependencies on mutable external state —. If an operation produces any of those effects or implements any of those rules, it belongs to a logical component.

A utility component **shall not** be referenced by components of layers of a lower level of abstraction than the layer in which it is defined, except in the case of the utility components on primitive types of the Access Layer.

A utility component **shall not** inherit from any component other than another utility component of its own layer. The utility is a grouper of pure static operations and does not participate in heterogeneous inheritance hierarchies; inheritance chains among utilities of the same layer (*e.g.* extending the catalog of operations of a base utility) are admissible, chains that cross types or layers are a violation of the model.

A utility component **shall** carry the `Utils` suffix in its name, so that its nature is immediately identifiable by any developer who consults the structure of the artifact.

Canonical nomenclature All utility components carry the `Utils` suffix — for example, `TemperatureUtils`, `QueryUtils`, `JsonUtils`, `ColorUtils`, `ValidationUtils`, `StringUtils`, `DateUtils`, `NumberUtils`.

Organization in the folder structure Utility components are grouped under the `/utils` subfolder within the folder of their corresponding layer:

```
/src
├── /repository/utils (Access utility components)
│   ├── StringUtils (Utility on a primitive type)
│   ├── DateUtils (Utility on a primitive type)
│   ├── NumberUtils (Utility on a primitive type)
│   └── QueryUtils (Access utility)
├── /business/utils (Business utility components)
│   └── TemperatureUtils (Business utility)
└── /api/utils (Interaction utility components)
    └── ColorUtils (Interaction utility)
```

Listing 21. Organization in folders: utility component

7.3.5 Transfer components

Definition A transfer component is a data structure that models the form of the information at a concrete point in the processing flow of the artifact, and that is used by the logical components to exchange information among themselves, both within a single layer and between layers. Unlike logical components — which *maintain* a state

that evolves in correspondence with the business process they model —, a transfer component *is* the data structure itself and does not model any business process. It may declare operations, provided they are **self-contained**: they operate exclusively on the data of the structure itself — querying them, transforming them, deriving them, or combining them — and do not access any other component of the artifact. Its function is to give that data a verifiable, named, and reusable form, together with the operations that the structure offers on itself.

Pragmatically, a transfer component is the answer to the question: *what form does the data have at this point of the processing flow?* When an access component retrieves temperature data from an external API, the result of that retrieval is a datum with a concrete form — a numeric value, a unit of measurement, and a timestamp. That form is modeled as a transfer component `Temperature` that may be used by any component of the artifact that needs to work with temperature data, regardless of the layer in which it is classified.

Distinction with respect to the other types: absence of business modeling The distinction between a transfer component and a logical component is not the presence of operations, but rather their **nature**: the operations of a transfer component are self-contained — they transform, query, or derive their own data — and do not model a business process of the domain nor orchestrate other components; that effective logic of the domain is the exclusive competence of the logical component. Its attributes describe the concept they model with the precision necessary so that any component that consumes them can interpret them without additional information.

Historically, the industry has used multiple terms to refer to data structures according to their context of use — `DTO`, `Entity`, `Model`, `POJO`, `Record`, `Schema`, `VO`, `Struct` — each with distinct usage connotations that have generated the semantic fragmentation described in §5.4. The XF model unifies all of these concepts under a single type: the transfer component. This unification does not imply that all data structures are equivalent in their semantics — it implies that all of them share the same structural nature in the model, and that the differences in semantics are managed by the logical components that process them, not by the classification of the transfer component itself.

Competence in the taxonomy and projection of framework structures This characterization fixes with precision the competence of the transfer component within the taxonomy. As opposed to the **logical component**, the transfer does not model a business process nor orchestrate other components; as opposed to the **utility component** — which groups pure and static operations, without an instance or data of its own —, the transfer is an *instantiable* structure whose operations act on the data that it itself carries. Consequently, the "rich" data structures — those that combine data and self-contained operations on that data — are transfer components, not logical components.

This competence closes the semantic gap between the model and the development framework: the structures that frameworks and standard libraries already provide with this nature — collections, date and time structures, futures and promises, interface controls, timers, result wrappers — **project** to transfer components of the

XF model, since they carry operations without modeling any business process of the artifact. The implementer classifies them as transfers in the layer corresponding to the concept they model, without the need to rewrite them: the model recognizes their structural nature and integrates them into the matrix $L \times T$ like any other transfer.

The following fragment illustrates a transfer component with self-contained operations:

```
// /src/repository/transfers/Temperature
// Transfer with data and self-contained operations on that data.
class Temperature {
    double celsius;       // own datum
    DateTime measuredAt;  // own datum

    // self-contained operations: query or derive the own data
    double toFahrenheit() { return this.celsius * 9 / 5 + 32; }
    double toKelvin()     { return this.celsius + 273.15; }
    int compareTo(Temperature other) { return Double.compare(this.celsius,
        other.celsius); }
    boolean isAbove(Temperature other) { return this.celsius > other.celsius; }

    // Does NOT access R, B, A or other components; does NOT model a business
        process.
    // Deciding whether to turn on the heating according to the temperature IS
        a business process:
    // it lives in a logical component (Business), not in the Temperature
        transfer.
}
```

Listing 22. Transfer component with self-contained operations

Unification of structures as a capability of the model The XF model **enables** the unification of structures as an emergent capability: a single transfer component may be shared by all the layers that work with the same concept of the domain of the artifact without the need to duplicate it per layer. The domain of the artifact includes concepts of the three layers — not only of the business — and the capability applies to all of them equally: to a business concept such as `User` or `Order`, to an access concept such as `Query`, `HttpRequest` or `Cursor`, and to an interaction concept such as `Button`, `MouseEvent` or `Viewport`. This capability contrasts with the traditional approach of defining distinct structures per layer for a single concept — a `UserEntity` in the Access Layer, a `UserModel` in the Business Layer, and a `UserDTO` in the Interaction Layer — which produces a proliferation of redundant types, duplication of attributes, and mapping operations between structures that add no value to the artifact.

The unification is a *capability*, not a prescription: the model does not oblige the implementer to unify transfers of the same concept in distinct layers, and two structurally equivalent transfers in distinct layers constitute a redundancy, not a violation. The model recommends it as a good design practice because it reduces the complexity of the artifact, eliminates the need for mapping operations between layers, and makes the evolution of the data model a change localized at a single point. The unification is *possible* because references to transfer components, even when crossing layer boundaries, always do so in the descending direction prescribed by the isolation principle — a higher layer consumes structures defined in lower layers, never the reverse — subject to the conditions of directionality and of non-modification that are developed below.

Directionality and non-modification of transfers between layers The transfer component presents a **direct referenceability between layers** that the other types of component lack. For the rest of the types, access to another layer is channeled through the injection component of the immediately lower layer (§7.3.3); a transfer component, on the other hand, may be referenced directly from another layer, even crossing more than one boundary. This referenceability does not constitute an exception to the isolation principle: it preserves its descending direction — a transfer component **may** be referenced by components of its own layer or of layers of a higher level of abstraction, and **shall not** be referenced by components of layers of a lower level of abstraction. The complete matrix of references permitted between layers is developed in §9.4.

Non-modification is a **design guideline**, not a prescription: the direct reuse of a transfer component between layers is appropriate when the structure is consumed without modifying its definition; when a layer needs added or removed attributes with respect to the structure received from a lower layer, the canonical option is to define a new transfer component in its own layer. The modification of the values of the attributes of an existing structure does not require defining a new one. The model does not mechanically operate this guideline: distinguishing a structural transformation from a mere reuse is not decidable by static analysis, but rather requires domain judgment (§9.4). The guideline may be stated as follows:

If a layer **consumes** a structure from a lower layer without modifying its definition, it **may** reference it directly. If a layer **transforms** the definition of a structure from a lower layer — by adding or removing attributes —, the canonical option is to **define a new transfer component** in its own layer that represents the result of that transformation.

This guideline favors each layer being responsible for the definition of the structures it produces, preserving the traceability of the data transformations throughout the processing flow of the artifact.

The following code fragment illustrates the application of both restrictions:

```
// Transfer component defined in the Access Layer

// /src/repository/transfers/User
class User {
    long id;
    String name;
    String email;
    String passwordHash; // sensitive attribute - shall not be exposed in the Interaction Layer
    DateTime createdAt; // audit attribute - not relevant in the Interaction Layer
    boolean isActive;
}

// The Business Layer consumes User without modifying it - it may reuse it directly
class UserBusiness {

    User getUser(String token) {
        Session session = B.session.decode(token);
        return R.identity.fetchUser(session.userId); // returns User from the Access Layer
    }
}
```

Listing 23. Transfer defined in the Access Layer

```
// Transfer component of the Interaction Layer - structural transformation
// The Interaction Layer transforms User - it shall define its own transfer
    component

// /src/api/transfers/UserResponse
class UserResponse {
    long id;
    String name;
    String email;
    // passwordHash removed - sensitive datum
    // createdAt removed - not relevant for the consumer
    // isActive removed - not relevant for the consumer
}

// The interaction logical component applies the transformation
class UserService {

    UserResponse getUser(String token) {
        User user = B.user.getUser(token);
        return new UserResponse(user.id, user.name, user.email); // transforms
            User into UserResponse
    }
}
```

Listing 24. Structural transformation of the transfer in the Interaction Layer

Normative properties The first property is **self-containment**: a transfer component may declare operations, but these operate exclusively on the data of the structure itself and do not model business processes nor access other components of the artifact. The modeling of the effective logic of the domain is the exclusive competence of the logical components.

The second property is **semantic independence from the framework**: the attributes of a transfer component describe the concept of the domain of the artifact, not the conventions of the development framework or of the technological ecosystem of implementation. Annotations, decorators, type attributes, or other syntactic constructs of the language that the framework requires for reasons of serialization, persistence, or deployment are admissible when they are strictly necessary for those ends and do not alter the semantics of the modeled structure. The model does not prescribe the mechanical verification of this property: the membership of an attribute in the domain of the artifact is a decision of the implementer, and the deviations that compromise portability — attribute types bound to a technology, field names altered by serialization annotations — are detectable from the name of the component itself or remain a matter of architectural review.

The third property is **reference directionality**: a transfer component may only be referenced by components of layers of a higher or equal level of abstraction than the layer in which it is defined. The rule applies uniformly to transfers of the artifact itself and of integrated external artifacts, without distinction.

Restrictions The operations of a transfer component **shall** be self-contained: they operate exclusively on the data of the structure itself. A transfer component **shall not** access any other component of the artifact from its operations — neither the injection components `R`, `B` and `A`, nor logical, generalization, or utility components of any layer.

A transfer component **shall not** model a business process of the domain. Every operation that implements the effective logic of the domain, coordinates several components, or produces effects outside the structure itself corresponds to a logical component, not to a transfer: it is that boundary — and not the mere presence of operations — that separates the transfer from the logical component.

The constructor of a transfer component, like the rest of its operations, **shall not** access other components of the artifact — in particular, it shall not access logical components through the injection components — nor depend on resources of the environment: it initializes the data of the structure itself from its parameters, from constant values, or from self-contained operations on that data.

A transfer component, if it inherits, **shall** inherit exclusively from another transfer component. Inheritability between transfers is permitted — it is the natural mechanism when a specialized transfer extends a base structure published by an XF library (§10.3) with its own attributes without losing the capacity to be treated as an instance of the base type — provided that the subclass preserves the structural modeling without function and respects the content restrictions prescribed for the type. The directionality of the inheritance between layers is additionally governed by the isolation principle (§6.2.2).

A transfer component **shall not** be referenced by components of layers of a lower level of abstraction than the layer in which it is defined.

Canonical nomenclature Transfer components **shall not** carry a suffix added to the domain concept they model: their name describes that concept directly, without an additional label of data structure type. Canonical examples: `User`, `Temperature`, `Schedule`, `Session`, `Product`. The verification is semantic and depends on the judgment about what constitutes the "domain concept" in the artifact under analysis.

The transfer components that are typically transmitted as exceptions of the language — `TemperatureOutOfRangeException`, `NetworkUnavailableException`, `AuthenticationFailedException`, `DatabaseConnectionException` — may carry the `Exception` suffix when the language or the convention of the ecosystem favors it, but the model does not prescribe it: the choice of the syntactic construct of transmission is the competence of the implementer (§9.5) and does not generate structural asymmetry in the formal domain $\mathbb{C}_T$.

Organization in the folder structure Transfer components are grouped under the `/transfers` subfolder within the folder of the layer in which they are defined:

```
/src
├── /repository/transfers (Access transfer components)
│   ├── User (User data structure -- origin in the Access Layer)
│   ├── Temperature (Temperature data structure)
│   └── Wrapper (API response wrapper structure)
│
├── /business/transfers (Business transfer components)
│   └── Session (Session structure -- defined and transformed in the Business
    Layer)
│
└── /api/transfers (Interaction transfer components)
    ├── UserResponse (Transformation of User for exposure in the Interaction
        Layer)
    └── TemperatureOutOfRangeException (Interaction exception)
```

Listing 25. Organization in folders: transfer component

The following table contrasts the traditional modeling with the unification of structures of the XF model.

Table 4. Contrast between the traditional modeling with multiple structures per layer and the modeling of the XF model with a single shared structure that circulates between layers. Each row represents the same conceptual information (`User`) as it is modeled in each paradigm. The XF model enables the unification of structures as a design capability; the definition of a new transfer in the face of a structural transformation is a guideline, not a prescription.

Layer where it is used	Traditional modeling	XF modeling
Access	`UserEntity` (with `@Entity`, JPA annotations, mapping to table)	`User` (single structure)
Business	`UserModel` (object enriched with business methods, validations)	`User` (the same; without redefinition)
Interaction	`UserDTO` / `UserViewModel` / `UserResponse` (serializable fields, without logic)	`User` (the same; except if it is structurally transformed, in which case it is `UserResponse` in Interaction)
Mapping operations between layers	`UserMapper`, `UserAssembler`, manual or automatic converters	None; the layers share the structure directly
Cost of evolution (adding an attribute)	Modify each structure per layer and all affected mappers	Modify the single structure in its layer of origin

7.4 Canonical folder structure

Definition The canonical folder structure is the physical materialization of the $L \times T$ classification matrix in the file system. Each element — layer folder, type subfolder and injection component — corresponds one-to-one to an element of the model: the first-level folders represent the abstraction layers, the second-level subfolders the functional types within each layer, and the injection components reside at the root of their layer folder as a centralized access point to their logic components.

Any folder structure whose subfolders organize the $L \times T$ matrix in this way constitutes an **artifact root**: the structural materialization of an **artifact** in the sense of §3.1.1 — a self-contained piece of code that formalizes a process, is structured into the three abstraction layers and runs in a single execution space. The correspondence structurally defines the artifact and is one-to-one: **an artifact materializes in exactly one artifact root, and every artifact root materializes exactly one artifact**. An artifact root is, in these terms, any point in the file tree from which a complete $L \times T$ matrix hangs.

What is normative is that matrix organization and the canonical names of its folders; the *physical location* of the root is not. The recommended location is the `/src` folder at the project root, but normatively an artifact root is **any** folder structure that organizes the $L \times T$ matrix, whatever the path that contains it. This abstraction is

what allows languages and ecosystems with different location conventions to project onto the model without exception.

That matrix organization has normative rank — not mere recommendation — for the same reason that motivates closed and exhaustive typing (§6.2.4): a folder structure that faithfully reflects the artifact's $L \times T$ matrix is structural documentation embedded in the project itself, inspectable without the need for additional tools. A developer accessing a well-formed XF artifact for the first time orients with the same speed regardless of the domain, the language or the framework — the organization is always the same.

Prescribed structure In its recommended location under `/src`, the folder structure of an XF artifact is as follows:

```
/projectName
├── /src
│   ├── /api (Interaction Layer)
│   │   ├── /general (Generalization components)
│   │   ├── /logic (Logic components)
│   │   ├── /transfers (Transfer components)
│   │   ├── /utils (Utility components)
│   │   └── A (Injection component)
│   │
│   ├── /business (Business Layer)
│   │   ├── /general (Generalization components)
│   │   ├── /logic (Logic components)
│   │   ├── /transfers (Transfer components)
│   │   ├── /utils (Utility components)
│   │   └── B (Injection component)
│   │
│   └── /repository (Access Layer)
│       ├── /general (Generalization components)
│       ├── /logic (Logic components)
│       ├── /transfers (Transfer components)
│       ├── /utils (Utility components)
│       └── R (Injection component)
└── [...] (Rest of the project files)
```

Listing 26. Prescribed structure

Folder naming convention and its normative justification The names of the three layer folders — `/repository`, `/business`, `/api` — and of the four type subfolders — `/general`, `/logic`, `/transfers`, `/utils` — are **established industry terms**, adopted by alignment with the type of component they host: `/repository` (from the *Repository* pattern [12], the most recognizable term for the role of the Access Layer), `/business` (a universal term for domain logic; aligned with the `*Business` suffix and the `B` injection), `/api` (the public interface of the artifact: it covers the systemic entry points `*Service` and the graphical views `*View`; aligned with the `A` injection), `/general` (from the generalization component; preferred over `base` or `abstract` to avoid semantic collisions), `/logic` (logic components, where the layer's effective logic resides), `/transfers` (transfer components; preferred over `dto`, `model`, `entity` or `vo` so as not to presuppose their connotations) and `/utils` (utility components; matches the `*Utils` suffix).

Once adopted, these names are a **verifiable and immutable normative contract**: conformance operates on the exact names — an artifact with `/access` instead of `/repository` is not recognized as conformant — and they are preserved in any

translation, adaptation or future extension of the model. Their invariability is the physical expression of the principle of technological agnosticism (§6.2.1).

File naming convention Every XF component **shall** be defined in a file whose name, without extension, matches the canonical name of the component class declared in that file. This one-to-one correspondence between file and component is the basis on which the model's structural verification operates: verification identifies the components by the file name — without the need to inspect their content — and resolves type and layer membership from the file's location in the canonical structure. The prescription is trivial in languages that impose it by specification on public classes and is required by the model in any language, regardless of the mechanism the language provides to make it effective.

Elements of the canonical tree When a layer contains components, these **shall** reside under the first-level canonical folder corresponding to their layer — `/api`, `/business` or `/repository`; layer folders without components are not required. The three injection components — `A`, `B` and `R` — **shall** reside at the root of their layer folder when the layer contains at least one logic component.

The four second-level subfolders — `/general`, `/logic`, `/transfers` and `/utils` — are the **canonical spaces** of the four component types with their own folder. They are not required to be materialized empty: a layer that does not contain components of a type need not create its subfolder. When it does contain them, those components **shall** reside in the canonical subfolder of their type. Conformance is evaluated by the location of the existing components, not by the presence of the folders; the type membership of each component is determined by its location under the corresponding canonical subfolder.

The internal subdivisions of the `/logic` folder within each layer, whose specific organization responds to the criterion proper to each layer and is defined in §7.2 together with the detailed description of each layer, **are recommended** but not required; an artifact that does not implement them remains conformant to the XF model.

Location of the execution start-point The artifact's **execution start-point** — file `main.ts`, `App.java`, `main.go` or equivalent according to the language or framework, when the latter does not manage it implicitly — **shall reside outside the artifact root**, typically at the project root. It is not a component classifiable in the $L \times T$ matrix — neither logic, nor generalization, nor utility, nor transfer, nor injection —, and the artifact root contains exclusively classified XF components; it therefore falls outside the canonical structure. Its definition and its role in the artifact's initialization are developed in §8.1.

Correspondence with the $L \times T$ matrix The following table establishes the correspondence between each element of the folder structure and its category in the $L \times T$ classification matrix:

Table 5. Correspondence between the canonical folder structure and the $L \times T$ classification matrix.

Folder	Layer (L)	Type (T)
`/repository/logic`	Access	Logic
`/repository/general`	Access	Generalization
`/repository/transfers`	Access	Transfer
`/repository/utils`	Access	Utility
`/repository/R`	Access	Injection
`/business/logic`	Business	Logic
`/business/general`	Business	Generalization
`/business/transfers`	Business	Transfer
`/business/utils`	Business	Utility
`/business/B`	Business	Injection
`/api/logic`	Interaction	Logic
`/api/general`	Interaction	Generalization
`/api/transfers`	Interaction	Transfer
`/api/utils`	Interaction	Utility
`/api/A`	Interaction	Injection

Components outside the canonical structure The presence of components outside the canonical structure of their artifact root — in folders not recognized by the model, directly under the artifact root or in arbitrary locations of the project — constitutes a violation of the model.

In artifacts in the process of migration toward the XF model, the not-yet-migrated components coexist with correctly classified XF components. The XF model does not prescribe a specific way to organize the non-migrated components, but it recommends that their location be clearly distinguishable from the XF structure to facilitate tracking of the migration process.

Projection of the artifact root onto different ecosystems Since the artifact root is defined by the organization of the $L \times T$ matrix and not by its physical path, it projects without friction onto the location conventions of any ecosystem or language. The model does not accommodate itself to these environments: it recognizes them as distinct materializations of one and the same abstraction. The effective artifact root is, in each case, the point in the file tree from which the complete $L \times T$ matrix hangs; from that point on, the sub-structure — layer folders, type subfolders `/general`, `/logic`, `/transfers`, `/utils` and injection components `R`, `B`, `A` — and the canonical names apply without variation.

- **Ecosystem-specific source root.** Maven or Gradle on the JVM impose a source root (`src/main/java/`, `src/main/kotlin/`); the $L \times T$ matrix hangs from it — `src/main/java/repository/`, `.../business/`, `.../api/`.
- **Corporate package prefix.** Languages that require a corporate package as a prefix of the *namespace* (`com.<company>.<artifact>` in Java) nest the matrix under that subtree — `src/main/java/com/company/artifact/repository/`, and so on.

- **Multi-artifact project.** A single file tree may contain several independent artifact roots — monorepos, multi-module projects, microservice groupings —, each with its own $L \times T$ matrix, as developed below.

The projection does not relax any prescription: the model identifies the artifact root by the presence of the $L \times T$ matrix, not by a literal path, and over each root the canonical names, the total and exhaustive $L \times T$ classification and the conformance rules apply in full.

Multiple artifact roots A single source tree may contain **several independent artifact roots** — that is, several artifacts coexisting in a single tree. This arrangement is common in contemporary projects: monorepos (Nx, Turborepo, pnpm workspaces), multi-module projects (Maven/Gradle, .NET solutions, Go workspaces), architectures with library / application separation, systems organized by DDD bounded contexts, or microservice groupings in a single repository.

Each artifact root is detected by the presence of its $L \times T$ matrix — at least one canonical layer folder (`repository`, `business` or `api`) as a direct child — and is **maximal by inclusion**: no root is a descendant of another root detected in the same tree.

The set of detected roots constitutes the set of XF artifacts to verify. **Verification is executed independently on each root**: conformance verification is applied to each root separately, with its own $L \times T$ matrix, its own `R` / `B` / `A` triple and its own violation detection. The cardinality "one injection per layer" (§7.3.3) applies per root, not over the tree: two roots may coexist with their own triples without violation.

Code outside any detected root falls outside the scope of the model. The source tree may contain non-XF directories — configuration files, build scripts, auxiliary code, vendored external libraries, code under migration, tests outside the canonical hierarchy — without the model intervening. The XF model governs only what is structurally declared XF (that is, what lives inside some detected root).

The boundary between roots is structural (not physical): two roots in the same tree are as independent as two artifacts in different repositories.

The following example illustrates a conformant arrangement with two artifact roots and one non-XF folder coexisting under a Maven / Gradle source root:

```
src/main/java
├── com/company/app          <-- artifact root "app"
│   ├── repository/
│   ├── business/
│   └── api/
├── com/company/core         <-- artifact root "core"
│   ├── repository/
│   ├── business/
│   └── api/
└── com/company/shared       <-- not a root: no layer folders
    └── helpers.java         <-- outside the model, not verified
```

Listing 27. Multiple artifact roots

The roots `app` and `core` are verified independently. The folder `com/company/shared` is not a root (it does not contain `repository`, `business` or `api` as a direct child)

and its content falls outside the scope of the model, regardless of whether the names of the files it contains match canonical suffixes or not.

8 Instantiation of the architecture

The instantiation of the architecture is the process by which an XF artifact passes from being a static structure of classified components to being a running system capable of processing interactions. This process is not exclusive to the XF model — every software artifact requires a bootstrap mechanism that initializes its components in the correct order before the system can operate. What the XF model contributes is a way of understanding that mechanism in terms of the formal modelling of processes, endowing it with conceptual meaning independently of the development framework or execution environment employed.

The philosophy of the XF model in relation to instantiation rests on a principle that must be stated precisely before developing the concrete prescriptions: the model prescribes the conceptual structure of the instantiation process — what shall happen, in what order and why — but not the physical form in which that process is implemented in each development framework or execution environment.

Every development framework implements automatic mechanisms for managing the lifecycle of its components — dependency injection containers, view lifecycle managers, periodic task schedulers, message consumers. These automatic mechanisms do not violate the XF model — they implement it, with their own nomenclature and without the developer having coded them explicitly. The developer who knows the XF model understands why the development framework manages those lifecycles in the way it does: an Android `Activity` is a view whose lifecycle is managed by the operating system; a Spring `@Scheduled` is a `BusinessTask` whose periodic execution is managed by the container; a Kafka `Consumer` is a service whose invocation cycle is managed by the microservices environment. In all of these cases, the development framework is implementing the spirit of the model — the developer only needs to recognize that correspondence and ensure that the logic implemented within those mechanisms respects the prescriptions of the corresponding layer and component type.

The code that the developer writes explicitly is indeed subject to the normative prescriptions of the model in their entirety. What the development framework manages before, during or after that entry point already satisfies the spirit of the model from the perspective of the execution environment.

The clause is organized as follows. Clause §8.1 describes the execution start point and its relationship with the initialization of the injection components. Clause §8.2 describes the complete lifecycle of the artifact in four phases — instantiation, initialization, execution and termination —, its normative prescriptions and the formal representation as an automaton. Clause §8.3 describes the element of the XF model that acts as the centralized control point of instantiation, the `XF` start-point element.

8.1 Execution start point

Definition The **execution start point** of an artifact is the first fragment of the developer's code that the execution environment invokes upon loading the artifact. It is the causal origin of all observable activity of the artifact in memory and the place where control crosses from the execution environment into the artifact: at that instant the artifact exists as a static structure of classified components — constructed in memory with their initial state σ_0, but not yet operational —. Making the artifact an active system, capable of processing the interactions for which it was designed, is the function of the initialization operation that the start point invokes, not of the start point itself.

The start point is a concept of the model, not a fixed entity of the language: it manifests physically in different forms according to the execution environment — global bootstrap function, bootstrap module, file designated by the environment configuration, handler function invoked by the infrastructure, static member of a class, or another mechanism of the underlying environment. The XF model formalizes the architectural function of the start point, but its normative scope does not reach it: the prescriptions of the model begin at the components and, where applicable, at the `XF` start-point element (§8.3). The start point belongs to the execution environment; the model describes its role without imposing conformance prescriptions on its code.

Architectural function The start point fulfils two inseparable architectural functions:

1. **Boundary with the execution environment.** The start point is the only place where control crosses from the execution environment into the artifact. At that instant the artifact is loaded in memory but not operational: its components exist as objects whose initial state σ_0 has not yet been validated by the bootstrap logic. The transition to operational is not produced by the start point, but by the initialization operation that is invoked from it.
2. **Origin of lifecycle orchestration.** The start point is the only place from which the **canonical initialization sequence** is legitimately invoked. This sequence makes the artifact operational and **shall** complete before any other activity of the artifact logic.

Normative prescription The XF model prescribes that the **initialization of the artifact** is materialized as the canonical sequence of invocations `R.init(); B.init(); A.init();` in ascending order of abstraction, and its **termination** as the sequence `A.terminate(); B.terminate(); R.terminate();` in reverse order. Initialization is the first operation executed on the artifact and termination the last: no operation of the artifact logic may precede initialization nor follow termination.

These sequences are invoked from the execution start point, which lies outside the normative scope of the model: the prescription falls on the injection components `R`, `B` and `A` — which **shall** expose the operations `init()` and `terminate()` — and on the order of the sequence, not on the code of the start point. The registration of termination with the shutdown mechanism of the environment, when the latter provides one, occurs at the start point; the model does not prescribe the availability of such mechanisms in all environments.

The prescription operates on the canonical sequence as an ordered whole; the model does not prescribe the physical site where it is invoked, nor whether the three invocations are performed directly from the start point or are encapsulated in a single operation that orchestrates them.

External boundary of the artifact As a consequence of initialization having to complete before any other activity of the artifact logic, no external consumer may invoke the operations of the artifact's Interaction Layer before initialization has completed. Until that moment, the entry points of the artifact are not operational: the Interaction Layer has not been initialized and the artifact is not ready to process interactions.

The boundary does not apply to the internal invocations between logical components during initialization itself: in `init()` the logical components are already instantiated — their instantiation completes atomically as an effect of loading the classes of the injection components `R`, `B` and `A` (§8.2) — and they may invoke one another through the injection components provided they respect the descending hierarchy between layers established in §6.2.2. The only invocation prohibited in all cases is the one produced from the constructor of a component: during instantiation, the static references may not yet be resolved and accessing them produces undefined behaviour.

Automatic mechanisms of the execution environment Modern execution environments implement automatic lifecycle-management mechanisms that may precede, follow or partially replace the developer's start point: dependency injection containers, view lifecycle managers, periodic task schedulers, message consumers. The XF model prescribes **which tasks shall be performed** — the existence of the canonical operations `init()` and `terminate()`, their invocation in the prescribed order, the architectural discipline of each component — and **does not prescribe to whom it falls to perform them**. The concrete attribution of each task between the developer and the execution environment is a shared responsibility whose limits are determined by the language, the development framework and the integrator's decision: when the environment assumes the instantiation, the configuration or the activation of a component, that portion of the cycle is satisfied by construction; when the environment does not assume it, it **shall** be coded by the developer in accordance with the canonical sequence of §8.2. The prescriptions of the model apply in full to the code that the developer writes *within* each component — its classification $L \times T$, its functional responsibility, its layer and type rules —, regardless of which actor activates the component from outside.

8.2 Lifecycle of the artifact

The lifecycle of an XF artifact describes the sequence of states through which the artifact transits from the moment the execution environment loads its classes until the moment the artifact process finishes. The XF model formalizes this lifecycle in four clearly differentiated phases — **instantiation**, **initialization**, **execution** and **termination** — each with precise responsibilities and with a well-defined transition between them. The first phase is *non-invocable* — it is an effect of loading the classes of the singleton —; the remaining three are governed by means of explicit operations of the injection component or of the `XF` architectural element.

Phase 0 — Instantiation The instantiation phase comprises the construction in memory of the three injection components `R`, `B` and `A` and, as an effect of the initialization of their static fields, the instantiation of all logical components of the artifact. Its objective is to leave the artifact in the **defined** state: each logical component exists as an object in memory, accessible through its canonical identifier in the corresponding injection component, and is in its initial state σ_0 — the state subsequent to the execution of the constructor and prior to the invocation of `init()`. The injection component possesses no mutable state of its own: σ_0 is a property of the instantiated logical component, not of the injection component that instantiates it.

Phase 0 is *non-invocable*: it occurs as an effect of the loading of the classes of the singleton by the execution environment, before the start point of the artifact invokes any operation. The developer does not govern it by means of a call; the model prescribes only the structural properties of the code that the phase materializes and the restrictions on the content of the constructors that the phase executes. At the end of phase 0, no external resource has yet been acquired by the artifact.

```
// Phase 0 is declarative code: the initialization of the
// static fields upon loading the classes R, B and A constructs
// all the logical components and leaves them in their initial state sigma_0.
// It requires no invocation whatsoever.

final abstract class R {
    public static final DatabaseRepository database = new DatabaseRepository();
    public static final IdentityRepository identity = new IdentityRepository();
    // ...
}

final abstract class B {
    public static final SessionBusiness session = new SessionBusiness();
    public static final UserBusiness user = new UserBusiness();
    public static final TemperatureBusiness temperature = new
        TemperatureBusiness();
    // ...
}

final abstract class A {
    public static final TemperatureService temperatureService = new
        TemperatureService();
    public static final TemperatureView temperatureView = new
        TemperatureView();
    // ...
}
```

Listing 28. Phase 0 — Instantiation

Every logical component of the artifact that needs to be accessible from other components **shall** be declared in the injection component of its layer. A logical component that is not declared in its injection component is not accessible from any other component of the artifact. The presence of non-exposed logical components is not a normative violation: there may exist internal logical components of a package or library that are deliberately not exposed to external consumers.

Normative prescriptions of phase 0:

The static references of the injection components **shall** be immutable — declared as `readonly`, `final` or the equivalent in the programming language employed. Their value is assigned in the declaration by means of the instantiation of the logical component and cannot be modified during the execution of the artifact.

The injection component **shall not** declare references to logical components of other layers. `B` declares only logical components of the Business Layer. `R` declares only logical components of the Access Layer. `A` declares only logical components of the Interaction Layer. Cross-declaration constitutes a violation of the isolation principle (§6.2.2).

The constructor of a logical component may initialize its internal attributes but **shall not** establish communication with external systems, invoke operations of other components nor execute logic that depends on resources of the execution environment. All bootstrap logic that depends on external resources belongs to the `init()` operation of the logical component, not to its constructor.

Phase 1 — Initialization The initialization phase comprises all the operations that the artifact shall complete before it can process its first external interaction. Its objective is to bring the artifact from the DEFINED state established in phase 0 to the **initialized** state — all components are initialized, their dependencies are resolved and the artifact is ready to process interactions.

The XF model prescribes that the initialization phase is performed by means of the sequential invocation of the `init()` operation of the three injection components in ascending order of abstraction:

```
R.init() // Initialization of the Access Layer

B.init() // Initialization of the Business Layer

A.init() // Initialization of the Interaction Layer
```

Listing 29. Phase 1 — Initialization: ascending order between layers

Each `init()` of an injection component **delegates** to the `init()` operation of the logical components of its layer, invoked in an order compatible with their internal dependencies; the choice of the concrete order is the implementer's decision:

```
// Typical body of B.init(): delegates to each business.init()
// in an order compatible with the internal dependencies (the
// choice of the concrete order is the implementer's decision)

public static void init() {
    B.session.init();
    B.user.init();
    B.temperature.init();
    // ...
}
```

Listing 30. Phase 1 — Initialization: typical body of the init() operation

This order is not arbitrary — it is a direct consequence of the principle of dependency between layers established in §6.2.2. The Business Layer depends on the Access Layer — its logical components may invoke repository operations during their initialization, and those repositories shall be initialized beforehand. Likewise, the Interaction Layer depends on the Business Layer — its logical components may invoke business operations during their initialization, and those business components shall be initialized beforehand. Inverting this order would produce accesses to non-initialized components whose behaviour is undefined.

The initialization phase is atomic from the perspective of the consumer of the artifact: the artifact shall not process any external interaction until `A.init()` has completed successfully. If any initialization operation fails — a database connection that cannot be established, a configuration file that cannot be read, a system resource that is not available — the artifact cannot reach the INITIALIZED state in a consistent state and shall terminate in a controlled manner, propagating the error to the execution environment. The XF model does not prescribe how that error shall be managed — it prescribes that it shall not be silenced nor ignored.

Normative prescriptions of phase 1:

Every logical component that requires bootstrap logic **shall** implement an `init()` operation separate from its constructor. The prohibition on housing that logic in the constructor corresponds to phase 0.

The body of the `init()` operation of the injection component **shall** be limited to invocations of the `init()` operation of the logical components declared as slots of the injection itself. The existence of a topological order that respects the internal dependencies between the logical components is a property derived from the model — not an independent prescription —: the acyclicity of the dependency graph follows from the principle of layer isolation (§6.2.2), and the existence of a layer-respecting topological ordering follows from that acyclicity; the concrete order in which the injection invokes its logical components is left to the implementer's decision, provided it is compatible with those dependencies.

Phase 2 — Execution The execution phase comprises the period during which the artifact actively processes the logic for which it was designed. This phase begins at the moment `A.init()` completes successfully and ends at the moment the artifact receives a shutdown signal or produces an uncontrolled exception.

During the execution phase, the XF model does not prescribe any additional behaviour beyond the prescriptions of the layers and the component types defined in clauses 6 and 7. The logical components process the operations that are invoked on them, maintain their state, communicate between layers through the injection components and propagate exceptions according to the established rules. The lifecycle of the individual components during execution is the responsibility of each logical component and of its injection component — not of the lifecycle of the artifact.

An important property of the execution phase is the **structural immutability of the artifact**: during execution, the set of active logical components — defined in the injection components — does not vary. No new instances of logical components are created, the existing ones are not destroyed and the static references of the injection components do not change. The artifact operates on the set of components that were defined and initialized during bootstrap; any structural modification requires a controlled termination and a new bootstrap.

Normative prescriptions of phase 2:

Phase 2 introduces no prescriptions of its own. The structural immutability described is not an independent prescription: it follows from the immutability of the static references of the injection components and from the prescription to declare in them every accessible logical component (phase 0). The rest of the properties applicable

during execution — layer isolation, directionality of the references, discipline of each component type — are the prescriptions of clauses §6, §7.2 and §7.3, which apply in full without phase-specific reformulation.

Phase 3 — Termination The termination phase comprises all the operations that the artifact shall complete before its process finishes, with the objective of releasing the resources acquired during initialization and leaving the system in a defined rest state for the next execution. Its objective is the symmetric inverse of the initialization phase: to bring the artifact from the INITIALIZED state to the TERMINATED state in an orderly and controlled manner.

The XF model prescribes that every XF artifact **shall** implement the controlled-termination mechanism by means of the `terminate()` operation of each injection component, invoked in the inverse order of initialization:

```
A.terminate() // Termination of the Interaction Layer

B.terminate() // Termination of the Business Layer

R.terminate() // Termination of the Access Layer
```

Listing 31. Phase 3 — Termination: descending order between layers

Symmetrically, each `terminate()` of the injection component **delegates** to the `terminate()` operation of the logical components of its layer, invoked in *inverse* order to that of their initialization:

```
// Typical body of B.terminate(): delegates to each business.terminate()
// in inverse order to that of B.init()

public static void terminate() {
    B.temperature.terminate();
    B.user.terminate();
    B.session.terminate();
    // ...
}
```

Listing 32. Phase 3 — Termination: typical body of the terminate() operation

This inverse order is equally a consequence of the principle of dependency between layers: the Interaction Layer shall terminate before the Business Layer to guarantee that no new interactions arrive while the business components are being terminated. Likewise, the Business Layer shall terminate before the Access Layer to guarantee that no business operation attempts to access a repository that has already closed its connections.

The prescription to implement the termination mechanism is normative and admits no exception: the absence of `terminate()` in the injection components of an artifact is a violation of the model, regardless of whether the execution environment guarantees its invocation. A `terminate()` that implements resource release but which the environment does not invoke in certain scenarios is a problem of the environment — not of the artifact. A `terminate()` that does not exist is indeed a problem of the artifact.

The effective registration of the controlled-termination mechanism that the execution environment provides — `SIGTERM` signal handlers on Unix systems, `onDestroy` on

Android, `onApplicationStop` on Spring, shutdown hooks in Node.js — depends on the environment and on the initialization model of the artifact; the XF model neither prescribes nor recommends it, considering it outside the structural scope of the architectural model. The XF model explicitly recognizes that in certain execution environments — serverless functions, containers destroyed without prior notice, processes interrupted by the operating system — the invocation of controlled termination cannot be guaranteed and depends on the infrastructure environment. The artifact **shall** implement the mechanism; the environment determines whether it can invoke it.

Normative prescriptions of phase 3:

Every injection component **shall** implement the `terminate()` operation. The absence of `terminate()` in an injection component constitutes a violation of the model regardless of whether the execution environment guarantees its invocation.

The order of invocation between the injections `A`, `B`, `R` is the inverse of that of initialization.

Additionally, every logical component that has acquired system resources during its initialization **shall** implement a `terminate()` operation that releases them in a controlled manner.

Resource symmetry. The relationship between the `init()` and `terminate()` operations is expressed formally as a resource symmetry: every resource acquired during `init()` shall be released during `terminate()`, and no resource shall be released in `terminate()` that has not been acquired in `init()`. This symmetry is the formal property that guarantees that the artifact does not produce resource leaks — memory leaks, unclosed connections, locked files — between successive executions. Expressed in terms of the automaton: the transition DEFINED → INITIALIZED acquires resources; the transition INITIALIZED → TERMINATED releases them. The TERMINATED state is functionally equivalent to the DEFINED state with respect to the system resources that the artifact occupies.

Formal representation of the lifecycle The lifecycle of the artifact may be represented formally as a state machine with three states and five transitions, as collected in the following figure:

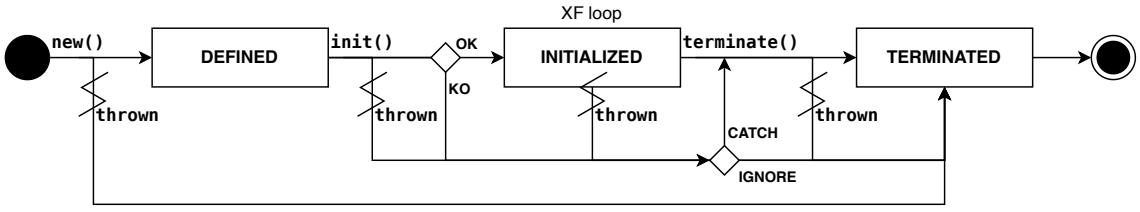

Figure 10. Automaton of the artifact lifecycle: three states — DEFINED, INITIALIZED and TERMINATED — and their transitions, triggered by the events new, init and terminate or by the occurrence of an exception. A caught exception (CATCH) leads to TERMINATED in a controlled manner — executing terminate —; an uncaught one (IGNORE), or any exception arising in instantiation or in termination itself, abruptly. Transitions not represented are not defined and constitute error conditions.

- **defined**: initial state of the artifact. The components exist as code structures but are not initialized. No logical component may be accessed.
- **initialized**: active state of the artifact. All components are initialized and the artifact processes interactions. The component structure is immutable.
- **terminated**: final state of the artifact; the process may finish. It is reached in one of two modes — **controlled**, executing terminate and releasing the acquired resources in an orderly manner, or **abrupt**, without executing terminate. Absorbing state: once reached, the artifact cannot return to the INITIALIZED state without a new bootstrap.

The transitions of the automaton are triggered by four types of event — `new`, `init`, `terminate` and the occurrence of an exception — and produce five formal transitions (the exception event triggers one transition to TERMINATED from DEFINED and another from INITIALIZED):

- **`new` (instantiation)**: (no previous state) → DEFINED. Entry transition into the automaton: the loading in memory of the logical components by the execution environment (Phase 0) brings the artifact to the initial state DEFINED.
- **`XF.init()` successful**: DEFINED → INITIALIZED. It requires that all `init()` operations of all injection components complete successfully.
- **`XF.init()` failed**: DEFINED → TERMINATED. Some `init()` operation of the injection components does not complete successfully — by reporting failure (KO) or by throwing an exception — before reaching INITIALIZED.
- **`XF.terminate()` controlled**: INITIALIZED → TERMINATED. Triggered by the shutdown signal of the environment or by a caught exception (CATCH); it executes the `terminate()` sequence and releases the resources in an orderly manner.
- **Exception in execution**: INITIALIZED → TERMINATED. Exception arising while the artifact processes interactions; its termination mode depends on whether it is caught (see *Termination modes*).

Termination modes. The artifact reaches TERMINATED in one of two modes, according to whether the exception is caught by the control loop of the artifact:

- **controlled** (CATCH): the exception is caught and the artifact executes the `terminate()` sequence, releasing the acquired resources in an orderly manner. It is also the mode of the normal termination triggered by the shutdown signal of the environment.
- **abrupt** (IGNORE): the exception is propagated to the execution environment and the process concludes without executing `terminate()`.

An exception arising during instantiation (phase 0) or during termination itself (phase 3) always leads to an abrupt termination: in phase 0 the control loop is not yet active to catch it; in phase 3 the artifact is already terminating and termination admits no recovery.

This formal representation is the basis upon which the conformance rules of the lifecycle are built. Understanding these flows helps to understand the state preconditions under which it is permitted to execute the initialization and termination operations of the components.

Table 6. Automaton of the artifact lifecycle: transitions between the three states DEFINED, INITIALIZED and TERMINATED triggered by the events `new`, `init` and `terminate` and by the occurrence of exceptions. The termination mode — controlled (executes terminate) or abrupt — depends on whether the exception is caught. Transitions not listed are not defined and constitute error conditions.

Source state	Event	Target state	Operational condition and mode
(no previous state)	new	DEFINED	The execution environment loads the classes of the injection components and instantiates the logical components as static fields (phase 0). An exception in this phase terminates the artifact *abruptly*.
DEFINED	init (success)	INITIALIZED	init_R, init_B and init_A complete successfully in ascending order.
DEFINED	init (failure)	TERMINATED	Some init_L does not complete — failure (KO) or exception — before reaching INITIALIZED. *Controlled* if it is caught and terminate is executed; *abrupt* if it is propagated.
INITIALIZED	terminate	TERMINATED	Shutdown signal of the environment or caught exception (CATCH); terminate_A, terminate_B and terminate_R complete in inverse order (*controlled*). An exception during terminate likewise concludes in TERMINATED, *abruptly*.
INITIALIZED	uncaught exception	TERMINATED	The exception is propagated to the environment (IGNORE); the process concludes without executing terminate (*abrupt*).
DEFINED	terminate	(no transition)	Not defined: an artifact that has not been initialized cannot be terminated.
INITIALIZED	init	(no transition)	Not defined: an artifact in operation cannot be re-initialized.
TERMINATED	(any)	(no transition)	Absorbing final state: the lifecycle of the artifact has concluded.

8.3 `XF` start-point element

The `XF` element is the **start-point element of the XF architecture**: the construct of the artifact that materializes, by means of two static operations, the state transitions of the aggregate lifecycle of the artifact. Its presence in an artifact is the explicit declaration that that artifact implements the XF model.

The `XF` element **is not a component**: it is not classified in any cell of the $L \times T$ matrix of the model and lies outside the domain of the classification function ϕ. It is a structural element of the artifact — analogous to the canonical folders — that the model recognizes by its canonical name and by its location under `/src`, not by a typological assignment. Its structural function consists of declaring two operations — initialization and termination — that orchestrate the canonical sequence of the aggregate lifecycle, and, where applicable, of housing the execution start point of the artifact when the language allows the latter to be a static member of a class.

Optionality The presence of the XF element in the artifact is **optional**. The model does not require declaring it: the execution start point (§8.1) may directly invoke the canonical initialization sequence (`R.init(); B.init(); A.init();`) and, symmetrically, the canonical termination sequence (`A.terminate(); B.terminate(); R.terminate();`), without the XF element being present.

When the implementer opts to declare it, the XF element is bound by the prescriptions that the following clauses establish: its canonical name, its location, the mandatory declaration of `init()` and `terminate()` and the canonicity of the body of those two operations. The presence of the component contributes no additional structural capability over the materialization without it, but it centralizes the orchestration of the lifecycle at a single point of invocation, explicitly declares the conformance of the artifact with the model, and simplifies the content of the start point.

Canonical location The file of the XF element resides at the root of `/src` at the same level as the folders of the three layers. This location is verified by construction: any file named XF located outside the canonical root constitutes a violation of the location prescriptions according to the concrete place where it is located, without the need for a prescription specific to the XF element.

```
/src
├── /api
├── /business
├── /repository
└── XF
```

Listing 33. Canonical location

Mandatory operations The XF element **shall** declare two static operations: `init()` and `terminate()`. The absence of either of them in an XF element present in the artifact constitutes a violation of the model.

The body of `XF.init()` materializes the transition DEFINED → INITIALIZED of the artifact lifecycle (§8.2, Phase 1) and **is closed** to the canonical sequence of invocations `R.init(); B.init(); A.init();` in ascending order of abstraction.

The body of `XF.terminate()` materializes the transition INITIALIZED → TERMINATED of the artifact lifecycle (§8.2, Phase 3) and **is closed** to the canonical sequence of invocations `A.terminate(); B.terminate(); R.terminate();` in inverse order.

```
// Mandatory content of the XF element
class XF {
    static void init()      { R.init(); B.init(); A.init(); }
    static void terminate() { A.terminate(); B.terminate(); R.terminate(); }
}
```

Listing 34. Mandatory operations

Structural openness of the class The model **does not close** the set of members of the XF element, nor does it restrict inheritance, nor does it restrict the instantiability of the class. The class may declare additional members and participate in inheritance hierarchies or be instantiable when the execution environment requires it. This openness allows XF to accommodate two normatively equivalent usage configurations:

1. **XF does not contain the execution start point of the artifact.** The start point resides outside XF (§8.1), and invokes XF.init() as the first operation of the developer's code and, optionally, registers XF.terminate() with the shutdown mechanism of the environment. In this configuration, XF contains exclusively the two mandatory operations.
2. **XF contains the execution start point of the artifact.** When the language allows the start point to be a static member of a class, the implementer may declare it within XF — as an additional member, distinct from init() and terminate() — and invoke init() and terminate() internally from it. In this configuration, XF may additionally include the annotations, decorators, inheritance or static configuration that the execution environment prescribes in order to recognize the class as the start point.

The choice between the two configurations is determined by the language and the execution environment, and within what the environment permits, by the implementer. The architectural discipline over the additional members that are declared is covered transversally by the rest of the model: instantiating logical components outside the injections, invoking operations of logical components outside the canonical pattern or referencing non-classified components constitute violations of the model.

Visibility of init() and terminate() The model does not prescribe a concrete visibility for init() and terminate(). The effective visibility is determined by the accessibility necessary for their legitimate invokers, and therefore by the underlying language: if the start point resides outside XF, both operations shall be accessible from that point and their visibility will be public (or the equivalent in the language); if the start point resides within XF, the operations are invoked internally and their visibility may be private. This nuance does not appear as two distinct prescriptive cases: it is the single generalization that the visibility is the minimum necessary for the legitimate invocation to be possible.

Invocation exclusivity The operations XF.init() and XF.terminate() **shall** be invoked **only** from the XF element itself — when the latter contains the execution start point of the artifact — or from the external start point of the artifact when the latter resides outside XF. No classifiable component of the artifact — logical, generalization, injection, utility or transfer — may invoke XF.init() nor XF.terminate(). The orchestration of the aggregate lifecycle of the artifact is reserved exclusively to the execution start point, whatever its materialization. In the case of the external start point, the invoker is not a classifiable component and lies outside the scope of the model by construction.

The XF element as a conformance contract The presence of the XF element in an artifact is the explicit declaration that that artifact implements the XF model. An artifact that contains the XF element is declaring that its structure is classified according to the taxonomy of the model, that its dependencies respect the principle of layer isolation and that its aggregate lifecycle is governed from a single control point. This semantic property justifies that the XF element be the first element that conformance-analysis tools inspect when evaluating an artifact: its presence or absence is the most immediate indicator of the implementation of the model.

9 Data flows

The data flow describes how information is transmitted and transformed between the components of an XF artifact during its execution. This flow is not a property that the XF model invents — it is an intrinsic property of every formal process and, therefore, of every software artifact. The XF model formalizes that property, establishes the constraints that guarantee the coherence of the flow with the layer isolation principle, and provides the prescriptions that allow its correctness to be verified statically.

The clause is organized as follows. Clause §9.1 describes the information transmission model as an intrinsic property of software. Clause §9.2 formalizes the types of data structures that the model recognizes. Clause §9.3 develops the principle of unification of data structures. Clause §9.4 prescribes the directionality and structural transformation rules of transfer components. Clause §9.5 formalizes the communication channel of transfer components as a concept unique to the model and prescribes the structural homogeneity property over $\mathbb{C}_T$.

9.1 Information transmission model

The transmission and transformation of data between components is an intrinsic property of every formal process automated by means of software. When a component executes its effective logic, it may require input data to operate — produced by another component — and produces output data as a result — consumed by another component. This exchange of information between components is the fundamental mechanism by which the components of an artifact cooperate to fulfill the purpose of the system.

This mechanism is not determined by the programming language or by the development framework — it is a property common to all business process management, independent of the modeling tools. A component that needs data to operate receives it as input parameters. A component that produces data as a result of its operation returns it as a return value. This input-output duality is universal and is present in any programming paradigm — object-oriented, functional, reactive, or imperative.

In the context of the XF model, the transmission of information between components is characterized by a single dimension: the **flow direction** — data may flow in the downward direction, from higher layers toward lower layers, or in the upward direction, from lower layers toward higher layers. The communication channel through which each concrete transmission is materialized (§9.5) is a single formal concept of the model, regardless of the syntactic construct that the language provides to deliver the datum to the caller.

The XF model recognizes two types of flow according to the direction, both well identified in software practice:

- **Downward flow**: a higher layer invokes an operation of a lower layer and receives the result through the communication channel of the invoked operation. It is the most frequent flow — the Interaction Layer invokes the Business Layer, which invokes the Access Layer. The datum delivered through the

communication channel may be a foreseen result or an anomalous condition; in both cases it is a transfer component (§9.5).

- **Upward flow**: a lower layer notifies a state change or a condition to a higher layer; this notification **never** employs direct upward invocation and is typically carried out by means of the observer pattern. The lower-layer component notifies its registered observers; each observer receives the notification through the communication channel of the callback method that it itself exposed; the datum carried by the callback is likewise a transfer component (§9.5).

Note on layer isolation. The existence of the upward flow does not contradict the downward isolation principle prescribed in §6.2.2. The dependency relation between layers is defined in terms of **structural knowledge**: a component A depends on a component B if and only if A knows the definition of B — that is, it declares its identifier, imports its class, or references its type in its source code. The isolation principle prohibits only this type of dependency in the upward direction: no component of a lower layer may know the definition of a component of a higher layer. The upward flow preserves this prohibition because it is typically materialized by means of the **observer pattern** and **event delegation**, described in §6.2.2: the higher layer knows the observable component of the lower layer and registers with it; the lower layer emits notifications through a generic mechanism — a typed list of observers, an event channel — without knowing the identity, the type, or the existence of its consumers. Therefore, the information flow ascends while the structural dependency remains downward.

The vehicle of information transmission in all these flows is always the same: the **transfer component** — the data structure that models the form of the information at a concrete point of the processing flow. The rules that govern the use of transfer components in each of these flows are the subject of the following clauses.

9.2 Types of data structures

The XF model recognizes two categories of data structures that are managed at the bit level in the layer where they are worked — a categorization consistent with the foundations of type theory in programming languages [40] and that the model adopts without modification.

The first category is the **primitive types**, defined in §3.1.15 in the sense of ISO/IEC/IEEE 24765:2017 [20]. They constitute the raw material on which all transfer components operate.

The second category is the **structures** — aggregations of primitive types and of other previously defined structures, which make it possible to model information of greater complexity. In the XF model, these structures are, in the strict sense, the transfer components: the data classes that model the concepts of the artifact domain in their various processing states; they may carry self-contained operations on their own data (§7.3.5).

The XF model introduces no additional distinction within the structures beyond the one already established by their classification in the $L \times T$ matrix — the layer in which they are defined. The proliferation of semantic categories that characterizes

traditional architectures — `DTO`, `Entity`, `Model`, `POJO`, `Record`, `VO`, `Schema` — has no place in the XF model. All structures are transfer components, regardless of their context of use, and their role in the data flow is managed by the logical components that process them, not by their classification.

9.3 Unification of data structures

Definition and scope of the artifact domain The XF model **enables** the **unification of data structures** as an emergent capability of the transfer framework: a single transfer component may be shared by all the layers that work with the same concept of the artifact domain, without the need to duplicate it per layer. The *artifact domain* comprises both the business concepts that the artifact models (`User`, `Order`, `Temperature`, `Schedule`) and the concepts of access to its external systems (`Query`, `HttpRequest`, `Cursor`, `Connection`) and the concepts of interaction with its consumers (`MouseEvent`, `Viewport`, `HttpResponse`). Unification is possible in any of these three subdomains.

Contrast with traditional modeling This capability contrasts directly with the predominant approach in traditional architectures, where each layer defines its own data structures to represent the same concept according to its flow direction and its abstraction level. With this approach, a single business concept may require up to six distinct structures in an artifact with three layers and bidirectional flow — one input structure and one output structure per layer — each with its own attributes, its own transformation operations, and its own independent maintenance.

In traditional modeling, developers maintain distinct structures per layer, reaching in the extreme schemes such as `UserRequestDTO`, `UserRequestModel`, `UserRequestEntity` for the input flow, and `UserResponseDTO`, `UserResponseModel`, `UserResponseEntity` for the output flow — six structurally equivalent structures for the same concept `User`, with five conversion operations between each adjacent pair. In XF modeling, a single transfer component may circulate through any layer in the upward direction. The transformation operations — $f(C_T) \to C'_T$ — do not produce a component of a distinct type but a modification of the data of the same component, with the attributes adjusted according to the needs of the logical component that processes it.

Nature of unification: capability, not prescription Unification is a *capability* that the model enables, not a normative prescription that the model imposes. The model does not require the implementer to unify transfers of the same concept across distinct layers: two structurally equivalent transfers in distinct layers constitute a redundancy, not a violation. Pragmatically, a single structure per concept reduces the complexity of the artifact, eliminates the mapping operations between equivalent structures, and makes the evolution of the data model a change localized at a single point; the model recommends unification as good practice, but the decision remains in the design domain, not in the normative domain. Unification is possible because transfer components are exempt from the strict layer isolation constraint: they may be referenced from layers of greater abstraction without violating the isolation principle, subject to the directionality constraints of transfers and to the structural transformation rule (§9.4).

9.4 Directionality and transformation of transfer components

A transfer component **may be referenced** by components of its own layer or of layers of a higher abstraction level, and **shall not be referenced** by components of layers of a lower abstraction level; the reference thus preserves the downward direction common to every dependency of the XF model. Unlike the rest of the component types — whose access from another layer is channeled through the injection component of the immediately lower layer (§7.3.3) —, the reference to a transfer component is direct and may cross more than one layer boundary. The unification of structures (§9.3) operates within this directionality, not against it.

The following table summarizes the permitted directionality of references to transfers between layers.

Table 7. Matrix of permitted directionality of references to transfer components between layers: the row indicates the referencing layer; the column, the layer where the transfer is defined. “Permitted” indicates an admitted reference; “Prohibited”, a non-admitted reference.

Component \ Transfer defined in	Interaction	Business	Access
Interaction Component	Permitted	Permitted	Permitted
Business Component	Prohibited	Permitted	Permitted
Access Component	Prohibited	Prohibited	Permitted

This constraint preserves the sequencing of the layers — information flows from layers of lower abstraction toward layers of higher abstraction, never in the inverse direction. A structure defined in the Interaction Layer cannot be referenced by components of the Business Layer or of the Access Layer — its existence is invisible to the lower layers. A structure defined in the Business Layer may be referenced by components of the Interaction Layer but not by components of the Access Layer. A structure defined in the Access Layer may be referenced by components of any layer.

The directionality constraint applies indistinctly to every transfer, regardless of the syntactic construct with which the language materializes its transmission: transfers delivered as language exceptions are transfer components of the same domain $\mathbb{C}_T$ as those delivered as an ordinary return (§9.5). A transfer defined in the Access Layer may be referenced by the Business Layer and by the Interaction Layer, regardless of the usual transmission construct. A transfer defined in the Business Layer may be referenced by the Interaction Layer but not by the Access Layer.

This property has a direct practical implication: the Interaction Layer is the only layer that can know and manage the complete set of error transfers of the artifact — both those defined in the Access Layer and those defined in the Business Layer and its own. This is consistent with the responsibility of the Interaction Layer to represent the results to the consumer of the artifact — including the error results — in a manner appropriate to the interaction protocol employed.

Structural transformation: design guideline Directionality determines *from which layers* a transfer component may be referenced, but it does not by itself resolve under what conditions the receiving layer may reuse the received transfer or, alternatively, define a new one. The XF model establishes the following design guideline — not a normative prescription — in terms of the criterion that distinguishes the two options: **direct reuse** is appropriate when the receiving layer consumes the component without modifying its **structural definition** (its set of attributes); the **definition of a new transfer** in the receiving layer is the canonical option when that layer needs a structure with attributes added or removed with respect to the received component. Every new structure of the artifact is classified as a transfer component (a structural prescription on transfer components and on the folder structure).

The modification of the **values** of the attributes — computing a derived value, formatting a string, converting a unit of measure — does not require defining a new component: the original transfer may be reused with adjusted values, provided that its structural definition does not change. Only when the **definition** of the structure changes — attributes added or removed — does the receiving layer need a new transfer to model the result.

9.5 Homogeneity of transfers and communication channel

Programming languages offer distinct syntactic constructs for a component to transmit information to the component that invokes it: explicit return by means of `return`, the throwing of exceptions by means of `throw` or `raise`, multi-value return of the type `(T, error)`, algebraic types such as `Result<T, E>` or `Either`, return codes, and global variables such as `errno`. Beneath this syntactic variety, all these constructs materialize the same underlying concept: the **communication channel** of the operation — the medium by which a component delivers information to its caller during execution. This clause formalizes the communication channel as a concept unique to the model and prescribes the homogeneity property on which the coherence of the model's prescriptions rests in the face of the syntactic variety of the language.

Syntactic materialization in the language The communication channel of an operation is materialized differently according to the paradigm of the implementation language:

- **Languages with structured exception handling** — Java, C#, C++, Python, JavaScript, Kotlin, Swift, Ruby — provide two syntactically differentiated primitives: the return primitive (`return` and equivalents) and the throwing primitive (`throw`, `raise`) caught with `try/catch` or equivalent.
- **Languages with algebraic types** — Rust, Haskell, Scala — materialize all output over the same return primitive, distinguishing the purpose of the transmitted datum by means of the type of the returned value: `Result<T, E>` with the constructors `Ok(T)` and `Err(E)`, or `Either E T` with `Right` and `Left`.
- **Languages with multi-value return** — Go — materialize the output as a tupled return: `(T, nil)` when the operation completes as foreseen and `(zero,`

`error`) when an anomalous condition occurs.
- **Convention-based languages** — traditional C — materialize the output by means of the return value and, when an anomalous condition occurs, by means of global variables (`errno`) or reserved return codes.

This variety of syntactic constructs does not introduce conceptual variety: in all cases, a component delivers information to its caller through its communication channel, and the datum delivered — whether a foreseen result or an anomalous condition — is always a transfer component.

From the language constructs to the XF transfer component The conceptual equivalence of the syntactic transmission constructs is the formal basis on which the XF model prescribes the homogeneity of transfer components. If the language constructs are distinct syntactic materializations of the same underlying concept, the data that circulate through them are entities of the same structural nature. The ordinary structure that an operation returns and the exception that an operation throws are not entities of a distinct type in the XF model: both are transfer components — they belong to the single set $\mathbb{C}_T$, they are classified in the same $L \times T$ matrix, and they are governed by the same directionality and structural transformation rules (§9.4). The difference between one and the other is exclusively the syntactic construct with which the transmission is materialized in a concrete invocation, not its category in the model. The formalization of the model absorbs the distinct syntactic paradigms without asymmetry: the communication channel is modeled as a single output signature of the operation, and the transported type always belongs to the same $\mathbb{C}_T$.

Structural homogeneity of transfers The XF model establishes as a normative property that **every transfer that circulates through the communication channel of an operation belongs to the same single domain** $\mathbb{C}_T$. There are no "normal-result transfers" and "error transfers" as two disjoint classes: there exists the single set of transfer components $\mathbb{C}_T$ of the artifact, and each transfer is admissible through the communication channel regardless of the purpose of the datum it transports and of the syntactic construct that the language uses to deliver it to the caller. The distinction between the purpose of the transmitted data — foreseen result or anomalous condition — is exclusively *of implementation* — which syntactic primitive of the language materializes the transmission — and **does not alter the structural nature, the classification in** $L \times T$**, nor the rules that govern the transfer component**.

This homogeneity has four direct implications, all of them corollaries of rules or guidelines already prescribed:

1. *Uniform directionality.* The downward directionality (§9.4) applies to every transfer that appears in the output signature of an operation, regardless of the syntactic construct that materializes the transmission. A transfer of the Access Layer may be delivered to the Interaction Layer by means of `return`, `throw`, `Result<T, E>`, or any other equivalent construct — the permitted direction is the same.

2. *Uniform structural transformation.* The guideline of §9.4 — defining a new transfer when a layer transforms the received structural definition — operates with the same force on transfers materialized as language exceptions. If an Access error transfer is transformed when passing through Business, the integrator defines a new transfer in Business.
3. *Uniform unification.* The unification capability enabled in §9.3 allows modeling a transfer that represents the same concept of the domain with a single component and delivering it by means of any syntactic construct without duplication. Duplicating the same conceptual transfer — one for the return construct and another for the exception construct — is contrary to the unification guideline unless they represent effectively distinct concepts of the domain; the choice between unification and duplication is a design decision, not a prescription of the model.
4. *Syntactic freedom of the construct.* The model does not attribute syntactic constructs to transfers. A transfer with the `Exception` suffix may be returned by means of `return` (typically to reify the error as a value processable by the caller), and an ordinary transfer may be thrown as an exception when the language and the semantics justify it. The choice of the construct for each transfer is the responsibility of the artifact designer.

Naming convention for error transfers The XF model does not prescribe additional naming conventions on the transfers that are habitually transmitted as anomalous conditions. The `Exception` suffix in their canonical identifier — useful for legibility and for analysis tools — is a convention that the implementer may freely adopt; an artifact that delivers errors as ordinary values by means of sum types `Result<T, E>`, or that uses ad-hoc names for error transfers, is fully conformant to the model. The purpose of the transmitted datum does not generate structural asymmetry in $\mathbb{C}_T$.

The following table collects the recurrent transfers that are transmitted as exceptions in XF artifacts.

Table 8. Recurrent transfers habitually transmitted as language exceptions, with the layer where they originate and their typical context. They are transfer components like any other: the `Exception` suffix is a naming convention, not a structural requirement. Informative and non-exhaustive table.

Canonical exception	Origin layer	Typical context
`NetworkException`	Access	Failure in the communication with a remote system: timeout, closed connection, unresolvable host.
`AuthenticationException`	Access	Invalid or expired credentials when authenticating against an external system.
`AuthorizationException`	Access or Business	Operation not permitted for the authenticated subject.
`ResourceNotFoundException`	Access	Resource requested from the remote system does not exist (HTTP 404, absent row).
`ResourceConflictException`	Access	State conflict when modifying a resource (HTTP 409, optimistic locking).
`ServerErrorException`	Access	Error of the remote system (HTTP 5xx, DB failure, etc.).
`ValidationException`	Business	Input data do not satisfy the structural rules of the domain.
`BusinessException`	Business	Semantic rule of the domain violated (insufficient balance, invalid state, etc.).
`ConflictException`	Business	State of the domain incompatible with the requested operation.
`InvalidArgumentException`	Interaction	Input parameter to the artifact malformed or absent.
`UnsupportedOperationException`	Interaction	Exposed operation not implemented by the component.

Native language exceptions The native exceptions provided by the implementation language (`Error` in JavaScript/TypeScript, `Exception` and subclasses in Java, Kotlin, or C#, `BaseException` and subclasses in Python, etc.) are, in the XF axiomatization, transfer components. The model does not require redefining them as transfers proper to the artifact: when a logical component throws a native exception, that exception is to all effects a transfer that travels through the communication channel — inherited from the language, but part of the same formal domain. Compatibility with inherited transfers is developed in clause §10.

Decision of the transmission construct as the implementer's responsibility The XF model does not prescribe the conditions under which a component shall transmit information by means of one syntactic construct or another. This is an

implementation decision that depends on the business rules of the domain that the artifact models. For example, if an inconsistent value is introduced as input to a business operation, the component may return an adjusted structure by means of `return` — indicating the error as part of the result — or throw an exception — interrupting the normal execution. Both options are valid in the XF model — the choice belongs to the implementer.

What the model does prescribe is that, whatever the chosen construct, the vehicle of transmission is always a transfer component — a well-defined, named data structure classified in the corresponding layer — never an untyped error nor a text message without formal structure.

Historical note The classical nomenclature of the error propagation models in software habitually distinguishes between "main channel" (the medium by which the result of an operation is delivered when it completes as foreseen) and "exception channel" (the medium by which anomalous conditions are transmitted). This duality reflects the syntactic separation between `return` and `throw` characteristic of languages with structured exception handling, and it has been carried over to architectural models as a property of the system. The XF model recognizes the widespread use of this nomenclature, but does not incorporate it into its formalization: the communication channel is a single unified concept, and the choice of the syntactic construct for each transfer remains completely in the implementer's domain.

10 Compatibility and interoperability

The XF model derives from a structural property of software established in §5.1: every software artifact, whether XF or not, models a formal process. This characterization implies that every component of any artifact carries out one of the three invariant functional responsibilities of the formal process — interaction, processing or access (§5.2). The $L \times T$ classification of the XF model is not proprietary: it is the formalization of that universal property of software. By the completeness property (§5.5), every component of any artifact is classifiable into exactly one cell of the matrix, regardless of whether the artifact that hosts it was built under the XF model or under a traditional architecture.

From this structural universality the notion of compatibility of the XF model is derived: integrating an external component does not consist in converting it to the model nor in relaxing the prescriptions of the layer that receives it, but in **recognizing the functional responsibility** that the component carries out, **assigning it its cell** in the $L \times T$ matrix of the integrating artifact and **applying the mechanism prescribed** by the model for that cell and that component type. The clause is organized into three internal clauses. §10.1 establishes the principle of compatibility and formalizes the four possible cases according to whether each artifact implements the model or not. §10.2 catalogs the mechanisms prescribed by integrated component type. §10.3 characterizes the ecosystem of XF libraries as a natural consequence of the model.

10.1 Principle of compatibility

Universality of the $L \times T$ classification Every formal process admits a canonical decomposition into three invariant responsibilities (§5.2); whether it classifies them explicitly or not, every software artifact carries out those three responsibilities. The $L \times T$ matrix of the XF model is the formalization of that decomposition: it organizes the three layers as an expression of the three responsibilities of the formal process, and the five functional types as an expression of the structural roles with which components contribute to each layer. By the completeness property derived in §5.5, all program logic belongs to exactly one of the three layers: there is no component of any artifact that escapes the $L \times T$ classification. The difference between an XF artifact and a traditional artifact is not that the former has a classification and the latter does not — both have it by the nature of software —; it is that the former makes it explicit by construction and the latter keeps it implicit in its proprietary nomenclature.

Operational definition of compatibility When an XF artifact A integrates a component of another artifact B, the implementer of the integrator performs three cognitive operations in order:

1. **Recognition**: identify the functional responsibility — interaction, processing or access — that the integrated component carries out according to the three-condition classification criterion (§7.2.1, §7.2.2, §7.2.3).
2. **Assignment**: classify the component into the corresponding cell of the $L \times T$ matrix of artifact A.
3. **Application of the mechanism**: integrate the component by means of the mechanism prescribed in §10.2 for that cell and that component type.

The first two operations — recognition and assignment — are architectural judgments of the integrator, not structural properties of the artifact, and therefore are not verified as independent prescriptions; the model verifies their material consequences (correct layer of the resulting XF component, fit of its functional responsibility to that of its layer) by means of the prescriptions applicable to its type and to layer isolation. The third operation — application of the mechanism — is materialized in the structural conformance of the resulting XF component with the prescriptions of its $L \times T$ cell.

Recognition and assignment are performed by the integrator, not by the integrated. This is the basis of the **bidirectionality** and the **non-invasiveness** of compatibility: no integrated artifact is modified to make its integration possible; each artifact retains the autonomy of its own classification space and only the integrator is responsible for the compatibility decisions over the dependencies it adopts.

Integration levels Compatibility operates at two levels according to the execution space of the artifacts involved. The **execution space** of an artifact is defined as the computational context in which its components operate — memory, system resources and process context. Two artifacts share an execution space if and only if their components can reference each other directly in memory without serialization or transmission through an external channel. The distinction is binary: it is either

shared or not shared.

According to whether they share their execution space or not, the integration between two artifacts is classified into two levels:

- *Application-level integration* when the execution spaces are disjoint — independent processes, serialized protocol communication. From the perspective of the XF artifact, the external artifact is an external system that the Access Layer accesses according to the prescriptions of §7.2.1; the compatibility mechanisms of §10.2 do not apply.
- *Artifact-level integration* when the artifacts share an execution space — same process, direct references in memory. The components of the integrated are candidates for integration through the mechanisms of §10.2.

The following table summarizes both integration levels.

Table 9. Integration levels between artifacts according to the execution space. Application-level integration is independent of the internal architecture of each artifact: two artifacts are compatible at this level from the moment they mutually know and satisfy their communication protocol. Artifact-level integration activates the compatibility prescriptions of the XF model.

Property	Application-level integration	Artifact-level integration
Execution space	Disjoint (independent processes)	Shared (same process)
Communication mechanism	Network protocol (HTTP, gRPC, WebSocket, messaging)	Direct references in memory, local invocation
Activation of the XF compatibility prescriptions	No — the integrated is an external system accessed via the Access Layer	Yes — the mechanisms of §10.2 apply
Typical example	Microservice invoking another via REST	Artifact importing an XF library in the same runtime

Matrix of cases At the artifact level, integration admits four possible combinations according to whether the integrating artifact and the integrated artifact implement the XF model or not. The four combinations are developed below and summarized in Table 10.

Case (A) — XF integrator, XF integrated The $L \times T$ classification of each component of the integrated is done by construction of the integrated itself. The integrator does not reclassify; it respects the cell that the integrated assigned to each component. The mechanisms of the catalog in §10.2 apply according to the type of the component being integrated — **injection** for logical components, **inheritance** for generalization components, **inheritance or direct use** for utility components and **direct use** for transfer components. The choice among the variants that each mechanism provides is at the discretion of the integrator.

Case (B) — XF integrator, non-XF integrated The integrated does not expose an $L \times T$ classification by construction; the integrator **shall** supply that classification through the recognition → assignment → application cycle. The resulting XF component may make contact with the integrated by means of any mechanism that the language or the development framework provides — encapsulation (delegation to an internal instance), inheritance (extension of a development framework class), annotation or decoration (metadata processed by the development framework), interface implementation (conformance to a development framework contract), among others. The choice of mechanism is the integrator's decision and does not alter the classification of the resulting XF component in its $L \times T$ matrix: the component remains XF and its cell is dictated by its functional responsibility. The specific prescriptions of each mechanism are developed in §10.2.

Case (C) — non-XF integrator, XF integrated The XF model does not prescribe over artifacts that do not implement it; the integrator is not subject to the prescriptions of the model. The XF integrated, by the principle of layer isolation (§6.2.2) and the prescription of canonical access (§7.3.3), exposes its logical components through its injection components with stable signatures independent of internal implementation details. The non-XF integrator accesses those interfaces directly. Although the model does not prescribe behavior to the integrator, it is recommended that A respect the initialization order of the integrated (§8.2) — invoke `R.init()`, then `B.init()`, then `A.init()` — so as not to produce undefined behavior in B; failure to follow this order does not constitute a violation of the model by A, which does not follow it.

Case (D) — non-XF integrator, non-XF integrated When neither of the two artifacts implements the model, the integration lies entirely outside the prescriptive scope of the XF model. The model states no prescription over this combination. Its inclusion in the matrix responds only to the exhaustiveness of the table.

Table 10. Matrix of artifact-level integration cases. The rows indicate whether the integrating artifact implements the XF model; the columns, whether the integrated artifact implements it. Each cell states what the model prescribes: the XF × XF case activates direct integration because the $L \times T$ classification of the integrated is inferable by construction; the XF × non-XF case requires the integrator to recognize the functional responsibility and apply the prescribed reformulation; the cases in which the integrator is not XF lie outside the prescriptive scope of the model.

	XF integrated	Non-XF integrated
XF integrator	**(A)** Classification inferred by construction; the mechanism is determined by the type of the integrated component (§10.2)	**(B)** The integrator **shall** recognize the responsibility, assign a cell and reformulate the functionality into its own XF component (§10.2.1)
Non-XF integrator	**(C)** Possible without normative criterion from the model; the integrated guarantees stable interfaces by construction	**(D)** Outside the prescriptive scope of the XF model

10.2 Prescribed mechanisms

The application phase of the compatibility cycle (§10.1) materializes the integration of the recognized and assigned component into an effective dependency between the integrating artifact and the integrated artifact. This clause establishes the mechanisms that the model prescribes for that materialization and delimits their scope of application.

The structure of the integration mechanisms reflects a distinction that the model elevates to a governing criterion: the integrated component may already be normalized in the XF vocabulary or not. The difference does not affect the principle — the recognition → assignment → application cycle is invariant — but it does condition the application phase. When the integrated exposes its $L \times T$ classification by construction, the mechanism is determined exclusively by the type of the component and the model prescribes it unambiguously. When the integrated does not expose that classification, the integrator **shall** define its own XF component that assumes the recognized functional responsibility; the syntactic mechanism with which that component makes contact with the external functionality is free within what the language and the development framework offer.

This syntactic freedom in the second case is not a concession: it is a direct consequence of the principle of technological agnosticism (§6.2.1). A model with the vocation of being applied uniformly over an ecosystem of heterogeneous development frameworks cannot legislate over the concrete primitive — annotation, inheritance, decorator, interface implementation, attribute delegation — that each development framework requires to activate. What is prescriptive is the result: the conformance of the resulting XF component with the rules of its $L \times T$ cell and with the axioms of the model. Any syntax of the integrated is admissible as long as that result is preserved.

The following table sets out the mechanism prescribed in each case.

Table 11. Mechanism prescribed by the XF model according to the XF normalization of the integrated component. When the integrated is not normalized in XF, the prescription operates over the result — the correct classification of the integrator's own XF component that assumes the responsibility — and not over the syntax of the contact, which is at the discretion of the integrator. When the integrated is already normalized in XF, the mechanism is determined by the type of the component and the model prescribes it normatively.

XF normalization	Component type	XF mechanism
Not normalized in XF	(any functionality)	Reformulation into the integrator's own XF component (§10.2.1)
Normalized in XF	Logical	Injection (§10.2.2)
Normalized in XF	Generalization	Inheritance (§10.2.3)
Normalized in XF	Utility	Inheritance or direct use (§10.2.3 / §10.2.4)
Normalized in XF	Transfer	Direct use (§10.2.4)

A cross-cutting property applies to the four mechanisms: the layer, type and directionality restrictions of the model (§6.2.2, §6.2.4, §9.4) operate with the same force over dependencies between artifacts as over dependencies internal to an artifact. The boundary between artifacts is not a zone of normative relaxation; it is a boundary of visibility, not of prescription.

10.2.1 Reformulation of functionality not normalized in XF

When the integrated does not expose an $L \times T$ classification — development framework, operating system development kit, traditional library, legacy module, code generated by a tool that does not know the XF model — the integrator recognizes the functional responsibility that the external functionality is going to carry out within its artifact and defines its own XF component classified into the corresponding $L \times T$ cell that makes contact with the external behavior by means of the syntactic mechanism that the language or the development framework offer. The type of the resulting XF component — logical, generalization, utility or transfer — is the integrator's decision according to the nature of the behavior that the traditional dependency provides; a generalization component is the canonical place to abstract technological dependencies common to several logical components of the same layer (*e.g.* a `RestRepository` that encapsulates a generic HTTP client from which the integrator's Access logical components inherit). The model adds no specific rules to verify the correctness of the assignment: the material consequences — correct layer of the XF component, fit of its functional responsibility to that of its layer, absence of leakage of primitives of the integrated development framework — are captured by the prescriptions applicable to its type, which operate with the same force over the component regardless of whether its behavior proceeds from the integrator's own implementation or from the encapsulation of an external dependency.

The layer assignment criterion is the same one that governs any logical classification of the model, without attenuation by the external origin: a functionality that knows a communication protocol with an external system and does not know business rules is assigned to the **Access Layer**; a functionality that knows domain concepts of the artifact and does not know external protocols or presentation mechanisms is assigned to the **Business Layer**; a functionality that defines an entry point to the artifact or produces a representation for the consumer is assigned to the **Interaction Layer**. When an external functionality implements responsibilities of more than one layer, those responsibilities **shall** be decomposed and reformulated into distinct XF components, one per layer.

Syntactic freedom of the contact The syntactic mechanism with which the resulting XF component makes contact with the external functionality is the integrator's decision. Attribute delegation — the XF component maintains an instance of the development framework class as an internal attribute and delegates operations to it — is the most direct and self-sufficient realization, but it is neither the only one nor necessarily the most appropriate. Inheritance from a base class of the development framework, decoration with annotations processed by the development framework container, implementation of an interface declared by the development framework, or any combination of the above are equally admissible when the language or the development framework require it to activate their infrastructure — automatic trans-

action management, discovery by reflection, dependency injection of the development framework, integration into the operating system component hierarchy.

What the model prescribes is the result, not the primitive: the resulting XF component **shall** fulfill all the prescriptions of its $L \times T$ cell, and in particular its functional responsibility — including the public interface it exposes — shall be limited to that of its layer (§7.3.1). The surface exposed to the rest of the artifact is the XF component, not the integrated: the upper layers do not know of the existence of the external functionality, they only know the XF component that reformulates it. The leakage of primitives of the integrated development framework (*e.g.* a `Repository` that exposes a JDBC `Connection` to its consumers) is a particular case of excess functional responsibility: the contract of the resulting XF component ceases to be "to offer the data abstraction of the domain" and becomes "to expose the driver", which falls outside the responsibility of the Access Layer correctly understood.

```
// Attribute delegation: the XF component maintains an instance
// of the library as an attribute and delegates operations to it.

class DatabaseRepository extends Repository {

    FrameworkClient client;

    void init(String url) {
        this.client = new FrameworkClient(url);
    }

    List<Record> get(String endpoint) {
        return this.client.call("GET", endpoint);
    }
}
```

Listing 35. Contact by attribute delegation

```
// Decoration with a development framework annotation: the XF component is
    declared
// with metadata that the container processes to activate transaction
// management, pooling or other development framework services.

@Repository
class IdentityRepository extends Repository {
    // ...
}
```

Listing 36. Contact by decoration with a development framework annotation

```
// Inheritance from a base class of the development framework: the XF
    component is
// integrated into the development framework hierarchy while retaining its XF
    role
// in the L x T matrix of the integrating artifact. Here a component of the
// Interaction Layer (View suffix) inherits from Flutter's base Widget to
// integrate into its render tree, without ceasing to fulfill its XF role:
// to present the domain state to the consumer of the artifact.

class TemperatureView extends StatelessWidget {

    // build() is the extension point that Flutter's Widget requires;
    // the View accesses the domain through the injection (Interaction ->
        Business)
    // and composes the view with framework primitives.
    Widget build(BuildContext context) {
        Temperature t = B.temperature.current();
        return Text(t.toFahrenheit() + " F");
    }
}
```

Listing 37. Contact by inheritance from a base class of the development framework

The three examples illustrate different syntaxes; none exhausts the possibilities nor constitutes the only admissible form for its case. The choice is dictated by the integrated development framework and the integrator's design criterion.

Inherited transfers: $L \times T$ classification without canonization In reformulating external functionality, the XF component receives, returns or propagates data types defined by the development framework, the operating system kit or the integrated library — `Connection` of a JDBC driver, `Response` of an HTTP client, `IOException` of an SDK, `Promise<T>` of the runtime, `Date` of the language. The model calls them **inherited transfers** and treats them like any other transfer component: they belong to $\mathbb{C}_T$ and the axiomatization does not recognize an external domain (§9.5). Two practical consequences follow from this:

- **They are classified in $L \times T$.** The integrator **shall** assign each inherited transfer that appears in the signature of its operations to the layer of lowest abstraction in which it is introduced (criterion of §6.2.4): an `IOException` that comes from a database driver belongs to the Access Layer; a `Date` used by the business logic, to the Business Layer. The rule reaches every inherited transfer, without exception.
- **They are not canonized.** They retain the identifier and structure of origin: the model **does not** prescribe renaming them with canonical suffixes nor relocating them. The canonical vocabulary (§7.3.5) obligates only the transfers that the integrator defines by itself.

10.2.2 Injection of XF logical components

When the integrated exposes logical components already normalized in XF, the integrator accesses them following the canonical pattern `<injection>.<component>.<operation>()` prescribed in §7.3.3. The pattern is satisfied both if the referenced injection is that of the integrated itself (*e.g.* `com.authxf.B.user.getCurrent()`) and if the integrator previously incorporates the external logical component into its own injection and accesses through it (`B.user.getCurrent()`). Both forms are conformant with the model; the model does not prescribe the choice.

Incorporation into the integrator's own injection — when the integrator opts for it — is subject to the typing rules of the slots: a logical component of the Business Layer of the integrated can only be incorporated into the `B` injection component of the integrator; an Access logical component, into `R`; an Interaction logical component, into `A`. The layer is a property of the component, not of the artifact that provides it, and incorporation does not alter it. This restriction is applied per-slot over the integrator's own injection component.

Incorporation admits several variants according to how the integrator combines the **lifecycle initialization** — delegated to the integrated or assumed by the integrator — and the **referencing** — alias to the instance managed by the integrated or new

instance. The model does not prescribe the variant: the choice is dictated by the coverage of the external catalog that the integrator needs to exhibit, whether the component's state must be shared or independent, and whether the presence in memory of the integrated's complete catalog is admissible. The derived combinations are valid and may coexist in the same injection with the integrator's own logical components.

The only proscription derives directly from the principle of uniqueness of the injection component per layer (§7.3.3): no logical component — own or external — may be instantiated outside an injection component, nor declared in an injection of a layer different from its own. The model covers the first aspect (creation shall occur within some `R`, `B` or `A`) and the second (the slot shall be in the injection of the corresponding layer), with the same force over integrated components as over own components.

The initialization order (§8.2) applies within each artifact over its own components. The dependency between artifacts is resolved in the invocation order: if an own logical component of the integrator depends on a logical component of the integrated during its initialization, the integrator **shall** order the `init` invocations respecting that dependency.

```
// Variant with shared state: the own injection maintains an
// alias to the instance managed by the integrated and combines that
// alias with the integrator's own logical components.
final abstract class B {

    // The integrator's own logical components.
    static readonly NetworkBusiness network = new NetworkBusiness();

    // Logical components of the integrated incorporated by reference.
    static readonly TemperatureBusiness temperature =
        com.myappxf.B.temperature;
    static readonly UserBusiness user = com.myappxf.B.user;
    static readonly SessionBusiness session = com.myappxf.B.session;

    static void init() {
        // the integrated initializes its own components.
        com.myappxf.B.init();

        // the integrator initializes its own.
        B.network.init();
    }

    static void terminate() {
        B.network.terminate();
        com.myappxf.B.terminate();
    }
}
```

Listing 38. Injection by reference of external logical components (shared state)

```
// Variant with isolated state: the own injection instantiates
// directly the logical component of the integrated and assumes its
// lifecycle.
final abstract class B {

    static readonly TemperatureBusiness temperature =
      new com.myappxf.TemperatureBusiness();

    static void init() {
        B.temperature.init();
    }

    static void terminate() {
        B.temperature.terminate();
    }
```

```
}
```

Listing 39. Injection by instantiation of an external logical component (isolated state)

10.2.3 Inheritance of components

Inheritance is the mechanism prescribed for integrating generalization components published by an XF artifact. The integrator defines its own component in the equivalent layer and specializes its behavior.

The layer restriction derives directly from the principle of layer isolation (§6.2.2): a component classified in a layer of the integrated can only be inherited by components of the **same layer** of the integrator. Inheritance between different layers constitutes a violation of the model.

```
// Inheritance of an XF generalization published by a library.
class IdentityRepository extends com.restxf.RestRepository {

    void init() {
        super.init("https://identity.xforg.com");
    }

    User fetchUser(String id) {
        return get("/users/" + id);
    }
}
```

Listing 40. Inheritance of an XF generalization published by a library

```
// Inheritance of an XF utility published by a library: the base
// is extensible (not final) and not instantiable due to the static
// nature of its operations; the derived one adds static operations
// to the catalog without altering the contract of the type.
class TemperatureUtils extends com.unitsxf.MeasurementUtils {

    static Temperature normalize(Temperature t) {
        // own operation that extends the catalog of MeasurementUtils
    }
}
```

Listing 41. Inheritance of an XF utility published by a library

10.2.4 Direct use

Direct use is the mechanism prescribed for integrating utility and transfer components published by an XF artifact. The integrator imports the component and uses it in its code without wrapping it or declaring it in any injection component: it instantiates the transfers with `new` when needed and invokes the utility operations statically.

The directionality restrictions of §9.4 operate over the transfers of the integrated with the same force as over its own: a transfer of the integrated defined in the Access Layer may be referenced by components of any layer of the integrator; a Business transfer, by Business or Interaction components of the integrator, but not by Access components. The structural transformation guideline of §9.4 applies indistinctly as a design criterion: if the integrator needs a structure with a definition different from the one the integrated publishes — adding or removing attributes — the canonical option is to define its own transfer component in the corresponding layer. The inheritability of the transfer (§7.3.5) is an alternative when the integrator requires

preserving the structural lineage while at the same time extending attributes; direct use and inheritance are both admissible as appropriate.

The locality restrictions of utilities (§7.3.4) apply equally: a utility defined in the Business Layer of the integrated can only be referenced by components of the Business Layer of the integrator, except for the already mentioned exception of utilities over primitive types.

```
// Direct use of the Temperature structure and the TemperatureUtils
// utility published by the xf-temperature library.
import com.temperaturexf.repository.transfers.Temperature;
import com.temperaturexf.business.utils.TemperatureUtils;

class TemperatureBusiness extends StatefulBusiness<Temperature> {

    void update(Temperature temperature) {
        if (TemperatureUtils.toCelsius(temperature) < 14)
            throw new TemperatureOutOfRangeException();
        R.server.saveTemperature(temperature);
        super.update(temperature);
    }
}
```

Listing 42. Direct use of an XF transfer and an XF utility published by a library

Combination of mechanisms The four mechanisms are not mutually exclusive. A single integrating artifact may simultaneously reformulate functionalities of several traditional development frameworks into its own XF components, inject logical components of an XF library into its injection, inherit generalization components of another, and directly use its transfers. Each mechanism operates under its own prescriptions without additional restrictions due to the combination. The only cross-cutting element is the governing rule: each resulting XF component, whatever the mechanism employed to build it, retains its $L \times T$ classification and shall satisfy the rules of its cell.

10.3 Ecosystem of XF libraries

Formal definition of an XF library An **XF library** is defined as a software artifact that implements the XF model and that exposes, through its public interfaces, one or more of its components for consumption by other XF artifacts. An XF library may freely combine components of any type — generalizations, transfers, utilities and, when it models a concrete domain, also logical components through its injection components.

An XF library is distinguished from a conventional XF artifact in its purpose: whereas a conventional XF artifact automates a complete formal process — with its three layers active and its own entry point — an XF library provides components that other artifacts incorporate to build their own logic.

Two properties characterize the quality of an XF library without forming part of the prescriptive body of the model. The model does not prescribe them as conformance rules: a library that does not satisfy them remains conformant with the XF model if its structure is so; what they add or subtract is ecosystem quality, not architectural conformance.

The first is **functional idempotence**: the interfaces that the library exposes should produce consistent and predictable results for the same inputs, independently of the artifact that consumes them or of the execution context in which they operate. A library with non-idempotent interfaces is structurally XF if it respects layers, types, nomenclature and lifecycle, but its usefulness as a reusable piece degrades because its consumers cannot reason about its behavior locally.

The second is **uniformity across technologies**: when an XF library is published in several technologies (*e.g.* a Java variant and a TypeScript variant of the same abstraction), the signatures of its operations should be conceptually equivalent in order to minimize the learning curve of the consumer who moves between languages. A library that does not satisfy this property — for idiomatic reasons of the language, for conventions of the target development framework or by the author's decision — remains conformant with the model; what it adds is heterogeneity complexity over the ecosystem, not normative non-compliance.

Types of XF libraries The only doctrinally significant distinction between XF libraries is that of the semantic commitment to a concrete domain. The distinction is not taxonomic in the manner of the component type — an XF library may freely combine components of any type and normally does so — but rather structural and functional, derived from a single question: *does the library implement the semantics of some concrete domain?* The presence or absence of logical components is the syntactic mark of the answer, and it has an immediate consequence on the lifecycle that the library imposes on the consumer. This distinction is descriptive: it enriches the vocabulary with which the XF standard discusses the ecosystem, without altering the conformance prescriptions of the model nor the integration mechanisms of §10.2, common to both types.

Utility libraries. They do not model any domain. Their contribution is reusable patterns, structures and operations without semantic commitment — a `RestRepository` generalization that abstracts the generic HTTP protocol, a `StringUtils` utility that operates over primitive types, an `HttpResponse` transfer that normalizes the representation of the result of a REST invocation, a `ValidationUtils` utility that verifies universal formats such as UUID or IBAN. As they contain no logical components they do not expose their own entry point: they require no orchestration of `init()` or `terminate()` on the part of the consumer, and they do not maintain state between invocations. Their value grows linearly with the number of artifacts that use them without imposing any domain decision on them, which makes them natural candidates for shared infrastructure cross-cutting to the industry — a good XF library for an HTTP client, cryptography, time handling or universal format validation has adoption potential in any artifact, whatever its domain.

Domain libraries. They model a subset of a concrete domain and encapsulate it as a reusable artifact — a federated identity provider, a corporate authentication scheme, a telemetry protocol, a thermoregulation library. They contain logical components whose singleton instances live in the injection components of the library and, consequently, expose their lifecycle to the consumer, who orchestrates their `init()` and `terminate()` respecting the dependencies with the rest of the artifact. Their value is concentrated in the artifacts that share that domain, and their impact is measured by the convergence they produce among heterogeneous implementations

over a single representation of the same domain — the inflection point where the industry stops reimplementing identity, telemetry or payment logic in each artifact and starts composing proven domain libraries.

The properties of the ecosystem The ecosystem of XF libraries has four properties that distinguish it from the ecosystems of traditional libraries. Unlike the quality properties stated earlier, which are descriptive, these ecosystem properties are consequences derivable from the prescriptions of the model: they emerge when the model is applied to a sufficiently broad set of artifacts and libraries, and they reflect the structure of the model itself, not aspirations of the author.

The first is **non-redundancy between libraries**. In the ecosystems of traditional libraries, the same functionality coexists replicated in multiple libraries with different interfaces — it is common to find several HTTP clients for Java, several validation libraries for JavaScript or several implementations of the observable pattern for Dart. The redundancy is attributed in part to the coupling of each library to a concrete development framework or paradigm, in addition to factors such as competition between projects and the temporal evolution of the ecosystem. In the XF ecosystem, a library that implements `RestRepository` conformant with the model is the reference implementation for that pattern in that technology — there is no reason for multiple incompatible implementations of the same pattern to exist in the same technology, because the model prescribes the interface that all of them shall satisfy.

The second is the **minimization of the size of artifacts**. When the most common structural patterns are published as XF libraries, the artifacts that consume them do not need to implement them internally. An artifact that inherits `RestRepository`, `StatefulBusiness` and `FormView` from XF libraries only needs to implement the effective logic of its own components — the concrete endpoints, the specific business rules, the particular visual composition. The contextual logic — the protocol, the observation pattern, the form management — is already in the libraries. This reduces the size of the code specific to each artifact and concentrates the development effort on what really differentiates each solution.

The third is **portability across technologies**. An XF library that implements `RestRepository` in Java and its equivalent in TypeScript expose the same abstraction with the same responsibilities. A developer who knows the XF abstraction of `RestRepository` can use either of the two implementations without additional learning curve — they only need to learn the syntax of the language, not the concept. This property is the one that allows the XF ecosystem to reduce the cost of context switching between technologies characterized in §1.2.

The fourth is **conformance verifiability**. Given that the XF model prescribes the interface that the components of each type shall satisfy, it is possible to verify statically whether a library is XF-compatible — whether its generalization components respect the layer restrictions, whether its transfer components contain no behavior logic, whether its utility components are pure and not instantiable. The conformance analysis tools can verify both the artifacts and the libraries they consume.

11 Conformance of the XF model

The conformance of an artifact with the XF model is the property that determines to what extent that artifact satisfies the normative prescriptions established in the preceding clauses. The present clause operationalizes those prescriptions — it converts them into verifiable conditions that allow the degree of conformance of any software artifact with the XF model to be computed deterministically, independently of the programming language, the development framework or the business domain of the artifact.

The formal nature of the XF model makes it possible for conformance to be algorithmically computable by static code analysis tools. Given an artifact and the catalog of rules of this clause, the conformance level is a function of the artifact — it does not depend on the interpretation of the analyst but on the mechanical application of the algorithm defined in §11.4. This property enables the automatic and deterministic verification of the conformance of any artifact, independently of the technology and of the reference architecture that the artifact projects onto the model.

The clause is organized as follows. Clause §11.1 defines the concepts specific to conformance — verifiable element, violation, conformance level and verifiability of the rules. Clause §11.2 describes the five conformance levels. Clause §11.3 presents the complete catalog of verifiable rules, organized into nine thematic groups derived from the preceding normative clauses. Clause §11.4 defines the algorithm for determining the conformance level and the conditions of applicability of the rules.

11.1 Conformance model

The verification of the conformance of an artifact with the XF model requires a set of precise concepts that allow the deviations of the artifact with respect to the prescriptions of the model to be described, classified and quantified. The central concept is the **conformance rule**: a verifiable condition that operationalizes a normative prescription of the model into a form evaluable over a verifiable element of the artifact. The closed and exhaustive set of rules that the model prescribes constitutes the **catalog** developed in §11.3; each rule of the catalog carries a canonical identifier in kebab-case format and is characterized by its **verifiability** — structural or semantic —, which determines whether the rule can be evaluated deterministically from observable syntactic properties of the source code (structural) or requires semantic interpretation of the functional responsibility (semantic). Verifiability honestly delimits the scope of static analysis tools and is dealt with in §11.1.4. Every rule of the catalog has the same normative status: its non-fulfillment produces a violation that affects the conformance level of the artifact.

The present clause defines the auxiliary concepts of the conformance model — verifiable element, violation, conformance level and verifiability of the rules — with the precision necessary for their application to be unambiguous.

11.1.1 Verifiable element

A **verifiable element** is any element of the artifact over which a conformance rule can be evaluated. The XF model recognizes two types of verifiable elements, which

are mutually disjoint — no element belongs simultaneously to more than one type.

A **code element** is any unit of code with its own identity — classes, modules or files of cohesive functions as defined in §12. It comprises two classes: the **component** — code element classified in the $L \times T$ matrix, the most frequent case, over which the majority of the catalog rules operate — and the **XF start-point element** `XF` — code element that orchestrates the life cycle of the artifact and is not classified in $L \times T$, over which the rules of Group 8 operate.

A **structural element** is any folder or subfolder of the file system of the artifact that forms part of its directory structure. The rules of Group 1 of the catalog operate exclusively over structural elements — they verify the existence and organization of the canonical folder structure without the need to inspect the content of any file.

Every rule of the catalog of §11.3 operates over exactly one type of verifiable element. This classification is indicated in the definition of each rule of the catalog (§11.3).

11.1.2 Violation

A **violation** is the determination that a verifiable element of the artifact does not satisfy the condition prescribed by a rule of the conformance catalog. Every violation is associated with exactly one rule and exactly one verifiable element — if ten components violate the same rule, there are ten independent violations. The presence of any violation is incompatible with full conformance with the model and therefore prevents the artifact from reaching level 4.

A rule is **applicable** to an artifact when there exists at least one verifiable element of the artifact over which the rule can be evaluated. A non-applicable rule does not produce violations — neither satisfied nor violated. The restriction of the domain of each rule to its type of verifiable element guarantees by construction that type-specific rules do not generate violations over elements outside their canonical classification.

11.1.3 Conformance level

The **conformance level** is a property of the artifact that determines to what extent it satisfies the normative prescriptions of the XF model. The conformance level is a discrete value in the set **{0, 1, 2, 3, 4}** — computable deterministically from two properties of the artifact: the code totality condition and the set of violations.

The **code totality condition** is satisfied when every component of the artifact is classified in the $L \times T$ matrix by means of the classification function ϕ. The `XF` element, when declared, is not a component and falls outside the totality predicate by construction. An artifact that does not satisfy this condition has unclassified components that coexist with XF components.

The conditions that determine each level are defined in §11.2.

11.1.4 Verifiability of the rules

The rules of the conformance catalog are classified according to their **verifiability**, which determines the type of analysis necessary to evaluate the rule.

A rule is **structurally verifiable** when its evaluation can be carried out deterministically from observable syntactic properties of the source code of the artifact — file names, location in the folder structure, import declarations between modules, suffixes and prefixes of identifiers, dependency graph between components. Structurally verifiable rules are those that static analysis tools can evaluate automatically with a precise result that is independent of the analyst who runs the tool.

A rule is **semantically verifiable** when its evaluation requires understanding the functional responsibility of the code — not only its syntactic form — in order to determine whether the implemented logic corresponds to the responsibility of the category to which the component is classified. Semantically verifiable rules require analysis additional to that of the static analysis tool: human architectural review, data flow analysis, analysis of the intent of the code, or combinations of the foregoing. Their evaluation is not trivially automatable.

The rules of the catalog are classified according to their **verifiability**: **structural** (decidable deterministically by static analysis over observable syntactic properties of the source code) or **semantic** (requires interpretation of the functional responsibility). The distinction delimits the scope of static analysis tools: structural rules are algorithmically evaluable; semantic rules are not.

Implications for the computation of conformance The verifiability of a violation — structural or semantic — has a **normative** effect on the conformance level: it determines the ceiling that the artifact can reach. A structural violation prevents exceeding level 2; in the absence of structural violations, a semantic violation prevents level 4 but permits level 3. The distinction additionally retains its methodological side — the structural violation is detected algorithmically by static analysis, the semantic one requires human intervention.

The conformance function is **deterministic** — given the complete set of violations, the level is uniquely determined —, but its **complete application is not automatable**: the semantic rules of the catalog require interpretation of the functional responsibility. From this follows a precise boundary for static analysis: a tool can determine on its own whether an artifact reaches **level 3** — which depends solely on structural violations —, but **cannot** affirm **level 4**, whose distinction with respect to level 3 rests on semantic violations. The tools of the XF ecosystem **shall not** report level 4 as definitive without the explicit qualification that the semantic rules of the catalog require human evaluation.

11.2 Conformance levels

The XF model defines five discrete, ordered and mutually exclusive conformance levels. Every software artifact belongs to exactly one of these levels at a given moment. The levels are ordered — a higher level implies the satisfaction of all the conditions of the lower levels plus additional conditions. The transition between levels is always the result of correcting violations or of classifying unclassified components — it does not require any additional action outside the normative scope of the model.

11.2.1 Level 0 — Non-conformant

An artifact is of level 0 when it contains no element classified in the $L \times T$ matrix. There is no recognizable XF structure — no injection component, no canonical folder, no component typed according to the taxonomy of the model. The rules of the catalog of §11.3 are not applicable because there is no XF structure over which to evaluate them.

Level 0 is the natural state of every traditional artifact — developed under any architecture or pattern other than XF. It is not an error state but the starting state of any artifact before the adoption of the model.

11.2.2 Level 1 — Partially conformant

An artifact is of level 1 when it contains at least one component classified in the $L \times T$ matrix and accessible through an injection component, but does not satisfy the code totality condition — there exist unclassified components that coexist with XF components. The rules of the catalog apply over the XF part of the artifact — the unclassified components do not activate conformance rules because they do not form part of the XF structure of the artifact.

Level 1 is the typical state of an artifact in the process of progressive migration towards the XF model. The components are classified and migrated incrementally without the need to refactor the complete artifact at once — each component that is correctly classified reduces the set of unclassified components and brings the artifact closer to the totality condition necessary to reach level 2.

11.2.3 Level 2 — Imperfectly conformant

An artifact is of level 2 when it satisfies the code totality condition — every component is classified in the $L \times T$ matrix — but contains at least one **structural** violation. The artifact has fully adopted the XF structure, but fails to fulfill at least one structural prescription; correcting all the structural violations raises it to level 3.

Level 2 is the typical state of an artifact in the process of refinement — the typing is complete but there remain structural violations to be corrected. Conformance analysis tools are especially valuable at this level because they can identify with precision which violations must be corrected and in which components they occur.

11.2.4 Level 3 — Structurally conformant

An artifact is of level 3 when it satisfies the code totality condition and contains no structural violation, even though it contains at least one semantic violation. Its structure — classification, directionality between layers, naming, location and uniqueness of the injection components — is wholly correct; the inconsistencies that remain belong to the functional responsibility of some component, whose evaluation requires semantic interpretation.

Level 3 is the highest level that a static analysis tool can certify on its own: it depends exclusively on structural violations, decidable deterministically over the source code. An artifact at level 3 is a **well-formed artifact** — it satisfies by construction the structural conditions on which the model makes its formal properties depend, such

as the inter-layer acyclicity of the dependency graph. The formal development of that equivalence as a theory is left as an object of future evolutions of the model.

11.2.5 Level 4 — Perfectly conformant

An artifact is of level 4 when it satisfies the code totality condition and contains no violation, neither structural nor semantic. It satisfies all the normative prescriptions of the model — structural and semantic — and can be analyzed, maintained and evolved by any developer trained in the model independently of the technology employed.

Level 4 is the target state of every XF artifact in production. Its certification is not completely automatable: it requires, over an artifact already verified at level 3 by static analysis, the human evaluation of the semantic rules of the catalog.

The following table summarizes the five levels with their determining conditions:

Table 12. Comparison of the five conformance levels of the XF model: determining conditions, operational semantics, what the artifact guarantees and which static analysis tools detect it.

Level	Designation	Totality	Violations	Semantics / detection
$\lambda = 0$	Non-conformant	No XF elements	N/A	Traditional artifact; no structural guarantee.
$\lambda = 1$	Partially conformant	Not satisfied	N/A	Classified and unclassified components coexist; the catalog does not apply until totality is resolved.
$\lambda = 2$	Imperfectly conformant	Satisfied	≥ 1 structural	Fails to fulfill at least one structural prescription; detectable deterministically by static analysis.
$\lambda = 3$	Structurally conformant	Satisfied	0 structural; ≥ 1 semantic	Well-formed artifact; structure wholly correct. Highest level certifiable by a static analysis tool.
$\lambda = 4$	Perfectly conformant	Satisfied	0 total	All prescriptions, structural and semantic, satisfied; equivalent to the idealized artifact. The $3 \to 4$ distinction requires human semantic evaluation.

11.3 Catalog of rules

The catalog of conformance rules is the exhaustive, closed set of verifiable conditions that operationalize the normative prescriptions of the XF model. Each rule of

the catalog derives directly from a prescription established in clauses 6 to 10 — it introduces no new requirements but rather expresses each prescription in the form of a condition verifiable by static analysis tools.

The catalog is organized into nine thematic groups. The grouping criterion is the **type of component or structural element** on which the rule operates — consistent with the expository order of §7.3 and §7.2, which describe the normative prescriptions in the context of each type. The class of prescription (nomenclature, dependency, inheritance, state, purity, lifecycle) is encoded as the second segment of the canonical identifier (for example, `logic-naming-repository` belongs to Group 3 by component type and to the *nomenclature* subgroup by prescription class), permitting filtering by class from analysis tools without the need to regroup.

The groups are not mutually independent: a single architectural deviation may activate rules of multiple groups simultaneously. The domain restriction of each rule (§11.4.2) guarantees that type-specific rules do not generate violations on elements outside their canonical classification, without the need for an independent propagation mechanism.

Each rule is identified by means of a **canonical identifier** in kebab-case format — the unique name with which analysis tools reference the rule — and is described by means of a precise sentence that indicates which condition it verifies. Its violation affects the conformance level of the artifact. The definition of each rule — its violation condition, the type of verifiable element on which it operates, and its normative reference — is developed in the catalog groups that follow.

The present catalog is normative and closed in the current version of the model. Future versions of the model may extend the catalog with new rules derived from the per-development-framework compatibility document and from the evolution of the XF library ecosystem, provided that each new rule derives from a normative prescription of the model and does not introduce requirements not contemplated in the normative body.

Table 13. Master catalog of conformance rules of the XF model: canonical identifier (`kebab-case`), thematic group, applicability domain over the type of verifiable element (**fld** = Folder, **cod** = code element), verifiability (**str** = Structural — decidable by deterministic static analysis —, **sem** = Semantic — requires interpretation of the functional responsibility) and reference to the clause that originates it. Non-compliance with any rule produces a violation that affects the conformance level.

Canonical identifier	Group	Dom.	Verif.	Normative origin
`structure-layer-mismatch`	1	fld	str	§7.4, §8.1
`structure-type-mismatch`	1	fld	str	§7.4
`structure-injection-missing`	1	fld	str	§7.3.3
`structure-injection-multiplicity`	1	fld	str	§7.3.3
`structure-component-naming`	1	fld	str	§7.4
`structure-domain-subdivision`	1	fld	sem	§7.2.1, §7.2.3
`layer-reference`	2	cod	str	§6.2.2
`layer-inheritance`	2	cod	str	§6.2.2
`layer-skip`	2	cod	str	§6.2.2
`logic-naming-repository`	3	cod	str	§7.3.1, §7.2.1

Continued on the next page

Table 13 (continued)

Canonical identifier	Group	Dom.	Verif.	Normative origin
`logic-naming-business`	3	cod	str	§7.3.1, §7.2.2
`logic-naming-service`	3	cod	str	§7.3.1, §7.2.3
`logic-naming-view`	3	cod	str	§7.3.1, §7.2.3
`logic-mismatch-repository`	3	cod	sem	§7.2.1
`logic-mismatch-business`	3	cod	sem	§7.2.2
`logic-mismatch-api`	3	cod	sem	§7.2.3
`logic-initialization-missing`	3	cod	str	§7.3.1, §8.2
`logic-termination-missing`	3	cod	str	§7.3.1, §8.2
`logic-constructor-mismatch`	3	cod	str	§8.2
`logic-inheritance`	3	cod	str	§7.3.1
`general-naming-repository`	4	cod	str	§7.3.2, §7.2.1
`general-naming-business`	4	cod	str	§7.3.2, §7.2.2
`general-naming-service`	4	cod	str	§7.3.2, §7.2.3
`general-naming-view`	4	cod	str	§7.3.2, §7.2.3
`general-mismatch-repository`	4	cod	sem	§7.3.2, §7.2.1
`general-mismatch-business`	4	cod	sem	§7.3.2, §7.2.2
`general-mismatch-api`	4	cod	sem	§7.3.2, §7.2.3
`general-injection-reference`	4	cod	str	§7.3.2
`general-domain-state`	4	cod	str	§7.3.1, §7.3.2
`general-instantiable`	4	cod	str	§7.3.2
`general-initialization-missing`	4	cod	str	§7.3.2, §8.2
`general-termination-missing`	4	cod	str	§7.3.2, §8.2
`general-constructor-mismatch`	4	cod	str	§8.2
`general-inheritance`	4	cod	str	§7.3.2
`injection-naming-r`	5	cod	str	§7.3.3
`injection-naming-b`	5	cod	str	§7.3.3
`injection-naming-a`	5	cod	str	§7.3.3
`injection-non-repository`	5	cod	str	§7.3.3
`injection-non-business`	5	cod	str	§7.3.3
`injection-non-api`	5	cod	str	§7.3.3
`injection-mismatch`	5	cod	str	§7.3.3
`injection-instantiable`	5	cod	str	§7.3.3
`injection-member-mutable`	5	cod	str	§7.3.3
`injection-member-public`	5	cod	str	§7.3.3
`injection-init-missing`	5	cod	str	§7.3.3, §8.2
`injection-terminate-missing`	5	cod	str	§7.3.3, §8.2
`injection-init-mismatch`	5	cod	str	§7.3.3
`injection-terminate-mismatch`	5	cod	str	§7.3.3
`injection-lifecycle-symmetry`	5	cod	str	§8.2
`injection-inheritance`	5	cod	str	§7.3.3
`utility-naming`	6	cod	str	§7.3.4
`utility-mismatch`	6	cod	sem	§7.3.4
`utility-instantiable`	6	cod	str	§7.3.4
`utility-member-instance`	6	cod	str	§7.3.4
`utility-mutable-state`	6	cod	str	§7.3.4
`utility-inheritance`	6	cod	str	§7.3.4
`transfer-naming`	7	cod	sem	§7.3.5
`transfer-dependency`	7	cod	str	§7.3.5
`transfer-business-logic`	7	cod	sem	§7.3.5
`transfer-inheritance`	7	cod	str	§7.3.5
`xf-init-missing`	8	cod	str	§8.3
`xf-terminate-missing`	8	cod	str	§8.3

Continued on the next page

Table 13 (continued)

Canonical identifier	Group	Dom.	Verif.	Normative origin
`xf-init-mismatch`	8	cod	str	§8.3
`xf-terminate-mismatch`	8	cod	str	§8.3
`lifecycle-logic-instantiation`	9	cod	str	§7.3.3
`lifecycle-logic-init`	9	cod	str	§7.3.3, §8.3
`lifecycle-logic-terminate`	9	cod	str	§8.3
`lifecycle-injection-init`	9	cod	str	§7.3.3, §8.3
`lifecycle-injection-terminate`	9	cod	str	§8.3
`lifecycle-xf-init`	9	cod	str	§8.3
`lifecycle-xf-terminate`	9	cod	str	§8.3

11.3.1 Group 1 — Folder structure

The rules of this group verify the existence and organization of the canonical folder structure prescribed in §7.4. They are the highest-level rules of the catalog — their verification does not require inspection of the content of any file, only analysis of the directory structure of the artifact.

The material existence of the four type subfolders — `/general`, `/logic`, `/transfers` and `/utils` — is not enumerated as an independent rule: a layer that contains no components of a type need not materialize the corresponding subfolder as an empty folder. When a layer contains components of a type, those components shall reside in the canonical subfolder of that type — a prescription verified by the location rules of groups 3, 4, 6 and 7 on the individual component, not on the presence of the folder.

Rules

1. `structure-layer-mismatch` *(structural)* — An element (file or folder) exists directly under an artifact root whose name is not canonical for that level (§7.4, §8.1).
2. `structure-type-mismatch` *(structural)* — An element (file or folder) exists directly under a layer folder whose name is not canonical for that layer (§7.4).
3. `structure-injection-missing` *(structural)* — A layer folder does not contain the canonical file of the injection component that corresponds to it (§7.3.3).
4. `structure-injection-multiplicity` *(structural)* — More than one file exists with the canonical name of an injection component under the root of the corresponding layer (§7.3.3).
5. `structure-component-naming` *(structural)* — A file declares an XF component whose canonical name does not match the name of the file without extension (§7.4).
6. `structure-domain-subdivision` *(semantic)* — The internal subdivision of `/repository/logic` or of `/api/logic` groups logical components by functional domain of the artifact, instead of by the legitimate subdivision criterion of its layer (§7.2.1, §7.2.3).

11.3.2 Group 2 — Layer isolation

The rules of this group verify compliance with the principle of layer isolation (§6.2.2) and with the directionality rule of transfers (§9.5) on any component of the artifact,

independently of its type. They operate homogeneously on the reference relation between components: given a pair (c, c') such that c references c', the rules verify the admissibility of the pair in terms of the layer difference of its endpoints. The verification is performed on the importing component — not on the imported one —, which allows violations to be attributed to the concrete file that introduces them.

Rules

1. `layer-reference` *(structural)* — A component references another component of a layer of higher abstraction level than its own (§6.2.2).
2. `layer-inheritance` *(structural)* — A component inherits from another component classified in a layer different from its own, in either direction (§6.2.2).
3. `layer-skip` *(structural)* — A component references another component of a lower abstraction level skipping the intermediate layer. Transfer components (§9.5) and utility components over primitive types defined in `/src/repository/utils` (§7.3.4) are excepted.

11.3.3 Group 3 — Logical components

They verify compliance with the normative prescriptions on logical components (§7.3.1, §7.2.1, §7.2.2, §7.2.3). They apply to the classes located in `/src/`*layer*`/logic/` or with canonical suffix `Repository`, `Business`, `Service` or `View`.

Nomenclature rules

1. `logic-naming-repository` *(structural)* — A logical component of the Access Layer does not bear the canonical suffix `Repository`.
2. `logic-naming-business` *(structural)* — A logical component of the Business Layer does not bear the canonical suffix `Business`.
3. `logic-naming-service` *(structural)* — A logical component of the Interaction Layer that implements a systemic interaction point does not bear the canonical suffix `Service`.
4. `logic-naming-view` *(structural)* — A logical component of the Interaction Layer that implements a graphical interaction point does not bear the canonical suffix `View`.

Content rules

1. `logic-mismatch-repository` *(semantic)* — A logical component classified in the Access Layer contains logic that does not correspond to the functional responsibility of that layer (§7.2.1).
2. `logic-mismatch-business` *(semantic)* — A logical component classified in the Business Layer contains logic that does not correspond to the functional responsibility of that layer (§7.2.2).
3. `logic-mismatch-api` *(semantic)* — A logical component classified in the Interaction Layer contains logic that does not correspond to the functional responsibility of that layer (§7.2.3).

Lifecycle rules

1. `logic-initialization-missing` *(structural)* — A logical component does not declare or inherit an invocable `init()` operation (§7.3.1, §8.2).
2. `logic-termination-missing` *(structural)* — A logical component does not declare or inherit an invocable `terminate()` operation (§7.3.1, §8.2).
3. `logic-constructor-mismatch` *(structural)* — A logical component implements non-trivial initialization logic within its constructor instead of in a separate `init()` operation (§8.2).

Inheritance rules

1. `logic-inheritance` *(structural)* — A logical component inherits from a component that is neither a logical component nor a generalization component of its same layer (§7.3.1, §7.3.2).

11.3.4 Group 4 — Generalization components

They verify compliance with the normative prescriptions on generalization components (§7.3.2, §7.4). They apply to the classes located in `/src/`*layer*`/general/`.

Nomenclature rules

1. `general-naming-repository` *(structural)* — A generalization component of the Access Layer does not end with the canonical suffix `Repository` (§7.3.2, §7.2.1).
2. `general-naming-business` *(structural)* — A generalization component of the Business Layer does not end with the canonical suffix `Business` (§7.3.2, §7.2.2).
3. `general-naming-service` *(structural)* — A generalization component of the Interaction Layer that abstracts behavior for systemic interaction points does not end with the canonical suffix `Service` (§7.3.2, §7.2.3).
4. `general-naming-view` *(structural)* — A generalization component of the Interaction Layer that abstracts behavior for graphical interaction points does not end with the canonical suffix `View` (§7.3.2, §7.2.3).

Content rules

1. `general-mismatch-repository` *(semantic)* — A generalization component classified in the Access Layer contains logic that does not correspond to the functional responsibility of that layer (§7.2.1).
2. `general-mismatch-business` *(semantic)* — A generalization component classified in the Business Layer contains logic that does not correspond to the functional responsibility of that layer (§7.2.2).
3. `general-mismatch-api` *(semantic)* — A generalization component classified in the Interaction Layer contains logic that does not correspond to the functional responsibility of that layer (§7.2.3).
4. `general-injection-reference` *(structural)* — A generalization component invokes operations of logical components through an injection component (§7.3.2).

5. `general-domain-state` *(structural)* — A generalization component declares mutable attributes that model concepts of the domain of the artifact (§7.3.1, §7.3.2).

Instantiation rules

1. `general-instantiable` *(structural)* — A generalization component does not explicitly prevent its direct instantiation by means of the mechanisms of the programming language (§7.3.2).

Lifecycle rules

1. `general-initialization-missing` *(structural)* — A generalization component does not declare or inherit an invocable `init()` operation (§7.3.2, §8.2).
2. `general-termination-missing` *(structural)* — A generalization component does not declare or inherit an invocable `terminate()` operation (§7.3.2, §8.2).
3. `general-constructor-mismatch` *(structural)* — A generalization component implements non-trivial logic within its constructor instead of in a separate `init()` operation (§8.2).

Inheritance rules

1. `general-inheritance` *(structural)* — A generalization component inherits from an XF component internal to the artifact that is not a generalization component of its same layer (§7.3.2).

11.3.5 Group 5 — Injection components

They verify compliance with the normative prescriptions on the injection components `R`, `B` and `A` (§7.3.3, §8.2). They apply to the files with canonical name `R`, `B` or `A` located directly at the root of `/src/repository`, `/src/business` and `/src/api`, respectively.

Nomenclature rules

1. `injection-naming-r` *(structural)* — The injection component of the Access Layer does not bear the canonical name `R` (§7.3.3).
2. `injection-naming-b` *(structural)* — The injection component of the Business Layer does not bear the canonical name `B` (§7.3.3).
3. `injection-naming-a` *(structural)* — The injection component of the Interaction Layer does not bear the canonical name `A` (§7.3.3).

Content rules

1. `injection-non-repository` *(structural)* — The injection component `R` declares as a static attribute a component that is not a logical component of the Access Layer (§7.3.3).

2. `injection-non-business` *(structural)* — The injection component `B` declares as a static attribute a component that is not a logical component of the Business Layer (§7.3.3).
3. `injection-non-api` *(structural)* — The injection component `A` declares as a static attribute a component that is not a logical component of the Interaction Layer (§7.3.3).
4. `injection-mismatch` *(structural)* — An injection component declares any member that is not a typed static reference to a logical component of its layer or the static operations `init()` or `terminate()` (§7.3.3).

Class structure rules

1. `injection-instantiable` *(structural)* — An injection component does not explicitly prevent its instantiation by means of the mechanisms of the programming language (§7.3.3).
2. `injection-member-mutable` *(structural)* — The static references to the logical components in the injection component are not immutable (§7.3.3).
3. `injection-member-public` *(structural)* — The static references to the logical components in the injection component do not have public visibility (§7.3.3).

Lifecycle rules

1. `injection-init-missing` *(structural)* — An injection component does not declare an invocable static `init()` operation (§7.3.3, §8.2).
2. `injection-terminate-missing` *(structural)* — An injection component does not declare an invocable static `terminate()` operation (§7.3.3, §8.2).
3. `injection-init-mismatch` *(structural)* — The body of the `init()` operation of an injection component contains statements other than invocations to `init()` of the logical components aggregated as slots in that injection (§7.3.3).
4. `injection-terminate-mismatch` *(structural)* — The body of the `terminate()` operation of an injection component contains statements other than invocations to `terminate()` of the logical components aggregated as slots in that injection (§7.3.3).
5. `injection-lifecycle-symmetry` *(structural)* — There exists a slot s of an injection component such that s`.init()` is invoked in the body of `init()` and s`.terminate()` is not invoked in the body of `terminate()`, or vice versa (§8.2).

Inheritance rules

1. `injection-inheritance` *(structural)* — An injection component inherits from any component, internal or external to the artifact (§7.3.3).

11.3.6 Group 6 — Utility components

They verify compliance with the normative prescriptions on utility components (§7.3.4). They apply to the classes located in `/src/`*layer*`/utils/` or with canonical suffix `Utils`.

Nomenclature rules

1. `utility-naming` *(structural)* — A utility component does not bear the suffix `Utils` (§7.3.4).

Content rules

1. `utility-mismatch` *(semantic)* — A utility component implements rules of the domain modeling of the artifact, invokes logical components through the injection components, or produces observable side effects (§7.3.4).

Class structure rules

1. `utility-instantiable` *(structural)* — A utility component does not explicitly prevent its instantiation by means of the mechanisms of the programming language (§7.3.4).
2. `utility-member-instance` *(structural)* — A utility component declares instance members (non-static attributes or methods) (§7.3.4).
3. `utility-mutable-state` *(structural)* — A utility component declares mutable attributes (§7.3.4).

Inheritance rules

1. `utility-inheritance` *(structural)* — A utility component inherits from a component that is not a utility component of its same layer (§7.3.4).

11.3.7 Group 7 — Transfer components

They verify compliance with the normative prescriptions on transfer components (§7.3.5, §9). They apply to the classes located in `/src/`*layer*`/transfers/`.

Nomenclature rules

1. `transfer-naming` *(semantic)* — A transfer component bears a suffix added to the domain concept it models (§7.3.5).

Structure and self-containment rules

1. `transfer-dependency` *(structural)* — A transfer component references an injection, logical, generalization or utility component (§7.3.5).
2. `transfer-business-logic` *(semantic)* — An operation of a transfer component models a business process of the domain (§7.3.5).
3. `transfer-inheritance` *(structural)* — A transfer component inherits from a component that is not a transfer component (§7.3.5).

11.3.8 Group 8 — XF start-point element

The rules of this group verify the normative prescriptions on the **start-point element** `XF` and the orchestration of the aggregate lifecycle of the artifact that its declaration

represents: the presence of the `init()` and `terminate()` operations and the canonicity of their bodies — a sequence of invocations `R.init(); B.init(); A.init();` in ascending order and `A.terminate(); B.terminate(); R.terminate();` in reverse order. The four rules share the `xf-` prefix because they verify the same code element: the `XF` start-point element. The exclusivity of lifecycle orchestration over the logical components — which components may instantiate them and invoke `init()`/`terminate()` on them — is verified per-component in Group 9.

Rules of the XF element

1. `xf-init-missing` *(structural)* — The `XF` element does not declare an invocable static `init()` operation (§8.3).
2. `xf-terminate-missing` *(structural)* — The `XF` element does not declare an invocable static `terminate()` operation (§8.3).
3. `xf-init-mismatch` *(structural)* — The body of the `init()` operation of the `XF` element contains statements other than the invocations `R.init()`, `B.init()`, `A.init()` in that order (§8.3).
4. `xf-terminate-mismatch` *(structural)* — The body of the `terminate()` operation of the `XF` element contains statements other than the invocations `A.terminate()`, `B.terminate()`, `R.terminate()` in that reverse order (§8.3).

11.3.9 Group 9 — Exclusivity of lifecycle orchestration

The rules of this group verify the exclusivity of the lifecycle operations in three senses parallel to the orchestration hierarchy of the artifact: the creation of instances of logical components and the invocation of `init()`/`terminate()` on logical components are reserved to the injection components; the invocation of `init()`/`terminate()` on the injections is reserved to the `XF` element; and the invocation of `init()`/`terminate()` on `XF` itself is reserved to `XF` itself or to the external execution start point of the artifact. They quantify over every component of the artifact and report as witness of the violation the invoking component, not the invoked one.

Rules on logical components

1. `lifecycle-logic-instantiation` *(structural)* — A component of the artifact other than the injection components instantiates a logical component by means of `new` (§7.3.3).
2. `lifecycle-logic-init` *(structural)* — A component of the artifact other than the injection component of the layer of the logical component invokes `init()` on a logical component (§7.3.3, §8.2).
3. `lifecycle-logic-terminate` *(structural)* — A component of the artifact other than the injection component of the layer of the logical component invokes `terminate()` on a logical component (§8.2).

Rules on injection components

1. `lifecycle-injection-init` *(structural)* — A component of the artifact other than the `XF` element invokes `init()` on an injection component (§7.3.3, §8.3).

2. `lifecycle-injection-terminate` *(structural)* — A component of the artifact other than the `XF` element invokes `terminate()` on an injection component (§8.3).

Rules on the `XF` element

1. `lifecycle-xf-init` *(structural)* — A component of the artifact other than `XF` itself invokes `init()` on the `XF` element (§8.3).
2. `lifecycle-xf-terminate` *(structural)* — A component of the artifact other than `XF` itself invokes `terminate()` on the `XF` element (§8.3).

11.3.10 Catalog closure

The catalog enumerates **71 rules** distributed across **9 thematic groups**. Non-compliance with any of them constitutes a violation and affects the conformance level λ of the artifact. The figures aggregated by domain (**fld**, **cmp**) and by verifiability (**str**, **sem**) are derivable from the master table and are reported, where appropriate, by the analysis tools.

The catalog is closed in version 1.0 of the model. Future versions may extend it by means of the addition of new rules in accordance with the normative procedure of evolution, provided that each new rule derives from a prescription established in clauses 6 to 10 and does not introduce requirements not contemplated in the normative body. The removal of an existing rule requires explicit normative justification and is recorded in Annex F (Revision history).

11.4 Determination of the conformance level

The determination of the conformance level of an artifact is an algorithmic procedure that produces a unique and deterministic result from two inputs: the artifact to be evaluated and the catalog of rules of §11.3. The procedure does not require interpretation — given the same inputs, it always produces the same result regardless of who executes it or of the tool that implements it.

11.4.1 Determination algorithm

The algorithm for the determination of the conformance level is executed in four sequential stages. Each stage produces a result that conditions the execution of the subsequent stages. The present clause prescribes the behavior that every static analysis tool conforming with the XF model shall implement.

Stage 1 — Inventory of verifiable elements The analyzer traverses the file system of the artifact and constructs the complete inventory of verifiable elements (§3.1.22): the structural elements (folders and subfolders under `/src`) and the code elements — classes, modules or files of cohesive functions, including the `XF` start-point element —, classifying the components by their location in the folder structure and by their nomenclature. The inventory retains the components for which the classification is not defined — unclassified components — for the purposes of the next stage.

Stage 2 — Verification of the totality condition The analyzer verifies whether the artifact satisfies the code totality condition defined in §11.1.3: every component of the artifact is classified in the matrix $L \times T$. The `XF` element, when declared, falls outside the totality predicate by not being a component.

- If the condition is satisfied, the algorithm continues at Stage 3.
- If it is not satisfied and there exists at least one classified component, the artifact is of **level 1** (partially conformant) and the algorithm terminates.
- If it is not satisfied and no classified component exists, the artifact is of **level 0** (non-conformant) and the algorithm terminates.

Stage 3 — Evaluation of the catalog of rules The analyzer evaluates the catalog of rules of §11.3 over the inventory of verifiable elements, respecting the applicability conditions of §11.4.2. Each rule-element pair whose rule is applicable and is not satisfied is registered as a violation.

Stage 4 — Determination of the level The analyzer classifies the registered violations according to their verifiability (§11.1.4) and applies the conditions of §11.2:

- If there exists at least one **structural** violation, the artifact is of **level 2** (imperfectly conformant).
- If no structural violation exists but there exists at least one **semantic** violation, the artifact is of **level 3** (structurally conformant).
- If no violation exists, the artifact is of **level 4** (perfectly conformant).

The final result of the algorithm is $\Lambda(\mathfrak{A}) \in \{0, 1, 2, 3, 4\}$ which completely characterizes the conformance state of the artifact with respect to the XF model. The formal properties of the algorithm are its determinism and its completeness.

The decision tree of Figure 11 summarizes the four stages in operative form.

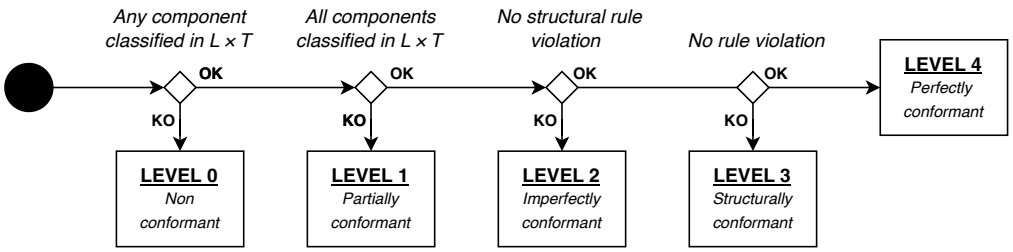

Figure 11. Decision tree that materializes the algorithm for the determination of the conformance level $\Lambda(\mathfrak{A}) \in \{0, 1, 2, 3, 4\}$. Each decision node evaluates a successive condition — presence of at least one classified component, satisfaction of the totality condition, absence of structural violations and absence of semantic violations — and each terminal branch assigns the corresponding level. The sequence is the deterministic operationalization of the algorithm for the determination of the conformance level.

11.4.2 Rule applicability

A rule is applicable to an artifact when there exists at least one verifiable element of the artifact on which the rule can be evaluated in a meaningful manner. A non-

applicable rule does not contribute to the set of violations: a rule whose applicability conditions are not satisfied generates no violations. For example, an artifact without generalization components produces no `general-mismatch-*` violations because no element exists on which to evaluate them; the corresponding rules are simply not applicable to the concrete artifact.

The present clause of the normative body prescribes the operational behavior that every static analysis tool conforming with the XF model shall implement.

12 Ontology of the architecture

The ontology of the XF architecture establishes the formal vocabulary of the model — the set of terms with a precise and normative definition that allows development teams, analysts, reviewers, and tools to communicate about the architecture of an XF artifact with the same understanding, independently of the technology used. This vocabulary is the materialization of the central objective of the model: to eliminate the terminological ambiguity that characterizes the current software development ecosystem.

The clause is organized into two internal clauses. Clause §12.1 presents the conceptual map of the model — a visual representation of the relationships between the fundamental concepts. Clause §12.2 presents the dictionary of terms — the precise definition of each term of the model, organized according to the expository order of the document, with local thematic grouping where it favors readability.

12.1 Conceptual map

The conceptual map in Figure 12 brings together in a single graph the fundamental concepts of the XF model and the semantic relationships that link them, from the artifact and the process it formalizes to the statement and the state that an operation transforms. It serves as a visual index to the dictionary of terms in §12.2, where each concept receives its precise normative definition.

12.2 Dictionary of terms

Concepts of the formal process and of the industry

Formal process Sequence of activities with precise semantics, well-defined inputs and outputs, and verifiable transition rules, which can be described with sufficient precision to be automated by software. Every formal process admits a canonical decomposition into three stages with distinct formal status: *processing* as the structurally necessary stage, and *interaction* and *access* as structurally optional but exhaustive and mutually exclusive channels of communication with the environment.

Reference: §5.1, §5.2.

Interaction stage First invariant stage of every formal process. It comprises the reception of the order to execute the process and the verification that the preconditions necessary for its start are satisfied. In the XF model, this stage corresponds one-to-one with the Interaction Layer of the artifact.

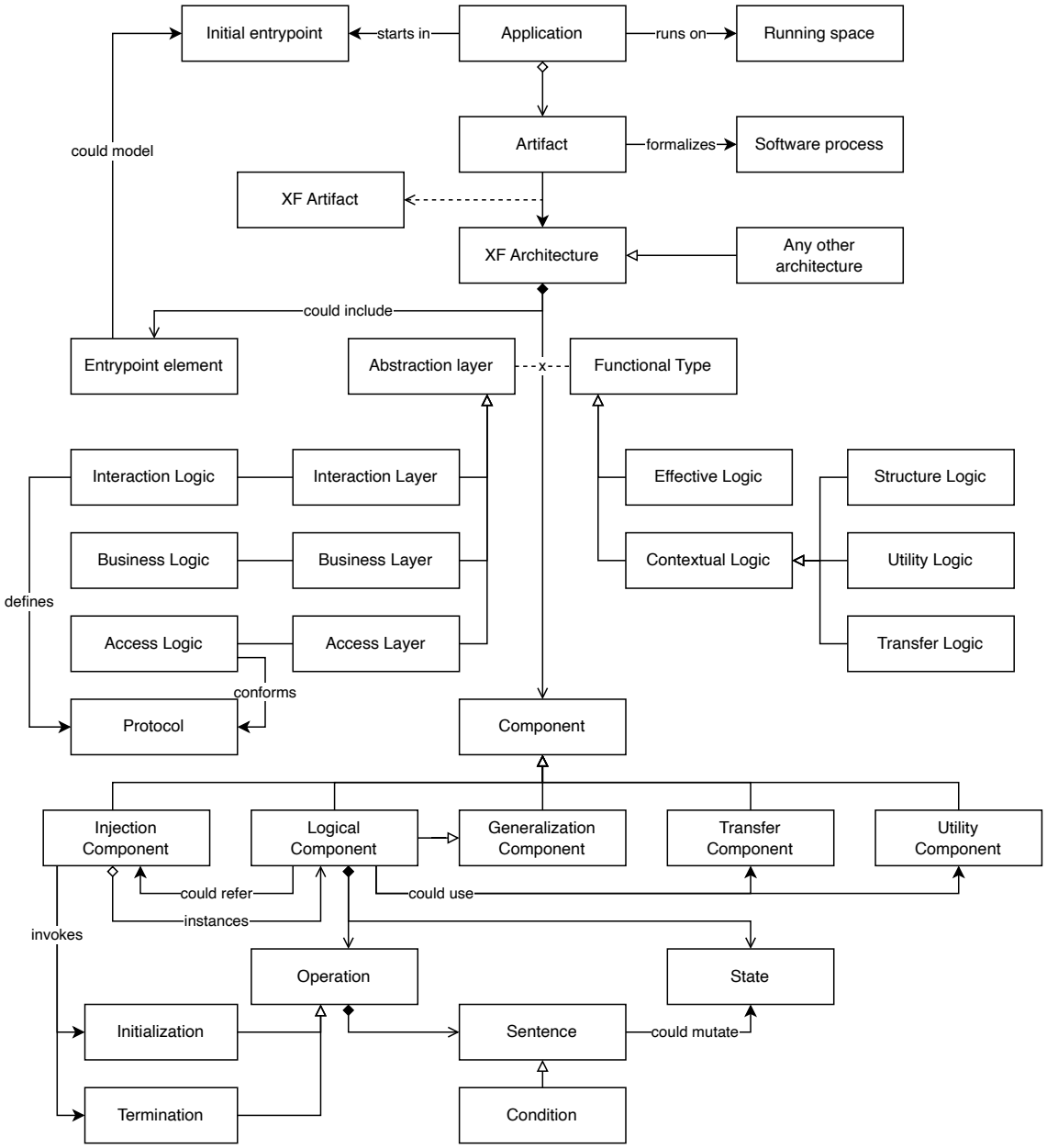

Figure 12. Conceptual map of the XF model: its fundamental concepts — artifact, architecture, layers, types, components, operations, statements, and state — and the semantic relationships that link them.

Reference: §5.2.

Processing stage Second invariant stage of every formal process. It comprises the ordered and deterministic execution of the operations that transform the inputs into the expected result. In the XF model, this stage corresponds one-to-one with the Business Layer of the artifact.

Reference: §5.2.

Access stage Third invariant stage of every formal process. It comprises the delegation of part of the processing to systems external to the process, through a

defined communication protocol. In the XF model, this stage corresponds one-to-one with the Access Layer of the artifact.

Reference: §5.2.

Heterotechnical synonymy Descriptive phenomenon of the problem that the XF model addresses: the use of different names to designate the same architectural concept in different development frameworks — for example, `Service` in one technology and `Manager` in another for the same type of component. Heterotechnical synonymy makes architectural communication between teams trained in different technological traditions impossible and constitutes one of the roots of the fragmentation of the ecosystem.

Reference: §5.4.

Interframework homonymy Descriptive phenomenon of the problem that the XF model addresses: the use of the same name to designate different architectural concepts in different development frameworks — for example, `Service` with the meaning of a business-logic component in one technology and the meaning of a systemic-exposure component in another. Interframework homonymy introduces ambiguity into architectural discussions between teams and constitutes one of the roots of the fragmentation of the ecosystem.

Reference: §5.4.

Semantic coupling to the development framework Descriptive phenomenon of the problem that the XF model addresses: the structural dependency of the artifact's architecture on the conventions, nomenclatures, and mechanisms of the development framework used. An artifact with semantic coupling to the development framework cannot be ported to another technology without complete architectural reconceptualization, and its review requires specific knowledge of the framework's conventions. The XF model is built precisely to remove this coupling, keeping the architecture as a stable layer above the framework.

Reference: §5.4, §6.2.3.

Implicit convergence Descriptive phenomenon observed by the XF model: the spontaneous tendency of the industry to converge toward the same set of component categories and the same layer stratification, regardless of the development framework used. Implicit convergence is the empirical evidence that justifies the viability of an agnostic architectural reference model: if the industry does in fact converge, the formalization of the convergence pattern is not only possible but economically efficient. The XF model makes this convergence explicit and normative.

Reference: §5.4.

Artifact, layers, and the $L \times T$ matrix

Artifact In the XF model, an artifact is the unit of software that automates a complete formal process. Every artifact has exactly three abstraction layers — Access,

Business, and Interaction — and a single execution space. The term "artifact" in the XF model is more precise than the general-use term "application" or "system" — an artifact is a self-contained architectural unit that can be instantiated, initialized, and terminated independently.

Reference: §6.1.

Application Executable unit resulting from loading one or more artifacts into a single execution space, typically a single operating-system process. The term *application* designates the executable functional unit; the term *artifact* designates the self-contained structural unit that the application loads and composes. An application may be composed of a single XF artifact or integrate several — own or external — in accordance with the mechanisms prescribed in §10.2.

Reference: §10.2.

Architectural reference model Set of design decisions applicable to a domain of recurring problems that, once adopted, establish constraints on the architecture of the systems that implement them and favor certain qualities of the resulting system. The XF model is an architectural reference model in this sense — it is not a design pattern or an architectural style, but a conceptual abstraction that establishes a common vocabulary, a universal organizational structure, and a set of guiding principles applicable to any software artifact.

Reference: §6.

Abstraction layer Horizontal division of the artifact that groups the components implementing the same stage of the formal process that the artifact automates. Each layer has exclusive responsibilities, well-defined boundaries, and unidirectional dependencies with respect to the adjacent layers. The XF model defines exactly three abstraction layers — Access, Business, and Interaction — which stratify the internal structure of the artifact in a dimension orthogonal to the seven layers of the OSI model (ISO/IEC 7498-1).

Reference: §6.1, §6.2.2, §5.5.

Access Layer Abstraction layer of the XF model corresponding to the access stage of the formal process. It contains all the components that encapsulate communication with systems external to the artifact. It is the only layer of the XF model that has direct contact with the Presentation Layer of the OSI model (Level 6). Its responsibility is protocol abstraction, data normalization, and the propagation of communication errors.

Reference: §7.2.1.

Business Layer Abstraction layer of the XF model corresponding to the processing stage of the formal process. It contains all the components that implement the domain logic of the artifact. It is the layer that defines the functional identity of the artifact — what it does, what concepts it manages, and what operations on those concepts are possible. It is the only layer of the model that may maintain domain state.

Reference: §7.2.2.

Interaction Layer Abstraction layer of the XF model corresponding to the interaction stage of the formal process. It contains all the components that define the entry points to the artifact — how users and external systems can invoke its business operations. It exists in every XF artifact, although it may be empty of logical components in artifacts without external interaction.

Reference: §7.2.3.

Component Atomic unit of classification of the XF model. Any unit of code with its own identity, a unique name within the artifact, and a classification defined in the $L \times T$ matrix. In object-oriented paradigms, an XF component corresponds to a class. In functional paradigms, to a module. Every component belongs to exactly one layer and exactly one type.

Reference: §7.1, §11.2.1.

Functional type Classification of a component according to the function it performs within its layer. The XF model defines exactly five functional types — Logical, Generalization, Injection, Utility, and Transfer — which form a closed and exhaustive classification system.

Reference: §6.2.4, §7.3.

$L \times T$ matrix Two-dimensional classification structure of the XF model formed by the Cartesian product of the set of layers L = {Access, Business, Interaction} and the set of functional types T = {Logical, Generalization, Injection, Utility, Transfer}. It produces fifteen possible classification categories. Every component of an XF artifact belongs to exactly one cell of the matrix.

Reference: §7.1.

Abstraction dimension First orthogonal dimension of the $L \times T$ matrix of the XF model. It stratifies the artifact into three layers — Access, Business, and Interaction — derived from the invariant stages of the formal process that the artifact automates. It is the dimension that orders the components according to their level of proximity to the boundaries of the artifact.

Reference: §6.1, §6.2.2.

Functional dimension Second orthogonal dimension of the $L \times T$ matrix of the XF model. It types the components of each layer into five functional categories — Logical, Generalization, Injection, Utility, and Transfer — with disjoint responsibilities. It is the dimension that orders the components according to the function they perform within their layer.

Reference: §6.2.4, §7.3.

Classification function ϕ Function that assigns to each component of the artifact its position in the $L \times T$ matrix. It is denoted ϕ: $\mathbb{C} \to L \times T$. The function ϕ is total over the set of components of the artifact — every component has exactly one classification; no component belongs to more than one category.

Reference: §6.2.4, §11.2.2.

Logical components and logic

Logical component Functional type that implements the effective logic of the layer to which it belongs. It is the main functional unit of each layer — the only type that may maintain state in the formal sense of the model and the only one whose instances are managed by the injection component. Canonical nomenclature: suffix `Repository` in the Access Layer, `Business` in the Business Layer, `Service` or `View` in the Interaction Layer.

Reference: §7.3.1.

Repository (logical component) Logical component of the Access Layer. It encapsulates communication with a concrete external system to which the artifact delegates part of its processing. It applies protocol abstraction and data normalization to the information that crosses the boundary of the artifact. Canonical nomenclature: suffix `Repository`. The XF model broadens the meaning of Fowler's *Repository* pattern [12] to cover every access to an external system — persistent, remote, messaging, hardware, or another environment —, not only persistence access (§7.4).

Reference: §7.3.1, §7.2.1.

Business (logical component) Logical component of the Business Layer. It implements the effective logic of a domain concept that the artifact automates — its business operations, its business conditions, and the domain state associated with the concept —. Canonical nomenclature: suffix `Business`.

Reference: §7.3.1, §7.2.2.

Service (logical component) Logical component of the Interaction Layer that defines a point of systemic interaction — a mechanism through which an external system can invoke operations of the artifact by means of a defined communication protocol. Canonical nomenclature: suffix `Service`.

Reference: §7.3.1, §7.2.3.

View (logical component) Logical component of the Interaction Layer that defines a point of graphical interaction — a visual element through which a human user can invoke operations of the artifact or receive information from it. It combines composition, style, and presentation logic. Canonical nomenclature: suffix `View`.

Reference: §7.3.1, §7.2.3.

Effective logic Minimal set of operations that justify the existence of a logical component in the artifact — those that cannot be abstracted into any other type of component without losing the specificity of the concept that the component models. Effective logic is inherent to the concept that the component models and cannot exist in any other component without losing its identity. It is distinguished from contextual logic in that it is irreducible to the concrete component.

Reference: §7.3.1.

Contextual logic Auxiliary logic that supports the achievement of the effective logic of a logical component, but that may be shared with other components, reused, or expressed generically. Contextual logic belongs to the generalization, utility, or transfer components according to its nature — not to the logical component. It is formally subdivided into three species according to the type of component that hosts it: **structural logic** in generalization components, **utility logic** in utility components, and **transfer logic** in transfer components.

Reference: §7.3.1.

Structural logic Species of contextual logic encapsulated in a generalization component, which abstracts a structural behavior pattern common to two or more logical components of the **same layer** and transmits it to them by inheritance. It is **parametric with respect to the domain**: its logic applies regardless of the concrete concept that each logical component inheriting it models. The presence in a generalization component of logic tied to a concrete concept of the artifact's domain indicates that it has ceased to be structural logic and shall be moved to the logical component to which it belongs.

Reference: §7.3.2.

Utility logic Species of contextual logic encapsulated in a utility component. It provides auxiliary operations — pure functions, stateless, without side effects — that can be invoked by any component of the layer to which the utility component belongs. Unlike structural logic, it is not transmitted by inheritance but by direct invocation.

Reference: §7.3.1, §7.3.4.

Transfer logic Species of contextual logic encapsulated in a transfer component. It comprises exclusively the structural representation of the concept that the component models — attribute declaration, type constraints, syntactic validations — and optionally transformation operations between equivalent representations of the same concept. It does not include behavioral logic over the domain.

Reference: §7.3.1, §7.3.5.

State Set of data of a logical component that have direct correspondence with information of the business domain and that determine the observable behavior of the component in response to the operations that invoke it. State does not include internal operational attributes of the component — connection references, control

flags, retry counters. Only logical components may maintain state in the formal sense of the model.

Reference: §7.3.1.

Operation Subset of the effective logic of a logical component that executes atomically in response to an event or invocation, producing a transformation of the component's state, a result for the invoker, or both. Each operation of a logical component corresponds to exactly one well-defined action on the concept that the component models.

Reference: §7.3.1.

Statement Each of the logical steps that make up an operation. Statements execute in order, and their set constitutes the complete implementation of the action that the operation models. A statement may be a data transformation, an invocation of a lower-layer component, the evaluation of a condition, or a mutation of the component's state.

Reference: §7.3.1.

Condition Specific statement that evaluates the component's state or the input parameters of an operation before executing the effective logic. Conditions constitute the execution preconditions of the operation — if a condition is not satisfied, the operation typically propagates an exception to the invoker, although other forms of handling — default value, retry — are admissible when the semantics of the process justify it. The XF model recommends that conditions be evaluated at the beginning of the operation, ideally before any other statement.

Reference: §7.3.1.

Generalization, injection, utility, and transfer components

Generalization component Functional type that abstracts structural behavior common to two or more logical components of the same layer. It does not implement effective logic of any concrete domain concept — its logic is parametric with respect to the domain and has no utility independent of the logical components that inherit it. Its scope is strictly limited to the layer in which it is defined.

Reference: §7.3.2.

Singleton Gathering Design pattern implemented by the injection components of the XF model. An extension of the Singleton pattern that aggregates at a single point the unique instances of a set of related components, guaranteeing the instance uniqueness of each logical component of the layer and providing a named, stable point of access to each of those instances.

Reference: §7.3.3.

Injection component Functional type that implements the Singleton Gathering pattern for the logical components of its layer. It is the only point of access to the logical components of its layer from any other component of the artifact. It is a non-instantiable class with immutable public static references to the unique instances of the logical components of its layer. The XF model defines exactly three injection components — `R` for the Access Layer, `B` for the Business Layer, `A` for the Interaction Layer.

Reference: §7.3.3.

`R` Canonical name of the injection component of the Access Layer. It aggregates the unique instances of all the Repositories of the artifact and manages their lifecycle. It is the only point of access to the artifact's external data from the Business Layer and from the Access Layer itself.

Reference: §7.3.3.

`B` Canonical name of the injection component of the Business Layer. It aggregates the unique instances of all the Business components of the artifact and manages their lifecycle. It is the only point of access to the artifact's domain logic from the Interaction Layer and from the Business Layer itself.

Reference: §7.3.3.

`A` Canonical name of the injection component of the Interaction Layer. It aggregates the unique instances of all the services and views of the artifact and manages their lifecycle. It is the only point of access for the consumption of the artifact by other Applications and human Users.

Reference: §7.3.3.

Slot (injection) Immutable public static reference, declared in an injection component, that points to the unique instance of a logical component of the corresponding layer. It is the mechanism prescribed by the model for the materialization of the Singleton Gathering pattern: the set of slots of an injection component constitutes the exhaustive enumeration of the accessible logical components of its layer.

Reference: §7.3.3.

Post-initialization immutability Structural property of the injection component that prescribes that the composition of slots — the identity of the referenced logical components — cannot be modified once the artifact has reached the INITIALIZED state. Post-initialization immutability is the basis on which instance uniqueness and the stability of references throughout the artifact's lifecycle rest.

Reference: §7.3.3, §8.2.

Utility component Functional type that provides auxiliary operations of scope local to its layer. Its operations are pure functions — stateless, without side effects, without dependencies on other components. It is neither instantiable nor inheritable.

Utility components over primitive types defined in the Access Layer constitute the only exception to the local scope — they may be referenced from any layer. Canonical nomenclature: suffix `Utils`.

Reference: §7.3.4.

Primitive type Primitive data type in the sense of ISO/IEC/IEEE 24765:2017 [20]: a type from which other types are built, present as a base type in most programming languages. In the XF model, primitive types enter the artifact through the Access Layer — the boundary with the Presentation Layer of the OSI model (Level 6) —, are prior to any semantic transformation, and constitute the raw material on which all layers operate. Exchange formats of the operating system or the execution environment whose structure is determined by the platform are treated as primitive types for the purposes of the model.

Reference: §7.3.4, §7.2.1.

Transfer component Functional type that models the shape of the information at a concrete point in the artifact's processing flow. It may declare self-contained operations over its own data, but it does not model business processes or access other components; its attributes describe the concept of the artifact's domain that it models: a business concept (`User`, `Order`), a concept of access to an external system (`Query`, `HttpRequest`), or a concept of interaction with a consumer (`Button`, `MouseEvent`). It is the only type of component that may cross layer boundaries, subject to the restrictions of directionality and structural non-modification. Canonical nomenclature: name of the concept without suffix; suffix `Exception` as a recommended convention for transfers that are typically transmitted as language exceptions.

Reference: §7.3.5.

Inherited transfer Data type defined by a development framework, operating-system kit, or integrated library that a resulting XF component receives, returns, or propagates without canonizing it into a transfer of the artifact's own. For the purposes of the model, every inherited transfer that appears in an XF operation signature is a transfer component ($\mathbb{C}_T$) classifiable in the $L \times T$ matrix, even if its identifier does not follow the canonical nomenclature of the model. It allows $\lambda = 3$ to be reached even when the code handles external identifiers.

Reference: §10.2.1.

Structure In the XF model, a grouping of primitive types or other structures that models a concept of the artifact's domain. Synonym of transfer component in its structural sense. The XF model unifies under this single term the proliferation of traditional semantic categories — `DTO`, `Entity`, `Model`, `POJO`, `Record`, `VO`, `Schema` — that designated the same type of element with different names depending on the context of use.

Reference: §9.2.

Structure, nomenclature, and guiding principles

Canonical folder structure Physical representation of the $L \times T$ matrix in the project's file system. It prescribes the organization of folders and subfolders that every XF artifact shall implement, so that the architecture is visible and inspectable without the need for additional tools. Its presence and correct organization is a necessary condition for the conformance of the artifact.

Reference: §7.4.

Artifact root First-level folder of the file system that contains the artifact. It is the anchor on which the canonical folder structure of the model operates. Its name is not prescribed by the model. It shall contain exclusively the `/src` subfolder and the execution start point, together with any technical element of the development framework or the language used.

Reference: §7.4.

Canonical nomenclature Set of names and naming patterns prescribed by the XF model for each category of component. The canonical nomenclature is part of the model and shall be respected independently of the naming conventions of the language or development framework used. Its consistency is a necessary condition for semantic interoperability between teams.

Reference: §7.1.

Principle of technology agnosticism Guiding principle of the XF model that establishes that the model imposes no restrictions on the programming language, the development framework, the programming paradigm, the execution platform, or the business domain of the artifact. Every artifact that automates a formal process can be organized according to the XF model independently of any technological consideration.

Reference: §6.2.1.

Principle of layer isolation Guiding principle of the XF model that establishes that every component of an XF artifact depends exclusively on components classified in its own layer or in layers of a lower level of abstraction. Dependencies between layers are strictly unidirectional in the downward direction. Transfer components are governed by this same directionality; the unification of structures between layers (§9.3) operates within the isolation principle, not as an exception to it, subject to the restrictions of directionality and non-modification.

Reference: §6.2.2.

Principle of precedence of the architecture over the tool Guiding principle of the XF model that establishes that the architectural structure of an XF artifact derives from the properties of the formal process it automates, and not from the conventions of the development framework, the programming language, or the execution platform used. The development framework is the instrument through which the architecture is implemented — not its origin or its determinant.

Reference: §6.2.3.

Principle of closed and exhaustive typing Guiding principle of the XF model that establishes that the component classification system is simultaneously closed — the set of recognized types is finite and non-extensible — and exhaustive — every component of any software artifact can be classified into exactly one of the fifteen categories of the $L \times T$ matrix.

Reference: §6.2.4.

Stratification, lifecycle, and data flows

Dependency In the XF model, any relationship between two components A and B in which A needs to know the definition or behavior of B in order to be compiled, instantiated, or executed. It encompasses operation invocation, inheritance, declaration of attributes of type B, and reference to the module or namespace of B. Every dependency is subject to the principle of layer isolation.

Reference: §6.2.2.

Inheritance Dependency relationship between a logical component and a generalization component of the same layer, through which the logical component acquires the structural behavior abstracted by the generalization component. In the XF model, inheritance is strictly restricted to components of the same layer — no component may inherit from a component of a different layer.

Reference: §7.3.2.

Instantiation Creation of an object in memory through the constructor of a component. In the XF model, the instantiation of logical components is an operation exclusive to the injection component of its layer — it occurs at class-load time through the declaration of the injection component's static references. Instantiation is distinguished from initialization — instantiation creates the object, initialization prepares it to operate.

Reference: §7.3.3, §8.3.

Initialization Execution of the start-up logic of a logical component — establishment of connections with external systems, loading of configuration, start-up of periodic tasks, establishment of the initial state. Initialization is implemented in the `init()` operation of the logical component and is invoked by the injection component of its layer in the order that the dependencies between components require. It is distinguished from instantiation — initialization prepares the component to operate, it does not create it.

Reference: §7.3.3, §8.3.

Termination Execution of the shutdown logic of a logical component — release of resources acquired during initialization, closing of connections, persistence of the state that must be preserved between executions. Termination is implemented in

the `terminate()` operation of the logical component and is invoked by the injection component in the reverse order of initialization.

Reference: §7.3.3, §8.3.

Protocol In the XF model, the set of rules and formats that govern the communication between two software artifacts over a communication channel. The management of protocols is a structural responsibility of the two boundary layers of the artifact, in complementary roles: the **Interaction Layer** *defines and exposes* the protocol through which external consumers invoke the artifact; the **Access Layer** *consumes* the protocols published by the external systems to which the artifact delegates part of its execution. No component of the Business Layer may know the details of any protocol. In the XF model the term has a broader meaning than in OSI terminology: it includes both network protocols and local-access protocols — system files, data buses, command console.

Reference: §7.2.1 y §7.2.3.

Protocol abstraction Property of the logical components of the Access Layer by which the details of the communication protocol with external systems are confined within the component and are invisible to the upper layers. Protocol abstraction is the property that makes the principle of technology agnosticism possible at the data level — a change of protocol is confined to the corresponding repository without requiring modifications in the Business Layer or the Interaction Layer.

Reference: §7.2.1.

Data normalization Syntactic transformation applied by the components of the Access Layer to the data received from external systems — it converts the protocol format into structured transfer components interpretable by the Business Layer. Normalization is exclusively syntactic — it does not apply semantic transformations to the data.

Reference: §7.2.1.

Business ontology Conceptual model that defines the structured set of key concepts handled by the artifact, their relationships, the operations permitted on them, and the rules that govern them. The business ontology is the formal description of the domain that the artifact automates and is represented in the XF artifact by the set of logical components of the Business Layer.

Reference: §7.2.2.

Business operation Subset of the effective logic of a business logical component that executes atomically in response to an event or invocation, producing a transformation of the component's state, a result for the invoker, or both.

Reference: §7.2.2.

Business condition Statement of a business operation that evaluates the component's state or the input parameters before executing the effective logic. It constitutes the execution precondition of the operation. If it is not satisfied, the operation propagates a business exception.

Reference: §7.2.2.

Business exception Transfer component that models an error condition specific to the artifact's domain — an unsatisfied business condition or an unresolved Access Layer dependency — and that is typically transmitted as a language exception. By model convention (§9.5, not a structural requirement) it carries the suffix `Exception`. The native exceptions of the implementation language (`Error`, `Exception`, `BaseException`, etc.) are, for the purposes of the XF axiomatization, equivalent transfer components (§10.2.1, inherited transfers), and do not need to be wrapped in a component of the artifact's own.

Reference: §7.2.2.

Communication exception Transfer component originated in the Access Layer that models an error condition in the communication with an external system — network failure, protocol error, system unavailability —. It is typically transmitted as a language exception and shall be intercepted by the Business Layer, which decides whether to propagate it, transform it into a business exception, or attempt a recovery within the domain logic. Its propagation without intermediation up to the Interaction Layer is not admissible under the principle of layer isolation (§6.2.2).

Reference: §7.2.2, §9.5.

Entry point In the XF model, any mechanism through which a consumer of the artifact — human user or external system — can invoke the business operations that the artifact implements. Entry points are the exclusive responsibility of the Interaction Layer. In the XF model the term has a more precise meaning than in general terminology — it does not designate the start of execution of the process but the mechanisms of external access to the artifact.

Reference: §7.2.3.

Presentation logic Set of algorithms and checks that determine how a view must behave in response to the data it receives and the events that occur on it — what must be displayed, when, and in what state. It does not include business rules or access to external systems.

Reference: §7.2.3.

Presentation state Set of data of a logical component of the Interaction Layer that determines its visual or response behavior in the face of the events it receives. It has no direct correspondence with information of the business domain — it models the dynamic context of the interaction.

Reference: §7.2.3.

Execution start point Fragment of code that the execution environment invokes to start the artifact's process — `main`, `App.main()`, `index.ts`, or equivalent depending on the language. It is the first developer code that receives control from the environment; its sole responsibility is to delegate the XF initialization and, at the end, the termination. It is not a component classifiable in the $L \times T$ matrix and resides outside the root of the artifact. It shall not be confused with the entry point (responsibility of the Interaction Layer) or with the `XF` start-point element (optional construct internal to the artifact that orchestrates the centralized initialization).

Reference: §8.1.

XF start-point element Optional code element of the artifact named `XF` that encapsulates the initialization and termination operations of the three injection components in a single centralized control point. *It is not a component*: it is not classified in the $L \times T$ matrix and falls outside the domain of the classification function ϕ, but it is a verifiable element of type *code element*, on which the rules of Group 8 of the catalog operate (§11.3). Its presence in an artifact is an explicit declaration that that artifact implements the XF model.

Reference: §8.3.

Artifact lifecycle Sequence of states through which an XF artifact transits from the moment the execution environment yields control to the developer's code until the artifact's process ends. The XF model formalizes three states — DEFINED, INITIALIZED, and TERMINATED — and five transitions between them.

Reference: §8.2.

defined state Initial state of the XF artifact. The components exist as code structures but are not initialized. No logical component may be accessed. The artifact enters this state at the moment the execution environment yields control to the developer's code.

Reference: §8.2.

initialized state Active state of the XF artifact. All components are initialized, their dependencies are resolved, and the artifact processes interactions. The component structure is immutable during this state. The artifact enters this state when `XF.init()` completes successfully.

Reference: §8.2.

terminated state Final state of the XF artifact. The resources have been released and the process may end. It is an absorbing state — once reached, the artifact cannot return to the INITIALIZED state without a new start-up.

Reference: §8.2.

Initialization order Sequence in which the logical components of an XF artifact are initialized. It is determined by the dependency relationships between components

and shall be a topological ordering of the dependency graph that satisfies two constraints — the layer constraint and the intra-layer dependency constraint.

Reference: §8.3.

Canonical initialization sequence Order prescribed by the XF model for invoking the initialization operations of the logical components of an artifact. It is determined as a topological ordering of the dependency graph between components constrained by the principle of layer isolation: the components of the Access Layer are initialized before those of the Business Layer, and these before those of the Interaction Layer. Within each layer, the order satisfies the intra-layer dependencies.

Reference: §8.3.

Canonical termination sequence Order prescribed by the XF model for invoking the termination operations of the logical components of an artifact. It is the inverse of the canonical initialization sequence: the components of the Interaction Layer are terminated before those of the Business Layer, and these before those of the Access Layer.

Reference: §8.3.

Circular dependency Cycle in the dependency graph between components that would make it impossible to determine a valid initialization order. Inter-layer circular dependencies do not arise in a well-formed XF artifact as a consequence of the principle of layer isolation (§6.2.2, operationalized by Group 2 of the catalog): inter-layer acyclicity is a property derived from the model, not an independent provision.

Reference: §8.3.

Data flow Mechanism for transmitting information between the components of an XF artifact during its execution. The XF model recognizes two types of flow according to their direction — upward or downward — and formalizes the communication channel as a single concept, regardless of the syntactic construct that the language provides for delivering the datum to the invoker.

Reference: §9.1.

Downward flow Data flow that goes from layers of higher abstraction toward layers of lower abstraction — the Interaction Layer invokes the Business Layer, which invokes the Access Layer. It is the invocation flow prescribed by the principle of layer isolation.

Reference: §9.1.

Upward flow Data flow that goes from layers of lower abstraction toward layers of higher abstraction. In the XF model, upward flow never uses direct upward invocation; it is typically carried out through the observer pattern.

Reference: §9.1.

Observer pattern Characteristic mechanism — recommended, not exclusive — for implementing upward flow between layers. A component of an upper layer registers as an observer of a component of a lower layer; the lower component notifies the observer of the occurrence of relevant events without knowing its concrete identity. The observer receives the notification through an interface defined by the upper-layer component. The observer pattern is the usual realization of upward communication; what the model prohibits, by the isolation principle, is the upward structural dependency and, with it, the direct upward invocation.

Reference: §9.1.

Structural homogeneity of transfers Normative property whereby every transfer that circulates through the communication channel of an operation belongs to the same single domain $\mathbb{C}_T$, regardless of the purpose of the datum it carries — expected result or anomalous condition — and of the syntactic construct that the language uses to deliver it. The distinction of purpose is exclusively one of implementation: it does not constitute disjoint classes of transfer, nor does it alter the classification in $L \times T$ or the rules that govern the transfer component.

Reference: §9.5.

Communication channel Medium through which a component delivers information to its invoker during the execution of an operation. The XF model formalizes the communication channel as a single concept: regardless of the syntactic construct that the language provides (`return`, `throw`, `Result<T, E>`, multi-value return, global variables such as `errno`), they all materialize the same medium for transmitting transfer components $C_T \in \mathbb{C}_T$. The distinction between the purpose of the transmitted datum — expected result or anomalous condition — is exclusively one of implementation; it does not generate structural asymmetry in the formal domain, and the rules of directionality and structural transformation apply indistinctly regardless of the construct chosen by the implementer.

Reference: §9.5.

Unification of data structures Emergent capability of the XF model: the same transfer component may be shared by all the layers that work with the same concept of the artifact's domain without the need to duplicate it per layer. It is a capability enabled by the exemption from strict layer isolation that the model grants to transfer components, subject to the directionality of transfers (§9.4) and to the structural transformation rule (§9.4). The model recommends unification as a good design practice, but does not impose it as a provision of the catalog.

Reference: §9.3.

Structural transformation rule Provision of the XF model that establishes that if a layer modifies the structural definition of a transfer component coming from a lower layer — by adding or removing attributes — it shall define a new transfer component in its own layer. The modification of the values of the attributes does not trigger this obligation.

Reference: §9.3.

Compatibility, conformance, and canonical suffixes

Execution space Computational context in which the components of an artifact operate — the memory, the system resources, and the process context that the artifact occupies during its execution. Two artifacts share the same execution space if their components can reference each other directly in memory without serialization or transmission through any external communication channel. The execution space is the criterion that determines the level of integration between artifacts.

Reference: §10.2.

Compatibility Property of the XF model that prescribes how an XF artifact integrates components from other artifacts: by recognizing the functional responsibility of the integrated component, assigning it a cell of its own $L \times T$ matrix, and applying the mechanism prescribed for that cell and that type of component. It is bidirectional — it applies indistinctly to the integration of external components into an XF artifact and to the integration of XF components into external artifacts — and non-invasive: no integrated artifact shall be modified in order to make its integration possible.

Reference: §10.1, §10.2.

Integrated component Component coming from an artifact different from the XF artifact that integrates it. By the principle of compatibility, the integrated component is classified in the $L \times T$ matrix of the integrating artifact according to its functional responsibility, regardless of the classification that the component has in its original artifact. The classification of the integrated component is a decision of the integrating artifact, not of the component.

Reference: §10.1, §10.2.

Reformulation Reorganization operation that makes conformant to the XF model, without altering the observable functional behavior, either a pre-existing artifact as a whole, or a functionality not normalized in XF that is integrated by reformulating it into a component of the artifact's own (§10.2.1). It operates on the file structure, the nomenclature, the declared dependencies, and the classification of the components; it does not constitute a rewrite of the code of the logical components. It is the prescribed path for a pre-existing artifact to reach conformance levels above zero.

Reference: §10.1, §10.2.1.

Technological mapping Document complementary to the normative specification of the model that prescribes how the concepts of the XF model are materialized in a concrete technology — language, development framework, or platform. Technological mappings resolve the specific tension between the model's provisions and the constraints of the technical environment. They are external to the normative specification of the model and do not extend the set of concepts of the model.

Reference: §6.2.3.

Application-level integration Form of integration between artifacts with disjoint execution spaces, which exchange information through an established communication protocol. It is independent of the internal architecture of each artifact.

Reference: §10.1.

Artifact-level integration Form of integration between artifacts that share an execution space, communicating directly through the interfaces of their components without the intermediation of any network protocol. The layer constraints of the XF model apply to the dependencies between artifacts integrated at this level with the same force as to internal dependencies.

Reference: §10.1.

XF library Software artifact that implements the XF model and that exposes, through its public interfaces, components of any type — generalization, transfer, utility and, when it models a concrete domain, also logical components through its injection components — for consumption by other XF artifacts. It is recommended that its interfaces be idempotent and independent of the implementation technology.

Reference: §10.3.

Conformance rule Verifiable condition that operationalizes a normative provision of the XF model into a form evaluable over a verifiable element of the artifact. Every rule has a canonical identifier in kebab-case format and is characterized by its **verifiability** — structural (decidable by deterministic static analysis) or semantic (requires interpretation of the functional responsibility) —. The breach of any rule of the catalog produces a violation that affects the conformance level of the artifact. The closed and exhaustive catalog is developed in §11.3.

Reference: §11.1, §11.3.

Canonical identifier String in kebab-case format that uniquely identifies a rule of the conformance catalog. It constitutes the reference key between the normative provision, the static analysis tools, and the conformance reports issued on an artifact. It is stable throughout the lifecycle of the model: a revision of the catalog may add rules with new identifiers but may not reassign an existing identifier to a different rule.

Reference: §11.1, §11.3.

Catalog (of rules) Closed and exhaustive set of conformance rules prescribed by the XF model, grouped by thematic groups. Its exposition in natural language is developed in §11.3. An artifact is conformant to the model if and only if it satisfies all the rules of the catalog applicable to its content.

Reference: §11.1, §11.3.

Structural verifiability Property of a rule of the conformance catalog indicating that the rule is decidable by deterministic static analysis over the file structure, the

nomenclature, and the declared dependencies of the artifact. Rules with structural verifiability can be implemented in a manner agnostic to the language and the development framework. Membership in this class is a declared property of each rule of the catalog.

Reference: §11.1, §11.3.

Semantic verifiability Property of a rule of the conformance catalog indicating that the rule requires interpretation of the content of the components and is not decidable by exclusively structural analysis. Its implementation in tools is not agnostic to the language.

Reference: §11.1, §11.3.

Verifiable element Any element of the artifact against which a conformance rule can be evaluated. The XF model recognizes two types of verifiable elements — code element and structural element — that are disjoint from each other.

Reference: §11.1.1.

Structural element Type of verifiable element corresponding to every folder or subfolder of the artifact's file system that forms part of its directory structure.

Reference: §11.1.1.

Violation Determination that a verifiable element of the artifact does not satisfy the condition prescribed by a rule of the conformance catalog. Every violation is associated with exactly one rule and exactly one verifiable element. The presence of any violation prevents the artifact from reaching level 4.

Reference: §11.1.2.

Code totality Condition that an artifact satisfies when every component of the artifact — except the `XF` element — is classified in the $L \times T$ matrix by means of the classification function ϕ. It is a necessary condition for the artifact to reach conformance level 2, 3, or 4.

Reference: §11.1.3.

Conformance function Λ Function that assigns to each artifact its conformance level with the XF model. It is denoted $\Lambda\colon \mathfrak{A} \to \{0, 1, 2, 3, 4\}$; its result for a concrete artifact is denoted $\lambda = \Lambda(\mathfrak{A})$.

Reference: §11.4.

Conformance level Discrete property of an artifact that determines the extent to which it satisfies the normative provisions of the XF model. It takes values in the set {0, 1, 2, 3, 4} — Non-conformant, Partially conformant, Imperfectly conformant, Structurally conformant, and Perfectly conformant. It is computable deterministically from the set of violations of the artifact and from the code-totality condition.

Reference: §11.2.

Well-formed artifact Artifact that structurally satisfies the minimal provisions of the XF model necessary for the formal properties of the model — layer directionality, totality of the classification function, uniqueness of the injection components — to be guaranteed by construction. It is the assumption under which the theorems of the model are derived, for example, the inter-layer acyclicity of the dependency graph (a property derived from the model). An artifact with conformance level 3 or 4 is necessarily well-formed.

Reference: §8.3, §11.4.

`Repository` Canonical suffix prescribed by the XF model for the name of the logical components of the Access Layer (§7.3.1). It also applies to the generalization components that abstract structural behavior of logical components of this layer. Examples: `DatabaseRepository`, `IdentityRepository`, `FileRepository`.

Reference: §7.3.1, §7.2.1.

`Business` Canonical suffix prescribed by the XF model for the name of the logical components of the Business Layer (§7.3.1). It also applies to the generalization components that abstract structural behavior of logical components of this layer. Examples: `UserBusiness`, `SessionBusiness`, `TemperatureBusiness`.

Reference: §7.3.1, §7.2.2.

`Service` Canonical suffix prescribed by the XF model for the name of the logical components of the Interaction Layer that implement a point of systemic interaction (§7.3.1). It also applies to the corresponding generalization components. Examples: `TemperatureService`, `AuthService`, `ScheduleService`.

Reference: §7.3.1, §7.2.3.

`View` Canonical suffix prescribed by the XF model for the name of the logical components of the Interaction Layer that implement a point of graphical interaction (§7.3.1). It also applies to the corresponding generalization components. Examples: `LoginView`, `MainView`, `ThermometerView`.

Reference: §7.3.1, §7.2.3.

`Utils` Canonical suffix prescribed by the XF model for the name of the utility components (§7.3.4) of any layer. It operationalizes the `utility-naming` rule of the conformance catalog. Examples: `StringUtils`, `DateUtils`, `TemperatureUtils`.

Reference: §7.3.4.

`Exception` Canonical suffix prescribed by the XF model for the name of the transfer components (§7.3.5) of the exception subtype. It applies to any layer according to the origin of the exception. Examples: `NetworkException`, `AuthenticationException`, `ValidationException`.

Reference: §7.3.5.

13 Discussion

The internal clauses are informative. Their purpose is to analyze the practical consequences of applying the XF model to industrial software projects, organized into five advantage vectors. The limitations of the model are stated in §14.1; the formal projection of the reference architectures consolidated in the literature is collected in §D.3.

13.1 Communication and common vocabulary

Every software artifact fulfills, in its deepest nature, a communicative function: it communicates to the team what the system does, to the incoming developer what has been built, and to the machine what must be executed. The terminological fragmentation documented in §5.4 — heterotechnical synonymy and interframework homonymy — systematically compromises that communicative function: teams with developers trained in distinct technological traditions cannot discuss architecture with the same vocabulary, even though the underlying structure is identical.

The XF model contributes three mutually reinforcing levels of improvement to architectural communication. The first is the **single, agnostic vocabulary**: each component of the artifact belongs to a concrete cell of the $L \times T$ matrix, identified by canonical nomenclature. A developer who says "a logical component of the Business Layer with a dependency on a repository" is describing with precision the same architectural reality regardless of the implementation technology, eliminating the mental translation that proprietary nomenclatures impose. Design discussion no longer carries the overhead of clarifying what each term refers to in each development framework.

The second is the **closed and exhaustive taxonomy**. Industry implicitly converges toward the same component categories because the formal processes it automates have the same invariant stages (§5.3); the model makes that convergence visible and closes it: the $L \times T$ matrix produces fifteen possible categories, five types across three layers, into which each component of the artifact is classified. The architectural review of code ceases to be an interpretive judgment and becomes a membership verification: is this component in the cell that its functional responsibility dictates?

The third is **documentary precision by construction**. The canonical folder structure (§7.4) and the prescribed nomenclature constitute architectural documentation embedded in the artifact itself, indistinguishable from the code it documents and, to the extent that the artifact satisfies the catalog, updated together with it by construction. A developer who opens an XF project recognizes its structure without needing to read external documentation: the layers are where they are prescribed, the types where they are prescribed, the components with the names that are prescribed. Traditional documentation — wikis, READMEs, diagrams — goes from being a parallel layer liable to fall out of date to a complementary layer that documents what is specific to the domain, not what the model already prescribes.

These three improvements have a practical corollary: work descriptions in agile

methodologies — Definition of Work and Definition of Done — gain in eloquence and lose in ambiguity when expressed in the vocabulary of the model. A task described as "implement an operation in a logical component of the Business Layer that verifies a precondition and delegates to a repository of the Access Layer" is unambiguously interpretable by any developer trained in the model, and measurable against the catalog of rules. The communication gap between analysis and implementation — a historical source of inconsistencies in software projects — is bounded by the shared formal vocabulary.

13.2 Reduction of operational costs

The operational consequences of terminological fragmentation documented in §1.1 translate into concrete costs: an increase in the context-switching time between projects with distinct technologies, an increase in the onboarding time of new developers, inconsistencies in the implementation of equivalent functionalities, and technical debt derived from the structural dependence of the artifact on the development framework. The XF model acts on each of those cost vectors. Table 14 summarizes the contrast between the traditional mechanisms and the mechanisms prescribed by the model for each vector, and identifies the property of the model that enables it; the following clauses develop each one in detail.

Table 14. Operational cost vectors of software development and their treatment under each paradigm. For each vector, the mechanism characteristic of the traditional approach is contrasted with the mechanism prescribed by the XF model, and the property of the model that enables the cost reduction is identified. The table is informative and operates as a reading map for the following clauses.

Cost vector	Traditional mechanism	XF mechanism	Property of the model that enables it
Context switching between technologies	Conventions specific to the development framework; architectural relearning per project.	Identical layer structure, types, nomenclature, and constraints regardless of the technology.	Principle of technological agnosticism (§6.2.1).
Onboarding of new developers	Time proportional to the singularity of the project's conventions.	Learning bounded to the domain logic; universal architectural conventions.	Canonical nomenclature and canonical folder structure (§7.3, §7.4).
Cost estimation and planning	Expert or analogy-based estimation, without a well-defined architectural unit.	Aggregate cost as the sum of the unit costs of the components classified in the $L \times T$ matrix.	Closed and exhaustive typing (§6.2.4).

Continued on the next page

Table 14 (continued)

Cost vector	Traditional mechanism	XF mechanism	Property of the model that enables it
Maintenance and evolution	Technological changes propagate to multiple layers; cumulative technical debt.	Change bounded to the corresponding layer without affecting the rest of the artifact.	Principle of layer isolation (§6.2.2).

Context switching between technologies A developer trained in the XF model who switches projects between distinct technologies faces a learning curve bounded exclusively to the technical substrate — language syntax, development-framework idioms, build tools. The architectural structure is the same across all XF artifacts, regardless of the technology: the same three layers, the same five types, the same nomenclature, the same dependency constraints. The cost of architectural orientation — historically proportional to the singularity of the project's conventions — tends to vanish because the conventions are universal.

Onboarding of new developers Analogously, the onboarding of a new team member is accelerated because the structure of the artifact is predictable. The developer does not need to learn the project's specific conventions before being able to contribute — they need to learn the domain logic that the artifact models, which is the only genuinely new thing. A team's capacity to scale through the addition of people trained in the model is decoupled from the project-specific learning curve, eliminating a classic bottleneck in organizations with high turnover or sustained growth.

Estimation and planning of development costs The mandatory classification of the artifact's components in the $L \times T$ matrix — produced during functional analysis — opens a path to cost estimation that traditional methods do not support. Each identified and classified component constitutes a unit of work with an implementation cost that is estimable independently; the sum of the unit costs produces the aggregate cost of the artifact without resorting to expert judgment, analogy-based estimation, or subjective weighting per use case. The map of XF components thus becomes a planning instrument with exact traceability of the amount of code developed and pending development, with a precision that traditional methods do not attain precisely because they lack an architecturally well-defined unit of cost.

Maintenance and evolution Provided that the contract of the access component is preserved, the principle of layer isolation (§6.2.2) guarantees by construction that a change localized in one layer does not propagate upward: the substitution of a mechanism for accessing an external system does not affect the business logic or the interaction layer. This property reduces the cost of evolving the artifact throughout its life cycle: when a technological decision becomes obsolete — a communication protocol changes, an external system is replaced, a library is discontinued — the impact is bounded to the corresponding layer without requiring a rewrite of the

entire artifact. Technical debt ceases to accumulate structurally and instead becomes localized at the technological boundary where it originated.

13.3 Normalization and standardization

The discipline of software engineering, despite its relative maturity, has historically lacked an equivalent to the normative regulation that characterizes consolidated engineering disciplines. The constitution of the field of *software architecture* as a research agenda with its own vocabulary and problems was formally delineated by Garlan [15]; the XF model is inscribed in that agenda and positions itself as a step in the normative direction: an architectural reference model in the sense of the ISO/IEC/IEEE 42010:2022 standard [22], in continuity with the standardization tradition to which the OSI model [17], the consolidated software-engineering vocabularies [20], the software life cycle processes [19], and the ISO/IEC 25010:2023 software quality models [24] belong.

The continuity with the OSI model is not only formal but pragmatic: what OSI contributed to communication between systems — a common vocabulary that replaced proprietary nomenclatures, implementation-independent interfaces, and cross-validation that enabled systematic teaching — the XF model contributes to the internal architecture of the artifact. The two stratifications are orthogonal and conceptually complementary: OSI standardizes communication between artifacts; XF standardizes the internal structure of each artifact.

Agnostic meta-model The XF model acts as a meta-model with respect to the prior reference architectures. The formal projection of Clean Architecture, Domain-Driven Design, Hexagonal Architecture, Onion Architecture, and layered architectures onto the $L \times T$ matrix — developed in §D.3 — demonstrates that these architectures are partial instances of the same underlying structure. The model contributes over the state of the art the simultaneous integration of three prescriptions that the prior architectures formulate partially or independently: a layer structure derived from first principles, a closed and exhaustive taxonomy of component types, and a technologically agnostic canonical nomenclature.

Deterministic verifiability The conformance of an artifact with the model is algorithmically computable by static analysis tools. The catalog of rules (§11.3) and the algorithm for determining the conformance level (§11.4) materialize this verifiability: given an artifact and the catalog, the conformance level is a function of the artifact, not of the analyst's interpretation. This property is the basis on which reproducible architectural audit processes are built, a recurring requirement in engineering disciplines with a consolidated normative tradition and materialized in frameworks such as ISO/IEC/IEEE 12207 [19], which prescribes reproducible verification in the software life cycle processes.

Technological independence as a criterion of durability Architectures coupled to concrete technologies age with those technologies. The XF model, derived from invariant properties of the formal process and not from the conventions of any development framework, offers industry a layer of abstraction whose validity is

independent of the life cycle of any particular technology. This property qualifies it as a stable basis for long-term standardization: an organization that adopts the model is not betting on the longevity of a concrete framework but on an abstraction whose validity is tied to the very nature of software as the automation of formal processes.

13.4 Application in generative code models

Code generation assisted by large-scale language systems has emerged as a vector of transformation of the contemporary software industry. The formal properties of the XF model — single vocabulary, closed taxonomy, deterministic verifiability — confer on the model a singular relevance in this context. The consequences developed below are hypotheses derived from the formal properties of the model and from the prescriptions of the preceding clauses (§6, §11, §12); their controlled empirical validation remains as a line of future work.

Architectural context of constant cost Code generation systems operate over a finite context window that represents the state of the artifact being worked on. The quality of the generated code depends directly on the quality and completeness of the context that the system can maintain. In an artifact without standardized architectural conventions, the generating system must infer the structure, the relationships between components, and the team's conventions before producing coherent code — a variable inference cost, proportional to the singularity of the project, subject to errors when the conventions are ambiguous or inconsistent.

In an artifact conformant with the XF model, that inference cost tends toward a constant: the canonical folder structure, the prescribed nomenclature, and the dependency constraints are the same in any artifact conformant with the model, regardless of domain and scale. The generating system can devote the available context space to the domain logic — the only genuinely artifact-specific element — instead of to the inference of architectural conventions. The consequence is direct: greater coherence of the generated code, lower probability of structural violations, and greater effective capacity for generation at industrial scale.

Functional specification vocabulary The formal ontology of the model (§12) acts as a functional specification language of high precision and low ambiguity. A specification expressed in the vocabulary of the model — "a logical component in the Business Layer with an operation that verifies a precondition and delegates to a repository" — describes with exactness what exists, where it resides, what it does, and what it depends on, without interpretive ambiguity.

This semantic precision enables the paradigm known as *Specification-Driven Development* [37] — in which the formal functional specification precedes and algorithmically drives the generation of the code — with a level of formality that proprietary taxonomies do not admit. The functional specification, expressed in the vocabulary of the model, ceases to be a text subject to interpretation and becomes a formal object algorithmically transformable into code. The generator does not infer the structure — it receives it as part of the specification — and concentrates on the domain logic.

The functional analyst and the generator share the same mental model because they share the same formal vocabulary.

Generation-time verification The deterministic verifiability of the catalog (§11.3) closes the generation–verification–correction cycle within the generation process itself. A generating system that knows the XF model can evaluate at generation time whether the produced code satisfies the catalog before delivering it to the developer, reducing the human-in-the-loop iterations to the dimensions that require high-level architectural judgment and transferring to the generating system the structural-adequacy verifications that have historically fallen on human review. The specification, the implementation, and the verification share the same structural vocabulary: the transformation between them reduces to instantiating that vocabulary in the syntax corresponding to each level.

13.5 Closing the conceptual gap between human and generated code

Code generation by artificial intelligence systems introduces a challenge of growing relevance in industry: the conceptual gap between the code produced by the machine and the human developer's capacity to understand, review, maintain, and evolve it. This gap is not one of the developer's technical competence but of the structural homogeneity of the code: when each generation produces a distinct architectural style — influenced by the diversity of patterns present in the training data and by the absence of explicit conventions in the context — the human reader must decipher the implicit conventions of each fragment before being able to operate on it. The cost of comprehension replaces the cost of authoring that generation had suppressed, partially nullifying the productivity gain.

The consequence is an architectural risk of an emergent nature: as the proportion of generated code in artifacts grows, the human responsibility over its structural correctness is pressured by the inability to inspect it rapidly. Architectural audit — historically a costly practice but bounded to specific moments of the life cycle — is now demanded continuously and at speeds that the human flow cannot sustain without a structural substrate that facilitates it.

The XF model as a common substrate The adoption of the XF model as the architectural contract of the artifact turns that conceptual gap into an operational equivalence between human reader and generated code. The human recognizes the structure of the generated code because it is the canonical structure of the model, not a structure specific to the produced fragment. The layers are where the model prescribes, the types are where the model prescribes, the components bear the names that the model prescribes. The architectural review of the generated code no longer requires deciphering implicit conventions and becomes a verification of membership in known cells within a known matrix. Figure 13 illustrates this transformation: on the left, an artifact without a common structure over which the human reviewer must reconstruct the implicit conventions of each generated fragment; on the right, an artifact conformant with the XF model whose canonical structure is directly recognizable and reduces the review to a verification of membership in known cells.

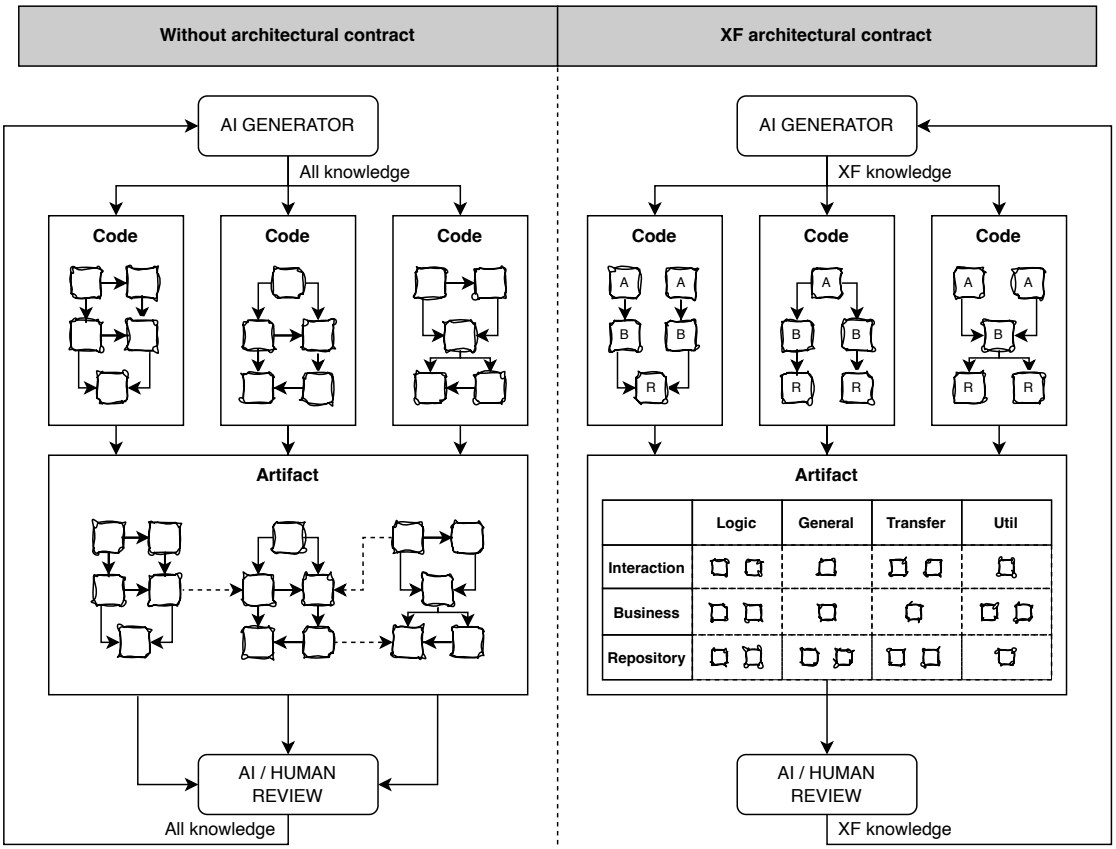

Figure 13. Closing the conceptual gap between the human developer and the code generated by automatic systems. On the left, an artifact without a common architectural contract: each generated fragment adopts distinct implicit conventions — nomenclature, folder structure, distribution of responsibilities — and the human reviewer must decipher them before being able to operate on the code, with a comprehension cost that is variable and proportional to the accumulated heterogeneity. On the right, an artifact conformant with the XF model: the layers, the types, and the nomenclature are the canonical ones prescribed by the model regardless of the fragment or the generating system that produced it, and the human review reduces to verifying the membership of each component in its corresponding cell of the $L \times T$ matrix, with a constant comprehension cost.

This property has a relevant historical parallel. The introduction of the ubiquitous language in Domain-Driven Design [11] resolved the communication gap between developers and domain experts through a shared vocabulary that both could use with the same precision. The XF model resolves the equivalent gap between the human developer and the code-generating system through a shared structural vocabulary that both can interpret with the same exactness. The ubiquitous language of DDD operated over the business domain; the vocabulary of the model operates over the architecture domain, completing the structural coverage of human–machine communication.

Reduction of continuous-audit effort The architectural audit of the generated code is offloaded onto static conformance analysis tools. The verification that requires architectural judgment — the adequacy of the functional responsibility assigned to each component, the choice of the correct classification cell — remains a human

task, but bounded to the subset of decisions that the catalog does not resolve algorithmically. The structural verification — canonical location, nomenclature, dependency constraints, life cycle — is completely automated. The ratio of human audit to automated audit is inverted with respect to the traditional review model, freeing human capacity for the decisions that genuinely require it.

Maintenance and evolution of code not written by humans The challenge that industry is beginning to face to an increasing degree — maintaining and evolving code that no human originally wrote — requires the structure of the code to be intelligible by construction, not through the developer's accumulated familiarity with a specific style. The XF model contributes that structural intelligibility: a developer who arrives at an XF artifact generated mostly by automatic systems can operate on it with the same ease with which they would operate on one written by humans, provided that both satisfy the architectural contract of the model. Authorship ceases to be a relevant property for maintainability when the structure is uniform.

This property has an organizational consequence worth stating: governance decisions about what code is generated, how it is reviewed, and how it is versioned can be made on objective and reproducible structural bases, not on qualitative criteria that depend on the reviewer's experience. The adoption of the model is, in this sense, an architectural response to the emergent risk of massive generation: it turns into a verifiable technical problem what is currently a human problem sustained by individual expertise.

14 Conclusions

This document has introduced the Cross-Framework Architecture model — XF — as an architectural reference model that organizes the internal structure of any software artifact through two orthogonal dimensions of classification: the abstraction dimension, which stratifies the artifact into three layers derived from the invariant stages of every formal process, and the functional dimension, which types the components of each layer into five categories with precise and non-overlapping responsibilities.

The central contribution of the model can be stated concisely: it simultaneously addresses three problems that have characterized the fragmentation of the software development ecosystem — the absence of a universal layer structure derived from first principles, the absence of a closed and exhaustive component taxonomy, and the absence of a canonical nomenclature common to all technologies —. Prior reference models solve each of these problems partially or in specific technological domains; the XF model solves them in an integrated manner, independently of the programming language, the development framework, the programming paradigm, the execution platform, and the functional domain of the artifact.

The formal properties of the model — completeness, strict separation, and directionality of the layers; total classification of components; prescription of canonical nomenclature — make it possible for the conformance of an artifact with the model to be deterministically computable by static analysis tools. The catalog of rules (§11.3) and the conformance level determination algorithm Λ (§11.4) materialize that

verifiability: given an artifact and the catalog, the conformance level is a function of the artifact, not of the analyst's interpretation.

The Discussion clause has developed the practical consequences of adopting the model along five vectors — architectural communication with a common vocabulary, reduction of operational costs, normalization and standardization, application in generative code models, and closing the conceptual gap between the human developer and code generated by automatic systems — and has honestly stated the limitations of the current specification. The combination of formal properties and operational consequences positions the model as a standardization instrument with an industrial and long-term vocation, in continuity with the consolidated tradition of normative reference models in software engineering.

14.1 Limitations of the model

The XF model, in its current version, has a set of limitations that must be honestly stated. Some are limitations of the specification that will be addressed in future versions; others are inherent to the model's approach and cannot be resolved without compromising its fundamental properties.

Tension with automatic dependency injection frameworks The model's injection component mechanism — based on immutable static references aggregated at a single point per layer — comes into tension with development frameworks that delegate the lifecycle of components to a container of the framework itself. In those contexts, the container may create multiple instances, manage visibility scopes, or resolve dependencies in ways that the model does not prescribe. Resolving this tension requires a specific analysis that determines how to implement the spirit of the model — instance uniqueness, centralized access point, initialization order — within the constraints of each framework's container. This analysis is the object of per-technology compatibility documents, external to the normative specification of the model.

Semantic verifiability of content rules The catalog includes content rules — of the `logic-mismatch-*` and `general-mismatch-*` type — that verify that a component does not incorporate logic foreign to its functional responsibility. These rules are conceptually precise, but their automatic verification requires semantic analysis of the source code that goes beyond the structural analysis of files and nomenclature. Their implementation in tools requires specific knowledge of the language and the library ecosystem employed and cannot be carried out in a manner completely agnostic to the technology. The model acknowledges this limitation and stratifies the catalog into two verifiability levels (structural and semantic) so that tools can honestly report the effective coverage of the analysis.

Absence of controlled empirical validation The controlled empirical validation of the impact of adopting the model has not yet been carried out under real development conditions.

Incomplete mathematical axiomatization The current version of the model formalizes the central concepts — the classification function, the computation of the conformance level, the initialization order — but does not provide an exhaustive axiomatization of all the prescriptions.

14.2 Pending empirical validation

The XF model is a formal proposal derived from first principles and from the systematic observation of the existing development ecosystem. Its theoretical foundation — the isomorphism between formal process and artifact, the derivation of the three layers from the invariant stages, the verifiability of the catalog of rules — has been developed rigorously in the preceding clauses. The controlled empirical validation of the impact of its adoption in real development contexts is, however, pending realization.

The main hypotheses that empirical validation would have to test are the following.

The **first hypothesis** is that adopting the model reduces the onboarding time of new developers into existing projects. The model predicts that a developer trained in XF can orient themselves in any XF artifact independently of its technological domain; if the prediction is correct, the onboarding time would be significantly lower in XF artifacts than in equivalent artifacts with proprietary conventions.

The **second hypothesis** is that adopting the model reduces the density of architectural defects in artifacts. The model predicts that the mandatory classification of components before their implementation and the automatic conformance verification during development reduce the number of incorrect design decisions that manifest as defects in production.

The **third hypothesis** is that adopting the model reduces the cost of context switching between projects with different technologies. The model predicts that the learning curve of a new XF artifact in a known technology is limited to learning the domain logic and does not require learning new architectural conventions.

The **fourth hypothesis** is that expressing functional requirements in the vocabulary of the model reduces the number of iterations needed between analysts and developers to produce an implementable specification. The model predicts that the formal semantics of the XF vocabulary eliminate the interpretive ambiguity that characterizes uncontrolled natural language in the definition of requirements.

The **fifth hypothesis**, derived from the analysis in clause §13.5, is that adopting the model reduces the cost of the architectural audit of code generated by automatic systems. The model predicts that the prescribed structural uniformity transforms the human review of generated code into a verification of membership in known cells, eliminating the cost of deciphering implicit conventions that mass generation imposes on the human reader.

Table 15 summarizes the five hypotheses with their prediction, a candidate metric, and a suggested study design, as an auditable agenda for the line of empirical validation.

Table 15. Empirical validation hypotheses of the XF model, their prediction, a candidate metric, and a suggested study design. The table operationalizes the model's empirical research agenda in auditable form: for each hypothesis, what the model predicts, what observable quantity would allow it to be tested, and what experimental configuration could produce that measurement.

Hypothesis	Model prediction	Candidate metric	Suggested study design
H1 Onboarding time	Significant reduction in the orientation time of new developers in XF artifacts relative to artifacts with proprietary conventions.	Mean time to first productive contribution.	Comparison between teams trained and not trained in XF on equivalent artifacts.
H2 Density of architectural defects	The mandatory prior classification and the automatic verification reduce the incorrect design decisions that manifest as defects in production.	Architectural defects per thousand lines of code.	Longitudinal analysis of XF artifacts against artifacts with proprietary conventions in the same domain.
H3 Cost of context switching	The learning curve of a new XF artifact in a known technology is limited to the domain logic.	Architectural orientation time when switching projects.	Cross-over study on developers who move between artifacts in different technologies.
H4 Analyst–developer iterations	The formal semantics of the XF vocabulary eliminate the interpretive ambiguity of uncontrolled natural language in the definition of requirements.	Number of iterations per requirement until an implementable specification.	Analysis of clarification tickets in projects with controlled XF vocabulary against projects with uncontrolled vocabulary.
H5 Cost of auditing generated code	The prescribed structural uniformity transforms the human review of generated code into a verification of membership in known cells.	Review time per unit of generated code.	Comparative study on generated code reviewed by humans on artifacts with and without an XF contract.

The design of controlled studies that test these hypotheses in real industrial contexts — with control groups trained in traditional architectures and experimental groups trained in the model — is the highest-priority line of empirical validation for the future development of the model.

14.3 Lines of future work

The XF model, in its current version, is a complete normative specification in the aspects it covers, but it leaves open lines of research and development that are the object of complementary documents and of future versions of the model.

Controlled empirical validation As developed in the preceding clause, the controlled empirical validation of the impact of adopting the model on team productivity, quality of the code produced, onboarding cost of new developers, long-term maintainability, and reduction of the audit gap of generated code is the highest-priority line of work. The studies would have to be designed with sufficient methodological rigor to be publishable in reference journals, so that the empirical validation can be cited together with the formal specification as the foundation of the model. The quantification of the economic impact of the fragmentation of the development ecosystem underscores the industrial relevance of this validation.

Research on closing the human–generated-code gap The conceptual gap between the human developer and code generated by automatic systems — analyzed in clause §13.5 — is an emergent phenomenon whose magnitude grows with the industrial adoption of assisted generation. Characterizing it quantitatively, modeling its evolution as the proportion of generated code in artifacts increases, and measuring the mitigating effect of the XF model on that phenomenon constitute an independent research agenda of immediate relevance. The intersection of code readability theory, cognitive load models of the reader, and the formal properties of the model is the natural space for that research.

Complete mathematical axiomatization The current version of the model includes formal expressions for the central concepts but does not provide an exhaustive mathematical axiomatization of all its prescriptions. Complete formalization in predicate logic — enabling the proof of properties such as the consistency of the classification system, the completeness of the catalog of rules, and the correctness of the conformance level determination algorithm — is a line of research that requires collaboration with the formal logic and software engineering communities.

Conformance analysis tools The catalog of rules and the conformance level determination algorithm provide the complete specification for the implementation of XF conformance static analysis tools. The envisaged tools include folder structure and nomenclature analyzers — implementable in a technologically agnostic manner —, inter-component dependency analyzers — implementable per language through import analysis —, semantic content analyzers — implementable through per-language abstract syntax tree analysis —, development environment plugins that provide real-time feedback, and conformance report generators that compute the λ level of the artifact.

Base projects and XF code generators A set of base projects implementing the canonical folder structure, the injection components, and the most common generalizations of the model for the most widespread development technologies, distributed as open-source repositories, would reduce the start-up cost of new XF

artifacts and would guarantee that the technical configuration is compatible with the model from the first line of code. Complementarily, code generators that receive a functional specification in XF vocabulary and produce code skeletons conformant to the model — with components correctly classified, located, and named, and verifiable against the catalog — constitute a promising line of development, especially in combination with generation systems based on language models, which would provide the inference of the domain logic over the prescribed architectural skeleton.

XF library ecosystem The development of an XF library ecosystem — components published and distributed in conformance with the model, both utility and domain-modeling (§10.3) — is the condition for the adoption of the model to be cumulative and economically efficient. The highest-priority libraries are those that abstract the most recurrent patterns of software development in the most widespread technologies. Their development as open-source projects, with automatic conformance verification, is the most effective way to drive the adoption of the model in industry. The reference trajectory is that of the OSI model: what OSI made possible for communication between systems — an ecosystem of interoperable protocol implementations over a common vocabulary — the XF model makes possible for the internal architecture of the artifact.

Conformance certification The model opens the possibility of a conformance certification system — both for people and for artifacts — analogous to the one that organizations such as the W3C administer for web standards. A competency certification would attest that a person is capable of developing artifacts conformant to the model; an artifact conformance seal would attest that an artifact satisfies the prescriptions of the model at a given level. The design of the certification system — criteria, certifying bodies, audit processes — is a line of work that transcends the technical scope of the model and requires the participation of the community of practice.

15 Bibliography

References

[1] Agha, G. A. (1986). *Actors: A model of concurrent computation in distributed systems*. MIT Press.

[2] Allen, R., and Garlan, D. (1997). A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, *6*(3), 213–249. https://doi.org/10.1145/258077.258078

[3] Bass, L., Clements, P., and Kazman, R. (2012). *Software architecture in practice* (3rd ed.). Addison-Wesley Professional.

[4] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). *Manifesto for agile software development*. https://agilemanifesto.org

[5] Bertalanffy, L. von (1968). *General system theory: Foundations, development, applications*. George Braziller.

[6] Booch, G. (1994). *Object-oriented analysis and design with applications* (2nd ed.). Benjamin/Cummings.

[7] Brooks, F. P. (1987). No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, *20*(4), 10–19. https://doi.org/10.1109/MC.1987.1663532

[8] Cockburn, A. (2005). *Hexagonal architecture*. alistair.cockburn.us. https://alistair.cockburn.us/hexagonal-architecture/

[9] Cunningham, W. (1992). The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, *4*(2), 29–30. https://doi.org/10.1145/157710.157715

[10] Dijkstra, E. W. (1968). The structure of the "THE"-multiprogramming system. *Communications of the ACM*, *11*(5), 341–346. https://doi.org/10.1145/363095.363143

[11] Evans, E. (2003). *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley Professional.

[12] Fowler, M. (2002). *Patterns of enterprise application architecture*. Addison-Wesley Professional.

[13] Fowler, M. (2004). Inversion of Control Containers and the Dependency Injection pattern. *martinfowler.com*. https://martinfowler.com/articles/injection.html

[14] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional.

[15] Garlan, D. (2000). Software architecture: A roadmap. In A. Finkelstein (Ed.), *The Future of Software Engineering* (pp. 91–101). ACM Press. https://doi.org/10.1145/336512.336537

[16] Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, *21*(8), 666–677. https://doi.org/10.1145/359576.359585

[17] International Organization for Standardization. (1994). *Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model* (ISO/IEC 7498-1:1994). ISO/IEC.

[18] International Organization for Standardization. (2013). *Information technology — Object Management Group Business Process Model and Notation* (ISO/IEC 19510:2013). ISO.

[19] International Organization for Standardization, International Electrotechnical Commission, and Institute of Electrical and Electronics Engineers. (2017). *Systems and software engineering — Software life cycle processes* (ISO/IEC/IEEE 12207:2017). ISO/IEC/IEEE.

[20] International Organization for Standardization, International Electrotechnical Commission, and Institute of Electrical and Electronics Engineers. (2017).

Systems and software engineering — Vocabulary (ISO/IEC/IEEE 24765:2017). ISO/IEC/IEEE.

[21] International Organization for Standardization. (2021). *ISO/IEC Directives, Part 2: Principles and rules for the structure and drafting of ISO and IEC documents* (9th ed.). ISO/IEC.

[22] International Organization for Standardization, International Electrotechnical Commission, and Institute of Electrical and Electronics Engineers. (2022). *Software, systems and enterprise — Architecture description* (ISO/IEC/IEEE 42010:2022). ISO/IEC/IEEE.

[23] International Organization for Standardization, International Electrotechnical Commission, and Institute of Electrical and Electronics Engineers. (2023). *Systems and software engineering — System life cycle processes* (ISO/IEC/IEEE 15288:2023). ISO/IEC/IEEE.

[24] International Organization for Standardization and International Electrotechnical Commission. (2023). *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model* (ISO/IEC 25010:2023). ISO/IEC.

[25] Jones, C. (2008). *Applied software measurement: Global analysis of productivity and quality* (3rd ed.). McGraw-Hill.

[26] Lewis, J., and Fowler, M. (2014). *Microservices: A definition of this new architectural term*. https://martinfowler.com/articles/microservices.html

[27] Liskov, B. H., and Wing, J. M. (1994). A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, *16*(6), 1811–1841. https://doi.org/10.1145/197320.197383

[28] Mandelbrot, B. B. (1982). *The fractal geometry of nature*. W. H. Freeman.

[29] Martin, R. C. (2008). *Clean code: A handbook of agile software craftsmanship*. Prentice Hall.

[30] Martin, R. C. (2017). *Clean architecture: A craftsman's guide to software structure and design*. Prentice Hall.

[31] Meyer, B. (1992). *Object-oriented software construction* (1st ed.). Prentice Hall.

[32] Milner, R. (1999). *Communicating and mobile systems: The π-calculus*. Cambridge University Press.

[33] Naur, P., and Randell, B. (Eds.). (1968). *Software engineering: Report on a conference sponsored by the NATO Science Committee*. NATO Scientific Affairs Division.

[34] Newman, S. (2015). *Building microservices: Designing fine-grained systems*. O'Reilly Media.

[35] Object Management Group. (2014). *Business process model and notation (BPMN) version 2.0.2* (OMG Document Number: formal/2013-12-09). OMG. https://www.omg.org/spec/BPMN/2.0.2

[36] Object Management Group. (2017). *Unified Modeling Language (UML) specification, version 2.5.1* (OMG Document Number: formal/2017-12-05). OMG. https://www.omg.org/spec/UML/2.5.1

[37] Ostroff, J. S., Makalsky, D., and Paige, R. F. (2004). Agile specification-driven development. In *Extreme Programming and Agile Processes in Software Engineering (XP 2004)* (LNCS 3092, pp. 104–112). Springer.

[38] Palermo, J. (2008). *The onion architecture*. jeffreypalermo.com. https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/

[39] Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, *15*(12), 1053–1058. https://doi.org/10.1145/361598.361623

[40] Pierce, B. C. (2002). *Types and programming languages*. MIT Press.

[41] Shaw, M., and Garlan, D. (1996). *Software architecture: Perspectives on an emerging discipline*. Prentice Hall.

[42] Szyperski, C. (2002). *Component software: Beyond object-oriented programming* (2nd ed.). Addison-Wesley Professional.

[43] Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, *s2-42*(1), 230–265. https://doi.org/10.1112/plms/s2-42.1.230

[44] United States Department of Defense. (1994). *Software development and documentation* (MIL-STD-498). U.S. Department of Defense.

[45] Wiener, N. (1948). *Cybernetics: Or control and communication in the animal and the machine*. MIT Press.

[46] Wirth, N. (1971). Program development by stepwise refinement. *Communications of the ACM*, *14*(4), 221–227. https://doi.org/10.1145/362575.362577

[47] Yourdon, E., and Constantine, L. L. (1979). *Structured design: Fundamentals of a discipline of computer program and systems design*. Prentice Hall.

A Annex A — Implementation examples: Access Layer

This annex is informative. Its purpose is to illustrate the application of the XF Architectural Model through a reference implementation. The content of this annex does not constitute a normative requirement.

This annex presents a reference implementation of the five types of components of the Access Layer. The examples belong to a hypothetical artifact for managing a connected thermostat: an IoT device that measures and controls the ambient temperature, communicates with a remote server to synchronize data, and is managed by an authenticated user. The same thread runs through Annexes B and C. For the sake of simplicity, the examples in the three annexes (A–C) do not internally

subdivide `/logic`; a real artifact would follow the subdivision guidelines of the corresponding layer (§7.2.1 for the Access Layer).

Note on the language of the examples. The choice of TypeScript for the examples in Annexes A–C is illustrative, not normative: it was preferred for the eloquence of its type system — interfaces, union types, access modifiers, and generics — to express with precision the structure of the components, the signatures of the operations, and the relationships between layers without syntactic noise. The XF model is technology-agnostic: any object-oriented language with an equivalent type system would materialize the same components with identical classification in the $L \times T$ matrix.

A.1 Transfer components

The transfer components of the Access Layer model the data structures as exposed by external systems — before any semantic transformation by the artifact — and the exceptions specific to communication failures. They live in `/src/repository/transfers`.

```
// /src/repository/transfers/RawTemperatureRecord.ts
export interface RawTemperatureRecord {
  readonly id: string;
  readonly value: number;
  readonly unit: string;       // as it arrives from the server: "C" or "F"
  readonly recordedAt: string; // ISO-8601 untransformed
}

// /src/repository/transfers/AuthTokenResponse.ts
export interface AuthTokenResponse {
  readonly token: string;
  readonly expiresInSeconds: number;
  readonly refreshToken: string;
}

// /src/repository/transfers/NetworkUnavailableException.ts
export class NetworkUnavailableException extends Error {
  constructor(public readonly endpoint: string, cause?: Error) {
    super(`Network unavailable: ${endpoint}`);
    this.cause = cause;
  }
}

// /src/repository/transfers/AuthenticationFailedException.ts
export class AuthenticationFailedException extends Error {
  constructor(public readonly reason: string) {
    super(`Authentication failed: ${reason}`);
  }
}
```

Listing 43. Transfer components of the Access Layer

A.2 Utility components

The utilities of the Access Layer operate on primitive types inherited from the OSI Presentation Layer and are, therefore, accessible from any layer of the artifact (§7.3.4). A string-manipulation utility (date parsing) is illustrated here that will be reusable throughout the artifact.

```
// /src/repository/utils/StringUtils.ts
export abstract class StringUtils {
  private constructor() {}
```

```
4
5   static parseIsoDate(s: string): Date {
6     const parsed = new Date(s);
7     if (isNaN(parsed.getTime())) {
8       throw new Error(`Invalid ISO-8601 date: ${s}`);
9     }
10    return parsed;
11  }
12
13  static isBlank(s: string | null | undefined): boolean {
14    return s === null || s === undefined || s.trim().length === 0;
15  }
16 }
```

Listing 44. Access utility component over primitive types

A.3 Generalization components

The generalizations of the Access Layer abstract behaviors common across the artifact's Repositories. The example models a remote Repository that encapsulates the HTTP request cycle, the uniform propagation of network errors, and the possibility of adding an optional authentication header. It lives in `/src/repository/general`.

```
1  // /src/repository/general/RemoteRepository.ts
2  export abstract class RemoteRepository {
3    protected abstract baseUrl(): string;
4
5    protected async request<T>(
6      path: string,
7      options: { method: string; body?: unknown; bearerToken?: string } = {
           method: 'GET' }
8    ): Promise<T> {
9      const url = `${this.baseUrl()}${path}`;
10     const headers: Record<string, string> = { 'Content-Type':
           'application/json' };
11     if (options.bearerToken) {
12       headers['Authorization'] = `Bearer ${options.bearerToken}`;
13     }
14
15     try {
16       const response = await fetch(url, {
17         method: options.method,
18         headers,
19         body: options.body ? JSON.stringify(options.body) : undefined
20       });
21
22       if (!response.ok) {
23         if (response.status === 401 || response.status === 403) {
24           throw new AuthenticationFailedException(`HTTP ${response.status}`);
25         }
26         throw new NetworkUnavailableException(url);
27       }
28
29       return await response.json() as T;
30     } catch (e) {
31       if (e instanceof AuthenticationFailedException) throw e;
32       if (e instanceof NetworkUnavailableException) throw e;
33       throw new NetworkUnavailableException(url, e as Error);
34     }
35   }
36
37   init(): void {}
38   terminate(): void {}
39 }
```

Listing 45. Generalization component: `RemoteRepository`

A.4 Logical components

The logical components of the Access Layer — the Repositories — encapsulate a concrete protocol for communication with an external system. The example presents two Repositories: `IdentityRepository` (authentication against an identity server) and `ServerRepository` (synchronization of temperature measurements with the telemetry server). Both extend `RemoteRepository`.

The `verifyToken` operation of `IdentityRepository` illustrates the complete pattern: preconditions (non-empty token), statements (delegation to the generalization for the remote call, normalization of the result), return typed by an `AuthTokenResponse`, and transmission of `AuthenticationFailedException` and `NetworkUnavailableException` as language exceptions.

```
// /src/repository/logic/IdentityRepository.ts
export class IdentityRepository extends RemoteRepository {
  protected baseUrl(): string {
    return 'https://identity.example.com/v1';
  }

  async verifyToken(token: string): Promise<AuthTokenResponse> {
    // Preconditions
    if (StringUtils.isBlank(token)) {
      throw new AuthenticationFailedException('empty token');
    }

    // Statements: delegation to the remote protocol
    const response = await this.request<AuthTokenResponse>('/verify', {
      method: 'POST',
      body: { token }
    });

    // Transfer return
    return response;
  }

  async revokeToken(token: string): Promise<void> {
    if (StringUtils.isBlank(token)) {
      throw new AuthenticationFailedException('empty token');
    }
    await this.request<void>('/revoke', { method: 'POST', body: { token } });
  }
}

// /src/repository/logic/ServerRepository.ts
export class ServerRepository extends RemoteRepository {
  protected baseUrl(): string {
    return 'https://telemetry.example.com/v1';
  }

  async fetchLastTemperature(deviceId: string, bearerToken: string):
      Promise<RawTemperatureRecord> {
    if (StringUtils.isBlank(deviceId)) {
      throw new NetworkUnavailableException('invalid deviceId');
    }
    return await this.request<RawTemperatureRecord>(
      `/devices/${deviceId}/temperature/last`,
      { method: 'GET', bearerToken }
    );
  }

  async saveTemperature(
    deviceId: string,
    record: RawTemperatureRecord,
    bearerToken: string
  ): Promise<void> {
    await this.request<void>(
      `/devices/${deviceId}/temperature`,
      { method: 'POST', body: record, bearerToken }
```

```
    );
  }

  // Primitive return: example of an operation that returns a number
  async countMeasurements(deviceId: string, bearerToken: string):
      Promise<number> {
    const result = await this.request<{ count: number }>(
      `/devices/${deviceId}/temperature/count`,
      { method: 'GET', bearerToken }
    );
    return result.count;
  }
}
```

Listing 46. Access logical components: `IdentityRepository` and `ServerRepository`

A.5 Injection component

The `R` component aggregates the unique instances of all the artifact's Repositories and manages their life cycle. It is the only access surface to the Access Layer from the rest of the artifact.

```
// /src/repository/R.ts
export abstract class R {
  // Definition: unique instances of the artifact's Repositories
  static readonly identity: IdentityRepository = new IdentityRepository();
  static readonly server: ServerRepository = new ServerRepository();

  private constructor() {} // not instantiable

  static init(): void {
    // Dependency order (in this example there are no internal dependencies)
    R.identity.init();
    R.server.init();
  }

  static terminate(): void {
    // Reverse order of initialization
    R.server.terminate();
    R.identity.terminate();
  }
}
```

Listing 47. Injection component of the Access Layer: `R`

B Annex B — Implementation examples: Business Layer

This annex is informative. Its purpose is to illustrate the application of the XF Architectural Model through a reference implementation. The content of this annex does not constitute a normative requirement.

This annex continues the connected-thermostat thread begun in Annex A and presents the five types of Business Layer components. The Business Layer orchestrates the domain logic: it manages the user session, transforms raw server data into coherent domain structures, validates business rules and maintains the observable state of the artifact's concepts. The process model that these components materialize is specified in §7.2.2.

B.1 Transfer components

The transfer components of the Business Layer model the domain concepts in their normalized and semantically meaningful form. They are distinct from the transfer components of the Access Layer (which reflect the raw server form): here the data already has domain types, canonical units and implicit coherence rules.

```
// /src/business/transfers/Temperature.ts
export interface Temperature {
  readonly value: number;
  readonly unit: 'C' | 'F';
  readonly recordedAt: Date;
}

// /src/business/transfers/Session.ts
export interface Session {
  readonly token: string;
  readonly expiresAt: Date;
  readonly refreshToken: string;
}

// /src/business/transfers/User.ts
export interface User {
  readonly id: string;
  readonly displayName: string;
  readonly deviceId: string;
}

// /src/business/transfers/TemperatureOutOfRangeException.ts
export class TemperatureOutOfRangeException extends Error {
  constructor(public readonly value: number, public readonly unit: 'C' | 'F') {
    super(`Temperature ${value}${unit} is out of acceptable range [14C, 32C]`);
  }
}

// /src/business/transfers/SessionExpiredException.ts
export class SessionExpiredException extends Error {
  constructor() {
    super('Session has expired; refresh required');
  }
}
```

Listing 48. Transfer components of the Business Layer

B.2 Utility components

The Business Layer utilities are pure functions over domain concepts. Unlike the Access utilities (which operate over primitives and are global), the Business utilities have local scope: they are only invoked from Business components.

```
// /src/business/utils/TemperatureUtils.ts
export abstract class TemperatureUtils {
  private constructor() {}

  static toCelsius(t: Temperature): number {
    return t.unit === 'F' ? (t.value - 32) * 5 / 9 : t.value;
  }

  static toFahrenheit(t: Temperature): number {
    return t.unit === 'C' ? t.value * 9 / 5 + 32 : t.value;
  }

  static isWithinAcceptableRange(t: Temperature): boolean {
    const celsius = TemperatureUtils.toCelsius(t);
    return celsius >= 14 && celsius <= 32;
  }
}
```

Listing 49. Business utility component: `TemperatureUtils`

B.3 Generalization components

The generalization of the Business Layer in this example is `StatefulBusiness<T>`, which abstracts the observable Business pattern: a component that maintains a state and notifies the registered observers when the state changes, illustrating upward communication without breaking the isolation between layers (§6.2.2).

```
// /src/business/general/StatefulBusiness.ts
export abstract class StatefulBusiness<T> {
  private observers: Array<(value: T) => void> = [];

  subscribe(observer: (value: T) => void): void {
    this.observers.push(observer);
  }

  unsubscribe(observer: (value: T) => void): void {
    this.observers = this.observers.filter(o => o !== observer);
  }

  protected notify(value: T): void {
    this.observers.forEach(o => o(value));
  }

  init(): void {}
  terminate(): void {
    this.observers = [];
  }
}
```

Listing 50. Generalization component: `StatefulBusiness<T>`

B.4 Logical components

The logical components of the Business Layer — the Businesses — model the domain concepts. The example presents three components: `SessionBusiness` (authentication state), `TemperatureBusiness` (temperature state and rules) and `UserBusiness` (active user identity). `TemperatureBusiness` illustrates the complete pattern: preconditions (acceptable range, valid session), delegation to the Access Layer through `R` and translation of Access exceptions into business exceptions, state mutation and notification to observers.

```
// /src/business/logic/SessionBusiness.ts
export class SessionBusiness extends StatefulBusiness<Session | null> {
  private current: Session | null = null;

  init(): void {
    super.init();
  }

  terminate(): void {
    this.current = null;
    super.terminate();
  }

  async login(token: string): Promise<Session> {
    // Preconditions
    if (StringUtils.isBlank(token)) {
      throw new SessionExpiredException();
    }
```

```
19
20      // Statements: delegation to Access + error translation
21      try {
22        const response = await R.identity.verifyToken(token);
23        const session: Session = {
24          token: response.token,
25          expiresAt: new Date(Date.now() + response.expiresInSeconds * 1000),
26          refreshToken: response.refreshToken
27        };
28        this.current = session;
29        this.notify(session); // notification to observers
30        return session;
31      } catch (e) {
32        if (e instanceof AuthenticationFailedException) {
33          throw new SessionExpiredException();
34        }
35        throw e;
36      }
37    }
38
39    // Primitive return: example of an operation that returns a boolean
40    isActive(): boolean {
41      return this.current !== null && this.current.expiresAt > new Date();
42    }
43
44    getCurrent(): Session | null {
45      return this.current;
46    }
47  }
```

Listing 51. Business logical component: `SessionBusiness`

```
1   // /src/business/logic/TemperatureBusiness.ts
2   export class TemperatureBusiness extends StatefulBusiness<Temperature> {
3     private current: Temperature | null = null;
4
5     init(): void {
6       super.init();
7     }
8
9     terminate(): void {
10      this.current = null;
11      super.terminate();
12    }
13
14    async update(t: Temperature): Promise<void> {
15      // Preconditions (transmitted as an exception)
16      if (!TemperatureUtils.isWithinAcceptableRange(t)) {
17        throw new TemperatureOutOfRangeException(t.value, t.unit);
18      }
19      const session = B.session.getCurrent();
20      if (session === null || !B.session.isActive()) {
21        throw new SessionExpiredException();
22      }
23
24      // Statements: delegation to Access, mutation, notification
25      const raw: RawTemperatureRecord = {
26        id: crypto.randomUUID(),
27        value: t.value,
28        unit: t.unit,
29        recordedAt: t.recordedAt.toISOString()
30      };
31      try {
32        await R.server.saveTemperature(B.user.getDeviceId(), raw, session.token);
33      } catch (e) {
34        if (e instanceof NetworkUnavailableException) {
35          // The local state is updated even if the remote synchronization fails
36          this.current = t;
37          this.notify(t);
38          throw e;
39        }
40        throw e;
41      }
```

```
42
43      this.current = t;
44      this.notify(t);
45    }
46
47    // Transfer return
48    getCurrent(): Temperature | null {
49      return this.current;
50    }
51  }
```

Listing 52. Business logical component: `TemperatureBusiness`

```
1  // /src/business/logic/UserBusiness.ts
2  export class UserBusiness extends StatefulBusiness<User | null> {
3    private current: User | null = null;
4
5    init(): void {
6      super.init();
7    }
8
9    terminate(): void {
10     this.current = null;
11     super.terminate();
12   }
13
14   setActive(user: User): void {
15     this.current = user;
16     this.notify(user);
17   }
18
19   getCurrent(): User | null {
20     return this.current;
21   }
22
23   getDeviceId(): string {
24     if (this.current === null) {
25       throw new SessionExpiredException();
26     }
27     return this.current.deviceId;
28   }
29 }
```

Listing 53. Business logical component: `UserBusiness`

B.5 Injection component

The `B` component aggregates the artifact's Businesses and manages their life cycle. The initialization order reflects the internal dependencies: `SessionBusiness` first, then `UserBusiness`, and last `TemperatureBusiness` (which depends on the active session and the user's device).

```
1  // /src/business/B.ts
2  export abstract class B {
3    // Definition: single instances of the artifact's Businesses
4    static readonly session: SessionBusiness = new SessionBusiness();
5    static readonly user: UserBusiness = new UserBusiness();
6    static readonly temperature: TemperatureBusiness = new TemperatureBusiness();
7
8    private constructor() {} // not instantiable
9
10   static init(): void {
11     B.session.init();
12     B.user.init();           // may depend on B.session
13     B.temperature.init();    // depends on B.session and B.user
14   }
15
16   static terminate(): void {
```

```
    B.temperature.terminate();
    B.user.terminate();
    B.session.terminate();
  }
}
```

Listing 54. Injection component of the Business Layer: `B`

C Annex C — Implementation examples: Interaction Layer

This annex is informative. Its purpose is to illustrate the application of the XF Architectural Model through a reference implementation. The content of this annex does not constitute a normative requirement.

This annex closes the connected-thermostat thread and presents the five types of components of the Interaction Layer. The Interaction Layer exposes the entry points of the artifact: a REST interface for integration with external systems (a `TemperatureService`) and a graphical view for the human user (`MainView`). Each component delegates exclusively to `B` for domain logic; none contains business rules nor directly accesses external systems. The process model of this layer is specified in §7.2.3. As the closing of the guiding thread, the annex also includes the `XF` element (§8.3), which materializes the execution start point of the artifact.

C.1 Transfer components

The transfer components of the Interaction Layer model the input and output formats specific to the entry points: HTTP request bodies, standardized responses, graphical interface events. They are distinct from the Business transfer components: here the data is adapted to the concrete input protocol, not to the domain. They live in `/src/api/transfers`.

```
// /src/api/transfers/TemperatureUpdateRequest.ts
export interface TemperatureUpdateRequest {
  readonly value: number;
  readonly unit: 'C' | 'F';
}

// /src/api/transfers/ApiResponse.ts
export interface ApiResponse<T> {
  readonly status: number;
  readonly data?: T;
  readonly error?: string;
}

// /src/api/transfers/InvalidParameterException.ts
export class InvalidParameterException extends Error {
  constructor(public readonly field: string, public readonly reason: string) {
    super(`Invalid request: field '${field}' ${reason}`);
  }
}
```

Listing 55. Transfer components of the Interaction Layer

C.2 Utility components

The Interaction utilities are pure functions over presentation concepts: output formatting, input parsing. Their scope is local to the Interaction Layer.

```ts
// /src/api/utils/FormatUtils.ts
export abstract class FormatUtils {
  private constructor() {}

  static formatTemperature(t: Temperature): string {
    return `${t.value.toFixed(1)} ${t.unit}`;
  }

  static formatDate(d: Date): string {
    return d.toISOString().substring(0, 19).replace('T', ' ');
  }
}
```

Listing 56. Interaction utility component: `FormatUtils`

C.3 Generalization components

The generalization of the Interaction Layer in this example models the common pattern of a REST service: the deserialization of the request body, the invocation of the logic delegated to Business, the serialization of the response, and the uniform capture of exceptions to produce coherent error codes.

```ts
// /src/api/general/RestService.ts
export abstract class RestService {
  protected async handle<TRequest, TResponse>(
    request: TRequest,
    operation: (req: TRequest) => Promise<TResponse>
  ): Promise<ApiResponse<TResponse>> {
    try {
      const data = await operation(request);
      return { status: 200, data };
    } catch (e) {
      if (e instanceof InvalidParameterException) {
        return { status: 400, error: e.message };
      }
      if (e instanceof SessionExpiredException) {
        return { status: 401, error: e.message };
      }
      if (e instanceof TemperatureOutOfRangeException) {
        return { status: 422, error: e.message };
      }
      if (e instanceof NetworkUnavailableException) {
        return { status: 503, error: e.message };
      }
      return { status: 500, error: 'Internal error' };
    }
  }

  init(): void {}
  terminate(): void {}
}
```

Listing 57. Generalization component: `RestService`

C.4 Logical components

The logical components of the Interaction Layer are the services (systemic consumers) and the views (human consumers). The example presents `TemperatureService` (a service that exposes the temperature update operation as a REST endpoint) and

`MainView` (a view that displays and refreshes the current temperature in the graphical interface). Both delegate to `B`; neither contains domain logic.

```
// /src/api/logic/TemperatureService.ts
export class TemperatureService extends RestService {
  init(): void {
    super.init();
  }

  terminate(): void {
    super.terminate();
  }

  // Systemic entry point: receives the request and delegates to Business
  async update(request: TemperatureUpdateRequest): Promise<ApiResponse<void>> {
    return this.handle(request, async (req) => {
      // Input preconditions (format validation, not domain validation)
      if (req.unit !== 'C' && req.unit !== 'F') {
        throw new InvalidParameterException('unit', 'must be C or F');
      }
      if (typeof req.value !== 'number' || isNaN(req.value)) {
        throw new InvalidParameterException('value', 'must be a number');
      }

      // Delegation to the Business Layer
      const temperature: Temperature = {
        value: req.value,
        unit: req.unit,
        recordedAt: new Date()
      };
      await B.temperature.update(temperature);
    });
  }

  // Transfer return: the client receives the current state formatted
  async getCurrent(): Promise<ApiResponse<{ formatted: string }>> {
    return this.handle({}, async () => {
      const current = B.temperature.getCurrent();
      if (current === null) {
        throw new InvalidParameterException('state', 'no temperature recorded
            yet');
      }
      return { formatted: FormatUtils.formatTemperature(current) };
    });
  }
}
```

Listing 58. Interaction logical component: `TemperatureService`

```
// /src/api/logic/MainView.ts
export class MainView {
  private displayedValue: string = '--';

  init(): void {
    // Subscription to the observable state of the Business Layer.
    // The View reacts to changes without actively invoking Business.
    B.temperature.subscribe(this.onTemperatureChange);
  }

  terminate(): void {
    B.temperature.unsubscribe(this.onTemperatureChange);
  }

  private onTemperatureChange = (t: Temperature): void => {
    this.displayedValue = FormatUtils.formatTemperature(t);
    this.render();
  };

  // Human entry point: the user requests to refresh the temperature
  async onRefreshClicked(): Promise<void> {
    try {
      const current = B.temperature.getCurrent();
```

```
      if (current !== null) {
        this.displayedValue = FormatUtils.formatTemperature(current);
        this.render();
      }
    } catch (e) {
      if (e instanceof SessionExpiredException) {
        this.displayedValue = 'Session expired. Please log in.';
        this.render();
      } else {
        this.displayedValue = 'Error reading temperature.';
        this.render();
      }
    }
  }

  // Primitive return: example of an operation that returns a string
  getDisplayedValue(): string {
    return this.displayedValue;
  }

  private render(): void {
    // Rendering on the graphical platform (omitted for brevity)
  }
}
```

Listing 59. Interaction logical component: `MainView`

C.5 Injection component

The `A` component aggregates the services and views of the artifact. It is the last injection component to be initialized in the artifact life cycle and the first to be terminated — this reflects that the Interaction Layer depends on the availability of the already-initialized lower layers.

```
// /src/api/A.ts
export abstract class A {
  static readonly temperatureService: TemperatureService = new
      TemperatureService();
  static readonly mainView: MainView = new MainView();

  private constructor() {}

  static init(): void {
    A.temperatureService.init();
    A.mainView.init(); // subscribes to B.temperature
  }

  static terminate(): void {
    A.mainView.terminate();
    A.temperatureService.terminate();
  }
}
```

Listing 60. Injection component of the Interaction Layer: `A`

C.6 `XF` start-point element

As a closing illustration, the `XF` element (§8.3) centralizes the initialization and termination of the artifact by invoking the three injection components in ascending and descending order respectively, and declares the entry point of the artifact in the same file when the language permits it. In TypeScript / Node, where the entry point is materialized as a `main()` function in the file declared by `package.json`, the canonical convention is to declare it in the same module as XF:

```ts
// /src/XF.ts
export abstract class XF {
  private constructor() {}

  static init(): void {
    R.init();
    B.init();
    A.init();
  }

  static terminate(): void {
    A.terminate();
    B.terminate();
    R.terminate();
  }
}

// Entry point of the artifact -- same file as XF
async function main(): Promise<void> {
  XF.init();
  try {
    // The artifact is ready: A.mainView observes B.temperature,
    // which delegates to R.server to synchronize with the external system.
    // The main loop of the execution environment manages the events.
    await runApplicationLoop();
  } finally {
    XF.terminate();
  }
}

main();
```

Listing 61. Start-point element: `XF`

D Annex D — Projection of external vocabularies onto the XF model

This annex is informative. Its purpose is to show the projection of the vocabularies of external frameworks and patterns onto the XF Architectural Model. The content of this annex does not constitute a normative requirement.

This clause sets out the projection of three classes of external vocabulary onto the XF model: the component nomenclature of the development frameworks most widespread in the industry (§D.1), the classic dependency-injection patterns (§D.2) and the core concepts of the reference architectures consolidated in the literature (§D.3). The three tables are informative and serve as a terminological bridge for the reader who arrives at the model from a specific technological or architectural tradition. The projection is semantic — the concept plays the same architectural role — even though the nomenclature and the implementation details differ between vocabularies.

D.1 Equivalences with development-framework nomenclature

Table 16. Equivalences between the canonical XF nomenclature and the terms used in popular development frameworks for each layer/component-type combination.

Layer	XF type	Spring (Java)	Angular (Web)	Flutter (Mobile)	Django (Python)	Node/Express
Access	Logical (`*Repository`)	`@Repository`	`Service` (HTTP)	`Repository`	`QuerySet/Manager`	`Model` (`DAO`)
Access	Generalization	abstract class	abstract class	abstract class	abstract `Model`	abstract class
Access	Injection (`R`)	`ApplicationContext`	`Injector` root	`GetIt/Provider`	`apps` registry	`container`
Access	Utility	static util class	utility function	static method	utility module	module exports
Access	Transfer	`POJO` / `DTO`	`interface`	`class` (data)	`dict/dataclass`	plain object
Business	Logical (`*Business`)	`@Service`	`Service` (`@Injectable`)	`BLoC/Cubit`	business in views	business module
Business	Generalization	abstract `Service`	abstract `Service`	abstract `BLoC`	abstract logic	abstract module
Business	Injection (`B`)	`@Configuration`	`Injector` domain	`Provider` tree	`services` module	`container`
Business	Utility	static helpers	utility function	static method	utils module	module exports
Business	Transfer	`Entity/Model`	`interface/class`	domain class	Django `Model`	plain object
Interaction	Logical (`*Service`)	`@RestController`	route component	(not applicable)	`View/ViewSet` (DRF)	route handler
Interaction	Logical (`*View`)	(not applicable)	`Component`	`Widget`	`TemplateView`	(not applicable)
Interaction	Generalization	abstract controller	base `Component`	abstract `Widget`	generic view	abstract handler
Interaction	Injection (`A`)	`WebApplicationContext`	`Injector` UI	`Provider` root	`urls` registry	express `app`
Interaction	Utility	template helpers	pipe / formatter	`Helper` static	template tag/filter	view helper
Interaction	Transfer	`@RequestBody/Response`	`ApiResponse`	`DTO` class	serializer DRF	req/res body

D.2 Equivalences with classic dependency-injection patterns

Table 17. Conceptual equivalence between classic dependency-injection patterns [13, 14] and the injection components of the XF model. The "Difference" column specifies the nuance of each pattern with respect to its projection onto the XF model.

Classic pattern	XF equivalent	Difference from the XF model
Service Locator [13]	`R`, `B`, `A`	XF has exactly three fixed locators (one per layer) accessed by static name, with no dynamic lookup and no runtime configuration.
Constructor Injection	(not applicable)	XF does not inject via the constructor: logical components access their dependencies through `R.x`, `B.y`, `A.z` in the body of the operations.
Setter Injection	(not applicable)	XF does not mutate the dependency graph after construction; the model guarantees structural immutability.
Singleton [14]	Logical component under injection	XF guarantees uniqueness by construction of the injection component, without requiring the Singleton pattern in the class.
Service Container (IoC)	$\mathtt{R} \cup \mathtt{B} \cup \mathtt{A}$	XF prescribes three containers with responsibility per layer, not a single expandable container.
Factory [14]	Initial instantiation by the injection component	The initial instantiation (static fields initialized when the singleton class is loaded) constructs the logical instances and establishes their initial state σ_0; the model exposes no invocable factories and does not separate construction into an operation with its own identifier.
DI Framework	Injection comp. + static dependency graph	XF resolves dependencies at compile-time through analysis of the dependency graph; without the need for a dependency-injection framework at run time.
Registry Pattern	Partial function I_l	The `R.id`, `B.id`, `A.id` is a registry with a finite, closed domain known at compile-time.

D.3 Projection of reference architectures

The XF model acts as a meta-model with respect to the reference architectures consolidated in the literature: it provides the formal vocabulary in which each architecture projects as a partial instance. The following tables show how the core concepts of each architecture are reformulated in terms of the $L \times T$ matrix. The projection is not a comparison between equivalent alternatives — it is the reformulation of each architecture's proprietary vocabulary in the invariant terms of the model, demonstrating that the predominant architectures are partial instances of the same underlying structure.

Clean Architecture Clean Architecture [30] prescribes the dependency-inversion principle and an organization into concentric layers without prescribing a closed taxonomy of components or a canonical nomenclature. Its projection onto the XF model is set out in the following table:

Table 18. Projection of the concepts of Clean Architecture onto the XF model: each construct is rewritten as its equivalent $L \times T$ category.

Clean Architecture concept	XF layer	XF type
Entity	Business	Transfer / Logical
Use Case (Interactor)	Business	Logical (`*Business`)
Input Boundary / Output Boundary	Business	Generalization
Controller	Interaction	Logical (`*Service`)
Presenter	Interaction	Logical (`*View`)
Gateway (Repository interface)	Business	Generalization
Gateway (Repository implementation)	Access	Logical (`*Repository`)
Frameworks & Drivers	Access	Logical (`*Repository`)

The dependency rule of Clean Architecture — dependencies always point inward, toward the policies — manifests in the XF model as the principle of isolation between layers (§6.2.2): every dependency descends from Interaction toward Business toward Access. Clean Architecture contributes the dependency-inversion principle; the XF model expresses it with closed typing of the elements that satisfy that principle.

Domain-Driven Design DDD [11] introduces a rich terminology for modeling the business domain: Aggregates, Repositories, Domain Services, Application Services, Value Objects, Bounded Contexts. The projection of this vocabulary onto the XF model is concentrated mainly in the Business Layer, as set out in the following table:

Table 19. Projection of the building blocks of Domain-Driven Design onto the XF model: each one is rewritten as its equivalent $L \times T$ category.

DDD concept	XF layer	XF type
Entity	Business	Transfer (with identity)
Value Object	Business	Transfer (immutable)
Aggregate	Business	Logical (`*Business`)
Aggregate Root	Business	Logical (stateful)
Domain Service	Business	Logical (`*Business`)
Application Service	Interaction	Logical (`*Service`)
Repository (interface)	Access	Generalization
Repository (implementation)	Access	Logical (`*Repository`)
Bounded Context	Business	Subdivision `/business/logic/instance`
Domain Event	Business	Transfer
Factory	Business	Utility (`*Utils`)
Specification	Business	Generalization

The Bounded Contexts of DDD project onto the XF model as subdivisions of the `/business/logic/instance` directory (§7.2.2), the region of the artifact where the model recognizes the functional domain as a legitimate criterion for subdivision. The Access and Interaction layers are organized instead by technical criteria orthogonal to the domain. DDD contributes the rich vocabulary of domain modeling; the XF model extends it with universal typing applicable also to the Access and Interaction layers and to the complete set of the artifact, regardless of the complexity of the domain.

Hexagonal Architecture Hexagonal Architecture [8] — also known as *Ports and Adapters* — introduces the principle of isolating the application core through abstract ports that the core exposes and concrete adapters that implement them. The projection of its vocabulary onto the XF model is set out in the following table:

Table 20. Projection of Hexagonal Architecture onto the XF model: ports and adapters are rewritten as $L \times T$ categories.

Hexagonal concept	XF layer	XF type
Application Core	Business	Logical (`*Business`)
Driving Port (primary port)	Interaction	Generalization (`*Service`)
Driving Adapter (primary adapter)	Interaction	Logical (`*Service`, `*View`)
Driven Port (secondary port)	Access	Generalization (`*Repository`)
Driven Adapter (secondary adapter)	Access	Logical (`*Repository`)

The Hexagonal model distinguishes port/adapter by direction — primary (driving) versus secondary (driven) — but unifies them conceptually under the input/output dichotomy. The XF model explicitly separates the input direction as the Interaction Layer and the output direction as the Access Layer, deriving them from the property of formal processes whereby invocation and the presentation of results constitute a structurally distinct stage from access to external systems. The projection preserves the Hexagonal principle of core isolation and adds to it the prescription of types per layer.

Onion Architecture Onion Architecture [38] organizes the system into concentric rings where dependencies always point toward the center — the domain — and never toward the outer rings. The projection of its rings onto the XF model is set out in the following table:

Table 21. Projection of Onion Architecture onto the XF model: each ring is rewritten as its equivalent $L \times T$ category.

Onion ring (from inside outward)	XF layer	XF type
Domain Model (central ring)	Business	Transfer + Logical
Domain Services	Business	Logical (`*Business`)
Application Services	Interaction	Logical (`*Service`)
Infrastructure	Access	Logical (`*Repository`)
User Interface	Interaction	Logical (`*View`)
Tests	(not applicable)	Excluded from the model

The dependency-toward-the-center rule of Onion manifests in the XF model as the descending directionality of the layer order Interaction → Business → Access. Onion contributes the circular topology as a visualization; the XF model formalizes it as an $L \times T$ matrix and prescribes the concrete types that populate each ring.

Layered architectures (Bass et al.) Layered architectures [3] establish the generic pattern of organization into horizontal layers with differentiated responsibili-

ties. The XF model is a concrete instance of this pattern with three layers derived from the invariant stages of formal processes, as set out in the following table:

Table 22. Projection of layered architectures onto the XF model: each traditional layer is rewritten as its $L \times T$ category.

Classic nomenclature (Bass et al.)	XF layer
Presentation Layer	Interaction
Application Layer (Service Layer)	Interaction
Business Logic Layer (Domain Layer)	Business
Persistence Layer (Data Layer)	Access
Database Layer	Access (external to the artifact)

The generic pattern of layered architectures does not prescribe how many layers a system must have nor what responsibilities each one must have. The XF model contributes that prescription — three layers derived from first principles — and complements it with closed typing and canonical nomenclature.

Synthesis The five projections above demonstrate that the XF model is not an alternative to the prior architectures, but the formalization of the common space that all of them describe implicitly with distinct vocabularies. Over the state of the art, the model contributes the simultaneous integration of three prescriptions that prior models formulate partially or independently: a layer structure derived from first principles, a closed and exhaustive taxonomy of component types, and a technologically agnostic canonical nomenclature. This integration is what makes it possible to express all the reference architectures in a single verifiable formal vocabulary.

E Annex E — Terminological equivalences with other vocabularies

This annex is informative. Its purpose is to record the terminological equivalences between the XF Architectural Model and other vocabularies. The content of this annex does not constitute a normative requirement.

This clause records the equivalences and differences between the term *Artifact* adopted by the XF model (defined in §3.1.1) and the analogous terms employed by other vocabularies consolidated in software engineering. The table allows the reader who arrives at the XF model from another terminological tradition to quickly recognise the correspondences and the semantic differences.

The following table synthesises these terminological equivalences.

Table 23. Terminological equivalences for the concept of a self-contained executable unit.

Vocabulary / Source	Term	Difference from the XF Artifact
OMG UML 2.5.1 [36]	*Artifact*	Supercategory: includes executables, models, documents, messages and other products of the process. The XF Artifact is the restriction to the executable and self-contained subset.
ISO/IEC/IEEE 24765:2017 [20]	*Software artifact*	Broad definition analogous to UML: any piece of information produced or consumed by the process. It does not require executability or stratification.
ISO/IEC/IEEE 12207:2017 [19]	*Software item*	Any element of the system's software (plans, procedures, programs, documentation). Vaguer than the XF Artifact.
ISO/IEC/IEEE 12207:2017 [19]	*Software unit*	Separately compilable piece: function, class or module. More granular: an artifact typically groups tens or hundreds of *units*.
ISO/IEC/IEEE 15288:2023 [23]	*System*	Combination of interacting elements with a stated purpose. Broader: a system may be composed of multiple artifacts.
MIL-STD-498 [44]	*CSCI*	Aggregation of software that satisfies an end-use function and is managed as a separate configuration unit. Semantically close, but archaic military terminology.
Szyperski [42]	*Software component*	Unit of composition with contractual interfaces and explicitly declared context dependencies, independently deployable. Shares the self-containment, but requires third-party composition that XF does not impose.
Parnas [39]	*Module*	Unit of decomposition that hides a design decision. More granular: an artifact contains several modules in the Parnas sense.
Lewis and Fowler [26]; Newman [34]	*Microservice*	Independently deployable service, modelled on a domain, typically HTTP. It captures self-containment and a single space, but limits the scope to the distributed style; it excludes mobile, scripts and embedded systems.
DevOps / Cloud-Native	*Deployable unit / artifact*	Self-contained, versioned and immutable package, ready for deployment. It captures self-containment and operability, but lacks a formal standard and does not prescribe internal structure.
Wirth [46]; Dijkstra [10]	*Program*	Composition of algorithms and data structures. Classic pedagogical conception preserved by the XF Artifact, to which it adds three structural conditions that the classic term does not prescribe.

As follows from the preceding table, **none of the vocabularies recorded simultaneously captures the three conditions that the XF model prescribes for an Artifact**: self-containment, exactly three abstraction layers derived from the invariant stages of the formal process, and a single execution space. This combination is the differential contribution of the XF vocabulary with respect to the prior literature.

The choice of the term *Artifact* — rather than coining a new one — is deliberate: the XF model positions itself as a specialisation of the OMG UML 2.5.1 concept, preserving continuity with the consolidated tradition of UML, ISO/IEC/IEEE 24765 and the unified-process methods, and adding the structural prescriptions that constitute the model's own normative content.

F Annex F — Revision history

This annex is informative. Its purpose is to record the revision history of the XF model normative document. The content of this annex does not constitute a normative requirement.

This clause records the chronology of revisions of the XF model normative document. Each row of the table captures the published version, its date of issue, a summary of the scope of the changes with respect to the previous version, and the editorial or substantive nature of the revision.

Table 24. Document revision history: version, date, and summary of the changes of each edition.

Version	Date	Scope of changes	Nature
1.0.0	June 9, 2026	First reference edition. Establishes the closed taxonomy of component types, the stratification into three abstraction layers derived from the invariant stages of formal processes, the catalogue of conformance rules and the determination algorithm for the conformance level.	Substantive

Future versions of the model shall be recorded in this table in ascending chronological order (the oldest edition first). The model's versioning policy is described in the Change policy row of the cover page. The "Nature" column takes one of three values: *Substantive* (changes to the taxonomy or the axioms, which increment the major version), *Editorial* (additions to the rule catalogue or reorganizations, which increment the minor version) and *Clarifying* (drafting refinements without alteration of the normative content, which increment the patch version).

Alphabetical index