ARCHITECTURE REFERENCE

ODT-ONT

USAREUR-AF

Ontology Design Principles

Architecture Reference

HEADQUARTERS
UNITED STATES ARMY EUROPE AND AFRICA
(USAREUR-AF)
Wiesbaden, Germany

20 MARCH 2026

DRAFT — UNOFFICIAL — NOT FOR OPERATIONAL USE

ONTOLOGY DESIGN PRINCIPLES

Governing principles for ontology design within the Palantir Foundry platform. Derived from Palantir DevCon5 (March 2026) "Advanced Ontology Deep Dive: Master the Underlying Primitives" and official Foundry community guidance.

Apply to all ontology modeling in this workspace: GDAP doctrine ontology, MIM information model, and any Foundry-backed Object Types.

Reference texts: *Domain-Driven Design* (Eric Evans), *Clean Architecture* (Robert C. Martin), *Effective Java Third Edition* (Joshua Bloch).

01/ DOMAIN DRIVEN DESIGN (DDD)

The Ontology is the API of the organization. Object Types, Links, and Actions must map directly to how the business domain thinks and operates — not to how data happens to be stored.

DDD Concept	Foundry Equivalent
Bounded Context	Ontology Project boundary
Aggregate Root	Object Type with backing dataset
Entity	Object Type instance
Value Object	Struct Type / Shared Property Type
Domain Event	Action Type
Repository	Ontology SDK (OSDK)
Ubiquitous Language	Object Type and Action naming

Rules: - Name Object Types and Actions using natural-language business concepts that form coherent sentences when composed - Define Object Types from the user's decision-making needs first, then map to backing data - Separate concerns between domains via project boundaries (Datasource → Integration → Ontology → Application)

02/ DON'T REPEAT YOURSELF (DRY) — USE THE RULE OF THREE

Avoid duplication, but do not abstract prematurely. Wait until a pattern appears three times before extracting a shared abstraction.

Rules: - Check existing Ontology before creating new Object Types — reuse first - Use Shared Property Types (SPTs) for properties that appear across multiple Object Types - Use Interfaces for behavioral contracts shared across types - Do NOT duplicate parent properties on child objects unless computationally necessary - Do NOT create helper abstractions for one-time operations — three similar definitions is better than a premature abstraction

---

03/ OPEN FOR EXTENSION, CLOSED FOR MODIFICATION

Object Types should be extensible without modifying existing definitions. Extend by adding, never by mutating.

Rules: - No version suffixes on Object Type names (`Message_v2` is an anti-pattern) — add properties or deprecate fully - Use Interfaces for polymorphism — behavioral contracts without shared inheritance hierarchies - Mark maturity levels: **Experimental** (unfinished), **Active** (stable), **Deprecated** (no production use) - Extensible code lists use the LackingCodeValue pattern — the code list itself remains closed, extension is through a defined escape hatch - Struct types group related properties into composite value objects that can be reused without modifying the parent type

---

04/ PRODUCER EXTENDS, CONSUMER SUPER (PECS) — COVARIANCE AND CONTRAVARIANCE

Data producers should emit the most specific type possible. Data consumers should accept the most general type they can work with.

Role	Variance	Foundry Example
Producer (backing dataset, pipeline)	Covariant — use specific types	Pipeline emits `MilitaryAircraft` (specific Object Type)
Consumer (application, workflow, AIP agent)	Contravariant — accept general types	Workshop reads via `Aircraft` Interface (general contract)

Rules: - Backing datasets and pipelines should produce the most specific Object Types with full property sets - Applications and consumers should read through Interfaces that define only the properties they need - Type-safe OSDK code should use Interface types for consumption and concrete Object Types for mutation - Action submission criteria should validate against the specific type, not the general interface

---

05/ COMPOSITION OVER INHERITANCE

Prefer composing Object Types through Links and shared properties over deep hierarchies. Foundry's type system is designed for flat composition, not deep inheritance trees.

Rules: - Model relationships through Link Types, not through inheritance hierarchies - Use Interfaces for shared shape (polymorphism) instead of abstract base Object Types - Use Shared Property Types and Struct Types for reusable property groups - Configure ALL meaningful Link Types — isolated Objects with no links signal poor design - Semantic link naming: use business-meaningful names, especially for same-type relationships (Manager/Direct Report, not Employee/Employee2) - Plural-side Link Type API names must be plural (`.subordinates.all()` not `.subordinate.all()`)

OBJECT TYPE MODELING RULES

Primary Keys

- **Must be string type** — no exceptions
- Must be inherently unique, derived only from object properties
- Must be a dedicated `id` column even when other unique columns exist
- Composite IDs should not be hashed — maintain readability for debugging
- Never infer Object Type properties from the primary key itself

Foreign Keys

- Follow naming pattern: `{foreign_object_type}_id` or `{link_api_name}_{foreign_object_type}_id`

Naming Conventions

- No abbreviations — spell out fully (Aircraft not AC, Cost Average not Cost AVG)

- No tag prefixes — use Groups instead (prefixes pollute API names)
- Consistent naming across scripts, datasets, and Object Types
- Event timestamps: `{verbed}_at_timestamp` (e.g., `created_at_timestamp`)
- Event authors: `{verbed}_by_user` (e.g., `created_by_user`)

Editability

- Do not mark Object Types as editable if backed by immutable truth sources or pipeline-generated datasets

Actions

- Restrict submission by user groups and validate business sense
- Turn off Revert Action unless explicitly needed — side-effects from Automate or external functions are hard to reverse

Metadata

- Assign Point of Contact (primary maintainer)
- Set health checks on backing datasets
- Mark maturity level (Experimental / Active / Deprecated)
- Configure aliases for alternate names used across business units
- Document Object Types, Actions, and Properties thoroughly

ANTI-PATTERNS

Anti-Pattern	Why It's Wrong	Correct Approach
Syncing raw source data directly to Ontology	Ontology is for decisions, not data dumping	Clean → Integrate → Ontology pipeline
Version suffixes on Object Type names	Creates fragmentation and API instability	Add properties or deprecate fully
Tag prefixes in names	Embeds into API names permanently	Use Groups for organization
Non-inherent primary keys	Rankings and runtime UUIDs break determinism	Derive from object properties only

Anti-Pattern	Why It's Wrong	Correct Approach
Making computed properties editable	Creates conflict between pipeline and manual edits	Separate computed from editable
Deep inheritance hierarchies	Tight coupling, hard to extend	Composition + Interfaces
Isolated Objects with no Links	Signals the object doesn't participate in decisions	Configure all meaningful relationships

APPLICABILITY

These principles apply to: - GDAP doctrine ontology (DoctrineElement, AlignmentPair, etc.) - MIM information model (MIMClass, MIMEnumeration, etc.) - All Foundry backend code generation - All OSDK-based application development - Any new Object Type, Link Type, Action, or Interface definition

Non-compliance must be documented with justification and a remediation plan.

SEE ALSO

- CDA Agents: Ontology Engineer — CDA ontology engineering doctrine
- CDA Core Principles — Principal 7 (Nine Canonical Object Type Varieties)
- GDAP: DoctrineElement → Foundry Mapping