46 · Domain Driven Design

Flutter Practice Notes

Contents

46 · Domain-Driven Design

Introduction

This module covers **Domain-Driven Design (DDD)** for Flutter: the **strategic** side (ubiquitous language, bounded contexts, context mapping) and the **tactical** building blocks (entities, **value objects**, **aggregates** + aggregate roots, domain services, **domain events**, DDD-flavored repositories, application services), plus the crucial judgment of **when DDD is worth it**. It walks fundamentals → tactical patterns → aggregates/invariants → repositories/events → a capstone modeling one bounded context. It deepens the domain layer of Clean Architecture (Module 40) and pairs with modular/feature-first (module/feature per bounded context — Module 44/Module 45).

Why this module exists

Most apps have simple domains where CRUD + basic entities suffice — but **complex domains** (finance, logistics, insurance, healthcare) have rich rules, ambiguous terminology, and tangled invariants that a thin model can't tame. DDD is the toolkit for **taming domain complexity**: a shared **ubiquitous language**, explicit **bounded contexts**, and a rich model (aggregates enforcing invariants, value objects making illegal states unrepresentable). Applied where warranted it's transformative; applied to a CRUD app it's overkill — so the module teaches both the patterns and the discernment.

Module map

#	File	Topic	Level
1	01_ddd_fundamentals.md	Ubiquitous language, bounded contexts, context mapping, when DDD pays	🔴
2	02_tactical_building_blocks.md	Entities, value objects, domain services, factories	🔴
3	03_aggregates_and_invariants.md	Aggregates, aggregate roots, consistency boundaries, invariants	🔴
4	04_repositories_and_domain_events.md	DDD repositories, domain events, application services	🔴
5	05_ddd_integration.md	Capstone: model one bounded context	🔴

> **Cross-references:** Clean Architecture (domain layer): Module 40. Feature-first / modular (context = feature/module): Module 44/Module 45. OOP (entities/objects): Module 03. Error handling (`Result`/invariants): Module 38. Immutability/value objects: Module 02.

Prerequisites

40 Clean Architecture (domain layer), 03 OOP, 02 Advanced Dart (immutability), 38 Error Handling (`Result`), 44/45 (contexts as features/modules).

What you'll be able to do after this module

- Build a ubiquitous language and split a domain into bounded contexts (+ context map).
- Model with entities, value objects, aggregates, domain services, and factories.
- Define aggregate roots + consistency boundaries that enforce invariants.
- Use DDD repositories, domain events, and application services correctly.
- Decide when DDD is worth it and apply it proportionally in Flutter.

Capstone

Bounded-context model: Model an "Ordering" context — value objects (`Money`, `Quantity`, `Address`), entities, an `Order` **aggregate** (root enforcing invariants: totals, status transitions), a domain service, a DDD **repository interface** (aggregate-granular), and **domain events** (`OrderPlaced`) — pure Dart, unit-tested, with a documented ubiquitous language and a note on how it maps to a feature/module.

Summary

DDD tames complex domains via a shared ubiquitous language, explicit bounded contexts, and a rich tactical model (value objects, entities, aggregates enforcing invariants, domain events, repositories). It deepens Clean Architecture's domain layer and maps to features/modules — powerful for complex domains, overkill for CRUD, so applied with judgment.

DDD Fundamentals (Language, Contexts, When)

> DDD's strategic core is about **taming complexity through language and boundaries**: build a **ubiquitous language** (one shared vocabulary used by domain experts, code, and conversations — no translation layer), split the domain into **bounded contexts** (explicit boundaries where a term has *one* precise meaning, each with its own model), and map how contexts relate (**context map**). The tactical patterns (aggregates, value objects) serve this. The essential judgment: DDD pays for **complex, rule-rich domains**, and is **overkill for CRUD** — so apply it where the domain complexity justifies the investment.

Introduction

This file covers strategic DDD — ubiquitous language, bounded contexts, context mapping — and the when-it-pays decision. These strategic ideas are more important than any single pattern; the tactical building blocks (later files) exist to serve them.

Why this concept exists

Complex domains fail not from missing patterns but from **ambiguous language** ("order" means five different things) and **one giant model** trying to serve every use. DDD fixes this by aligning code with a precise shared language and carving the domain into contexts, each internally consistent. Without this, models tangle, terms collide, and business rules scatter.

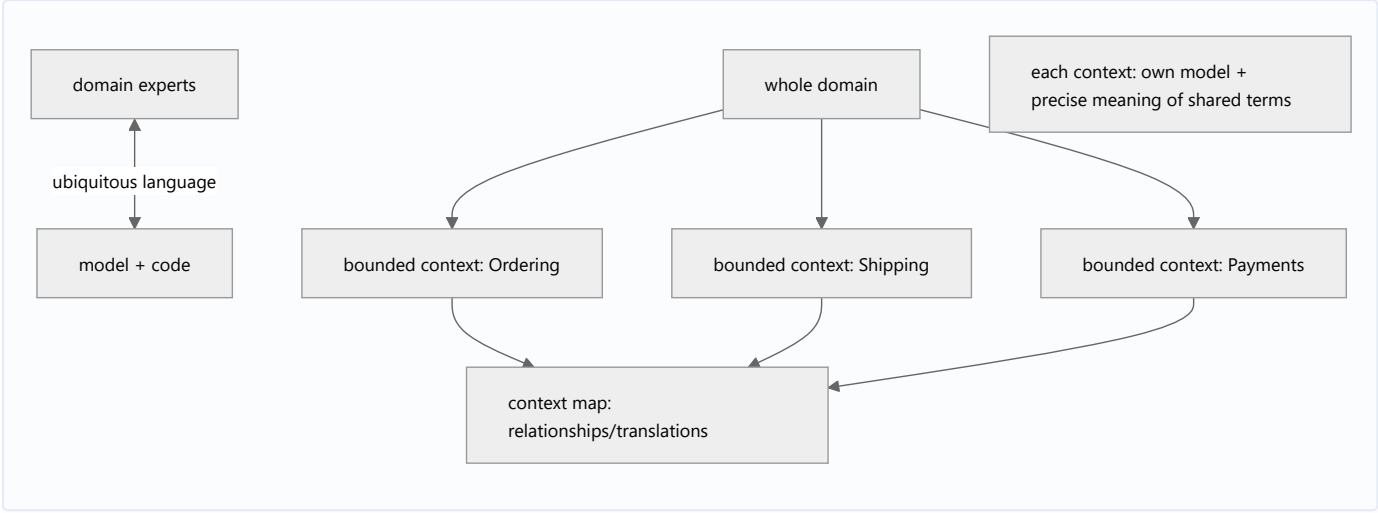
Real-world analogy

A **hospital** has many contexts using the word "patient" differently: in **billing** a patient is an account with insurance; in **clinical care** a patient is a body with symptoms; in **scheduling** a patient is a time-slot occupant. Forcing one "Patient" model to serve all three creates a monstrosity. DDD says: **each department (bounded context) keeps its own precise meaning**, everyone in that department speaks the **same words** (ubiquitous language), and a **directory (context map)** describes how departments hand off.

Problem Statement

For an e-commerce domain, identify bounded contexts (catalog, ordering, shipping, payments), define the ubiquitous language within one, note where the same term means different things across contexts, and decide whether the domain is complex enough to justify DDD. You'll do strategic modeling + a go/no-go call.

Internal Working



- **Ubiquitous language**: a **single, rigorous vocabulary** shared by developers and domain experts, used in **conversation, docs, and code** (class/method names *are* the language). No dev-vs-business translation. When a term is fuzzy, you refine it *with the experts* — the language and model co-evolve. This is DDD's beating heart.
- **Bounded context**: an **explicit boundary** within which a model + language are **internally consistent** — a term has **one precise meaning**. The same word ("customer," "product," "order") may mean **different things in different contexts**, and that's fine *because* they're separate models. A bounded context is where you'd draw a **feature/module boundary** (Module 44/Module 45).
- **Why boundaries**: one universal model for a big domain collapses under conflicting meanings/rules. Splitting into contexts lets each model be **small, precise, and consistent**, evolving independently.
- **Context mapping**: describe **how contexts relate** and translate across boundaries — patterns like **Shared Kernel** (shared subset), **Customer/Supplier**, **Conformist**, **Anti-Corruption Layer (ACL)** (translate an external/other context's model so it doesn't pollute yours), **Open Host/Published Language**. The map is the strategic architecture of the domain.
- **Strategic > tactical**: getting **language + boundaries** right matters more than any aggregate/value-object detail; tactical patterns implement a well-chosen context.
- **When DDD pays (the judgment)**:
  - **Yes**: **complex, rule-rich, evolving** domains (finance, logistics, insurance, healthcare, marketplaces) with domain experts and long lifespan — where miscommunication + tangled rules are the real cost.
  - **No / overkill**: **CRUD**, simple/technical, or short-lived apps — the language/boundary/aggregate ceremony exceeds the payoff; use straightforward Clean/MVVM.

    - **Partial**: apply DDD **only to the complex core** ("core domain"), keep supporting/generic subdomains simple.
    - **Core vs supporting vs generic subdomains**: invest DDD effort in the **core domain** (competitive differentiator); buy/simplify **generic** ones (auth, notifications).
- **Flutter angle**: a Flutter client often models a **subset/view** of the domain; full DDD tactical modeling is usually most valuable when the app owns significant domain logic (offline-first, rich client rules) — otherwise the client mirrors server contexts. Bounded contexts still guide **feature/module** decomposition.

Memory Representation

Not runtime — a **conceptual map**: the ubiquitous language (glossary), the set of bounded contexts (each a self-consistent model), and the context map (relationships/ACLs). Code names embody the language.

Compiler Behavior

Ubiquitous-language naming makes code self-documenting; bounded contexts become module/package boundaries the compiler can enforce (Module 45).

Runtime Behavior

Not applicable — strategic modeling shapes structure/communication, not runtime.

Flutter Engine Behavior

Not applicable.

Dart VM Behavior

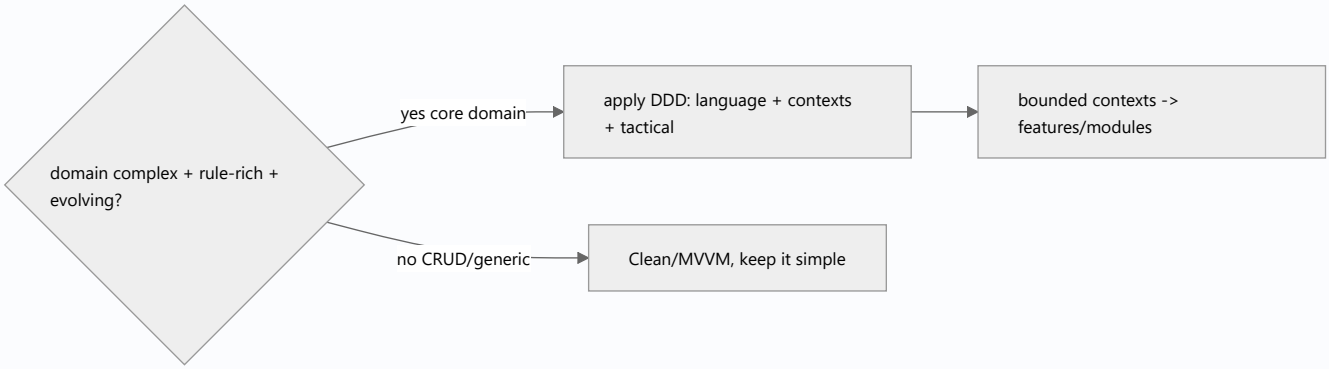
Not applicable.

Examples

```
E-commerce bounded contexts (same word, different meaning per context):
  Catalog context:   "Product" = sellable listing (name, price, description, images)
  Ordering context:  "Product" = an ordered line item (sku, qty, price-at-time)
  Shipping context:  "Product" = a physical package (weight, dimensions, fragile?)
  -> three DIFFERENT models; do NOT force one universal "Product"

Ubiquitous language (Ordering): Order, LineItem, place(), cancel(), OrderPlaced (event)
  -> these exact terms appear in code, docs, and expert conversations.
```

```
When-DDD-pays checklist:
  complex + rule-rich + evolving + long-lived + domain experts  -> YES (esp. the CORE domain)
  CRUD / simple / technical / short-lived                       -> NO (Clean/MVVM is enough)
  mixed -> apply DDD to the core domain only; keep generic subdomains simple
```

Diagrams



Common Mistakes

Mistake	Why it's wrong	Fix
Applying full DDD to a CRUD app	Overkill ceremony	Use Clean/MVVM; DDD only for complex domains
One universal model for the whole domain	Terms collide, model collapses	Split into bounded contexts
Dev/business translation layer	Miscommunication, drift	Ubiquitous language everywhere (code included)
Ignoring context relationships	Uncontrolled coupling	Context map + ACL at boundaries
DDD everywhere equally	Wasted effort on generic parts	Invest in the core domain; simplify generic
Tactical patterns without strategy	Aggregates around a wrong model	Get language + boundaries first
Letting an external model pollute yours	Foreign concepts leak in	Anti-Corruption Layer translates

Best Practices

- Build a **ubiquitous language** with domain experts and use it **in code, docs, and conversation**; refine language + model together.
- Split the domain into **bounded contexts** (each a precise, consistent model); accept that shared terms mean **different things** across contexts; draw the **context map** (+ ACLs at boundaries).
- **Get strategy right before tactics**; map **bounded contexts to features/modules** (Module 44/Module 45).
- **Apply DDD proportionally**: to the **complex core domain**, not CRUD/generic subdomains — decide with the when-it-pays judgment.

Performance

Not a runtime concern. The "performance" is **communication + change-velocity**: a shared language reduces miscommunication defects; clean contexts localize change and evolve independently. Over-applying DDD *slows* teams (ceremony) — proportionality is the efficiency lever.

Advantages / Disadvantages

- **+** Tames complex domains, shared precise language (fewer defects), independent evolvable contexts, aligns code with business, guides module boundaries.
- **−** Significant investment (experts, modeling), overkill for simple domains, requires ongoing language discipline, easy to misapply (tactics without strategy).

Interview Questions

1. 🟢 **What is a ubiquitous language?** — One rigorous vocabulary shared by developers and domain experts, used in code, docs, and conversation — no translation layer between business and code.
2. 🟢 **What is a bounded context?** — An explicit boundary where a model + language are internally consistent (a term has one meaning); the same word can mean different things in different contexts.
3. 🟡 **Why not one universal model for the whole domain?** — Conflicting meanings/rules make it collapse; bounded contexts keep each model small, precise, and consistent.
4. 🟡 **What is context mapping (and an ACL)?** — Describing how contexts relate/translate; an Anti-Corruption Layer translates another context's/external model so it doesn't pollute yours.
5. 🟡 **When is DDD worth it, and when overkill?** — Worth it for complex, rule-rich, evolving core domains with experts; overkill for CRUD/simple/generic — use Clean/MVVM there.
6. 🔴 **Why is strategy more important than tactics in DDD?** — Aggregates/value objects around the wrong language/boundaries don't help; getting language + contexts right is what tames complexity.
7. 🔴 **How do bounded contexts relate to features/modules in Flutter?** — A bounded context is a natural feature/module boundary; the context map guides decomposition and cross-module contracts.

Senior Engineer Tips

- Start with the language and the boundaries, with domain experts in the room; the tactical patterns are worthless if they model the wrong words or straddle contexts.
- Resist one-model-to-rule-them-all: let "customer/product/order" mean different things per context — that separation is the feature, not a bug.
- Apply DDD to the core domain only and keep generic subdomains (auth, notifications) simple; spending DDD effort uniformly is a classic waste.

Architect Perspective

Strategic DDD is domain architecture: a shared language that eliminates translation loss, and bounded contexts that keep each model precise and independently evolvable — mapping directly onto features/modules and their contracts. It's the intellectual complement to Clean/feature-first/modular: those give you *where* code lives; DDD gives you *what the domain actually is* and *how to carve it*. Its power is proportional to domain complexity, so the architect's job is to apply it to the complex core and keep the rest simple (02_tactical_building_blocks.md, Module 44, Module 45).

Summary

- Strategic DDD: a **ubiquitous language** (shared, in-code) + **bounded contexts** (precise, consistent models; same term differs across contexts) + a **context map** (relationships/ACLs).
- Strategy > tactics; bounded contexts map to features/modules.
- Apply DDD to **complex, rule-rich core domains**; it's **overkill for CRUD/generic** — decide proportionally.

Revision Notes

- Ubiquitous language: one vocabulary in code/docs/conversation (names = language), co-evolved with experts.
- Bounded context: explicit boundary, internally consistent model, one meaning per term (differs across contexts); maps to feature/module.
- Context map: Shared Kernel/Customer-Supplier/Conformist/ACL/Published Language; strategy > tactics.
- When DDD pays: complex/rule-rich/evolving core domain → yes; CRUD/simple/generic → no (Clean/MVVM); invest in core, simplify generic.

Practice Questions

1. How does the same term differ across bounded contexts, and why is that OK?
2. Why is the ubiquitous language central to DDD?
3. When is DDD overkill, and what do you use instead?

Coding Questions

1. Draft a ubiquitous-language glossary for one bounded context.
2. Split a domain into bounded contexts + a context map (with an ACL).
3. Make a go/no-go DDD decision for two given apps with justification.

Mini Project

Strategic model (Flutter/domain): For an e-commerce domain, identify bounded contexts (catalog/ordering/shipping/payments), write a ubiquitous-language glossary for the Ordering context, note where shared terms (product/customer) mean different things across contexts, draw a context map (with one ACL), and make a DDD go/no-go call for the core vs a generic subdomain. Acceptance: contexts identified + a context map; ubiquitous glossary for one context; cross-context term differences noted; ACL identified; proportional DDD decision (core vs generic) justified.

Tactical Building Blocks (Entities, Value Objects, Services)

> Within a bounded context, DDD models with distinct building blocks: **entities** (things with a persistent **identity** that changes over time — a `Customer`, an `Order`), **value objects** (immutable things defined **only by their attributes**, with no identity — `Money`, `Email`, `DateRange`; two are equal if their values match), **domain services** (stateless operations that don't naturally belong to a single entity/VO — `TransferFunds`), and **factories** (encapsulate complex creation, ensuring a valid object is born). The highest-leverage habit: **push rules into value objects** so illegal states are unrepresentable (a `Money` can't be negative currency-mismatched; an `Email` is always valid).

Introduction

This file covers the tactical vocabulary that populates a bounded context: entities vs value objects (the key distinction — identity vs value), domain services, and factories. These implement the model the strategic layer defined (01_ddd_fundamentals.md).

Why this concept exists

Complex domains need precise object semantics. Conflating identity and value (making everything an entity) loses the power of immutable, self-validating value objects; scattering rules across the UI/services instead of into VOs/entities makes them untestable and inconsistent. These building blocks give each concept the right shape so rules live in the model.

Real-world analogy

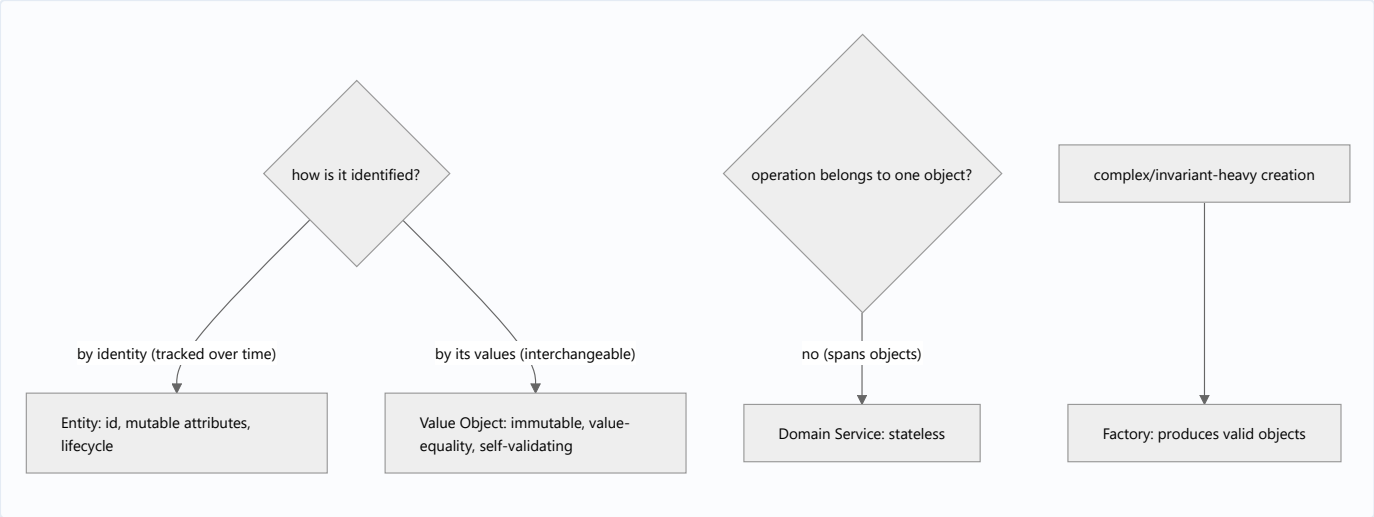
An **entity** is a **person**: they have a persistent identity (a passport number) even as their attributes change (address, hair color) — you track *the same person* over time. A **value object** is a **banknote's value**: a $20 is defined entirely by "20 USD" — one $20 is interchangeable with another (no identity), and it's immutable (you don't mutate a $20 into a $50; you get a different value). A **domain service** is a **currency exchange desk** — an operation involving multiple values that belongs to no single note.

Problem Statement

Model an ordering context's pieces: decide which concepts are entities (identity) vs value objects (value), make value objects self-validating/immutable (`Money`, `Quantity`, `Email`), add a domain service for a multi-entity operation, and use a factory for complex creation. You'll classify +

implement the building blocks.

Internal Working



- **Entity**: has a **stable identity** (an id) that persists across attribute changes; **mutable** over its lifecycle; equality is by **identity**, not attributes (two customers with the same name are different customers). Model things you **track over time**.
- **Value object (VO)**: **immutable**, defined **entirely by its attributes**, **no identity**; equality is by **value** (`Money(20,'USD') == Money(20,'USD')`). **Self-validating** in the constructor (an `Email` that exists is valid; a `Money` enforces currency/non-negativity). VOs make **illegal states unrepresentable** and are **freely shareable/side-effect-free**. **Prefer VOs** — most "fields" (money, dates, ranges, quantities, ids, addresses) are better as VOs than primitives (avoids "primitive obsession").
- **Entity vs VO (the core call)**: ask **"do I care *which* one it is (identity), or only *what* it is (value)?"** Identity → entity; value → VO. Getting this right is the most consequential tactical decision.
- **Domain service**: a **stateless** operation expressing domain logic that **doesn't belong to a single entity/VO** (e.g., `TransferFunds(from, to, amount)` , `PricingService`). Named in the ubiquitous language; contains **domain rules**, not app orchestration (that's an application service — 04_repositories_and_domain_events.md). Don't overuse — prefer putting behavior **on entities/VOs**; use services only for genuinely cross-object logic.
- **Factory**: encapsulates **complex or invariant-heavy creation** so objects are **born valid** (a `create()` /named constructor that enforces rules, or a factory method on an aggregate root). Keeps construction logic out of callers.
- **Behavior belongs in the model**: put **rules/methods on entities and VOs** (rich model), not in procedural services or the UI — the anti-anemic-model principle (Module 40).
- **Dart fit**: VOs = immutable classes with value equality (`==` / `hashCode` , or `equatable` /records/ `freezed`), validation in the constructor (throw/ `Result`); entities = classes

with an `id` + identity equality; services/factories = plain classes/functions — all **pure Dart** (no framework).

Memory Representation

VOs are small immutable values (safe to share/cache; equal-by-value). Entities hold an id + mutable state. Services are stateless (no fields beyond dependencies). Factories are transient constructors. VOs replace scattered primitives, concentrating validation.

Compiler Behavior

Value equality requires overriding `==` / `hashCode` (or using `equatable` / `freezed` /records); constructors enforce validity (throw or return `Result`). Immutability via `final` fields/ `const`. All plain Dart (testable).

Runtime Behavior

VOs, once constructed, are always valid (validated at birth) and never mutate; entities mutate through methods that preserve their rules; services compute; factories reject invalid inputs at creation.

Flutter Engine Behavior

None — pure domain (no widgets/IO).

Dart VM Behavior

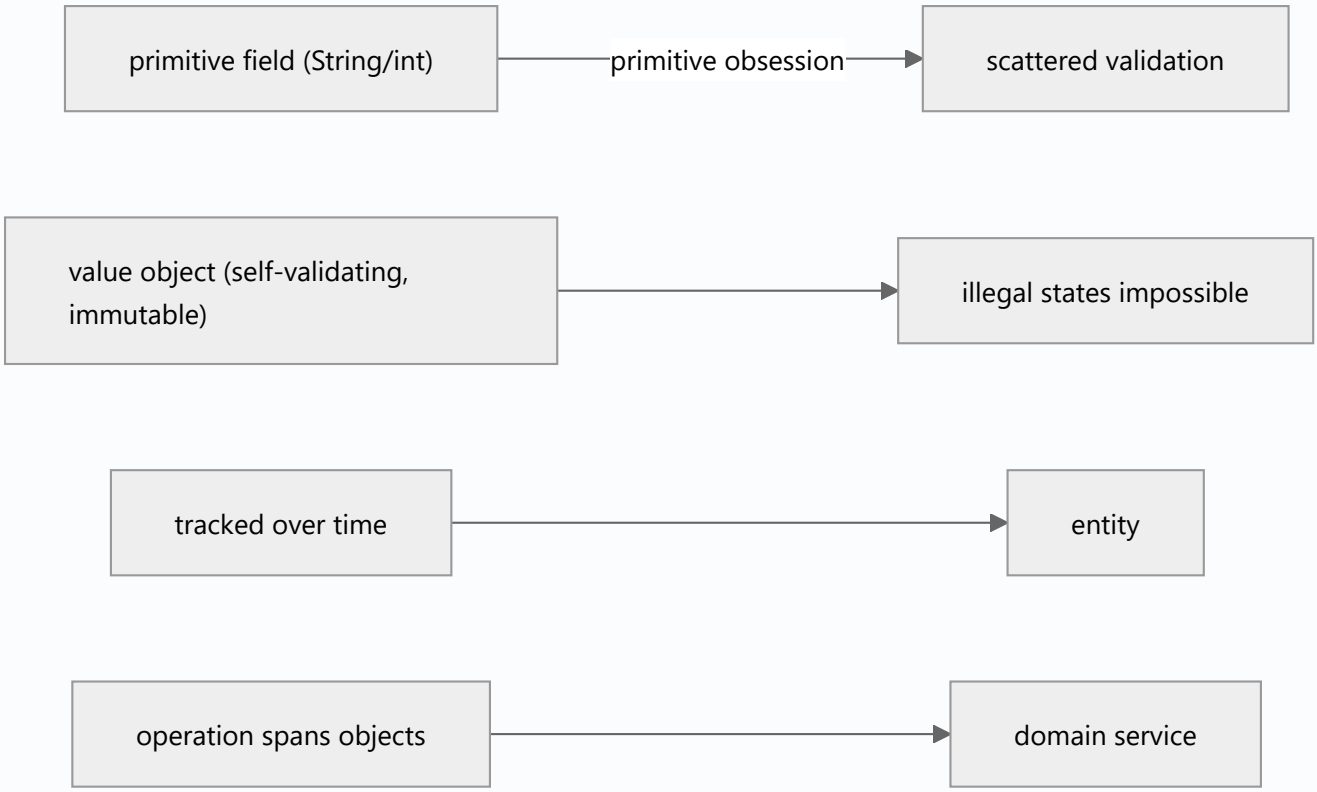
Immutable VOs are cheap, shareable, and comparison-friendly; pure-Dart building blocks give fast unit tests.

Examples

```
// VALUE OBJECT — immutable, value-equality, self-validating (illegal states impossible)
class Money {
  final int amountCents; final String currency;
  Money(this.amountCents, this.currency) {
    if (amountCents < 0) throw ArgumentError('Money cannot be negative');
    if (currency.length != 3) throw ArgumentError('Bad currency');
  }
  Money operator +(Money o) {
    if (currency != o.currency) throw ArgumentError('Currency mismatch'); // rule in the VO
```

```
    return Money(amountCents + o.amountCents, currency);
  }
  @override bool operator ==(Object o) =>
      o is Money && o.amountCents == amountCents && o.currency == currency; // value equality
  @override int get hashCode => Object.hash(amountCents, currency);
}

class Email {                                  // another self-validating VO (no invalid Email exists)
  final String value;
  Email(this.value) { if (!_re.hasMatch(value)) throw ArgumentError('Invalid email'); }
  static final _re = RegExp(r'^[\w.+-]+@[\w-]+\.\w+$');
}

// ENTITY — identity + mutable lifecycle; behavior (rules) on the entity
class Customer {
  final String id;                             // identity
  Email email; String name;                    // mutable attributes
  Customer(this.id, this.email, this.name);
  @override bool operator ==(Object o) => o is Customer && o.id == id; // identity equality
  @override int get hashCode => id.hashCode;
}

// DOMAIN SERVICE — stateless, spans objects (belongs to no single entity/VO)
class PricingService {
  Money priceFor(List<LineItem> items, Discount d) => /* domain pricing rules */ Money(0, 'USD');
}

// FACTORY — complex/invariant-heavy creation producing a valid object
class OrderFactory {
  Order createEmptyFor(Customer c) => Order.newFor(c.id); // encapsulates valid construction
}
```

Diagrams



Common Mistakes

Mistake	Why it's wrong	Fix
Primitive obsession (String email, int money)	Scattered/duplicated validation	Value objects (self-validating)
Everything an entity	Loses VO immutability/value-equality	VO when identity doesn't matter
Mutable value objects	Breaks value semantics/sharing	Immutable VOs (new instance on change)
Anemic entities (data bags)	Rules leak to services/UI	Put behavior/rules on entities/VOs
Overusing domain services	Procedural, anemic model	Prefer methods on entities/VOs; services only for cross-object
App orchestration in a domain service	Wrong layer	Domain rules only; orchestration → application service
No factory for complex creation	Invalid objects/duplicated logic	Factory ensures born-valid

Best Practices

- Distinguish **entities (identity, mutable, lifecycle)** from **value objects (value, immutable, self-validating, no identity)** by asking "do I care *which* or only *what*?"

- **Prefer value objects** over primitives (kill primitive obsession); validate in the constructor so **illegal states are unrepresentable**; use **value equality**.
- Put **behavior/rules on entities and VOs** (rich model); use **domain services** only for genuinely cross-object stateless logic (domain rules, not app orchestration).
- Use **factories** for complex/invariant-heavy creation (born-valid objects); keep all building blocks **pure Dart** (testable).

Performance

Immutable VOs are cheap and shareable; value-equality comparisons are fast for small VOs. Rich models add no runtime cost vs anemic ones. Pure-Dart building blocks give the fastest tests. Excessive tiny allocations are rarely an issue relative to correctness gains.

Advantages / Disadvantages

- + Precise semantics, self-validating VOs (fewer bugs), rich testable model, illegal states unrepresentable, clear cross-object logic via services.
- − More types (VOs for many fields), value-equality boilerplate (mitigated by `freezed` / `equatable` /records), judgment on entity-vs-VO and service usage.

Interview Questions

1. 🟢 **Entity vs value object?** — Entity has a persistent identity and mutable lifecycle (equality by id); a value object is immutable, has no identity, and is equal by its attributes.
2. 🟢 **Why prefer value objects over primitives?** — They centralize validation (self-validating), make illegal states unrepresentable, and carry domain meaning — avoiding "primitive obsession."
3. 🟡 **How do you decide entity vs VO?** — Ask whether you care *which* instance (identity → entity) or only *what* it is (value → VO).
4. 🟡 **When do you use a domain service?** — For stateless domain logic that spans multiple objects and belongs to no single entity/VO — not for app orchestration.
5. 🟡 **What's a factory for?** — Encapsulating complex/invariant-heavy creation so objects are born valid, keeping construction logic out of callers.
6. 🔴 **Why put behavior on entities/VOs instead of services?** — To avoid an anemic model — rules live with the data they govern, staying consistent and testable; services are for genuinely cross-object logic only.
7. 🔴 **How do you implement value equality + immutability in Dart?** — Immutable (`final` / `const`) fields + `==` / `hashCode` overrides (or `equatable` / `freezed` /records), validating in the constructor.

Senior Engineer Tips

- Wrap almost every "field" (money, email, dates, quantities, ids) in a self-validating value object; it eliminates a whole class of validation bugs and makes the model speak the domain.
- Keep entities rich (behavior + invariants on them) and reach for domain services sparingly; a domain full of services and anemic data bags is a procedural program wearing DDD clothes.
- Use `freezed` / `equatable` /records to remove value-equality boilerplate so the cost of "more VOs" is low and you actually create them.

Architect Perspective

The tactical building blocks give a bounded context a precise, rule-bearing model: value objects make invalid states impossible and carry domain meaning, entities track identity + lifecycle with behavior on them, domain services capture cross-object rules, and factories guarantee valid creation. This rich model is the substance of Clean Architecture's domain layer and the material aggregates organize into consistency boundaries. Preferring VOs and rich entities (over primitives and anemic bags) is the highest-leverage tactical habit (03_aggregates_and_invariants.md, Module 40).

Summary

- Entities = identity + mutable lifecycle (equality by id); value objects = immutable, value-equality, self-validating, no identity — prefer VOs over primitives.
- Domain services = stateless cross-object domain logic (not orchestration); factories = born-valid creation.
- Put behavior/rules on entities/VOs (rich model, illegal states unrepresentable); all pure Dart, testable.

Revision Notes

- Entity: identity (id), mutable, lifecycle, equality by id. VO: immutable, value-equality, self-validating (constructor), no identity — prefer over primitives (no primitive obsession).
- Entity-vs-VO: care *which* (identity→entity) or *what* (value→VO). Behavior/rules on entities/VOs (anti-anemic).
- Domain service: stateless, cross-object domain rules (not app orchestration). Factory: complex/invariant creation → born valid. All pure Dart; value equality via `==` / `hashCode` / `freezed` / `equatable` /records.

Practice Questions

1. Classify five concepts as entities or value objects and justify.

2. Why does a self-validating value object eliminate bug classes?
3. When is a domain service appropriate vs putting logic on an entity?

Coding Questions

1. Implement a self-validating, value-equal `Money` VO with `+`.
2. Model an entity with identity equality + behavior (a rule method).
3. Add a domain service for a cross-object operation and a factory for creation.

Mini Project

Tactical building blocks (Flutter/domain): For an ordering context, implement value objects (`Money`, `Quantity`, `Email` — immutable, self-validating, value-equality), entities with identity + behavior (`Customer`, `LineItem`), a domain service (`PricingService`) for a cross-object rule, and a factory for valid creation — all pure Dart, unit-tested. Acceptance: correct entity-vs-VO classification; VOs immutable + self-validating + value-equal (illegal states impossible); behavior on entities/VOs (no anemic bags); domain service only for cross-object logic; factory yields valid objects; unit-tested without a device.

Aggregates & Invariants

> An **aggregate** is a cluster of entities + value objects treated as **one consistency boundary**, accessed only through its **aggregate root** (a single entity that guards the whole cluster's **invariants**). Outsiders hold references to the **root only**; all changes go **through root methods** that keep the aggregate valid at all times (`order.addLine(...)` enforces "total = sum of lines," "can't modify a shipped order"). The design rules: keep aggregates **small**, enforce **invariants inside the boundary transactionally**, and reference **other aggregates by id** (not object) — so each aggregate is an independently consistent, loadable, savable unit.

Introduction

This file covers the pivotal DDD pattern: aggregates as consistency boundaries, the aggregate root as the single entry point and invariant guardian, and the rules for sizing aggregates and referencing across them. It organizes the tactical building blocks (02_tactical_building_blocks.md) into consistency units.

Why this concept exists

In a rich model, "which objects must always be consistent together, and who enforces that?" is the hard question. Without aggregates, invariants (an order's total, a project's task limits) get enforced ad hoc across the codebase and drift. Aggregates answer it: draw a **consistency boundary**, funnel changes through **one root**, and guarantee the cluster is **always valid** — making consistency a structural property, not a hope.

Real-world analogy

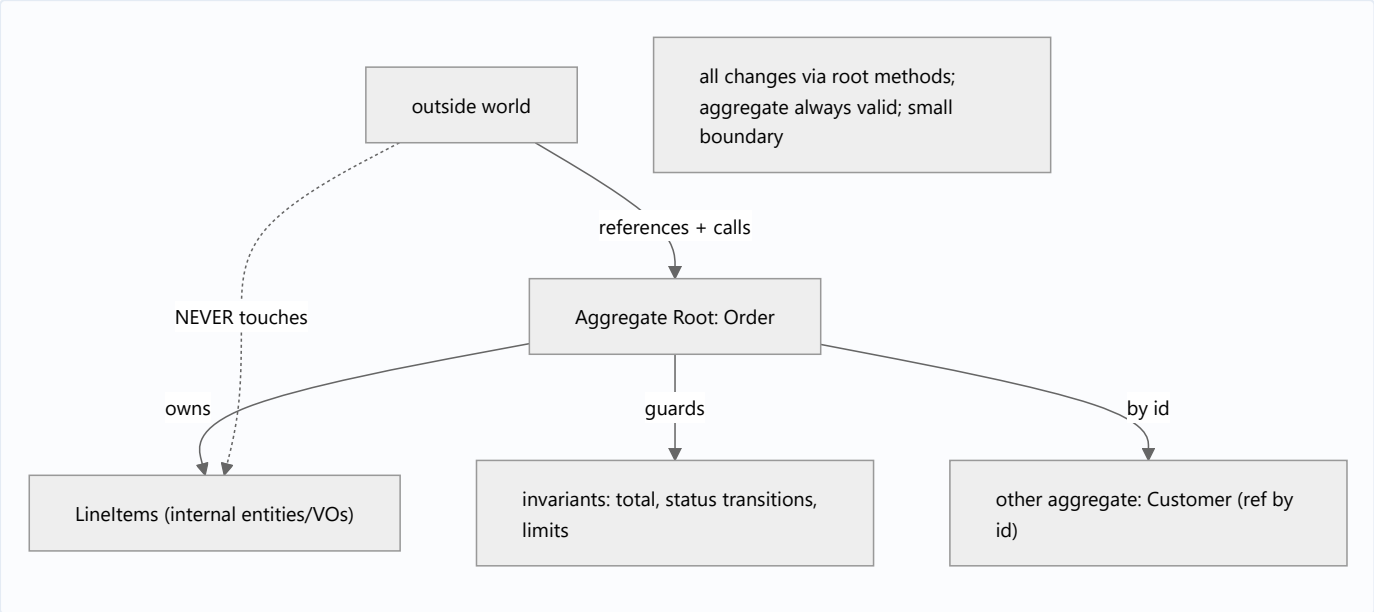
An aggregate is a **shopping cart with a single checkout clerk (the root)**: you can't reach into the cart and rearrange items directly — every add/remove goes **through the clerk**, who re-checks the rules (item limits, price totals) on every change and never lets the cart reach an invalid state. Other carts (aggregates) are referenced by **their receipt number (id)**, not by physically holding them. Keep each cart manageable (small); a cart holding the entire store is unmanageable.

Problem Statement

Model an `Order` aggregate: the `Order` root guards line items + status, enforcing invariants (total = sum of lines; only `draft` orders can add lines; `placed` → `shipped` transitions are valid), exposing only intent methods, referencing `Customer` by id, and keeping the aggregate small. You'll define

the boundary, root, invariants, and cross-aggregate references.

Internal Working



- **Aggregate**: a **cluster of related entities + VOs** that must stay **consistent together**, treated as a **single unit** for loading, saving, and rule-enforcement — the **consistency/transaction boundary**.
- **Aggregate root**: **one entity** that is the **sole entry point**. Outsiders reference/obtain **only the root**; **internal members are not directly accessible** from outside. The root **enforces all invariants** for the aggregate.
- **All changes through the root**: mutations happen via **root methods** (intent-named: `addLine`, `place`, `ship`), each of which **checks/preserves invariants**. No external code mutates internal parts directly — that's how consistency is guaranteed.
- **Invariants (the point)**: business rules that must **always** hold within the aggregate (an order's total equals its lines; can't add lines to a shipped order; a project can't exceed N members). The root guarantees they hold **after every operation** — the aggregate is **never invalid**.
- **Keep aggregates small**: prefer **small aggregates** (often just a root + a few VOs/child entities). Large aggregates cause contention, big loads, and blurred consistency. If two things don't need to be **immediately consistent**, they belong in **separate aggregates** (eventual consistency between them).
- **Reference other aggregates by id, not by object**: an `Order` holds a `customerId`, **not** a `Customer` object. This keeps aggregates **independently loadable/savable**, prevents huge object graphs, and clarifies boundaries. Cross-aggregate consistency is **eventual** (via domain events — 04_repositories_and_domain_events.md), not transactional.
- **One transaction per aggregate**: a single transaction should modify **one aggregate**; changes spanning aggregates happen across transactions (eventually consistent). This bounds

locking/contention.

- **Repositories are aggregate-granular**: you get/save **whole aggregates** through a repository per aggregate root (04_repositories_and_domain_events.md).
- **Dart implementation**: the root is an entity whose **internal collections are encapsulated** (private lists, `UnmodifiableListView` getters), mutations only via methods that **validate invariants** (throw or return `Result`), holding **ids** to other aggregates.

Memory Representation

An aggregate is an object graph rooted at the root entity: the root + owned child entities/VOs (encapsulated), plus **ids** (not objects) to other aggregates. It's loaded/saved as a unit; internals aren't exposed. Invariants are code in the root guarding state transitions.

Compiler Behavior

Encapsulation (private fields, unmodifiable getters, no setters) makes bypassing the root a compile-time impossibility for outsiders. Invariant checks run in root methods (throw/ `Result`).

Runtime Behavior

Every mutation passes through the root and re-establishes invariants, so the aggregate is **always valid** at rest. A rule violation is rejected (throw/ `Result`) rather than producing an invalid state. Cross-aggregate updates happen in separate steps (eventual consistency).

Flutter Engine Behavior

None — pure domain.

Dart VM Behavior

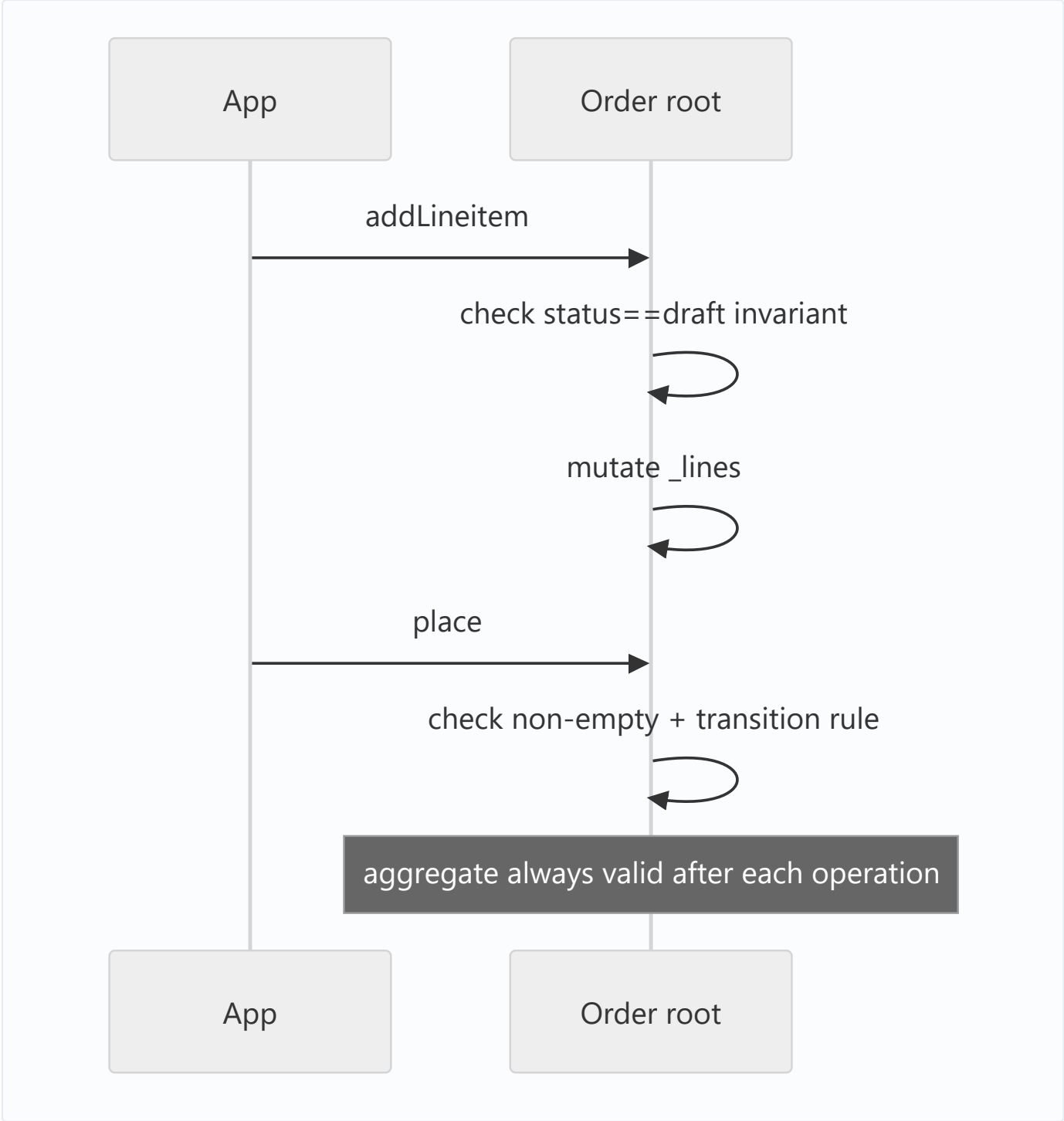
Pure-Dart aggregates give fast, exhaustive invariant tests (no device). Small aggregates keep object graphs and loads light.

Examples

```
// AGGREGATE ROOT — Order guards its lines + status; all changes via methods; refs by id
class Order {
  final String id;
  final String customerId;              // reference OTHER aggregate by id (not Customer object)
  final List<LineItem> _lines = [];     // encapsulated internal members
  OrderStatus _status = OrderStatus.draft;
```

```
  Order.newFor(this.id, this.customerId);

  List<LineItem> get lines => List.unmodifiable(_lines);   // no external mutation
  OrderStatus get status => _status;
  Money get total => _lines.fold(Money(0,'USD'), (s, l) => s + l.amount); // derived invariant

  void addLine(LineItem line) {          // intent method enforces invariants
    if (_status != OrderStatus.draft) {
      throw StateError('Cannot modify a $_status order');   // invariant: only draft is editable
    }
    _lines.add(line);
  }

  void place() {
    if (_lines.isEmpty) throw StateError('Cannot place an empty order');   // invariant
    _transition(OrderStatus.placed);
  }
  void ship() => _transition(OrderStatus.shipped);

  void _transition(OrderStatus to) {     // valid status transitions only (invariant)
    const allowed = {
      OrderStatus.draft: {OrderStatus.placed},
      OrderStatus.placed: {OrderStatus.shipped, OrderStatus.cancelled},
    };
    if (!(allowed[_status]?.contains(to) ?? false)) {
      throw StateError('Illegal transition $_status -> $to');
    }
    _status = to;
  }
}
// Outsiders hold an Order (root) and call addLine/place/ship — never touch _lines directly.
```

Diagrams


App
Order root
addLineitem
check status==draft invariant
mutate _lines
place
check non-empty + transition rule
aggregate always valid after each operation
App
Order root

Common Mistakes

Mistake	Why it's wrong	Fix
Mutating internal members from outside	Bypasses invariants → invalid state	All changes via root methods; encapsulate internals
Referencing other aggregates by object	Huge graphs, blurred boundaries	Reference by id
Huge aggregates	Contention, big loads, unclear consistency	Keep small; split by immediate-consistency need
Multiple aggregates per transaction	Contention/coupling	One aggregate per transaction (eventual across)
Exposing internal collections (mutable)	External mutation	Unmodifiable getters, no setters
Anemic root (rules elsewhere)	Invariants drift	Invariants enforced in the root
Expecting immediate cross-aggregate consistency	Not the model	Eventual consistency (domain events)

Best Practices

- Define an **aggregate = consistency boundary**; expose **only the root**, encapsulate internals, and route **all changes through root methods** that **enforce invariants** (always-valid).
- Keep aggregates **small** (root + a few members); put things that need **immediate consistency** together, everything else in **separate aggregates**.
- **Reference other aggregates by id** (not object); target **one aggregate per transaction**; use **eventual consistency + domain events** across aggregates.
- Enforce invariants **inside the boundary** (throw/ `Result`); make repositories **aggregate-granular** (04_repositories_and_domain_events.md); keep it **pure Dart** + unit-tested.

Performance

Small aggregates keep loads/saves and locking light (one aggregate per transaction reduces contention). By-id references avoid loading giant object graphs. Invariant checks are cheap. Large aggregates are the main perf/consistency hazard — split them.

Advantages / Disadvantages

- **+** Structural consistency (always-valid aggregate), clear boundaries, encapsulated invariants, bounded transactions/loads, testable rules.
- **−** Boundary-sizing judgment, by-id references (no navigation convenience), eventual consistency across aggregates (more complex flows), encapsulation discipline.

Interview Questions

1. 🟢 **What is an aggregate and its root?** — A cluster of entities/VOs forming one consistency boundary, accessed only through a single aggregate root that guards its invariants.
2. 🟢 **How are aggregates changed?** — Only through root methods (intent-named), each enforcing invariants — outsiders never mutate internal members directly.
3. 🟡 **Why reference other aggregates by id, not object?** — To keep aggregates independently loadable/savable, avoid huge graphs, and keep boundaries clear (cross-aggregate consistency is eventual).
4. 🟡 **Why keep aggregates small?** — Large aggregates cause contention, big loads, and blurred consistency; only immediately-consistent data belongs together.
5. 🟡 **What's the transaction rule?** — One transaction should modify one aggregate; changes across aggregates happen in separate transactions (eventual consistency).
6. 🔴 **How do you guarantee an aggregate is never invalid?** — All mutations go through root methods that check/preserve invariants and reject violations (throw/ `Result`) — encapsulation prevents bypass.
7. 🔴 **How does cross-aggregate consistency work then?** — Eventually, coordinated via domain events rather than a single transaction spanning aggregates.

Senior Engineer Tips

- Draw the consistency boundary by asking "what must be true together at all times?" — that set is one aggregate; everything else is another aggregate reached by id.
- Encapsulate ruthlessly (private collections, unmodifiable getters, no setters); an aggregate that leaks its internals can't guarantee its invariants.
- Default to small aggregates and eventual consistency across them; the instinct to make one big aggregate for convenience is the top cause of contention and invariant drift.

Architect Perspective

Aggregates turn invariants into a structural guarantee: a consistency boundary with a single guardian root, changed only through intent methods, referencing peers by id, kept small and transactionally bounded. This is where DDD's "always-valid model" lives, and it directly shapes persistence (aggregate-granular repositories), transactions (one aggregate each), and cross-aggregate flows (events, eventual consistency). Getting aggregate boundaries right is the most consequential tactical-design decision in a complex domain (02_tactical_building_blocks.md, 04_repositories_and_domain_events.md).

Summary

- Aggregate = consistency boundary accessed only via its root, which guards invariants; all changes go through root methods (always-valid).
- Keep aggregates small (only immediately-consistent data together); reference other aggregates by id; one aggregate per transaction; eventual consistency across.
- Encapsulate internals; enforce invariants inside; repositories are aggregate-granular; pure Dart + unit-tested.

Revision Notes

- Aggregate = cluster + consistency boundary; **root** = sole entry point + invariant guardian; all mutations via root methods (intent-named), always-valid.
- Keep small (immediate-consistency set only); reference other aggregates **by id**; **one aggregate per transaction**; cross-aggregate = eventual (domain events).
- Encapsulate internals (private, unmodifiable getters); enforce invariants (throw/ `Result`); aggregate-granular repositories; pure Dart, unit-tested.

Practice Questions

1. What determines which objects belong in one aggregate?
2. Why reference other aggregates by id and use one transaction per aggregate?
3. How does the root guarantee the aggregate is never invalid?

Coding Questions

1. Implement an `Order` aggregate root guarding line + status invariants (via methods).
2. Encapsulate internal collections (unmodifiable getters, no setters) and reference `Customer` by id.
3. Unit-test that illegal operations (add to shipped, empty place, bad transition) are rejected.

Mini Project

Order aggregate (Flutter/domain): Model an `Order` aggregate with a root enforcing invariants (total = sum of lines, only `draft` editable, valid status transitions, no empty place), exposing only intent methods (`addLine` / `place` / `ship` / `cancel`), encapsulating internal line items (unmodifiable getters), and referencing `Customer` by id. Unit-test that valid ops succeed and every invariant violation is rejected. Acceptance: single root entry point (internals encapsulated); all changes via methods enforcing invariants (always-valid); small aggregate; by-id cross-aggregate reference; illegal operations rejected in tests; pure Dart.

Repositories & Domain Events

> Two DDD patterns connect the aggregate model to the wider system: a **repository** provides the illusion of an **in-memory collection of aggregates** — one repository **per aggregate root**, getting/saving **whole aggregates** by identity (`orders.findById(id)` / `orders.save(order)`), hiding persistence entirely. **Domain events** capture **something meaningful that happened** in the domain (`OrderPlaced` , `PaymentReceived`) as immutable facts an aggregate emits; other parts of the system (often other aggregates/contexts) **react** to them — the mechanism for **eventual consistency** across aggregate boundaries. **Application services** orchestrate use cases (load aggregate → call its methods → save → publish events) without holding domain rules.

Introduction

This file covers the domain's boundary to persistence (repositories) and to the rest of the system (domain events), plus the thin orchestration layer (application services) that ties a use case together. It completes the tactical toolkit begun in the aggregates file (03_aggregates_and_invariants.md).

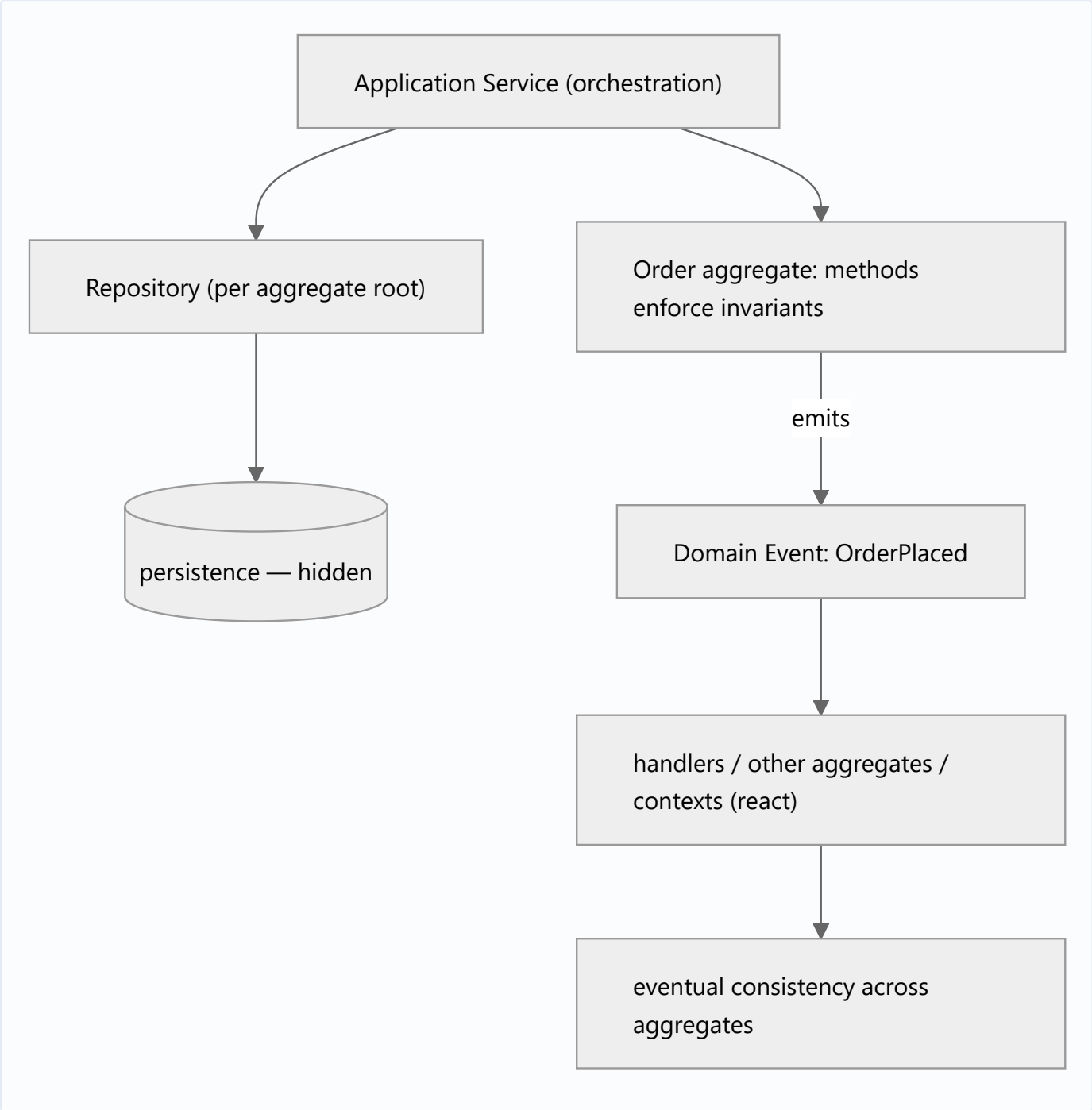
Why this concept exists

Aggregates need to be **persisted/retrieved** without leaking storage into the domain (repositories), and complex domains need to **coordinate across aggregate boundaries** without giant transactions (domain events → eventual consistency). Application services provide a place for **orchestration** so the domain stays pure and the presentation stays thin.

Real-world analogy

A **repository** is a **librarian for a specific kind of item**: you ask for a book by its id or hand one back to be shelved — you never see the shelving system (storage). A **domain event** is an **announcement on the PA** ("Order #42 has been placed") — a fact broadcast so interested departments (shipping, notifications) can react in their own time (eventual consistency). An **application service** is the **front-desk coordinator** who takes a request, fetches the right item from the librarian, has it do its thing, files it back, and posts the announcement — without knowing the item's internal rules.

Internal Working



- **Repository (DDD sense)**: an abstraction that behaves like an **in-memory collection of aggregates** — `findById`, `save`, sometimes query methods returning aggregate roots. **One repository per aggregate root** (Orders, Customers), operating on **whole aggregates** (load/save the root + its internals as a unit). The **interface lives in the domain**; the **impl in the data layer** (maps to DB/API — Module 40). It **hides persistence** so the domain never knows about SQL/HTTP.
  - *(Note: this is the DDD repository — collection-of-aggregates. The Clean-Architecture "repository" (Module 40) is a broader data-access abstraction; in a DDD app they align: aggregate-granular, domain-defined interface.)*

- **Domain event**: an **immutable fact** representing **something significant that happened** (`OrderPlaced(orderId, at)`, `FundsTransferred`). Named in **past tense** (ubiquitous language). An aggregate **records/emits** events as part of its behavior; a dispatcher **publishes** them after the aggregate is saved; **handlers react** (update another aggregate, notify, integrate). Events are the backbone of **eventual consistency** across aggregates/contexts (no giant cross-aggregate transaction — 03_aggregates_and_invariants.md).
- **Emitting events**: the aggregate root appends events to an internal list during operations; the application service **collects + dispatches** them **after a successful save** (so events reflect committed facts). Keep event payloads **small** (ids + key data), not whole aggregates.
- **Application service (use-case orchestration)**: a thin coordinator for a use case: **load** aggregate(s) via repositories → **invoke** aggregate methods (domain rules run there) → **save** → **publish** domain events. It **holds no domain rules** (those are in aggregates/domain services) and **no presentation** (that's the ViewModel). It's the DDD analog of a Clean **use case/interactor** (Module 40).
- **Transaction alignment**: an application-service operation typically = **one transaction on one aggregate** + emitting events; reactions to those events run in **their own transactions** (eventual).
- **Consistency model**: **inside an aggregate = strong/immediate** (root invariants); **across aggregates = eventual** (via events). This split is what keeps aggregates small + transactions bounded.
- **Dart fit**: repository = abstract class in domain (`Future<Order?> findById`, `Future<void> save(Order)`), impl in data; domain events = immutable classes; a simple event dispatcher/bus publishes to handlers; application service = plain Dart class using repositories + dispatcher, returning `Result`.

Memory Representation

Repository interface = domain abstraction; impl holds data sources/mappers. An aggregate holds a small list of pending domain events. The dispatcher holds handler registrations. Application services are stateless orchestrators holding injected repos + dispatcher.

Compiler Behavior

Repository/domain-event/handler are plain Dart types; the domain compiles without persistence/framework (interfaces only). Events are immutable value classes.

Runtime Behavior

App service: load → mutate aggregate (invariants enforced) → save (transaction) → publish collected events → handlers react (separate transactions/eventual). A failed save means no events are published (facts must be committed first).

Flutter Engine Behavior

None — domain/application layer is pure Dart; the client may subscribe to events for UI reactions if it owns domain logic.

Dart VM Behavior

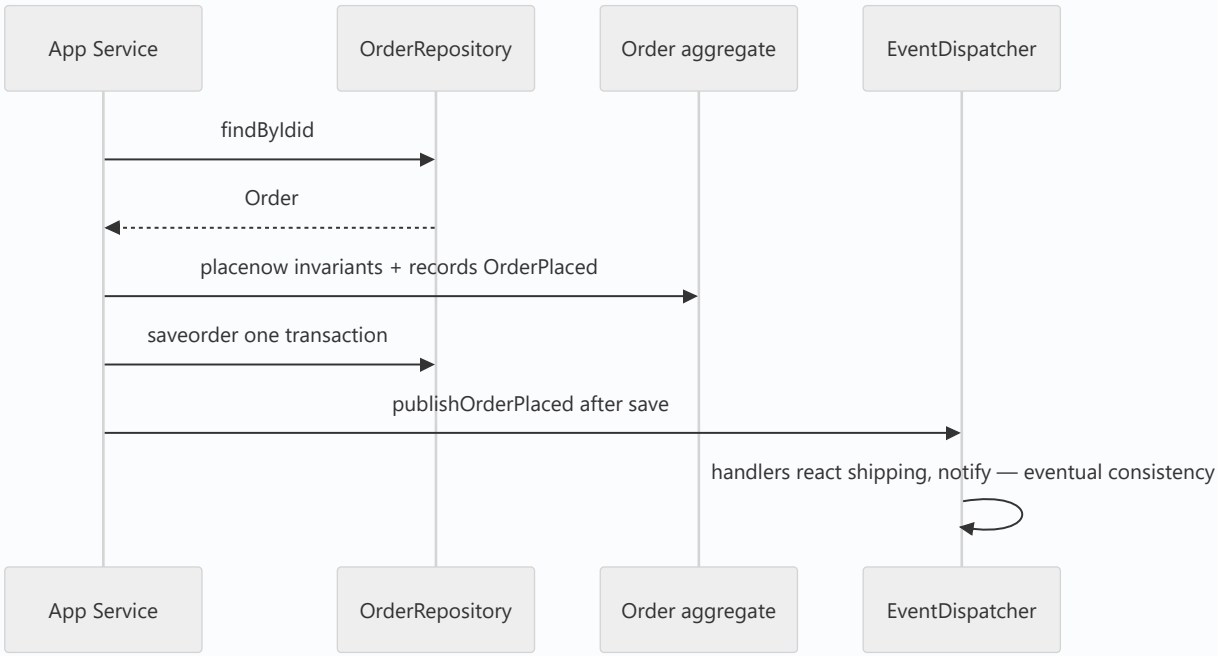
Pure-Dart repositories (interface)/events/services give fast unit tests (with fake repos + captured events); event dispatch is in-process (or bridged to a message bus in distributed systems).

Examples

```
// REPOSITORY — domain interface, one per aggregate root, whole-aggregate granularity
abstract class OrderRepository {
  Future<Order?> findById(String id);
  Future<void> save(Order order);       // persists the whole aggregate (root + internals)
}

// DOMAIN EVENT — immutable, past-tense fact
class OrderPlaced {
  final String orderId; final DateTime placedAt;
  const OrderPlaced(this.orderId, this.placedAt);
}

// AGGREGATE records events as part of behavior (from aggregates file)
class Order {
  final _events = <Object>[];
  List<Object> pullEvents() { final e = List.of(_events); _events.clear(); return e; }
  void place(DateTime now) {
    if (lines.isEmpty) throw StateError('empty order'); // invariant (strong, inside aggregate)
    _transition(OrderStatus.placed);
    _events.add(OrderPlaced(id, now));                  // record the fact
  }
  // ...
}
```

```
// APPLICATION SERVICE — orchestrates: load -> call -> save -> publish (no domain rules here)
class PlaceOrderService {
  final OrderRepository orders; final EventDispatcher events; final Clock clock;
  PlaceOrderService(this.orders, this.events, this.clock);

  Future<Result<void>> call(String orderId) async {
    final order = await orders.findById(orderId);
    if (order == null) return Failure(NotFoundFailure('order'));
    order.place(clock.now());          // domain rules run in the aggregate
    await orders.save(order);          // one transaction, one aggregate
    for (final e in order.pullEvents()) events.publish(e); // publish AFTER save (committed facts)
    return const Success(null);
  }
}
// Handler (elsewhere) reacts to OrderPlaced -> schedule shipping (its own aggregate/transaction) =
eventual consistency
```

Diagrams



Common Mistakes

Mistake	Why it's wrong	Fix
Repository per entity (not per aggregate)	Breaks aggregate consistency	One repo per aggregate root; whole-aggregate ops
Persistence details in the domain	Couples domain to storage	Interface in domain, impl in data
Publishing events before save	Broadcasts uncommitted facts	Publish after successful save
Fat event payloads (whole aggregates)	Coupling/bloat	Small events (ids + key data)
Domain rules in the application service	Anemic domain	Rules in aggregates/domain services
Presentation logic in app service	Wrong layer	Presentation in ViewModel
Cross-aggregate strong consistency (big txn)	Contention/coupling	Events → eventual consistency

Best Practices

- Define **one repository per aggregate root**, operating on **whole aggregates**, with the **interface in the domain** and impl in data (hides persistence).
- Emit **small, immutable, past-tense domain events** from aggregate behavior; **publish after a successful save**; use events for **cross-aggregate eventual consistency**.
- Keep **application services thin** (load → invoke aggregate → save → publish); **no domain rules** (aggregates/domain services) and **no presentation** (ViewModel).
- Align **one transaction ≈ one aggregate**; strong consistency **inside** aggregates, eventual **across**; keep it **pure Dart** + testable (fake repos, capture events).

Performance

Aggregate-granular repos + one-aggregate transactions bound locking/contention; small events keep dispatch cheap; eventual consistency avoids expensive cross-aggregate transactions. In-process events are near-free; distributed buses add latency but decouple. Pure-Dart layer = fast tests.

Advantages / Disadvantages

- **+** Persistence-ignorant domain, aggregate-consistent saves, decoupled cross-aggregate coordination (events), thin testable orchestration, bounded transactions.
- **−** Eventual-consistency complexity (ordering, retries, idempotent handlers), event/dispatcher plumbing, discipline to keep rules out of app services, publish-after-save care.

Interview Questions

1. 🟢 **What is a DDD repository?** — An abstraction that behaves like an in-memory collection of aggregates (findById/save), one per aggregate root, operating on whole aggregates, hiding persistence.
2. 🟢 **What is a domain event?** — An immutable, past-tense fact that something significant happened in the domain, emitted by an aggregate for others to react to.
3. 🟡 **Why publish events after save, not before?** — Events represent committed facts; publishing before a successful save could broadcast something that didn't actually happen.
4. 🟡 **How do domain events enable eventual consistency?** — Cross-aggregate reactions run in their own transactions triggered by events, avoiding a single transaction spanning aggregates.
5. 🟡 **What belongs in an application service vs the domain?** — App service orchestrates (load/invoke/save/publish) with no domain rules; rules live in aggregates/domain services; presentation lives in the ViewModel.
6. 🔴 **Why one repository per aggregate root, not per entity?** — Aggregates are the consistency/loading unit; per-entity repos would let you save internals independently and break invariants.
7. 🔴 **How does the DDD repository relate to the Clean-Architecture repository?** — They align: aggregate-granular, domain-defined interface, data-layer impl — DDD just specifies the granularity (per aggregate root) and semantics (collection of aggregates).

Senior Engineer Tips

- Make repositories aggregate-granular and load/save whole aggregates; per-entity repos silently reintroduce the invariant drift aggregates exist to prevent.
- Record events inside aggregate methods but publish them from the application service only after the save commits; and make handlers idempotent, since events may be redelivered.
- Keep application services skeletal (orchestration only); the moment a rule creeps in there, the domain is going anemic — push it back into the aggregate/domain service.

Architect Perspective

Repositories and domain events are the aggregate model's connections outward: repositories hide persistence and preserve aggregate-granular consistency; domain events + application services coordinate across aggregates via eventual consistency without giant transactions. Together with aggregates they complete the tactical toolkit — strong consistency inside boundaries, eventual across, orchestration thin, domain pure. This maps cleanly onto Clean Architecture (repos/use cases) and modular/bounded-context boundaries (events crossing contexts, often via an ACL) (03_aggregates_and_invariants.md, Module 40, Module 45).

Summary

- Repository = in-memory-collection-of-aggregates abstraction, one per aggregate root, whole-aggregate ops, interface in domain / impl in data (hides persistence).
- Domain events = immutable past-tense facts emitted by aggregates; published after save; drive cross-aggregate eventual consistency.
- Application services orchestrate (load→invoke→save→publish), holding no domain rules/presentation; one transaction ≈ one aggregate.

Revision Notes

- Repository: per aggregate root, `findById` / `save` whole aggregates; interface in domain, impl in data (hides persistence); aligns with Clean repo.
- Domain event: immutable, past-tense fact; recorded in aggregate methods, published by app service **after save**; small payloads (ids); drives eventual consistency across aggregates.
- Application service: thin orchestration (load→invoke aggregate→save→publish), no domain rules/presentation; one txn ≈ one aggregate; strong inside/eventual across; idempotent handlers; pure Dart.

Practice Questions

1. Why one repository per aggregate root operating on whole aggregates?
2. Why publish domain events after save, and how do they enable eventual consistency?
3. What does an application service do, and what must it not contain?

Coding Questions

1. Define an `OrderRepository` interface (domain) + an application `PlaceOrderService`.
2. Emit an `OrderPlaced` event from the aggregate and publish it after save.
3. Unit-test the service with a fake repository + captured events (no device).

Mini Project

Repository + events + service (Flutter/domain): For the Order aggregate, define an `OrderRepository` interface (domain; whole-aggregate `findById` / `save`), have the aggregate emit `OrderPlaced`, and write a thin `PlaceOrderService` that loads → `place()` → saves → publishes the event (after save), returning `Result`. Add a handler reacting to `OrderPlaced` (eventual consistency). Unit-test with a fake repo + captured events. Acceptance: aggregate-granular repo (domain

interface); events immutable/past-tense, published after save; app service orchestrates only (no rules/presentation); one-aggregate transaction + eventual cross-aggregate reaction; unit-tested with fakes.

DDD Integration (Capstone: Model One Bounded Context)

> Bring it together by fully modeling **one bounded context** end-to-end: a **ubiquitous language** glossary, **value objects** (self-validating, immutable), **entities** (identity + behavior), an **aggregate** (root enforcing invariants), a **domain service** for cross-object rules, a **repository interface** (per aggregate root), **domain events**, and a thin **application service** — all **pure Dart, unit-tested**, and mapped to a **feature/module** (Module 44/Module 45) as the domain layer of Clean Architecture (Module 40). This shows DDD not as isolated patterns but as a coherent model that tames one complex context — applied proportionally.

Introduction

This module capstone assembles strategic + tactical DDD into a single, coherent bounded-context model and situates it within the app's architecture. It demonstrates how the pieces fit, how they're tested, and how the context maps to a feature/module — the practical payoff.

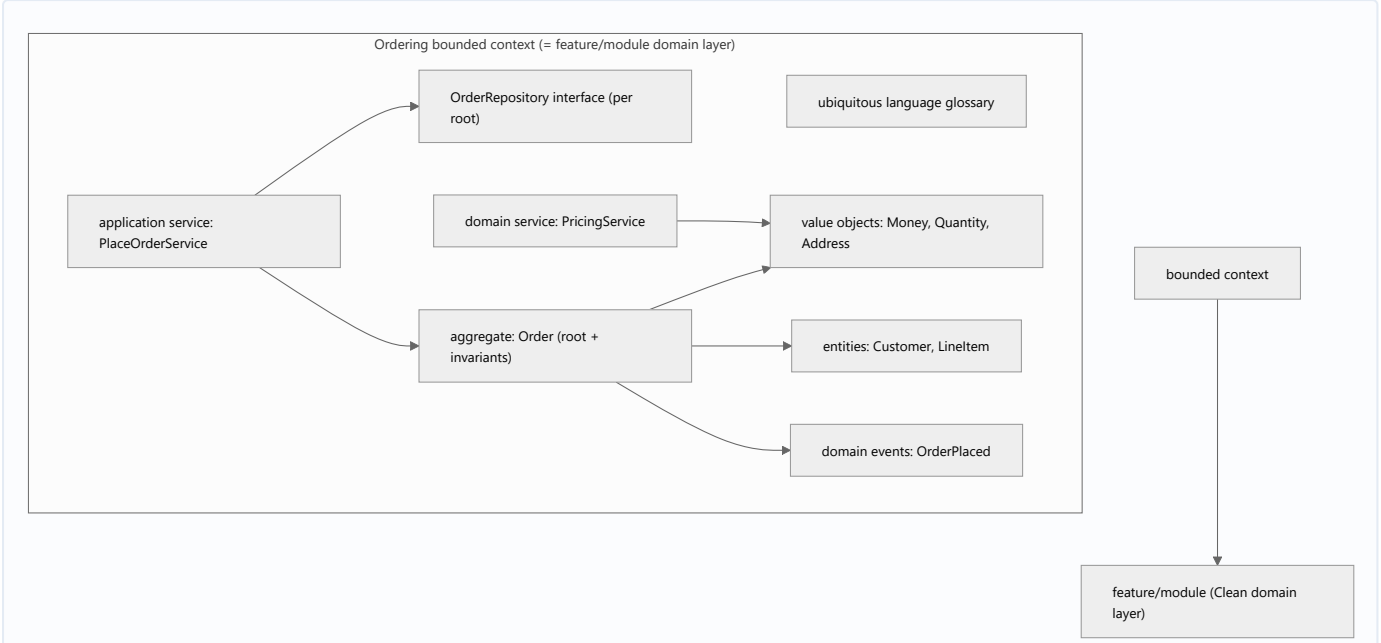
Why this concept exists

DDD's value appears only when the pieces cohere into one well-modeled context, expressed in the ubiquitous language and enforcing its invariants. This capstone provides that coherent example and shows DDD as the **rich domain layer** of the app's existing Clean/feature-first/modular structure — not a separate universe.

Real-world analogy

It's **building one fully-functioning department to a professional standard**: a shared glossary everyone uses (ubiquitous language), precise tools and parts (value objects/entities), a supervised workflow that can't produce defects (aggregate + invariants), specialists for cross-cutting tasks (domain services), a stockroom (repository), an announcement system (events), and a coordinator (application service) — all fitting into the larger company (Clean/feature/module) as one well-run unit.

Internal Working



- **Strategic setup** (01_ddd_fundamentals.md): choose **one bounded context** (Ordering), write its **ubiquitous language** glossary; this context = a **feature/module** whose **domain layer** is this model.
- **Tactical model** (02_tactical_building_blocks.md): **value objects** (`Money` , `Quantity` , `Address` — immutable, self-validating), **entities** (`Customer` , `LineItem` — identity + behavior), a **domain service** (`PricingService`) for cross-object rules, **factories** for valid creation.
- **Aggregate + invariants** (03_aggregates_and_invariants.md): the `Order` **aggregate** — root guards line items + status, enforcing invariants (total = sum of lines, only draft editable, valid transitions, non-empty place); small; references `Customer` by id.
- **Repository + events + app service** (04_repositories_and_domain_events.md): `OrderRepository` interface (per aggregate root, whole-aggregate), `OrderPlaced` domain event, thin `PlaceOrderService` (load → `place()` → save → publish).
- **Fit into the architecture**: this whole model is the **domain layer** of a **feature/module** (Clean — Module 40); the **data layer** implements `OrderRepository` ; the **presentation layer** (MVVM — Module 43) calls the application service. Cross-context interaction goes via **contracts/ACL** (Module 45).
- **Purity + testing**: the entire model is **pure Dart** (no framework/IO), **unit-tested** exhaustively (VO validation, aggregate invariants, service, application-service flow with fake repo + captured events) — the DDD payoff realized as fast, confident tests.
- **Proportionality (honest)**: this depth suits a **complex core domain**; a CRUD/generic subdomain should stay simple (01_ddd_fundamentals.md). DDD is applied **where it earns its cost**.

Memory Representation

The context is a cohesive object model: VOs (immutable values), entities (identity + state), the `Order` aggregate (encapsulated internals + invariants + pending events), repository interface (domain), and a stateless app service. It lives as one feature/module's domain layer.

Compiler Behavior

Pure-Dart domain compiles without framework/IO; VOs enforce validity + value equality; aggregate encapsulation prevents invariant bypass; repository/events are domain types. The context is enforceable as a module boundary (Module 45).

Runtime Behavior

App service loads the aggregate, invokes behavior (invariants enforced), saves (one transaction), publishes events after save; handlers react (eventual consistency). Invalid operations are rejected, never producing invalid state.

Flutter Engine Behavior

None — pure domain; the presentation layer (outside this model) touches the engine and calls the application service.

Dart VM Behavior

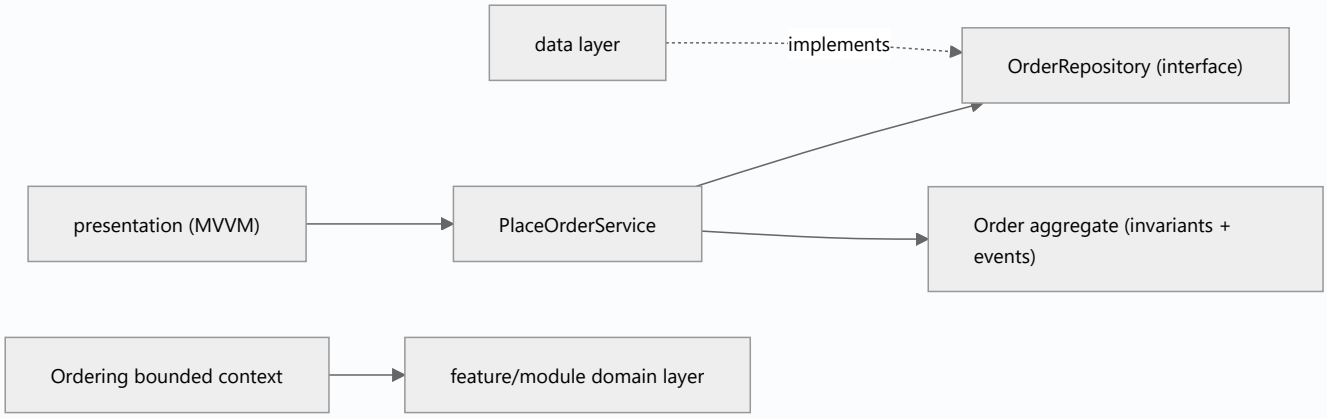
Fastest test tier: VO/entity/aggregate/service/app-service tests run in pure Dart with fakes — no device/binding.

Examples

```
// One coherent bounded-context model (Ordering) — pure Dart, ubiquitous-language names

// Value objects (self-validating, immutable) + entity (identity) — from tactical file
// class Money {...}  class Quantity {...}  class Customer {id, ...}  class LineItem {...}

// Aggregate root (invariants) + events — from aggregates/events files
class Order { /* root: addLine/place/ship; guards invariants; records OrderPlaced */ }

// Repository (domain interface, per aggregate root)
abstract class OrderRepository { Future<Order?> findById(String id); Future<void> save(Order o); }

// Application service (thin orchestration) — the entry point the presentation calls
class PlaceOrderService {
  final OrderRepository orders; final EventDispatcher events; final Clock clock;
  PlaceOrderService(this.orders, this.events, this.clock);
  Future<Result<void>> call(String id) async {
    final o = await orders.findById(id);
    if (o == null) return Failure(NotFoundFailure('order'));
    o.place(clock.now());                // domain rules/invariants in the aggregate
    await orders.save(o);                // one aggregate, one transaction
    for (final e in o.pullEvents()) events.publish(e); // after save
    return const Success(null);
  }
}
```

```
// Exhaustive, device-free tests — the DDD payoff
test('Money rejects negative + mismatched currency', () { /* VO validation */ });
test('Order enforces invariants (empty place / transitions / edit-when-shipped)', () { /* aggregate */ });
test('PlaceOrderService: places, saves, publishes OrderPlaced', () async {
  final repo = FakeOrderRepo(draftOrderWithLines());
  final events = CapturingDispatcher();
  final r = await PlaceOrderService(repo, events, FixedClock()).call('1');
  expect(r, isA<Success>());
  expect(events.published.single, isA<OrderPlaced>());
});
```

Diagrams



Common Mistakes

Mistake	Why	Fix
DDD depth on a CRUD/generic subdomain	Overkill	Apply to complex core only; keep generic simple
Framework/IO in the domain model	Breaks purity/testability	Pure Dart; repo interface only
Anemic aggregate (rules elsewhere)	Invariants drift	Rules/invariants in the aggregate
Ubiquitous language not in code	Translation drift	Name types/methods in the language
Rules in the application service	Anemic domain	Orchestrate only; rules in aggregate/service
Ignoring context boundary	Model leaks across contexts	Map context → feature/module; ACL across
Skipping tests	Loses the payoff	Unit-test VOs/aggregate/service

Best Practices

- Model **one bounded context** coherently: ubiquitous-language names, self-validating **value objects**, rich **entities**, an **aggregate** enforcing invariants, **domain services** for cross-object rules, a **per-root repository interface**, **domain events**, and a thin **application service**.
- Keep it **pure Dart** and **unit-test exhaustively** (VO validation, aggregate invariants, service flow with fakes) — the DDD payoff.
- Situate it as the **domain layer of a feature/module** (Clean); data implements the repo, presentation (MVVM) calls the app service; cross-context via **contracts/ACL**.
- **Apply proportionally** — DDD depth for the **complex core**, simplicity for CRUD/generic.

Performance

Runtime-neutral vs a simpler model; the payoff is correctness (always-valid aggregates), communication (shared language), and **test speed** (pure-Dart, exhaustive). Aggregate/transaction sizing keeps persistence efficient. Over-applying DDD costs dev time — proportionality is the efficiency lever.

Advantages / Disadvantages

- **+** Coherent, always-valid, exhaustively-testable model of a complex context; shared language; fits Clean/feature/module cleanly; tames complexity.
- **−** Significant modeling investment; overkill for simple domains; requires discipline (purity, invariants-in-aggregate, language) and proportional application.

Interview Questions

1. 🟢 **What does a fully-modeled bounded context contain?** — Ubiquitous language + value objects, entities, an aggregate (root + invariants), domain services, a per-root repository interface, domain events, and a thin application service.
2. 🟢 **Where does this model sit in the app's architecture?** — As the domain layer of a feature/module (Clean); data implements the repository, presentation (MVVM) calls the application service.
3. 🟡 **How is the model tested, and why is it fast?** — Pure-Dart unit tests of VOs/aggregate/service (with fakes + captured events) — no framework/IO, so the fastest tier.
4. 🟡 **How do you keep the model always-valid?** — Self-validating VOs + aggregate invariants enforced in root methods; the application service orchestrates but holds no rules.
5. 🟡 **How does a bounded context map to code structure?** — To a feature/module boundary; cross-context interaction via contracts/ACL (no model leakage).
6. 🔴 **When would you not model to this depth?** — For CRUD/generic subdomains — keep them simple; reserve DDD for the complex core (proportionality).
7. 🔴 **How do strategic and tactical DDD combine here?** — The ubiquitous language + context boundary (strategic) frame the model; the tactical patterns (VOs/entities/aggregate/services/repo/events) implement it coherently.

Senior Engineer Tips

- Model one context deeply and correctly (language → VOs → entities → aggregate → repo/events/service) rather than sprinkling patterns across the whole app; depth where it matters beats shallow DDD everywhere.

- Keep the whole model pure Dart and unit-test the invariants exhaustively; the confidence from fast, complete domain tests is the concrete reason to invest in DDD.
- Slot it in as a feature/module's domain layer and reach it via an application service from MVVM; DDD isn't a replacement for Clean/feature-first — it's the rich version of their domain layer, applied to the complex core.

Architect Perspective

This capstone shows DDD as a coherent domain model for one complex bounded context, sitting as the rich domain layer of the app's Clean/feature-first/modular structure: strategic boundaries + ubiquitous language framing a tactical model of value objects, entities, invariant-guarding aggregates, repositories, and events. It's the deepest expression of the architecture band — the "what the domain truly is" layer — applied proportionally to the core domain where taming complexity pays off, and mapped cleanly onto features/modules with contracts across contexts (Module 40, Module 44, Module 45).

Summary

- Model one bounded context coherently: ubiquitous language + VOs + entities + an invariant-guarding aggregate + domain services + per-root repository + domain events + thin application service.
- Keep it pure Dart, unit-test exhaustively; situate as the domain layer of a feature/module (Clean), with data implementing the repo and MVVM calling the app service.
- Apply proportionally — DDD depth for the complex core, simplicity for CRUD/generic; cross-context via contracts/ACL.

Revision Notes

- Bounded context = feature/module domain layer: ubiquitous language + VOs (self-validating) + entities (identity+behavior) + aggregate (root+invariants) + domain services + per-root repository interface + domain events + thin application service.
- Pure Dart + exhaustive unit tests (VO/aggregate/service, fakes + captured events); data implements repo, MVVM calls app service; cross-context via contracts/ACL.
- Apply proportionally (complex core only); strong consistency inside aggregate, eventual across; DDD = rich domain layer of Clean/feature/modular.

Practice Questions

1. What are all the pieces of a fully-modeled bounded context, and how do they connect?
2. Where does this model live relative to Clean/feature/modular architecture?

3. How do you decide how much DDD depth to apply?

Coding Questions

1. Assemble an Ordering context (VOs + entities + `Order` aggregate + `OrderRepository` + `OrderPlaced` + `PlaceOrderService`), pure Dart.
2. Unit-test VOs, aggregate invariants, and the application-service flow (fake repo + captured events).
3. Show how the presentation (MVVM) + data layers connect to this domain model.

Mini Project

Bounded-context model (capstone — Flutter/domain): Fully model the Ordering context: a ubiquitous-language glossary; value objects (`Money` , `Quantity` , `Address`); entities (`Customer` , `LineItem`); the `Order` aggregate (root enforcing invariants); a `PricingService` ; an `OrderRepository` interface (per root); an `OrderPlaced` event; and a thin `PlaceOrderService` — all pure Dart, exhaustively unit-tested (VO validation, aggregate invariants, service flow with fake repo + captured events). Document how it maps to a feature/module and note the proportionality decision. Acceptance: coherent context in ubiquitous language; self-validating VOs + rich entities; aggregate always-valid (invariants in root); per-root repo interface; events published after save; thin app service (no rules/presentation); pure Dart + exhaustive tests; mapped to feature/module; proportional-application note.