Pragmatic DDD Guide, short version

Nov 1, 2026 · @kodebot

This is the short version of the Pragmatic DDD Guide for the team: every rule, what goes wrong without it and what we do instead, with no code. The long guide has the same rules with the same numbers, each with the code that goes wrong, the code that works, the Spring feature used and the reasons. Read this one first, and go to the long one when a rule needs more than a paragraph. Read Parts A and B today, C and D tomorrow, and Part E when the situation comes up.

We design the inside of a microservice with a light form of Domain-Driven Design (DDD): write the code in the business's own words, and put each business rule inside the object it is about. The aim is that every rule lives in exactly one place, and a design mistake can be fixed inside one feature of one microservice. The stack is Spring Boot 4 with Spring Data MongoDB on Amazon DocumentDB.

Terms

Term	Meaning
Business action	One change the user asks for with one button, such as "close claim"
Entity	A business object with its own ID; its details change, its identity does not
Value object	A value with no ID, defined by its contents, never changed after it is created, such as `Money`
Aggregate	A group of objects always loaded and saved together; in DocumentDB, one document. The top object is the root
Repository	The one class that loads and saves one kind of aggregate
Application service	A class with one method per business action, running that action's steps
Adapter	The one class that talks to another microservice or system
Integration event	A versioned event that other microservices receive, such as `ClaimClosedV1`
Outbox, inbox	The event stored with the change and sent afterwards; the IDs of messages already processed
Design note	A one-page description of a feature that the design group approves before coding starts

Closing a claim, end to end

A caseworker presses Close claim. The controller checks the permission and calls the application service with the claim's ID and the version the user saw. The service loads the claim, checks the version, asks the note feature how many notes need action, and calls the claim's `close` method with that count and today's date. The claim checks its rules, changes its status and records a `ClaimClosed` event; if a rule refuses, it throws a business error that the shared error handler turns into `422` with a code. The service saves the claim and the outbox entry for the event in one transaction, then writes the audit record. A scheduled sender publishes `ClaimClosedV1` to the message broker, where the payment microservice picks it up and stops the payments. Every rule below points at one of these steps.

Part A. Where a rule lives

1. Put a business rule in the entity method that makes the change, not in a service.

A rule in a service is copied the next time another caller needs the same change, and the copies drift. The entity method takes the facts it needs as parameters, so no caller can skip the check: the controller, the nightly job and the message listener all call the same method.

2. Change an entity's fields only through its own methods; no setters.

A setter skips the rule and the event; a method with the business name checks the rule and records the event. All fields are private, and a build check fails on a public setter in an entity.

3. A service method only runs the steps: load, check the version, get facts, call the entity, save, audit.

A service method has no `if` about status, amounts, dates or counts; those belong in the entity or a value object. The fixed order is a convention we chose: when every action has the same six steps, a missing version check or

audit stands out in review. Calls to other systems happen before the transaction opens, and the time comes from the `Clock` bean.

4. Name classes, methods and errors with the business's own words.

"Claim has open actions" tells a caseworker reading the log what happened; "status 3" tells nobody. The design note, the code, the error code and the conversation with the business use the same words.

5. Put a calculation that spans several objects in a domain service, and only when it has worked examples.

A domain service is a plain class that never loads or saves, for a calculation such as weekly entitlement from claim type, rate table and number of children, with a table of worked examples in the design note. If you cannot write that table, the logic is one or two `if`s and belongs in the entity.

6. Give every ID and business value its own type; check its format once, where the value is created.

A bare string or decimal is checked in three places and in none on the import job, and two IDs swap without a compile error. `ClaimId`, `Money`, `FileName` and the rest refuse to exist with a bad value, so once one exists it is valid everywhere. A bare value arrives only at the edges and becomes a type there.

Part B. Aggregates

7. Keep a child inside the aggregate only if the parent's rules need it, it has a maximum count, it changes with the parent and it means nothing alone.

An aggregate is one document, written atomically, so a rule inside it is always true; between collections a rule is true only at the moment it is checked. Notes inside the claim make two caseworkers adding notes collide on the version, and 4,000 notes approach the 16 MiB document limit, so notes are their own aggregate holding the claim's ID. Copying the old system's tables one to one is the most common way to get this wrong.

8. Store another aggregate's ID, never the object itself.

Storing the object loads it on every read, 21 queries for 20 notes, and DocumentDB has no DBRefs. Store the ID; when the note feature needs something about the claim, it asks the claim's application service.

9. Save one aggregate per transaction.

A transaction over 10,000 claims hits DocumentDB's one-minute limit and its 32 MB log, and one failure undoes all of it. One save per aggregate, each atomic on its own. Saving two aggregates in one transaction needs an approved exception in the design note (rule 27).

10. Give every list inside an aggregate a maximum size.

Nothing else stops a claim with 600 evidence items, a page that times out and a document heading for 16 MiB. The design note says the maximum and the entity refuses the next one; a list that cannot have a maximum is its own collection. A file never goes in the document, only its reference.

11. Refuse a save that was made from an old version of the record.

Otherwise the second save silently overwrites the first and nobody is told. Every change carries the version the user saw; the entity refuses an old one with a friendly error, and the document's version field refuses it again in the database. Both become a `409`, and the screen says "changed by someone else, reload".

12. Make every change safe to repeat: a repeated request adds or counts nothing twice.

A request is repeated after a timeout, because the caller cannot tell whether the first one landed. The version check covers changes from a screen, an idempotency key covers creates, the inbox covers messages, and the entity's own check, ignoring a payment that is not newer than the one it holds, is the fourth guard and the only one that holds on every path.

13. Store money as an exact decimal and time in UTC, and count days in UK dates.

Floating point gives penny errors, local timestamps are an hour wrong twice a year, and a date stored as a timestamp reads back a day early on a UTC server. Money is an exact decimal, instants are UTC, and a rule that counts days works in dates stored as text.

Part C. Features, microservices and events

14. Use another feature only through its application service, passing IDs and simple values.
Reaching into another feature's storage breaks when that feature changes, and spreads its rule into two places. Ask its application service with an ID and get back a number or a small record, never an entity or a repository. Calls go one way; the other direction uses events. A build check enforces it.

15. Check permissions in the controller, never in the service or the entity.
A message listener also closes claims and has no user, so a check in the service needs a fake system user and means nothing. The controller knows the user and checks; a test proves a user without the permission gets `403`; services and entities never see a user.

16. Talk to another system only through one adapter class that translates its data into our types.
When the other team renames a field, only the adapter changes. Nothing from another system appears outside its adapter, and nobody reads another microservice's database. The adapter is tested against a fake of the other system.

17. Call another microservice before you open the transaction, never inside it.
A four-second call inside the transaction holds the session and its locks for four seconds, and the transaction fails whenever the network does; had the call made a change, a failed commit could not undo it. Get every outside fact first, then open the transaction and save. The object store counts as another system.

18. Retry a read; never retry a change.
A retried change lands twice when only the reply was lost, and the other system's audit trail shows two changes we cannot undo. A read may retry; a change is sent exactly once; a system that keeps failing is left alone for a short time by a circuit breaker.

19. Save an event in the same transaction as the change it reports, and send it afterwards.
Publish then save, and a failed save leaves an event about nothing; save then publish, and a failed publish loses the event. The entity records the event, the outbox entry is saved in the same transaction, and a scheduled sender publishes it afterwards. Nobody publishes any other way.

20. Only add fields to a published event; to remove or rename one, publish a new version.
A renamed field reads as null in the receiving microservice, which acts on the wrong date. Add freely; for a rename publish the next version beside the old one until every receiver has moved. Every integration event has a written schema; a domain event used only inside our microservice can change at any time.

21. Write every listener so a message that arrives late, twice or out of order does no harm.
No broker delivers exactly once and in order. An inbox of processed message IDs skips a repeat, and the entity ignores anything older than what it holds. The listener only translates the message and calls an application service; a message that keeps failing goes to a dead-letter queue.

22. Change several microservices as a chain of steps joined by events, with a named step for failure.
A transaction cannot cover two databases, so if their change succeeds and ours fails, the two disagree. Each step changes one microservice and sends an event that starts the next. A failed step cannot be rolled back, so the design names what happens instead, as a business action. Each such process needs design group approval and an architecture decision record.

Part D. Process

23. Get the design note approved before you code a feature with business rules.
A wrong aggregate boundary found in review means a data migration. The design group decides what each aggregate contains, the rules with worked examples and any two-aggregate transaction; engineers decide endpoints, classes and tests.

24. Mark a feature Simple (data with no rules) or Rich (states and rules) in its design note.
A lookup list of 12 codes does not need versions, events and an outbox. A Simple feature is data with no business rules, and its checks may sit in the service. The moment it gains a rule it becomes Rich and gets an entity method

for it.

25. Fix a mistake with a safe step today and the proper fix by a date.

Do the safe step now, tell the design group the same day, create a ticket with a date, then do the proper fix with a test, so the same mistake fails the build next time.

26. Keep personal data out of logs, error messages and events.

Logs and events are copied to monitoring tools and other teams' systems that our data protection rules do not cover. Value objects hide personal data when printed, and events carry IDs. If personal data reached production logs, report it through the data protection process.

Part E. Situations you will meet

27. When one button must save two aggregates, ask whether they are one; else react to an event; only then save both in one approved transaction.

Two plain saves leave half the work saved when the second fails. Three answers, in the order we chose: if the second thing is really a child, it goes inside the document; if it can follow a moment later, our own event goes through the outbox and a listener creates it; only when the user must see both at once, one transaction, written in the design note as an exception to rule 9. Never across microservices.

28. Keep a rule across all records of a kind true with a unique index, and turn the index error into a business error.

"A claimant has at most one open claim" is not a rule about one aggregate, and check-then-save lets two users race past it. A partial unique index holds the rule, the service keeps the exists check for a friendly error, and the duplicate-key error is translated into a business error at the edge.

29. Create an entity through a named factory method that checks the rules and raises the event; keep the constructor private.

Creation is a business action: the factory checks eligibility and the claim type and records `ClaimOpened`. The only constructor calls are inside the entity and in Spring Data when it loads a document. We make our own IDs.

30. Read for a screen through a query class on the collections; never build a screen out of aggregates.

Loading 50 aggregates for a list is 51 round trips and getters that exist only for screens. Reads have no rules to protect, so a query class reads the collections directly, across aggregates, and returns exactly the fields the screen needs. A query never changes anything and never holds a rule.

31. Search and reports are queries too; give a slow one its own read collection filled by events.

A report over two million documents at request time takes minutes. Start with a query; when it has been measured too slow, keep a read collection that the domain events fill, rebuildable from the outbox. Nothing is ever changed through it.

32. Do not store on a parent what a child aggregate knows; read it when needed, or copy it by event and accept a short delay.

A flag on the claim set by the note feature drifts and couples the two features. Normally the query asks the notes collection when reading; when the flag must be on the claim, copy it through an event, a moment behind, and say so in the design note. For the real rule, `close` still asks for the count at the moment of closing.

33. Never delete a business record; add a business action that marks it, with a reason.

A deleted claim leaves the audit trail, the notes and the events already sent pointing at nothing. A cancel action with a reason keeps the record and hides it from lists. Removal under data protection is a separate approved process, and a build check fails on any other delete.

34. Run a time-based rule from a scheduled job that calls the same entity method, one record per transaction.

A job that updates the collection skips the rule, the event, the audit and the version. The job is another caller of the entity, one record per transaction, taking today from the `Clock` bean. On more than one instance it first takes a lock document.

35. When a process has its own steps and deadlines, give it its own aggregate.
Process flags on the claim fill its methods with process `if`s, and nobody can see the whole process. A `ClaimClosure` aggregate holds the state of the process and reacts to the events; the claim keeps its own rules.

36. Change an aggregate's shape behind a flag, with a migration that uses the entity's own methods.
A release-night script that fails halfway leaves two shapes that drift. Release the new shape behind a flag, run a rerunnable migration job one record per transaction through the entity's methods, turn the flag on, then remove the old fields in a last release.

37. Test the entity with no database, the service with a fake repository, and the repository alone against a real database.
Starting the whole microservice for every test costs ten seconds a test and buries the rule. Most tests are entity tests, because that is where the rules are; one controller test per endpoint proves the permission; adapters are tested against WireMock; the repository against MongoDB in Testcontainers, with a smoke test against a real cluster in the pipeline.

38. Show two aggregates on one screen with two save buttons; one button that saves two things is a design smell to raise.
One button is one request that saves two aggregates, an exception to rule 9 nobody approved. Each part of the screen saves its own aggregate with its own button and version. When a screen keeps needing one button, tell the design group: the boundary is wrong, or the action deserves a name of its own (rule 27).

Who to ask

Ask a senior engineer the same day when you do not know where code belongs. Ask the design group, before you code, for a new or changed aggregate, a two-aggregate transaction or a new link between microservices. Tell the receiving teams before you change a published event. Report a security or personal data problem through the incident process at once.