THE WORKSHOP

DDD Modelling

Half a day with a clarified Event Storming wall, a whiteboard, and the correct people — to land on aggregates, bounded contexts, and a context map the team will actually re-read. Where Event Storming an Architecture produces the first cut of boundaries, this is the session that sharpens them.

2026-08-07
barkingiguana.com/writing/the-workshop-ddd-modelling/

A facilitator's playbook — print, annotate, and keep on the table.

Contents

A monolith that touches everything is a monolith because nobody drew the lines. DDD Modelling is the workshop where the team draws bounded contexts (the places where one part of the business becomes a different part) so the code can stop pretending those edges don't exist. Worked example: Drawing the Boundaries.

DDD Modelling

DDD Modelling turns a clarified Event Storming Architecture wall into a context map: bounded contexts with named vocabularies, aggregates with stated invariants, value objects separated from entities, and explicit integration patterns at every crossing. Half a day, design the implementers will re-read. Sometimes called strategic DDD, tactical DDD, or context mapping, depending on which half of Eric Evans's Blue Book (*Domain-Driven Design: Tackling Complexity in the Heart of Software*, Evans, 2003) the session is leaning on. Confused with Event Storming an Architecture: Architecture produces the first cut of boundaries from a flow; this session sharpens them, names the integration patterns, and closes the "we'll decide later" notes. Often paired with C4 Modelling in the same week: DDD decides *where the boundaries are*, C4 decides *what sits inside them*.

At a glance

- *Who, for how long:* a facilitator with DDD experience, developers and architects, a domain expert with veto power, a product lead, and optionally a tech lead from an adjacent team. Four to eight people, half a day.
- *What you walk out with:* aggregate cards with named invariants, bounded contexts with named vocabularies, value objects separated from entities, an integration pattern on every crossing, a one-page context map, and a short list of anti-corruption-layer spikes with owners and dates.
- *When to reach for it:* an Event Storming Architecture session has happened and the team is about to commit code against the boundaries, or a monolith is being split and the seams need names not gestures. Not for unagreed flows (run Event Storming an Architecture first), unfamiliar domains, container/component-level work (C4 Modelling territory), or sessions where no domain expert can attend.

What's It For

The Architecture event-storm ended with a wall the team nodded at. A fortnight later, three things have happened.

One: two developers have picked up stories that touch the Order aggregate, and they're making inconsistent assumptions about what Order owns. One is writing the cancellation logic inside Order; the other is writing it inside Billing, reasoning that refunds are a Billing concern. Both are defensible; they're working from the same photograph of the same wall.

Two: the "we'll put an anti-corruption layer between us and the carrier API" line has been marked on the wall but nobody has made it real. The first integration has gone live and the carrier's vocabulary (*consignment, leg, segment*) is now in the Delivery aggregate. A week from now somebody will notice and the extraction will be a weekend.

Three: someone has started using the phrase *"the Order service"* to mean both the Order aggregate and the Fulfilment context, because the team hadn't separated the words yet, and now the mis-use is on its way into a JIRA epic.

DDD Modelling is the session that prevents each of these. It forces the aggregate boundaries to be stated in terms of invariants rather than in terms of "what seemed obvious on the wall." It forces the integration patterns to be picked from a small, named set, with their costs explicit. It forces the bounded context *names* to be committed, because the names become the vocabulary the team will use for the next year.

This is denser work than the Architecture event-storm. Architecture explores; Modelling decides.

Reach for it when:

- An Event Storming Architecture session has happened and the team is about to commit code against the boundaries
- A monolith is being split and the seams need to be named, not just gestured at
- A new bounded context is being proposed and the team has to decide what it owns, what it subscribes to, and who it integrates with
- Two existing contexts have started to leak vocabulary at their boundary and someone needs to pick an integration pattern
- The team's ubiquitous language has drifted and half the room uses a word to mean different things

What It's Not For

Skip it when:

- You don't yet have an agreed flow. Run Event Storming an Architecture first, or Process Level before that.
- The domain is unfamiliar to half the room. DDD vocabulary will become a hammer in search of nails.
- You're at the container/component level rather than the context level. That's C4 Modelling territory.
- You don't have a domain expert who can veto an aggregate on business grounds. The developers will design elegant structures that don't match reality.

Trade-offs to weigh before booking the room:

Benefits

- Aggregates defined by invariants rather than by intuition: the code has a test, not a feeling
- Named bounded contexts that stabilise the team's vocabulary for the next year
- Explicit integration patterns at every crossing, with their costs visible
- A one-page context map the team actually re-reads
- A short list of anti-corruption layer spikes, owned and dated
- Disagreements argued in the room rather than in pull requests six weeks later

Costs

- 4–8 people × half a day, plus a prep pass on the Architecture wall the day before

- The DDD learning tax: teams new to the vocabulary pay a steep first-session cost
- Political cost when the boundaries cross team lines and raise ownership questions
- Recurring cost: models rot; re-modelling every six to twelve months is normal

Failure modes

- Aggregates named without invariants, which means they're named without boundaries
- Anti-corruption layers marked but never built: carrier vocabulary leaks in anyway
- The context map photographed and then filed; the backlog keeps using the old vocabulary
- The team conflates bounded context with microservice and draws Conway's Law as if it were the design
- Invariants written that are wishful rather than enforceable
- Context maps drawn but never re-read

Stop signals

- The domain expert can't attend; every aggregate will be designed in their absence
- The Architecture wall is still in dispute; modelling on top of disputed events produces disputed models
- Half the room can't define *aggregate* or *value object*; brief for 45 minutes before continuing, or reschedule

Stopping and scheduling a DDD primer before the session is not failure. Running a Modelling session where half the vocabulary doesn't land is.

Definitions & Background

DDD vocabulary used in this session

Six terms carry the weight. None are new if you've read the DDD Blue Book; the definitions below are the working versions used in the session itself.

Aggregate. A *consistency boundary* with a single root entity that external code must reference through. One transaction modifies one aggregate; cross-aggregate consistency is eventual. *"The cart total equals the sum of line items"* defines a cart aggregate. Aggregates are small by default; big aggregates are the most common design smell.

Aggregate root. The single entity inside the aggregate that everything outside is allowed to reference. External code cannot reach inside the aggregate to grab a child entity directly; you go through the root. This is the rule that makes the boundary actually mean something: without an enforced root, the cluster is a folder, not an aggregate.

Invariant. A rule that must always hold for an aggregate to be valid. Invariants define aggregate boundaries: if a rule spans two events, those events probably belong in the same aggregate. Invariants are the test: if you can't state one, the boundary is decorative.

Value object. A thing defined by its values, not its identity. Money (£3.50 GBP), a date range, an address. Two value objects with the same values *are* the same thing; they're interchangeable. Value objects simplify everything they touch because they have no lifecycle.

Entity. A thing with identity that persists through changes of state. A subscription, a box, a subscriber. Even if every field changes, it's still the same subscription. Entities have lifecycles; value objects don't.

Bounded context. A linguistic and design boundary around a cluster of aggregates that share a vocabulary. Inside the context, every word has exactly one meaning. Across contexts, the same word can legitimately mean different things (a *customer* in Billing is not the same thing as a *customer* in Support). The name of the context is the name of its vocabulary.

Context map integration patterns

The patterns split into two kinds, which teams routinely confuse:

- Power relationships describe *who has agency over the model.* Customer-Supplier and Conformist are about whether the downstream team can influence upstream changes; they're not integration mechanisms.
- Integration mechanisms describe *how two contexts actually connect.* ACL, Shared Kernel, Open-Host Service, Published Language, Separate Ways, Partnership.

In practice, most real context maps use Anti-Corruption Layer plus Customer-Supplier plus Separate Ways. The other patterns name situations the team is already in; they're not picked from a catalogue.

Power relationships:

- Customer-supplier. Upstream and downstream are different teams; the downstream team's needs are heard but not binding. Explicit coordination, clear direction of dependency.
- Conformist. Downstream accepts upstream's model as-is because translating would cost more than the contamination. Transitional, often on the way to an ACL.
- Big Ball of Mud. Brian Foote and Joseph Yoder's term for a sprawling, undisciplined, vocabulary-free codebase. The honest label for an existing legacy system whose vocabulary you don't trust and don't intend to clean up. Usually paired with an ACL on every context that touches it.

Integration mechanisms:

- Shared kernel. Two contexts share a small, jointly-owned piece of code or schema. Tightly coupled, high-trust, small surface. Use sparingly; most shared kernels want to grow.
- Anti-corruption layer. Downstream context translates upstream's vocabulary into its own, and keeps upstream's language out of its model. The pattern for external systems and any Big Ball of Mud you must integrate with.
- Open-host service. Upstream publishes a stable protocol for any downstream to consume. Usually paired with a published language: a shared schema (JSON Schema, Protobuf, or similar) that defines the contract.
- Published language. A documented, versioned contract for cross-context communication. The artefact; Open-host service is the posture.
- Separate ways. Two contexts deliberately don't integrate. Sometimes the correct answer: the cost of integration exceeds the value.
- Partnership. Two contexts share *mutual success or failure*: changes on either side require coordination on both, and either side failing hurts both. High-cost, high-trust, reserved for genuinely

interdependent work. Distinct from Customer-Supplier (which is a power relationship); Partnership names a shared fate.

Inputs

- A clarified Architecture wall or its equivalent: an agreed flow with aggregate candidates dotted, boundary lines drawn, and crossings marked as commands or events.
- A list of the external systems involved.
- A list of the vocabulary the team currently argues about.
- The DDD Blue Book or an equivalent reference in the room (not to read from, but to point at when a term is disputed).
- Half a day with no interruptions and the right people in the room (see *Who's Needed*).

Outputs

What lands on the wall and the page at the end:

- Aggregate cards. One card per aggregate, with the name and the single invariant that justifies its existence. *Order: "an order is exactly one of: pending, confirmed, cancelled."* If you can't name one invariant, you haven't got an aggregate.
- Context boundaries. Drawn around groups of aggregates that share vocabulary. The boundary is a line *and a name card*. The name is the vocabulary.
- Crossing notes. Every line that crosses a context boundary is annotated: direction, shape (command / event / query), and integration pattern.
- A one-page context map. Every context as a labelled box, every crossing as a labelled arrow with its pattern name. Photographable and re-readable.
- A short list of anti-corruption layer spikes, with owners and dates.
- A list of committed next spikes with owners and dates.

These outputs feed straight into:

- C4 Modelling. DDD Modelling decides where the boundaries are; C4 decides what sits inside them at the container/component level. Natural pair within the same week.
- Event Storming a Process. Process Level is upstream input. If the Modelling session reveals the flow is wrong, you're really in a Process Level conversation.
- Event Storming a Domain. Big Picture is where cross-context hotspots show up. Revisit when the Modelling session turns up a concept that doesn't fit any existing context.
- Example Mapping. When an invariant is contested, Example Mapping is the session that turns the argument into concrete rules and examples.
- Assumption Mapping. The integration patterns you pick are full of assumptions about the upstream context's behaviour. Pull them apart before committing to a shared kernel or an open-host service.

Who's Needed

Facilitator. DDD experience strongly preferred, or pair a generalist facilitator with a tech lead who knows the patterns. This session is vocabulary-heavy; a facilitator who can't sort out the *aggregate vs entity* confusion will stall when it matters.

Developers and architects. The heart of the room. These are the people who'll write the code; the aggregates and context boundaries need to land in their hands in front of each other. Four is a good number, six is the upper edge.

Domain expert with veto power. Not to propose aggregates, but to stop them. When the developers design an elegant Order aggregate that ignores that the business treats cancelled-and-refunded differently from cancelled-and-credited, the domain expert is the person who says so. If you can't get veto-level domain expertise in the room, postpone; designing without it produces a model that has to be rebuilt on first contact with the business.

Product lead. To turn the output into backlog shape in the days after and to carry the context names into every future product conversation.

Optional tech lead from an adjacent team. When your boundaries touch another team's boundaries, a representative from that team prevents the context map from being drawn unilaterally. They also prevent you from choosing an integration pattern that the other team can't actually support.

Group size: 4–8. Above eight and the argument space fragments; below four and you're missing a perspective (usually the domain expert, and it shows in the result).

Who to leave out:

- Non-technical stakeholders beyond the domain expert. They'll struggle with the vocabulary and absorb facilitator attention that the design needs.
- Observers. DDD sessions are loud and argumentative; observers warp the argument. If they want the output, read the close-out.
- Teams that own contexts not in scope. They'll derail the session by pulling at their own contexts. Invite them only if their context is on the map.

How To Run It

Phase	Duration	Materials	Key question
Orient to the wall	15 min	Architecture wall	"Do we all still agree?"
Name aggregates and invariants	45 min	Aggregate cards, invariant notes	"What must change together? Why?"
Identify value objects	20 min	Value object notes	"Does identity matter here?"
Draw bounded contexts	30 min	Context boundaries, names	"Where does the vocabulary change?"
Mark crossings and pick patterns	45 min	Crossing notes, pattern cards	"How do they talk? Who depends on whom?"
Anti-corruption layer decisions	20 min	ACL sketches	"Where do we need a translation?"
Write the context map	15 min	Single-page map	"What does this look like on one page?"
Commit next spikes	15 min	Owner + date list	"Who owns what next?"
Total	~3h 45min in a 4-hour block. Plan a buffer; the invariant phase almost always overruns.		

Most first-time sessions spend too long on aggregates and too little on integration patterns. The facilitator's clock-watch target: at the halfway mark, contexts must be drawn. If contexts aren't drawn by then, the integration phase will collapse.

The three artefacts

Three artefacts build through the session; keep them separate, visibly, on different parts of the wall.

- Aggregate cards. One card per aggregate, with the name and the single invariant that justifies its existence. *Order: "an order is exactly one of: pending, confirmed, cancelled."* If you can't name one invariant, you haven't got an aggregate.
- Context boundaries. Drawn around groups of aggregates that share vocabulary. The boundary is a line *and a name card*. The name is the vocabulary.
- Crossing notes. Every line that crosses a context boundary is annotated: direction, shape (command / event / query), and integration pattern.

The session alternates between cluster-level work on the whole wall and card-level work on individual aggregates. The facilitator's rhythm: broad pass, deep dive on the hardest candidate, broad pass, deep dive on the next, finally broad pass again to check the shape.

Phase 1: Orient to the wall (15 min)

Walk the Architecture wall out loud. Point at every aggregate candidate, every drawn boundary, every crossing note. Invite corrections.

> *"From this point on, the wall is the ground truth. If we disagree with the Architecture session's output, we're running a different session. Today we're sharpening, not redrawing."*

Small corrections are fine; major re-litigation is not. If the Architecture wall has genuinely drifted, end the session and schedule another Architecture pass first.

What to watch for:

- Stale dots. Aggregate candidates that the team has since re-evaluated. Update before you commit.
- Silent disagreement. A developer not speaking isn't agreement. Ask by name: *"You were at the Architecture session; does this wall still match what you remember?"*
- Absent crossings. Crossings that weren't marked on the Architecture wall but that the team has since realised exist. Add them now; don't pretend they're not there.

Phase 2: Name aggregates and invariants (45 min)

This is the session's core. For each aggregate candidate on the wall, the room writes a card with two things: the aggregate's name and one invariant.

> *"An aggregate isn't a set of events; it's the rule that keeps the set honest. I want a name and the one rule that would break if we let the events drift. If you can't state the rule in a sentence, the aggregate isn't one yet."*

The invariant test is strict. *"Orders must be consistent"* is not an invariant; it's a wish. *"An order is exactly one of: pending, confirmed, cancelled"* is an invariant, because it tells you what states are illegal. *"A refund's amount never exceeds the original payment"* is an invariant. *"A subscription's paused state and its next-delivery-date field cannot both be set"* is an invariant.

When the room produces multiple invariants for one aggregate, write all of them on the card. When the room produces one invariant that spans two aggregates, you have a clue: those two probably want to merge, or the invariant is at a higher level than the aggregate.

What to watch for:

- The god aggregate. One card with seven invariants. *"Order"* is almost always this. Split where the invariants don't all depend on each other.
- The wishful invariant. *"The subscription's state is consistent with the subscriber's preferences."* Too vague. Push: *"Give me a specific illegal combination."*
- Aggregates defined by data. *"The aggregate is whatever's in the* `orders` *table."* That's the current database; it's not the aggregate. Ask what *rule* forces the data together.
- Hidden entities. A concept referenced on the wall but not an aggregate candidate. Usually a sub-entity inside a larger aggregate. Park it and return to it during value-object identification.

Produce 5–8 aggregate cards for a single-context session, 10–20 for a multi-context session. If you have more, you're either modelling at the wrong scope or you've missed the clustering.

Phase 3: Identify value objects (20 min)

Walk each aggregate card. For every concept the aggregate mentions, ask:

> *"Does this have an identity that persists, or is it defined by its values?"*

Money, addresses, date ranges, quantities, SKUs, delivery windows, pause periods: usually value objects. Subscriptions, boxes, subscribers, substitutions: usually entities.

The test: *"Would two of these with identical values be the same thing?"* If yes, value object. If no, entity.

Value objects go on a second wall area, one note per value object. When a value object is shared across aggregates (Money, DateRange, Address), put it in a neutral area; it's often a candidate for the shared kernel if two contexts both need it.

What to watch for:

- Entities disguised as value objects. *"SubscriberPreferences"* looks like a value object until someone points out that the history of changes matters for the recommender. Identity matters, so it's an entity.
- Value objects disguised as entities. *"A refund"* is often modelled as an entity because the team has given it an ID. Check whether the ID does any work: if the refund is only ever referenced through its payment, it might be a value object inside Payment.
- The UUID reflex. Developers reach for IDs because the database will need them. That's an implementation detail. Design the model; let the repository add IDs later.

Phase 4: Draw bounded contexts (30 min)

Now the marker. For each cluster of aggregates that shares a vocabulary, draw a thick line around the group and write the context's name on a card inside.

> *"The line is the edge of a vocabulary. Inside the line, every word has exactly one meaning. Outside the line, the same word can legitimately mean something different. The name on the card is the name of the vocabulary."*

The test for a context boundary: *"Does the word subscriber (or customer, or order, or box) mean the same thing on both sides of this line?"* If it does, the line is probably wrong or redundant. If it doesn't, the line is real, and the anti-corruption layer debate starts.

In our running Greenbox example (the produce-box subscription startup the narrative series follows) the common contexts come out as: Subscription (the subscriber's commitments and choices), Fulfilment (picking, packing, delivery-day operations), Billing (money, invoices, refunds), Supply (producers, stock, substitutions), Notifications (messages going out), Identity (accounts, authentication). These will vary by product; don't copy them without justifying each.

What to watch for:

- The microservice-shaped context. Teams name contexts after services they already have. Sometimes correct, often Conway's Law leading the design. Ask: *"If we had no services today, would we still draw this boundary here?"*
- Contexts named after teams. If the context is called *"the platform team's context"*, you're drawing the org chart. Rename to a domain noun.

- One giant context. When the whole wall is one context, the team is modelling at too fine a scale; they're really inside one context already. Zoom out or pick a different scope.
- Too many tiny contexts. Five contexts for six aggregates. The team is confusing aggregate with context. Merge.

Phase 5: Mark crossings and pick patterns (45 min)

For every line that crosses a context boundary, the team picks an integration pattern from the menu. Write the pattern name on the crossing.

Walk each crossing systematically. For each, ask three questions:

> *"Which direction does the dependency go? Which context changes to accommodate the other?"*

> *"Is the crossing a command, an event, a query, or a shared data structure?"*

> *"Which pattern from the menu fits? And what does that pattern cost?"*

The most common patterns you'll use: anti-corruption layer for external integrations and for crossing into legacy or third-party-shaped contexts; customer-supplier for internal team-owned contexts where the team is cooperative; published language with an open-host service for contexts that fan out to many consumers; separate ways where integration cost exceeds value.

Explicitly name the cost of each pattern. Shared kernels are coupling taxes paid forever. Conformist is a linguistic tax. Anti-corruption layers are build-and-maintain taxes. Published languages are versioning taxes. A pattern picked without its cost stated is a decision the team will regret quietly.

What to watch for:

- The implicit shared kernel. Two contexts share a database table and nobody has named it a shared kernel. Make it explicit; either commit to the coupling or separate.
- The theoretical ACL. An anti-corruption layer marked on the wall but never committed as a real spike or story. Turn it into a named deliverable before the session ends.
- Bidirectional customer-supplier. Both sides think they're the supplier. Someone has to concede or you're in partnership territory, which is much more expensive.
- The unnamed crossing. A line crosses a boundary with no pattern written on it. Stop and fill it in; the unnamed crossing is where the next subtle bug lives.

Phase 6: Anti-corruption layer decisions (20 min)

A focused pass on every anti-corruption layer you've marked. For each one:

> *"What does the external vocabulary look like on their side? What does our vocabulary look like on ours? What's the translation?"*

Write two columns on a note, *their words* and *our words*, and fill in the translation. *Stripe's 'charge' becomes our 'payment capture'. Stripe's 'customer' becomes our 'subscriber-with-payment-method' inside Billing only.* The translation is the layer's specification.

Anti-corruption layers are also where the team quietly accepts the cost of owning the translation. Name an owner for each ACL. Undowned ACLs drift.

What to watch for:

- The incomplete translation. Only half the external vocabulary is mapped. Finish the pass or mark the gap explicitly.
- The ACL as procrastination. *"We'll put an ACL in front of it."* Fine, but *when*. Owners and dates on every ACL before the session ends.
- The ACL that isn't one. A team marks an ACL but their code plans to use the external types directly. Clarify: if you're not translating, you're conformist. Name it honestly.

Phase 7: Write the context map (15 min)

One page. Every context as a labelled box, every crossing as a labelled arrow with its pattern name. The team looks at it together and corrects the errors.

> *"If this map can't fit on one page, the scope is wrong or the map is wrong. One page is the artefact."*

The one-page map is what gets photographed, redrawn in a diagramming tool, pinned on the team's wall, and referenced for the next year. A map that's six pages of UML never gets re-read.

What to watch for:

- The map that doesn't match the wall. Someone simplifies for the one-pager and loses a real crossing. Keep the wall as truth; the map is a summary, not a replacement.
- The map that over-claims. Patterns marked as "published language" when no published language has been written. Mark them as *planned* if they're aspirational.

Phase 8: Commit next spikes (15 min)

The session ends on commitments, not summaries. Name the three or four things that have to happen in the next fortnight to keep the model honest:

- *Write the first ACL against the carrier API.*
- *Name the Subscription context officially in the codebase; rename the package.*
- *Schedule a follow-up DDD Modelling session on Supply, which got thin treatment today.*
- *Share the context map with the adjacent team's tech lead for feedback.*

Every commitment has an owner and a date.

Worked example

See Domain-Driven Design: Drawing the Boundaries for Charlotte walking the Greenbox team through their first serious DDD session, including the moment the room realises their Order aggregate is really three separate aggregates held together by a shared table, and the relief of naming them.

What Can Go Wrong

The god aggregate. One aggregate is absorbing the wall. *Recovery: "Tell me the one invariant. Now tell me which of its state changes depend on that invariant and which don't. The ones that don't are a different aggregate."* *Stop if:* A second split attempt reproduces the god aggregate. The scope is wrong or the domain expert is missing.

Wishful invariants. The room writes invariants that are aspirations rather than rules. *Recovery: "Give me a state the aggregate could be in where this rule is violated. If you can't, the rule isn't doing any work."* *Stop if:* Every invariant the team produces is wishful. The domain isn't well enough understood for Modelling; run more Event Storming first.

The anti-corruption layer as procrastination. Every hard integration is marked ACL and left. *Recovery:* Force specificity: *"What words are we translating? Who owns the translation? When is it built?"* *Stop if:* The team can't commit owners. The ACLs will drift; better to postpone the session than to produce a map of unowned translations.

The bounded-context-per-microservice trap. Someone keeps arguing boundaries should match services. *Recovery: "Services are an implementation. Boundaries are vocabulary. If the current services were all deleted tomorrow, where would the vocabulary actually change?"* *Stop if:* The argument won't end. The session is really about service boundaries, which is a different conversation.

Vocabulary drift during the session. Halfway through, the room starts using context names inconsistently. *Recovery:* Point at the context name card: *"Inside this line, we use this word. Outside it, we use the other word. Let's replay that sentence."* *Stop if:* The drift keeps happening after three corrections. The names on the cards are wrong; workshop them before continuing.

The context map nobody will re-read. The session produces a six-page diagram. *Recovery: "One page. If it doesn't fit, cut the detail. The map is what we point at in January; the wall is what we referred to today."* *Stop if:* The team insists on the long version. They're producing artefacts, not a working model. Stop and rediscuss the output goal.

Next Steps

The session ends; the work begins.

Facilitator's close-out (same day):

- Photographs: the aggregate-and-invariants wall, the bounded-context layout with names visible, each anti-corruption layer's translation notes.
- A transcribed, one-page context map in a diagramming tool. Every context named, every crossing labelled with its integration pattern.
- A short summary message: the contexts, the aggregates, the integration patterns, the committed spikes with owners and dates.

The tech lead's week:

This is where the pattern earns its cost.

- Make the context names real. Rename packages, services, tracker components to match. If the code still says *"OrderService"* when the context is now *"Fulfilment"*, the model has already started rotting.
- Write the first anti-corruption layer. Pick the highest-risk external integration and build the ACL within a fortnight. Every week the ACL isn't there, external vocabulary leaks further.
- Walk the context map with adjacent teams. Their tech leads will challenge the map at its edges, exactly where challenge is most valuable.

- Turn the aggregates into code structure. Each aggregate becomes a package, a module, or a directory with a clear invariant in its README (or equivalent). Aggregates that exist only on the wall will diverge from the code within weeks.
- Revisit the Supply (or thinnest) context. One context always gets thin treatment; schedule a follow-up session on it within the month.

Ongoing, the team:

- Pin the one-page context map where the team works. When a story crosses three contexts, the map makes the crossings visible and forces the choice: split the story, or accept the coordination cost.
- Re-open the model when new aggregates emerge. A feature that doesn't fit is often a new aggregate wanting to exist; don't cram it into an existing one.
- Track vocabulary drift. When someone starts using *"customer"* to mean the Billing subscriber *and* the Support subject again, schedule a vocabulary pass.
- Re-model every six to twelve months. The domain moves; the model should follow.

Where to go next:

- Event Storming an Architecture. Architecture produces the first cut of boundaries; DDD Modelling sharpens them. Always run Architecture first; run Modelling the week after.
- C4 Modelling. DDD Modelling decides where the boundaries are; C4 decides what sits inside them at the container/component level. Natural pair within the same week.
- Event Storming a Process. Process Level is upstream input. If the Modelling session reveals the flow is wrong, you're really in a Process Level conversation.
- Event Storming a Domain. Big Picture is where cross-context hotspots show up. Revisit when the Modelling session turns up a concept that doesn't fit any existing context.
- Example Mapping. When an invariant is contested, Example Mapping is the session that turns the argument into concrete rules and examples.
- Assumption Mapping. The integration patterns you pick are full of assumptions about the upstream context's behaviour. Pull them apart before committing to a shared kernel or an open-host service.

Variants

DDD Modelling has two usable levels and the choice shapes the day.

Level	Scope	Duration	Output
Single-context *(default)*	One bounded context already identified	3–4 hours	Aggregates, invariants, value objects, the context's internal language
Multi-context *(harder)*	A map across 3–6 contexts	Full day	Full context map with integration patterns at every crossing

Single-context (default). One bounded context already identified. Three to four hours. Output: aggregates, invariants, value objects, and the context's internal language. This is what most teams need for their first serious DDD session, and the rest of this post is calibrated to it.

Multi-context (harder). A map across three to six contexts in a full day. Output: a full context map with integration patterns at every crossing. This is where Modelling sessions historically blow up: they overrun, the later contexts get thin treatment, and the context map comes out with decisions on the early contexts and hand-waves on the later ones. A sequence of single-context sessions with a short multi-context capstone usually beats the full day.

Start with single-context unless you have a tested team and a full day.

About this playbook

This playbook is part of *The Workshop*, a reference series of facilitator playbooks published at barkingiguana.com. The canonical, up-to-date version lives at barkingiguana.com/writing/the-workshop-ddd-modelling/.

These posts are LLM-aided. Backbone, original writing, and structure by Craig. Research and editing by Craig + LLM. Proof-reading by Craig.