DDD, Event Storming, Event Sourcing, and CQRS in dddesigner

A Practical Guide for Users

Introduction

Domain-Driven Design (DDD) is an approach to building complex software systems in which architecture is shaped primarily by the **language and boundaries of the domain**, rather than by database structure or chosen technologies. The goal of DDD is to express business rules and processes accurately in software without obscuring them behind implementation details.

dddesigner is an environment for designing complex systems: from dividing a system into **Bounded Contexts** and exploring the domain through **Event Storming**, to designing **Aggregates**, multi-step processes (**Sagas**), read models (**CQRS Projections**), and verifying business scenarios (**Business Actions / Use Cases**). The model can generate callable application code for Go, C#, Java, Kotlin, TypeScript, and Python, with optional HTTP adapters and an owned GENERATED.md manifest.

This guide combines theory and practice:

1. **Theory** briefly explains DDD, Event Storming, Event Sourcing, and CQRS independently of any particular tool.
2. **Practice** shows how dddesigner implements these principles through an end-to-end online store example.
3. **Behavior verification** shows how a business intention is connected to the model and checked before implementation.

> The terminology in this guide follows Eric Evans's *Domain-Driven Design* and Vaughn Vernon's *Implementing Domain-Driven Design*. Concise explanations are also available in the application interface through the **“?” (Help)** button on the toolbar of the relevant view.

Part I. Strategic Design

1.1. Ubiquitous Language

A Ubiquitous Language is an agreed vocabulary shared by developers and domain experts. No additional translation should be needed between a discussion of business processes and the code. For example, if the team says “place an order,” the code should express that action as `PlaceOrder`, not as an abstract `CreateEntity(type=1)`.

The Ubiquitous Language does not have to be the same throughout an entire company. It applies within a particular model. For example, “customer” may refer to different entities with different attributes and rules in sales and support contexts. Such differences help identify context boundaries.

1.2. Bounded Context

A **Bounded Context** is an explicitly defined boundary within which a model and its terms have unambiguous, consistent meanings. Instead of one unified model for an entire large system, DDD proposes several specialized models. Each model describes its part of the domain precisely, while the ways in which the models interact are defined explicitly.

In an online store, separate Bounded Contexts might include **Ordering**, **Catalog**, **Payment**, **Inventory**, and **Shipping**.

In dddesigner

A context can be created in the **Project** tree (**Contexts → “+”**) or directly on the **Context Map** canvas. Each context has a name, a description, and a set of aggregates. Assigning an aggregate to a context identifies the part of the model to which it belongs.

1.3. Context Map and Strategic Patterns

A **Context Map** shows relationships between Bounded Contexts. The type of each relationship defines the nature of the integration, the distribution of responsibility, and the direction of influence between teams or systems:

Pattern	Meaning
Shared Kernel	A shared part of the model that both teams change by agreement
Customer-Supplier	The customer team states integration requirements, and the supplier team accounts for them in its development cycle
Conformist	The dependent context adopts the supplier's model without being able to influence it

Pattern	Meaning
Anti-Corruption Layer (ACL)	A translation layer that protects the internal model from an external or legacy model
Open Host Service (OHS)	A context provides a stable interface for multiple consumers
Published Language	A documented interchange format, often used with OHS
Partnership	Two teams plan changes together and coordinate releases
Separate Ways	A deliberate decision not to integrate when independent implementations cost less than a shared connection

In dddesigner

On the **Context Map** canvas, contexts are linked with the **Connect** tool. The relationship type—`customer_supplier`, `partnership`, `shared_kernel`, `acl`, `ohs`, `conformist`, or `published_language`—is selected in the **Properties** panel on the right. In the DSL, relationship participants are written as `kind(upstream, downstream)`. The events (`events[]`) that form the contract between contexts are specified there as well. Events are selected from the model catalog rather than entered as arbitrary text.

The same map is described in the DSL as follows:

```
context_map {
  customer_supplier(ordering, catalog) {
    description: "Ordering consumes product catalog"
    events: [catalog.ProductPriceChanged, catalog.ProductDiscontinued]
  }

  acl(ordering, legacy_billing) {
    description: "Anti-corruption layer for legacy billing system"
  }

  ohs(catalog, ordering) {
    events: [catalog.ProductCreated, catalog.ProductPriceChanged]
  }
}
```

The collapsible **Model** panel on the left contains a common catalog of model objects and relationships. From it, you can navigate to an object on the canvas and check whether all required elements are represented on the map.

Part II. Event Storming—Exploring the Domain

2.1. Why Use Event Storming

Event Storming, introduced by Alberto Brandolini, is a collaborative format for domain experts and developers. Participants explore processes through **Domain Events**—significant facts that have already occurred.

Instead of class diagrams, participants place colored sticky notes on a large board and arrange events chronologically. This approach helps teams:

- reconstruct the actual process using the knowledge of the whole team rather than one participant's view;
- discover gaps and contradictions, because different names or interpretations of the same event become immediately visible;
- identify potential context boundaries where terminology, rules, or team responsibilities change.

2.2. Basic Elements and a Common Color Scheme

Sticky note	Color	Wording	Example
Domain Event	orange	a fact that occurred, usually phrased in the past tense	`PaymentReceived`, `OrderPlaced`
Command	blue	an intention or action that may produce an event	`ProcessPayment`, `PlaceOrder`
Policy (reactive rule)	purple	“When an event occurs, execute a command”	“When `PaymentReceived` → `ShipOrder`”
Actor	yellow	a participant who initiates a command	“Customer,” “Warehouse operator”
Read Model	green	information on which a user bases a decision	“Shopping cart page”
External System	pink	an external source or recipient of data	“Payment gateway”
Hot Spot	red	an open question, disputed area, or risk	“What happens when an item is out of stock?”

Work usually begins with the **Big Picture** stage: participants record events freely without forcing them into a strict structure. During **Process Modeling**, they arrange events over time and add actors, commands, and reactive rules. If needed, the team proceeds to the **Design Level**, where it refines aggregate boundaries and assigns commands and events to them.

2.3. Event Storming in dddesigner

Event Storming opens as a separate view through the **View** menu or command palette. Detailed data structure is intentionally hidden here: the canvas focuses on event sequence and domain language, while the aggregate's detailed structure is edited in the **Aggregate** view.

- **Command** (blue), **Event** (orange), and **Policy** (purple) sticky notes are created on the canvas and form a flow: `command → emits → event`, `event → triggers → policy`, `policy → send → command`. Instead of sending a command, a policy may explicitly publish a new event through `emit`, but it must not republish the event that triggered it.
- Commands and events are synchronized with aggregates. Once semantically linked, they become part of the shared project model and appear in the **Aggregate** view, so their names do not need to be entered again.
- An empty canvas displays a prompt to add commands and events.
- The **“Open…”** button on a sticky note takes you to the tactical editor, where the aggregate structure can be described in detail.

The Event Storming result becomes the initial layer of the shared model, rather than a separate diagram that must be transferred manually after the workshop. The same model can then be refined consistently in the **Aggregate** view.

Part III. Tactical Design: Aggregates

3.1. Entities and Value Objects

- An **Entity** has a persistent identity. Its properties may change completely, yet it remains the same object. For example, an order line named `OrderLine` may have its own identifier.
- A **Value Object (VO)** has no identity of its own and is compared by the values it contains. Value Objects are usually immutable. Examples include `Money`, `Address`, and `Quantity`. Rules that belong to one concept fit naturally within a Value Object—for example, “quantity must be positive.”

3.2. Aggregate and Aggregate Root

An **Aggregate** is a group of Entities and Value Objects that change consistently. Its boundary determines which business rules must be verified within one transaction.

Every Aggregate has exactly one **Aggregate Root**. External code accesses the Aggregate through its Root and does not modify internal objects directly. Other Aggregates usually store only the Root's identifier, rather than a direct object reference. This reduces coupling and allows Aggregates to be loaded and changed independently.

Design Aggregates to be as compact as possible without violating the required consistency. Excessively large Aggregates increase the amount of data loaded and the likelihood of conflicts during concurrent changes. They also often combine several independent consistency boundaries.

3.3. Commands, Events, and Invariants

- A **Command** expresses an intention to change state, such as `SubmitOrder`. If executing a Command would violate an Invariant, the Aggregate must reject it.
- A **Domain Event** describes a significant fact that has already occurred, such as `OrderPlaced`. When the structure of stored Events changes, their versions and the transformation of old representations into new ones must be considered (see section 4.2).
- An **Invariant** is a business rule that must hold after every operation on an Aggregate. Examples include “the order total cannot be negative” and “a cancelled order cannot be shipped.”

3.4. Policies, Repositories, and Domain Services

- A **Policy** defines a reaction to an Event: when a particular fact occurs, the system sends the next Command (`send`) or explicitly publishes a new Event (`emit`). The reaction may remain within one Aggregate or connect several Aggregates.
- A **Repository** provides an interface for loading and saving Aggregates by identifier, for example `FindById` and `Save`.
- A **Domain Service** implements a domain operation that does not naturally belong to a particular Entity or Value Object. An example is a calculation that uses data from several domain objects.

3.5. The Tactical Model in dddesigner

The **Aggregate** tab can be opened by double-clicking an Aggregate in the Project tree or through the Context Map. The canvas displays the Aggregate's structural and behavioral elements: **Root, Entity, VO, Enum, Command, Event, Policy, Reference, Repository**, and **Domain Service**. Relationships show the model's composition and interactions—for example, `composition`, `emits`, and `triggers`.

Elements are created as follows:

Element	Where to create it
Entity, Value Object, Enum	**Structure** panel or creation commands in the tree
Command, Event, Policy	**Behavior** panel
Reference—a link to another Aggregate	**Infra** panel; select the target Aggregate in the dialog
Repository, Domain Service	**Infra** panel; one Repository is provided for an Aggregate type
Attributes of Root, Entity, VO, Command, and Event	**Properties** section, **"+Attribute"** command; enter the name and type on one line
Invariants, methods, and back-references	**Properties** section of the selected element

A `Command → emits → Event` relationship is created by selecting the Event from a list. This prevents a Command from referring to an Event that does not exist in the model. The Event that triggers a Policy and its explicit action—`send` for a Command or `emit` for an Event—are selected in the same way.

The following DSL defines an `Order` Aggregate corresponding to a model created on the canvas:

```
aggregate Order {
  root Order {
    id: OrderId
    customer_id: CustomerId
    status: OrderStatus
    lines: OrderLine[]
  }

  entity OrderLine {
    id: OrderLineId
    product_id: catalog.ProductId
    quantity: Quantity
    unit_price: money
  }

  value Quantity {
    value: int
    invariant { value > 0 }
    invariant { value <= 1000 }
  }

  enum OrderStatus { Draft Placed Paid Shipped Delivered Cancelled }

  command PlaceOrder {
    input {}
    emits OrderPlaced
    rejects "empty order", "not in draft status"
    invariant "must have at least one line" { count(lines) > 0 }
    invariant "must be draft" { status == OrderStatus.Draft }
  }

  event OrderPlaced v1 {
    order_id: OrderId
    placed_at: timestamp
    total: money
  }

  event OrderReadyForPayment v1 {
    order_id: OrderId
    total: money
  }

  policy OrderPlacedPolicy {
    on OrderPlaced {
```

```
      when status == OrderStatus.Placed { send payment.ChargeCustomer; }
    }
  }

  repository OrderRepository {
    Save(order: Order)
    FindById(id: OrderId) -> Order?
  }
}
```

The canvas and DSL are two views of the same model. Canvas changes are reflected in the DSL; DSL changes are parsed, validated, and transferred to the canvas. The underlying data is stored in PostgreSQL as JSON.

Part IV. Storing State as Events

4.1. The Core Idea

Event Sourcing is a way of storing state in which the source of truth is not an Aggregate's current state, but the sequence of Events that led to it. State is reconstructed by applying those Events in order.

The main benefits are:

- **Change history.** The system stores a sequence of meaningful changes, not only the final state.
- **State replay.** Given a complete history and correct handlers, the state of an Aggregate can be reconstructed at a particular point in time.
- **Diagnostics.** The Event sequence helps explain system behavior and why state changed.
- **Read-model construction.** Events can update multiple specialized read models (see Part V).

This approach makes Event schema evolution more difficult. Stored Events are immutable and may remain in use far longer than the code version that created them. Old versions must therefore remain readable, and transitions between versions must be defined explicitly.

4.2. Event Versions and Data Transformation

Every Event in dddesigner has a version: `v1`, `v2`, and so on. When an Event's structure changes, its previous version is retained and a new version is added separately. An `upcast` block describes the data transformation from the old version to the new one:

```
event OrderPlaced v1 {
  order_id: OrderId
  placed_at: timestamp
}

event OrderPlaced v2 {
  order_id: OrderId
  placed_at: timestamp
  customer_id: CustomerId       // new field
}

upcast from v1 to v2 {
  order_id    -> order_id
  placed_at   -> placed_at
  customer_id -> CustomerId.zero()   // default value
}
```

Diagnostic `D006` detects gaps in the version sequence, such as `v1` and `v3` without `v2`. This error blocks code generation and pushing changes to Git until the sequence is corrected. A separate type is created for each version, such as `OrderPlacedV1` and `OrderPlacedV2`, and conversion between versions is implemented as a separate function.

4.3. Configuring Event Sourcing

The mode is set in Project properties or directly in the DSL:

```
project Shop {
  event_sourcing: true
  default_broker: nats
}
```

At present, `event_sourcing: true` marks an architectural decision in the model. To generate Chronacta event store adapters, schemas, and projection manifests, set `codegen.eventstore: chronacta` in the codegen stack (default `none` leaves generated code unchanged). See CHRONACTA.md.

4.4. Chronacta runtime (opt-in)

When `codegen.eventstore: chronacta` is enabled, DDDesigner emits:

- `.dddesigner/chronacta-manifest.json` — aggregate contracts, streams, schemas, and `projection_graph`
- **Event** JSON Schemas (`deploy/chronacta/schemas/`) with `x-dddesigner-*` extensions
- **Command** JSON Schemas (`deploy/chronacta/commands/`) — application contracts; **not** registered in Chronacta Schema Registry
- Aggregate adapters: `Load` → fold → validate → `append(expected_version)` (plus idempotency / runtime metadata)
- Server-side projection manifests for **simple** read models (CEL/builtin)
- Durable subscription worker scaffolds for **complex** read models
- Postgres read-model `schema.yaml` + connector configs when a projection has `target: postgres` (columns and `source:` come from **handler updates** `field = event.x`, not from attributes alone)
- Dry-run `apply.sh` and a baseline `compatibility-report.json`

A separate flag `deployment.chronacta.enabled` adds only a local Compose profile (`deploy/chronacta/compose.yaml`). It does **not** enable contracts and does **not** register schemas automatically.

CLI with no remote calls:

```
dddesigner chronacta diagnostics --model model.json
dddesigner chronacta compatibility --model model.json --previous .dddesigner/chronacta-manifest.json
```

Blocking diagnostics (Chronacta artifacts are skipped): unresolved command/projection events, conflicting stream patterns, invalid event versions, impossible postgres mappings. Details: CHRONACTA.md.

Without `eventstore: chronacta`, projection scaffolds and SQL migrations are generated exactly as before. Canvas **Event Flows** remain a design view and are not the Chronacta runtime.

4.5. Immutability: corrections, compensation, and retention

After a domain event is **successfully appended**, it is not edited or deleted through the normal application API — whether you use Chronacta or your own event store. Fixes and reversals are modeled as **new facts** in the domain model.

Intent	Model in dddesigner	Storage effect
Fix wrong data	Command `CorrectOrderAddress` → event `OrderAddressCorrected`	New append to aggregate stream
Business reversal	Command / saga compensation → `OrderRefunded`	New append; saga may orchestrate steps
Schema change	New event version + `upcast` (§4.2)	Old records stay; readers upcast on load
Read model cleanup	Projection handler `delete`	SQL row removed; **event log unchanged**
Physical purge	Operator procedure (Chronacta scavenge or your retention job)	Ops-only; consumers must tolerate or rebuild

With and without Chronacta

The **same model** applies in both cases. Chronacta enforces append-only at the server; without Chronacta you enforce it in your `EventStore` implementation (INSERT-only table, no UPDATE/DELETE on events from app code).

Automatic correction pairs

When `event_sourcing: true`:

- Adding a new domain event (except `*Corrected`, `*Refunded`, etc.) automatically creates **`Correct{Event}`** and **`{Event}Corrected`** with fields copied from the source event plus optional `reason`.
- On an existing event, use **Properties → Add correction pair** if the pair was not created yet.
- Diagnostic **W018** warns when an event has no correction pair.

Rename generated names in the model if the domain language differs (`CorrectOrderAddress` / `OrderAddressCorrected` instead of `CorrectOrderPlaced` / `OrderPlacedCorrected`).

Compensation and refunds are **not** auto-generated for every event — model them explicitly as domain events or saga compensations (`OrderRefunded`, `RefundPayment`).

```
event OrderPlaced v1 {
  order_id: OrderId
  shipping_address: string
```

```
}

event OrderAddressCorrected v1 {
  order_id: OrderId
  shipping_address: string
  reason: string?
}

command CorrectOrderAddress {
  emits [ OrderAddressCorrected ]
}
```

Part V. CQRS and Read Models

5.1. The Core Idea of CQRS

Command Query Responsibility Segregation (CQRS) separates models responsible for changing data from models intended for reading it.

The write model consists of Aggregates and enforces Invariants. A read model is shaped for particular queries, screens, or API responses. A structure suitable for enforcing business rules is usually inconvenient for complex lists, filtering, and aggregation. Conversely, a denormalized read model must not be used as the basis for decisions that change state.

CQRS works well with Event Sourcing: stored Events can update several specialized read models. However, CQRS does not require Event Sourcing and can be applied independently.

When Projections are updated asynchronously, the read model may temporarily lag behind the write model. This **eventual consistency** must be considered in user journeys and error handling.

5.2. Projections in dddesigner

The **Projections** tab displays the processing chain:

FROM EVENT → HANDLER → READ MODEL

- The **Handler** button adds a handler and synchronizes the `from_events` list, so the source Event list does not have to be updated separately.
- Values for `from_events` and `trigger_event` are selected from the model's Domain Event catalog.
- Projection **attributes** only declare the read-model shape (field name and type). They do **not** define data sources.
- **Mapping** “projection field ← event field” is set on the **handler**: updates such as `order_id = event.order_id`, `status = event.status`. Codegen (including Chronacta Postgres `schema.yaml` / `source:`) uses those expressions.
- A handler may also delete a row (`delete`) for ordinary SQL-scaffold projections; for `target: postgres`, `delete` and literals are not supported.

In projection Properties, **Read model storage** chooses the default (Chronacta JSON / in-app worker) or **Postgres (Chronacta connector)**. Requires `codegen.eventstore: chronacta`.

With `codegen.eventstore: chronacta`, **simple** handlers (`field = event.field`) become Chronacta server-side projection manifests; **complex** handlers become durable subscription workers that call the generated `{Name}Apply` scaffold. Set `target: postgres` on a projection to map the same fields/handlers into Chronacta Postgres `schema.yaml` + connector artifacts (sqlgen owns DDL; app `migrations/projections` is skipped). See §4.4 and CHRONACTA.md.

```
projection OrderSummary {
  from_events [ ordering.OrderCreated, ordering.OrderPlaced, ordering.OrderCancelled ]

  fields {
    order_id: ordering.OrderId
    status: ordering.OrderStatus
    total: money
    placed_at: timestamp?
  }

  handler on ordering.OrderCreated {
    order_id = event.order_id
    status   = OrderStatus.Draft
  }
```

```
  handler on ordering.OrderPlaced {
    status     = OrderStatus.Placed
    total      = event.total
    placed_at  = event.placed_at
  }
}
```

From a `projection` definition, the generator creates a read-model structure, an Event application handler, and a SQL migration for the Projection table. The model structure is generator-managed, while the handler and SQL migration are scaffolds: each is created only when the corresponding file does not exist. Manual changes are therefore preserved.

Part VI. Multi-Step Processes: Saga and Process Manager

6.1. The Purpose of Sagas

An Aggregate defines a boundary of strict transactional consistency. Changes across several Aggregates—especially those in different Bounded Contexts—are usually coordinated through a sequence of Commands and Events, not one shared transaction.

For example, placing an order may involve reserving stock, processing payment, and confirming the order. A **Saga** or **Process Manager** coordinates such a process. The coordinator receives Events, sends Commands to process participants, and tracks current state. Consistency between individual steps is eventual.

When a completed step must be reversed from a business perspective, the process sends a **compensating Command**. For example, it may execute `ReleaseStock` after a failed payment. Compensation does not literally undo a completed transaction; it creates a new action that offsets the transaction's business consequences.

6.2. Orchestration and Choreography

- **Orchestration** uses a dedicated coordinator. It sends Commands, receives result Events, and determines the process's next step.
- **Choreography** has no single coordinator. Each participant responds to other participants' Events, and the process sequence emerges from those reactions.

Orchestration makes a long-running process easier to track, but concentrates its logic in one component. Choreography removes the central point of control, but as the number of participants grows, it becomes harder to understand the overall flow and handle failures.

6.3. Sagas in dddesigner

The **Saga** canvas uses four node types:

Node	Appearance	Purpose
START	purple	Entry point; `starts_on` specifies the Event that starts the process
STEP	blue	Main step: send a Command with `send`, wait for an Event with `wait_for`, or publish an Event with `emit`
ON FAIL	dark red	Saga-level compensation; no more than one is allowed
STEP FAIL	orange	Compensation for an individual step; no more than one per step is allowed

```
saga PlaceOrderFlow orchestration {
  starts_on ordering.OrderPlaced

  step ValidateInventory {
    send inventory.ReserveStock(order_id: event.order_id, lines: event.lines)
    wait_for inventory.StockReserved timeout 30s
  }

  step ProcessPayment {
    send payment.ChargeCustomer(order_id: event.order_id, amount: event.total)
    wait_for payment.PaymentCompleted timeout 60s
  }

  compensate HandleFailure {
    send inventory.ReleaseStock(order_id: event.order_id)
    send payment.RefundCustomer(order_id: event.order_id)
```

```
  }
}
```

The **“Rollback”** button creates compensation at the appropriate level for the selected element. Selecting the **START** point or the whole Saga creates an **ON FAIL** node. Selecting a particular step creates a **STEP FAIL** node. This keeps whole-process compensation separate from the failure handling of an individual step.

Part VII. Business Actions and Behavioral Verification

7.1. From business intention to model

Commands, Events, Policies, Sagas, and Projections describe parts of a system's behavior. On their own, however, they do not answer a basic question: **what business action is the user performing, and what result should the system produce?**

A **Business Action (Use Case)** is a named scenario such as `CreateUser` or `PlaceOrder`. It is the recommended starting point for design: capture the actor, goal, and intended business outcome before inventing a technical flow. As the model develops, the same Business Action is connected to a starting Command and becomes a verifiable contract:

```
Business Action
        ↓
Command
        ↓
Events, Policies, Sagas, and Projections
        ↓
Expected and forbidden outcomes
```

A Business Action is created only after its starting Command exists: creation requires a name and an unambiguously resolved start command; actor, goal, and outcome assertions can then be refined. The starting point is command-only: it must resolve unambiguously to an existing Command. Domain or Integration Events and Gateway routes are not executable Use Case starts; HTTP is configured separately as an optional exposure of the command-backed application callable. Assertions target real model elements: Events, Policies, Sagas, Projections, and Projection field-update rules.

7.2. Business Action DSL

Business Actions are stored in the virtual DSL file `use_cases.ddd`. For example:

```
use_case CreateUser {
  description: "Register a new user"
  actor: "Visitor"
  input {
    user_id: uuid;
    email: string;
    display_name: string?;
  }
  start command Identity.User.RegisterUser;
  expected event Identity.User.UserRegistered;
  expected projection UserDirectory;
  expected projection_update UserDirectory.id = "event.user_id";
  forbidden event Identity.User.RegistrationRejected;
}
```

`expected` means that the declared outcome must be reachable from the starting point. `forbidden` means that the outcome must not appear in the trace. References must resolve to existing model elements; arbitrary strings are not a substitute for model relationships.

The `input` block declares the typed application parameters. For a `start command`, these fields must match the target Command input by name, type, order, and optionality; diagnostic `D053` reports a mismatch. The block may be omitted for compatibility, in which case code generation derives the input from the target Command.

A Business Action describes an intention and its verification criteria rather than copying the sequence of Commands and Events. When the domain flow changes, the scenario continues to point to model objects and makes the expectations that need review explicit.

7.3. Structural simulation

Direct structural simulation can inspect a Command or Event. A Business Action always starts from its declared Command and shows the declared chain:

```
Command accepted
  → event created
  → policy fired
  → next command or event
  → projection handler found
  → projection update declared
```

The result includes the detailed causal trace (including parent links and assertion direction metadata), created Domain Events, updated Projections, started Sagas, waiting Sagas, and available compensations. For a Business Action, it also shows a `Passed` or `Requires work` verdict and the result of every assertion.

The simulator is **structural**, not a runtime integration test. It checks whether declared model relationships are reachable, but it does not execute:

- Aggregate business code or handlers;
- Policy conditions or Invariants;
- Projection update expressions;
- database queries or read-model state;
- real queues, brokers, or external systems.

A passing simulation therefore means that the model describes the expected path consistently. It does not replace unit, integration, or end-to-end tests of the application.

7.5. Business Action Canvas

Selecting a Business Action opens a read-only canvas built from the same structural simulation used by verification. The graph starts at the declared Command and follows declared model relationships: command `emits`, qualified Policy triggers and `send` / `emit` actions, Saga starts and steps, and Projection handlers. Matching prefers canonical `context.Aggregate.Member` identity; ambiguous bare names do not match. A Saga advances past `wait_for` only when the matching qualified Event is reached, so a wait boundary is visible and cannot be crossed merely because another step exists.

Structural expectations check only whether declared model elements are reachable or absent. They do not execute generated code, invariants, policy conditions, persistence, integrations, or HTTP requests. A Business Action without expectations is reported as not configured rather than passed.

The canvas shows every reachable branch. `expected` and `forbidden` assertions remain the manual business meaning layered on top of the computed graph. Moving nodes changes only the saved layout for that Business Action; users do not draw duplicate causal edges on this canvas. Conditions and invariants are displayed as not evaluated because structural simulation does not execute runtime code.

The canvas is deliberately different from Event Flows and Saga design: those views edit architecture, while Business Action Canvas explains one selected scenario in the current model.

7.4. Generated Business Actions

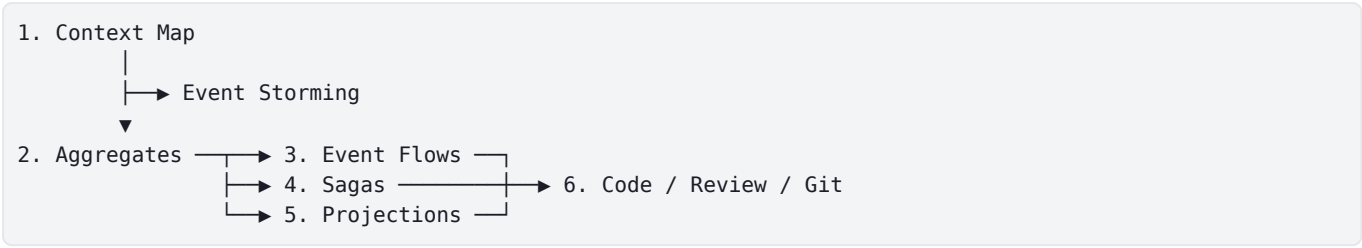
Every generation result includes one owned `GENERATED.md`. It records the generation version and language, every owned and scaffold file with a plain-language description of its purpose, each generated Business Action callable and typed input, and any configured HTTP method, path, authorization mode, and adapter path.

Part VIII. The dddesigner Interface: Workflow

8.1. Main Panels

- **Project** is a tree for navigating and creating Contexts, Aggregates, Sagas, and Projections. When the Project is selected, the **Properties** panel displays code-generation and Git connection settings.
- **Canvas** displays the structure and relationships of the current view. Each view is responsible for one model level: for example, a Command is edited in **Aggregate**, while **Event Flows** provides an overview of flows.
- **Properties** is a contextual panel for names, references, attributes, Invariants, warnings, and workflow stages. Relationships are selected from lists rather than entered manually.
- **DSL editor** is a textual view of the same model. By default, it appears as a narrow panel at the bottom of the center column and can show either the strategic model or the current Aggregate.
- **Model** is a collapsible catalog of objects and relationships with quick navigation to an element on the canvas.
- **Generated Code / Reviews** displays generated output and model discussions. The model itself is not edited there.
- **Status bar** shows connection and synchronization state, generation configuration, warnings, model version, and undo/redo commands.
- **Toolbar** creates and connects objects for the current canvas.

8.2. Recommended Workflow



The diagram shows the recommended sequence. When nothing is selected on the canvas, **Properties** displays a six-step checklist from **Context Map** through **Code / Review / Git**. Event Storming opens separately through the **View** menu or command palette. It is useful before Aggregate details are introduced and whenever new processes are explored.

Deployment topology is opt-in and does not alter the DDD model; it is absent by default, so existing projects keep their monolith output. Use the optional **Service Map** workspace view to choose topology, project-wide internal transport, Compose project, and service boundaries. It remains outside the six-step workflow. Without explicit boundaries, it shows the inferred service-per-context default; it also shows cross-service context relations, codegen diagnostics, and deployment-only previews. Changes save through the canonical Design Model, while previewed files become persisted artifacts only after **Generate**.

Workflow runtime is also opt-in. With `workflow: temporal`, DDDesigner generates Temporal workflow/activity scaffolds for orchestration sagas in Go, C#, Java, Kotlin, TypeScript, and Python. Choreography sagas remain broker/event-handler based, and messaging transport remains configured separately from the workflow runtime.

8.3. Checking Model Quality

dddesigner detects structural and semantic problems through `GET /api/v1/projects/:id/diagnostics`. Diagnostics appear in **Properties**; selecting a message navigates to the corresponding object.

Code	Severity	Check
D001	Error	duplicate Aggregate name
D002–D003	Error	an Aggregate has no Root, or its Root has no `id` field
D004	Error	a Command publishes an undeclared Event
D005	Error	a Reference points to a missing Aggregate
D006	Error	a gap exists in an Event version sequence
D007	Error	a Policy `emit` action refers to an unresolved Event
D008	Error	a Policy `send` action refers to an unresolved Command
D009	Error	a Policy action has an unknown kind, lacks a required target, or specifies both targets
D010	Error	a Command name is duplicated within an Aggregate
D011	Error	a Policy name is duplicated within an Aggregate
D012	Error	a Saga name is duplicated in the model
D013	Error	a Projection name is duplicated in the model
D014	Error	a Domain Event name-and-version pair is duplicated within an Aggregate
D050–D052	Error	a Business Action has an invalid name, start, or assertion
D053	Error	a Business Action input does not match its starting Command
W001	Warning	an Aggregate contains more than seven Entities
W002	Warning	a Command has no Invariants
W003	Warning	an Integration Event has no description
W004	Warning	a cross-context Reference is not protected by an ACL
W005	Warning	an Event has no more than one field
W006	Warning	a Command publishes no Events
W007	Warning	a Policy has no triggering Event
W008	Warning	a Policy's triggering Event is not found in the Aggregate
W009–W011	Warning	a Projection is not linked to existing Events
W012	Warning	a Saga has no triggering Event
W013	Warning	a Policy has no actions
W014	Warning	a Policy publishes the same Event that triggers it

Code	Severity	Check
W015	Warning	a Policy action with the same kind and target is duplicated
W016	Warning	a Saga's triggering Event is not found in the model
W017	Warning	a Saga has no steps
W018	Warning	an Event has no correction pair while event_sourcing is enabled

The **View → Model Health** panel supplements these checks by analyzing large Aggregates, isolated Contexts, and cycles in the Context Map. You can navigate directly from the report to the affected object.

8.4. DDD Guide—The Built-In Assistant

The **View → DDD Guide** panel combines two sources of guidance:

1. **Model rules** find missing elements and suggest ready-made actions that can be applied with one click.
2. **Answers to questions** first use built-in rules and reference material. If those are insufficient and a language model is available, it explains the concept or recommendation without inventing objects that do not exist in the Project.

For example, the assistant notices an excessively large Aggregate, a Command without an expected result, or an incomplete Saga. It helps review a design, but it does not replace the team's architectural decisions.

8.5. Code Generation and Git

The **Generated Code** panel supports Go, C#, Java, Kotlin, TypeScript/Node, and Python. The language, HTTP framework, database adapter, message broker, and dependency-injection mechanism are configured in Project properties. Before using generated code in production, consider the maturity level shown in the interface for the selected stack.

Optional deployment topology

A project may opt into `monolith`, `modular_monolith`, or `microservices`; each project selects `https` or `grpc` for internal transport. In the optional Service Map view, explicit service boundaries assign every Bounded Context exactly once; without them, codegen infers one service per context. The view's preview is codegen-derived and is not a promise of persisted artifacts before **Generate**. For valid non-monolith deployment, generation emits owned OpenAPI/proto contracts, a deployment README, topology/preview/dependency metadata, and scaffold Dockerfiles plus a root Compose file. The generator does not yet create complete language-specific server or client stubs; Kubernetes, Helm, service mesh, and mTLS are not provided, and not every deployment-schema field fully changes the renderer. Merge package dependency metadata into the user-owned project descriptor.

Generated files belong to two groups:

- **Generator-managed** (`owned`, ↻ icon, `DO NOT EDIT`)—the domain model, Command and Use Case contracts, service ports, Saga definitions, read models, and HTTP routes. They are replaced on regeneration.
- **Scaffolds** (`scaffold`, ✎ icon)—Command handlers, Saga executors, Projection handlers, migrations, and Repository adapters. They are created only if the file does not already exist, so manual changes are preserved.

Persistence generation is now available for all target languages at the repository-adapter level. Go keeps its existing adapters (`pgx`, `sqlc`, `ent`, `gorm`). C# emits EF Core bridge scaffolds for `efcore`; Java emits JPA/Spring Data bridge scaffolds for `jpa`; Kotlin emits Exposed bridge scaffolds for `exposed`; TypeScript emits Prisma bridge scaffolds for `prisma`; Python emits SQLAlchemy bridge scaffolds for `sqlalchemy`. These files intentionally do not generate a complete database schema or force generated domain objects to become ORM entities. Instead, they define a small store boundary that the application team wires to `DbContext`, `EntityManager`, Exposed transactions, Prisma Client, or SQLAlchemy Session and maps to the domain model.

The generator also writes ORM dependency hints to `.dddesigner/generated-dependencies.json`. Treat this JSON as build metadata: merge the listed packages into the user-owned project descriptor, then implement the scaffold store with the concrete runtime configuration, transactions, migrations, and indexes required by the application.

For every Business Action with an unambiguously resolved `start command`, the generator emits an owned application wrapper for all six supported languages. The wrapper defines the Use Case input, assembles the existing Command, and delegates it to a typed handler interface. It contains no generated business decisions: Aggregate invariants remain in the domain model and the scaffold Command handler remains the manual extension point. Business Actions with optional `expose http METHOD "/path" auth authenticated;` also produce a transport adapter that decodes typed input and invokes the callable. The adapter declares the application HTTP endpoint; production HTTPS is provided by TLS termination at a reverse proxy, ingress, or load balancer such as Caddy. Legacy Event and Gateway starts produce diagnostics and no runtime Business Action files because the canonical contract is command-only. For exposed actions, each target emits an explicit route-registration function/module and DI composition hook; authenticated is the default when `auth` is omitted, while `public`, `authenticated`, and `admin` select distinct guards. Auth middleware, identity/role lookup, handlers, repositories, and framework startup remain scaffold extension points. `.dddesigner/generated-dependencies.json` lists

framework packages that must be merged into the user-owned build descriptor. Unexposed actions generate no HTTP adapter or route. Regeneration is deterministic for identical canonical model and configuration.

The output can be downloaded as a ZIP archive or pushed to a connected Git repository together with the DSL files from `design/`.

8.6. Additional Tools

- **Tools → Model releases** provides named model snapshots and comparisons of objects added, removed, or renamed between releases.
- **Tools → Export** supports PNG and SVG for presentations, plus Markdown, Mermaid, PlantUML, and C4/Structurizr for repository documentation.
- The **Command palette** (`Ctrl+K` or `Ctrl+P`) searches Aggregates, Events, Sagas, Projections, gateways, and DSL files, and provides quick access to checks, releases, and exports.
- **Tools → People** sends email invitations with the architect, developer, and viewer roles; viewer grants read-only access. The same dialog holds **Agent keys** for external AI assistants (see § 8.7). Active participants appear in the title bar.
- **Undo and redo** use `Ctrl+Z` and `Ctrl+Shift+Z` to restore previous model states. History is stored on the server and remains available after the page is reloaded.

8.7. Designing with an external AI assistant

DDDesigner lets you refine a domain model not only in the web workspace, but also in the tools where you already talk to an assistant: a code editor, a separate chat, or automation scripts. The assistant does not replace the editor. It proposes changes; DDDesigner validates them against model rules and persists them only after an explicit safe cycle—preview first, then apply.

Why this exists

It is natural to refine a model in conversation: “add an ordering context”, “link the command to the event”, “check the saga for gaps”. An external assistant is good at drafting such steps, but it must not freely rewrite other people’s projects. Access therefore rests on a **personal agent key** bound to **one project** and to **your account**. The assistant’s rights are never wider than yours on that project.

How to obtain a key

1. Open the project (diagram) in the workspace.
2. Choose **Tools → People**.
3. Scroll to the **Agent keys** section.
4. Give the key a clear name (for example, “work laptop”) and choose **read only** or **read and write**.
5. Click **Create agent key**. The full secret is shown **once**. Copy it immediately to a safe place.
6. Close the dialog. The secret cannot be shown again—only revoke and create a new key.

Any participant with at least the developer role may create a key. Viewers do not see this section. An architect may revoke other people’s keys on the project when access must be cut off quickly.

The key acts as you: the same organisation and project isolation rules apply as for a normal sign-in. Requests to another project with this key are rejected.

Where to paste the key

Paste the secret **outside** DDDesigner—into the environment where the assistant runs:

- cloud MCP configuration for Cursor, Claude, Codex, or Gemini (`https://app.dddesigner.com/mcp` with header `X-Agent-Key`);
- ChatGPT Custom GPT action authentication (header secret);
- the shell environment for DDDesigner command-line tools (URL, key, and project identifier).

The workspace **People → Agent keys** section shows the MCP URL and a copyable configuration template (placeholder secret only). MCP clients do **not** need a separate project id: it is already bound into the key.

Do not commit the key to a code repository or paste it into public chats. If you suspect a leak, open **People → Agent keys → Revoke** and create a new key.

How the dialogue should proceed

The recommended order is the same in every external tool:

1. **Model summary** — load a compact picture of the current state and version (`whoami` / `get_summary`).
2. **Operation catalogue** — when planning edits (`ops_catalog`); do not invent operation names.
3. **Proposal** — a set of precise operations or an edit to the textual design description.
4. **Preview** — dry-run without saving (`preview_ops`); the system returns errors and warnings.
5. **Apply** — only after a successful preview and your explicit approval (`apply_ops`).
6. **Diagnostics** — another quality check (`get_diagnostics`).

DDDesigner remains the source of truth: canvas and textual description stay in sync, and history and collaboration continue as with manual editing.

Precise operations versus textual description

- **Precise operations** suit small steps: add a command, link it to an event, rename a field.
- **Textual design description** (the DDDesigner design language) suits larger pieces: a whole aggregate, a saga, a projection (`list_dsl_files` / `get_dsl` / `put_dsl`).

Code generation: workspace and assistant

You can obtain a code scaffold in two ways—the pipeline is the same:

1. In the workspace: **Generated Code** panel → **Generate** (preview, ZIP, Git push).
2. Via cloud MCP: `codegen_preview` → `list_codegen_files` → `get_codegen_file` for selected paths.

Fix model errors before generating. Over MCP, do not dump the whole tree—fetch only the files you need. MCP does not push to Git or download a ZIP in the current version; use the workspace buttons for that.

Built-in guide versus external assistant

The in-app **DDD Guide** already suggests model gaps by rules and answers questions. An external AI connects **in addition** when you want a free-form dialogue in a familiar editor. Both paths converge on the same model; the external path requires an agent key.

Full setup (Cursor, Claude, ChatGPT, Codex, Gemini, CLI) and the complete MCP tool list: Working with AI. Short handbook for the assistant itself: HANDBOOK.md.

Part IX. Example: Designing an Online Store from Intention to Verifiable Contract

The recommended workflow is a loop: a business intention sets direction at the beginning and returns at the end as a contract against the completed model.

Step 1. PlaceOrder Business Intention

After the `PlaceOrder` Command exists, create the `PlaceOrder` Business Action in the top-level **Business Actions** folder, select that required starting Command, set the actor to `Customer`, and describe the goal: place an assembled order and receive confirmation. Assertions may be added after the relevant Events and read models exist.

Step 2. Context Map

Create the `ordering`, `catalog`, `payment`, `inventory`, `shipping`, and `legacy_billing` Contexts. Model `ordering` ⇄ `catalog` as **customer_supplier**, expose catalog capabilities through **OHS**, and protect access to `legacy_billing` with an **ACL**. Derive these boundaries from the responsibilities needed by `PlaceOrder` and adjacent Business Actions.

Step 3. Event Storming for PlaceOrder

Use **Event Storming** to unfold the selected intention into a chronological story:

`PlaceOrder` (Command) → `OrderPlaced` (Event) → reservation → payment → confirmation or compensation.

Event Storming is not a separate final artifact here. It discovers the elements and relationships needed to fulfil the Business Action.

Step 4. Order Tactical Model

In the **Aggregate** view, define the `Order` Root, `OrderLine` Entity, `OrderId` and `Quantity` Value Objects, `OrderStatus`, Commands, and versioned Events. Capture the `PlaceOrder` Invariants: the order contains lines and is in an allowed state.

Step 5. Process and Read Model

Create `PlaceOrderFlow`, triggered by `ordering.OrderPlaced`: `inventory.ReserveStock` → `payment.ChargeCustomer` → `ordering.ConfirmOrder`. Add `ReleaseStock` and `RefundCustomer` compensations. Build the `OrderSummary` Projection for the “My orders” screen.

Step 6. Complete the Business Action

Return to the same `PlaceOrder` action. Select `ordering.Order.PlaceOrder` as its start, then add `OrderPlaced`, `PlaceOrderFlow`, `OrderSummary`, and its field updates as expected outcomes. Mark `OrderCancelled` as forbidden in the successful scenario.

Run structural simulation. **Passed** means the declared relationships make every expected object reachable. **Requires work** identifies the part of the contract that the model does not support.

Step 7. Flows, Generation, and Runtime Tests

Use **Event Flows** to inspect the end-to-end chain and event sources. In **Generated Code**, select a technology profile and generate the domain structure, contracts, application wrapper for `PlaceOrder`, its optional HTTP adapter, and `GENERATED.md`. Business rules, integrations, and runtime tests remain the developers' responsibility.

Conclusion: Design Quality Checklist

- [ ] Each Bounded Context has its own vocabulary, and identical terms are not conflated across Contexts.
- [ ] Every relationship on the Context Map has a specified type, not merely a direction.
- [ ] Event Storming was conducted before tactical design; key Events and their names were agreed with domain experts.
- [ ] Aggregates remain small, and other Aggregates are generally referenced by identifier.
- [ ] Every Command has Invariants, or the reason they are unnecessary is documented explicitly.
- [ ] Domain Events are versioned without gaps; old schemas are transformed into new ones through `upcast`.
- [ ] Read models are designed for particular screens or API responses rather than copying Aggregates.
- [ ] Long-running processes are modeled as Sagas, with compensations for failures.
- [ ] Key business intentions are captured as command-backed Business Actions once their starting Commands exist.
- [ ] After modeling, Business Actions are connected to starts and expected or forbidden outcomes, and their structural simulations pass.
- [ ] Warnings in **Model Health** and **Properties** have been resolved or consciously accepted.
- [ ] The language, HTTP stack, database, and message broker have been selected before code generation.
- [ ] If deployment is enabled, its topology, transport, boundaries, contracts, and scaffolds have been reviewed.
- [ ] If an external assistant is used, a personal agent key was created, the secret is kept out of the repository, changes go through `preview_ops` before apply, and generated code is fetched via `codegen_*` or the **Generated Code** panel when needed.

Glossary

Term	Meaning
Ubiquitous Language	the shared language of experts and developers within a Context
Bounded Context	a boundary within which a model and its terms have unambiguous meanings
Context Map	a map of relationships between Bounded Contexts
Aggregate / Aggregate Root	a transactional consistency boundary and its single external entry point
Entity	an object with persistent identity
Value Object	a usually immutable object without its own identity, compared by value
Domain Event	a domain fact that has already occurred
Command	a request to change state that may be rejected
Policy	a rule that responds to an Event and initiates the next process step
Repository	an interface for saving and loading an Aggregate as a whole
Domain Service	a domain operation that does not naturally belong to one Entity or Value Object
Invariant	a business condition that an Aggregate must preserve after every operation
Event Storming	collaborative domain exploration through a sequence of Events
Event Sourcing	storing state as a sequence of Events
Upcasting	transforming an old Event version into the current schema
CQRS	separation of write and read models
Projection / Read Model	a query-specific read model built by Event handlers
Saga / Process Manager	a model of a multi-step process spanning several Aggregates
Orchestration / Choreography	a process with a central coordinator or a distributed chain of Event reactions
Compensation (ON FAIL / STEP FAIL)	an action that offsets an entire Saga or an individual step

Term	Meaning
ACL (Anti-Corruption Layer)	a translation layer between external and internal models
OHS (Open Host Service)	a stable, published interface of a Context
Agent key	a personal secret for an external assistant; bound to one project and your account; for MCP use `https://app.dddesigner.com/mcp` with `X-Agent-Key`
Change preview	dry-run of model edits without saving
Apply changes	persist validated edits into the model

Concise help for each canvas view is available in the application through the "?" button on the toolbar. Implementation details are documented in the design workflow, principles, and design-language documents. Working with an external AI assistant is described in § 8.7 and in Working with AI.

8.4. NATS and Kafka codegen status

NATS and Kafka are generated adapters/scaffolds, not dddesigner runtime integrations. The generated ports are owned contracts; the broker-specific adapter is a scaffold that receives a user-owned minimal client interface or callbacks. Wire that bridge in the composition root using the official client library, then inject the adapter into the application. Generated async surfaces are `Task` in C#, `CompletionStage` / `CompletableFuture` in Java, `suspend` in Kotlin, `Promise` in TypeScript, and `async` / `await` in Python. Topics remain generated constants and payloads remain bytes.

The outbox transport is separate from the event store and Event Sourcing. An outbox records messages for publication after a transaction; Event Sourcing stores the authoritative sequence of domain events and rehydrates aggregates. The generated broker surface does not generate consumer groups, offsets, schema registry configuration, replay, or connection lifecycle management.

Use the generated `.dddesigner/generated-dependencies.json` entries as build metadata only. Add the official library and adapt its API in user-owned composition code: `NATS.Client.Core` or `NATS.Client` and `Confluent.Kafka` for C#, `io.nats/jnats` and `org.apache.kafka/kafka-clients` for Java/Kotlin, `nats` and `kafkajs` for TypeScript, and `nats-py` and `aiokafka` for Python.