DDD, Event Storming, Event Sourcing и CQRS в dddesigner

Практическое руководство для пользователей

Введение

Domain-Driven Design (DDD, предметно-ориентированное проектирование) — подход к созданию сложных программных систем, в котором архитектура опирается прежде всего на **язык и границы предметной области**, а не на структуру базы данных или выбранные технологии. Цель DDD — точно выразить в программе бизнес-правила и процессы, не заслоняя их особенностями инструментов.

dddesigner — среда для проектирования сложных систем: от разделения системы на **ограниченные контексты** (Bounded Contexts) и исследования предметной области с помощью **Event Storming** до проектирования **агрегатов**, многошаговых процессов (**Saga**), моделей чтения (**CQRS Projections**) и проверки бизнес-сценариев (**Business Actions / Use Cases**). На основе модели можно генерировать код на Go, C#, Java, Kotlin, TypeScript и Python, опциональные HTTP-адаптеры и управляемый манифест `GENERATED.md`.

Руководство сочетает теорию и практику:

1. **Теория** кратко объясняет DDD, Event Storming, Event Sourcing и CQRS без привязки к инструменту.
2. **Практика** показывает, как эти принципы реализованы в dddesigner, на сквозном примере интернет-магазина.
3. **Проверка поведения** показывает, как бизнес-намерение связывается с моделью и проверяется до реализации.

> Терминология руководства опирается на книги Эрика Эванса *Domain-Driven Design* и Вона Вернона *Implementing Domain-Driven Design*. Краткие пояснения также доступны в интерфейсе приложения: их можно открыть кнопкой **«?» (Help)** на панели инструментов соответствующего представления.

Часть I. Стратегическое проектирование

1.1. Единый язык (Ubiquitous Language)

Единый язык — согласованный словарь, которым пользуются и разработчики, и эксперты предметной области. Между обсуждением бизнес-процессов и кодом не должно требоваться дополнительного перевода. Например, если команда говорит «разместить заказ», это действие следует выразить в коде как `PlaceOrder`, а не как абстрактное `CreateEntity(type=1)`.

Единый язык не обязан быть одинаковым во всей компании. Он действует в пределах конкретной модели. Например, слово «клиент» в контексте продаж и в контексте поддержки может обозначать разные сущности с разными атрибутами и правилами. Такие различия помогают определить границы контекстов.

1.2. Ограниченный контекст (Bounded Context)

Ограниченный контекст — явно обозначенная граница, внутри которой модель и её термины имеют однозначный и непротиворечивый смысл. Вместо единой модели для всей большой системы DDD предлагает несколько специализированных моделей. Каждая из них точно описывает свою часть предметной области, а способы взаимодействия между ними задаются явно.

В интернет-магазине отдельными ограниченными контекстами могут быть **Ordering** (оформление заказов), **Catalog** (каталог товаров), **Payment** (платежи), **Inventory** (складской учёт) и **Shipping** (доставка).

В dddesigner

Контекст можно создать в дереве **Проект** (**Контексты → «+»**) или непосредственно на холсте **Context Map**. Для каждого контекста указываются имя, описание и входящие в него агрегаты. Привязка агрегата к контексту показывает, какой части модели он принадлежит.

1.3. Карта контекстов (Context Map) и стратегические паттерны

Карта контекстов показывает связи между ограниченными контекстами. Тип каждой связи определяет характер интеграции, распределение ответственности и направление влияния между командами или системами:

Паттерн	Смысл
Shared Kernel	Общая часть модели, которую обе команды изменяют по согласованию
Customer-Supplier	Команда-заказчик формулирует требования к интеграции, а команда-поставщик учитывает их в своём цикле разработки
Conformist	Зависимый контекст принимает модель поставщика без возможности влиять на неё
Anti-Corruption Layer (ACL)	Слой преобразования, который защищает собственную модель от внешней или унаследованной модели
Open Host Service (OHS)	Контекст предоставляет стабильный интерфейс для нескольких потребителей
Published Language	Документированный формат обмена, часто используемый вместе с OHS
Partnership	Две команды совместно планируют изменения и координируют выпуск версий
Separate Ways	Осознанный отказ от интеграции, когда независимая реализация обходится дешевле общей связи

В dddesigner

На холсте **Context Map** контексты соединяются инструментом **Connect**. Тип связи — `customer_supplier`, `partnership`, `shared_kernel`, `acl`, `ohs`, `conformist` или `published_language` — выбирается на панели **Properties** справа. В DSL участники связи записываются как `kind(upstream, downstream)`. Там же указываются события (`events[]`), составляющие контракт между контекстами. События выбираются из каталога модели, а не вводятся произвольным текстом.

В DSL та же карта описывается так:

```
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]
  }
}
```

Сворачиваемая панель **Model** слева содержит общий каталог объектов модели и их связей. Из неё можно перейти к объекту на холсте и проверить, все ли необходимые элементы отражены на карте.

Часть II. Event Storming — исследование предметной области

2.1. Зачем нужен Event Storming

Event Storming, предложенный Альберто Брандолини, — формат совместной работы экспертов предметной области и разработчиков. Участники исследуют процессы через **события предметной области** (Domain Events) — значимые факты, которые уже произошли.

Вместо диаграмм классов участники размещают цветные стикеры на большой доске и выстраивают события в хронологическом порядке. Такой подход помогает:

- восстановить реальный ход процесса, опираясь на знания всей команды, а не на представление одного участника;
- обнаружить пробелы и противоречия: разные названия или трактовки одного события становятся заметны сразу;
- определить возможные границы контекстов по изменениям в терминологии, правилах и ответственности команд.

2.2. Базовые элементы и распространённая цветовая схема

Стикер	Цвет	Формулировка	Пример
Domain Event	оранжевый	свершившийся факт, обычно в прошедшем времени	`PaymentReceived`, `OrderPlaced`

Стикер	Цвет	Формулировка	Пример
Command	синий	намерение или действие, способное привести к событию	`ProcessPayment`, `PlaceOrder`
Policy (реактивное правило)	сиреневый	«Когда происходит событие, выполнить команду»	«Когда `PaymentReceived` → `ShipOrder`»
Actor	жёлтый	участник, который инициирует команду	«Клиент», «Оператор склада»
Read Model	зелёный	информация, на основе которой пользователь принимает решение	«Страница корзины»
External System	розовый	внешняя система — источник или получатель данных	«Платёжный шлюз»
Hot Spot	красный	открытый вопрос, спорное место или риск	«Что произойдёт, если товара нет на складе?»

Обычно работа начинается с этапа **Big Picture**: участники свободно фиксируют события, не выстраивая их в строгую структуру. Затем на этапе **Process Modeling** события располагают по времени и дополняют акторами, командами и реактивными правилами. При необходимости команда переходит к **Design Level**, где уточняет границы агрегатов и распределяет между ними команды и события.

2.3. Event Storming в dddesigner

Event Storming открывается как отдельное представление через меню **View** или палитру команд. Подробная структура данных здесь намеренно скрыта: холст сосредоточен на последовательности событий и языке предметной области, а детальное устройство агрегата редактируется в представлении **Aggregate**.

- Стикеры **Command** (синие), **Event** (оранжевые) и **Policy** (сиреневые) создаются на холсте и образуют поток: `command → emits → event`, `event → triggers → policy`, `policy → send → command`. Политика может вместо команды явно опубликовать новое событие через `emit`, но не должна повторно публиковать событие, которое её запустило.
- Команды и события синхронизируются с агрегатами. После смысловой привязки они становятся частью общей модели проекта и появляются в представлении **Aggregate**, поэтому повторно вводить их имена не требуется.
- На пустом холсте отображается подсказка о добавлении команд и событий.
- Кнопка **«Открыть…»** на стикере переводит в тактический редактор, где можно подробно описать структуру агрегата.

Результат Event Storming становится исходным слоем единой модели, а не отдельной схемой, которую после рабочей сессии приходится переносить вручную. Затем эту же модель можно последовательно уточнить в представлении **Aggregate**.

Часть III. Тактическое проектирование: агрегаты

3.1. Сущности и объекты-значения

- **Сущность (Entity)** обладает устойчивой идентичностью. Её свойства могут полностью измениться, но она останется тем же объектом. Например, позиция заказа `OrderLine` может иметь собственный идентификатор.
- **Объект-значение (Value Object, VO)** не имеет собственной идентичности и сравнивается по содержащимся в нём значениям. Обычно такие объекты делают неизменяемыми. Примеры: `Money`, `Address`, `Quantity`. В объекте-значении удобно определять правила, относящиеся к одному понятию: например, «количество должно быть положительным».

3.2. Агрегат и корень агрегата

Агрегат (Aggregate) — группа согласованно изменяемых сущностей и объектов-значений. Его граница определяет, какие бизнес-правила должны проверяться в рамках одной транзакции.

У каждого агрегата есть один **корень агрегата (Aggregate Root)**. Внешний код обращается к агрегату через корень и не изменяет его внутренние объекты напрямую. Другие агрегаты обычно хранят только идентификатор корня, а не прямую объектную ссылку. Это уменьшает связанность модели и позволяет независимо загружать и изменять агрегаты.

Проектируйте агрегаты как можно компактнее, но не нарушайте требуемую согласованность. Излишне крупные агрегаты увеличивают объём загружаемых данных и вероятность конфликтов при параллельных изменениях. Кроме того, они нередко объединяют несколько самостоятельных границ согласованности.

3.3. Команды, события и инварианты

- **Команда (Command)** выражает намерение изменить состояние, например `SubmitOrder`. Если выполнение команды нарушило бы инвариант, агрегат должен её отклонить.
- **Доменное событие (Domain Event)** описывает уже произошедший значимый факт, например `OrderPlaced`. При изменении структуры сохраняемых событий необходимо учитывать их версии и преобразование старых представлений в новые (см. раздел 4.2).
- **Инвариант (Invariant)** — бизнес-правило, которое должно выполняться после завершения любой операции над агрегатом. Например: «итоговая сумма заказа не может быть отрицательной» или «отменённый заказ нельзя отгрузить».

3.4. Политики, репозитории и доменные службы

- **Политика (Policy)** задаёт реакцию на событие: когда происходит определённый факт, система отправляет следующую команду (`send`) или явно публикует новое событие (`emit`). Такая реакция может оставаться внутри агрегата или связывать несколько агрегатов.
- **Репозиторий (Repository)** предоставляет интерфейс для загрузки и сохранения агрегатов по идентификатору, например `FindById` и `Save`.
- **Доменная служба (Domain Service)** реализует операцию предметной области, которую нельзя естественно отнести к конкретной сущности или объекту-значению. Пример — расчёт, использующий данные нескольких доменных объектов.

3.5. Тактическая модель в dddesigner

Вкладку **Aggregate** можно открыть двойным щелчком по агрегату в дереве проекта или через карту контекстов. На холсте отображаются структурные и поведенческие элементы агрегата: **Root, Entity, VO, Enum, Command, Event, Policy, Reference, Repository** и **Domain Service**. Связи показывают состав модели и взаимодействие её элементов: например, `composition`, `emits` и `triggers`.

Элементы создаются следующим образом:

Элемент	**Где создать**
Entity, Value Object, Enum	Панель **Structure** либо команды создания в дереве
Command, Event, Policy	Панель **Behavior**
Reference — ссылка на другой агрегат	Панель **Infra**; целевой агрегат выбирается в диалоговом окне
Repository, Domain Service	Панель **Infra**; для типа агрегата предусматривается один репозиторий
Атрибуты Root, Entity, VO, Command и Event	Раздел **Properties**, команда **«+Атрибут»**; имя и тип вводятся в строке
Инварианты, методы и обратные ссылки	Раздел **Properties** выбранного элемента

Связь `Command → emits → Event` создаётся выбором события из списка. Благодаря этому команда не может ссылаться на отсутствующее в модели событие. Так же выбираются событие, запускающее политику, и её явное действие: `send` для команды либо `emit` для события.

Ниже приведён агрегат `Order` в DSL. Он соответствует модели, созданной на холсте:

```
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?
  }
}
```

Холст и DSL — два представления одной модели. Изменения на холсте отражаются в DSL, а изменения в DSL разбираются, проверяются и переносятся на холст. Сами данные хранятся в PostgreSQL в формате JSON.

Часть IV. Хранение состояния в виде событий

4.1. Основная идея

Event Sourcing — способ хранения состояния, при котором источником данных служит не текущее состояние агрегата, а последовательность приведших к нему событий. Состояние восстанавливается последовательным применением этих событий.

Основные преимущества:

- **История изменений.** Система хранит последовательность значимых изменений, а не только конечное состояние.
- **Воспроизведение состояния.** При наличии полной истории и корректных обработчиков можно восстановить состояние агрегата на определённый момент.
- **Диагностика.** Последовательность событий помогает анализировать поведение системы и причины изменения состояния.
- **Построение моделей чтения.** События можно использовать для обновления нескольких специализированных моделей чтения (см. часть V).

Такой подход усложняет развитие схем событий. Сохранённые события неизменяемы и могут использоваться значительно дольше версии кода, которая их создала. Поэтому старые версии должны оставаться доступными для чтения, а переход между версиями необходимо определять явно.

4.2. Версии событий и преобразование данных

Каждое событие в dddesigner имеет версию: `v1`, `v2` и далее. При изменении структуры события прежняя версия сохраняется, а новая добавляется отдельно. Блок `upcast` описывает преобразование данных из старой версии в новую:

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

```
event OrderPlaced v2 {
  order_id: OrderId
  placed_at: timestamp
  customer_id: CustomerId       // новое поле
}

upcast from v1 to v2 {
  order_id    -> order_id
  placed_at   -> placed_at
  customer_id -> CustomerId.zero()   // значение по умолчанию
}
```

Диагностика `D006` обнаруживает пропуски в последовательности версий, например наличие `v1` и `v3` без `v2`. Такая ошибка блокирует генерацию кода и отправку изменений в Git, пока последовательность версий не будет исправлена. Для каждой версии создаётся отдельный тип, например `OrderPlacedV1` и `OrderPlacedV2`, а преобразование между версиями оформляется отдельной функцией.

4.3. Настройка Event Sourcing

Режим задаётся в свойствах проекта либо непосредственно в DSL:

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

Сейчас `event_sourcing: true` фиксирует архитектурное решение в модели. Чтобы генерировать адаптеры Chronacta, JSON-схемы и манифесты проекций, включите `codegen.eventstore: chronacta` в codegen stack (по умолчанию `none` — codegen без изменений). См. CHRONACTA.md.

4.4. Runtime Chronacta (opt-in)

При `codegen.eventstore: chronacta` DDDesigner генерирует:

- `.dddesigner/chronacta-manifest.json` — контракт агрегатов, streams, schemas и `projection_graph`
- JSON Schema **событий** (`deploy/chronacta/schemas/`) с расширениями `x-dddesigner-*`
- JSON Schema **команд** (`deploy/chronacta/commands/`) — контракт приложения, **не** регистрируется в Schema Registry Chronacta
- адаптеры агрегатов: `Load` → fold → validate → `append(expected_version)` (+ idempotency / runtime metadata)
- server-side projection manifests для **простых** read models (CEL/builtin)
- scaffold durable subscription workers для **сложных** read models
- `schema.yaml` и connector для Postgres, если у проекции задано `target: postgres` (колонки и `source:` берутся из **handler updates** `field = event.x`, а не только из списка атрибутов)
- dry-run `apply.sh` и baseline `compatibility-report.json`

Отдельный флаг `deployment.chronacta.enabled` добавляет только локальный Compose profile (`deploy/chronacta/compose.yaml`). Он **не** включает контракты и **не** регистрирует схемы автоматически.

CLI без удалённых вызовов:

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

Блокирующие диагностики (артефакты Chronacta не генерируются): неразрешённые события команд/проекций, конфликтующие stream patterns, некорректные версии событий, невозможные postgres-маппинги. Подробности: CHRONACTA.md.

Без `eventstore: chronacta` scaffold проекций и SQL-миграции генерируются как раньше. Canvas **Event Flows** — design view, не runtime Chronacta.

4.5. Неизменяемость: исправления, компенсация и retention

После **успешной записи** доменное событие не редактируется и не удаляется через обычный API приложения — и с Chronacta, и со своим event store. Исправления и отмены моделируются как **новые факты**.

Намерение	Модель в dddesigner	Эффект в хранилище
Исправить данные	Команда `CorrectOrderAddress` → событие `OrderAddressCorrected`	Новый append в stream агрегата
Бизнес-отмена	Команда / компенсация саги → `OrderRefunded`	Новый append; сага координирует шаги

Намерение	Модель в dddesigner	Эффект в хранилище
Эволюция схемы	Новая версия события + `upcast` (§4.2)	Старые записи сохраняются; читатели делают upcast
Очистка read model	Projection handler `delete`	Строка SQL удалена; **журнал событий не трогаем**
Физическая очистка	Операторская процедура (scavenge Chronacta или свой retention)	Только ops; потребители должны пережить или rebuild

С Chronacta и без

Одна и та же модель. Chronacta принудительно append-only на сервере; без Chronacta — INSERT-only в своём `EventStore` (без UPDATE/DELETE событий из кода приложения).

Автоматические пары коррекции

При `event_sourcing: true`:

- при добавлении доменного события (кроме `*Corrected`, `*Refunded` и т.п.) автоматически создаются `Correct{Event}` и `{Event}Corrected` — поля копируются из исходного события, добавляется необязательное `reason`;
- для существующего события — **Properties → Add correction pair**, если пара ещё не создана;
- диагностика **W018**, если у события нет пары коррекции.

При необходимости переименуйте сгенерированные имена под язык домена (`CorrectOrderAddress` / `OrderAddressCorrected` вместо `CorrectOrderPlaced` / `OrderPlacedCorrected`).

Компенсации и возвраты **не** генерируются для каждого события — моделируйте явно (`OrderRefunded`, компенсация саги).

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

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

command CorrectOrderAddress {
  emits [ OrderAddressCorrected ]
}
```

Часть V. CQRS и модели чтения

5.1. Основная идея CQRS

Command Query Responsibility Segregation (CQRS) разделяет модели, отвечающие за изменение данных, и модели, предназначенные для их чтения.

Модель записи состоит из агрегатов и обеспечивает соблюдение инвариантов. Модель чтения формируется под конкретные запросы, экраны или ответы API. Структура, удобная для проверки бизнес-правил, обычно неудобна для сложных списков, фильтрации и агрегации. В свою очередь, денормализованная модель чтения не должна использоваться как основа для принятия решений об изменении состояния.

CQRS хорошо сочетается с Event Sourcing: сохранённые события могут одновременно обновлять специализированные модели чтения. Однако CQRS не требует Event Sourcing и может применяться независимо от него.

Если проекции обновляются асинхронно, модель чтения может некоторое время отставать от модели записи. Такая **согласованность с задержкой** должна учитываться в пользовательских сценариях и обработке ошибок.

5.2. Проекции в dddesigner

На вкладке **Projections** показана цепочка обработки:

FROM EVENT → HANDLER → READ MODEL

- Кнопка **Handler** добавляет обработчик и синхронизирует список `from_events`, поэтому перечень исходных событий не нужно обновлять отдельно.
- Значения `from_events` и `trigger_event` выбираются из каталога доменных событий модели.
- **Атрибуты** проекции описывают только форму модели чтения (имя и тип поля). Они **не** задают источник данных.
- **Маппинг** «поле проекции ← поле события» задаётся в **handler**: updates вида `order_id = event.order_id`, `status = event.status`. Именно эти выражения генератор использует при codegen (в том числе для Chronacta Postgres `schema.yaml` / `source:`).
- Обработчик может также удалять запись (`delete`) — для обычных SQL-scaffold проекций; для `target: postgres` `delete` и литералы не поддерживаются.

В Properties проекции поле **Хранилище модели чтения** выбирает default (JSON Chronacta / worker) или **Postgres (коннектор Chronacta)**. Нужен `codegen.eventstore: chronacta`.

События коррекции (`*Corrected`) и компенсации (`*Refunded`) обрабатываются так же — через append в журнал и новый handler; исходные события не изменяются (§4.5).

При `codegen.eventstore: chronacta` простые обработчики (`field = event.field`) становятся server-side манифестами Chronacta; сложные — durable subscription workers с вызовом `{Name}Apply`. Укажите `target: postgres`, чтобы те же поля и handlers попали в `schema.yaml` и connector Chronacta для Postgres (DDL через sqlgen; scaffold `migrations/projections` не создаётся). См. §4.4 и 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
  }
}
```

Из определения `projection` генератор создаёт структуру модели чтения, обработчик применения событий и SQL-миграцию для таблицы проекции. Структура модели управляется генератором, а обработчик и SQL-миграция служат заготовками: они создаются только при отсутствии соответствующих файлов. Благодаря этому внесённые вручную изменения сохраняются.

Часть VI. Многошаговые процессы: Saga и Process Manager

6.1. Назначение саг

Агрегат задаёт границу строгой транзакционной согласованности. Изменения нескольких агрегатов, особенно принадлежащих разным ограниченным контекстам, обычно координируются не одной общей транзакцией, а последовательностью команд и событий.

Например, оформление заказа может включать резервирование товара, проведение платежа и подтверждение заказа. Для координации такого процесса используют **сагу** или **менеджер процесса (Process Manager)**. Координатор получает события, отправляет команды участникам процесса и отслеживает его текущее состояние. Между отдельными шагами действует согласованность с задержкой.

Если выполненный шаг необходимо отменить с точки зрения бизнеса, процесс отправляет **компенсирующую команду**. Например, после неудачного платежа можно выполнить `ReleaseStock`. Компенсация не отменяет уже завершённую транзакцию буквально, а создаёт новое действие, которое компенсирует её последствия для бизнеса.

6.2. Оркестрация и хореография

- **Оркестрация** предполагает наличие выделенного координатора. Он отправляет команды, получает события о результатах и определяет следующий шаг процесса.
- **Хореография** не имеет единого координатора. Каждый участник реагирует на события других участников, а последовательность процесса складывается из этих реакций.

Оркестрация упрощает отслеживание состояния длительного процесса, но сосредоточивает его логику в одном компоненте. Хореография устраняет единый центр управления, однако с ростом числа участников затрудняет анализ общего потока и обработку сбоев.

6.3. Саги в dddesigner

На холсте **Saga** используются четыре вида узлов:

Узел	Обозначение	Назначение
START	фиолетовый	Точка входа; `starts_on` указывает событие, запускающее процесс
STEP	синий	Основной шаг: отправка команды `send`, ожидание события `wait_for` или публикация события `emit`
ON FAIL	красно-коричневый	Компенсация на уровне всей саги; допускается не более одной
STEP FAIL	оранжевый	Компенсация отдельного шага; допускается не более одной на шаг

```
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)
  }
}
```

Кнопка **«Откат»** создаёт компенсацию соответствующего уровня в зависимости от выбранного элемента. Если выбрана точка **START** или вся сага, создаётся узел **ON FAIL**. Если выбран конкретный шаг, создаётся узел **STEP FAIL**. Это помогает не смешивать компенсацию всего процесса с обработкой сбоя отдельного шага.

Часть VII. Бизнес-действия и проверка поведения системы

7.1. От бизнес-намерения к модели

Команды, события, политики, саги и проекции описывают части поведения системы. Но сами по себе они не отвечают на вопрос: **какое бизнес-действие выполняет пользователь и какой результат система должна дать?**

Бизнес-действие (Business Action / Use Case) — именованный сценарий, например `Создать пользователя` или `Оформить заказ`. Его создают после появления начальной команды: задают имя и однозначно разрешённую команду, затем уточняют актора, цель и ожидаемый бизнес-результат. Бизнес-действие превращается в проверяемый контракт:

```
Бизнес-действие
        ↓
Команда
        ↓
События, политики, саги и проекции
```

```
        ↓
Ожидаемые и запрещённые результаты
```

Бизнес-действие создаётся только после начальной команды: она должна однозначно разрешаться в существующую команду. Начальная точка всегда имеет вид `start command`; доменные и интеграционные события, а также маршруты Gateway не являются запускаемыми Use Case. HTTP настраивается отдельно как необязательная экспозиция command-backed сценария. Утверждения выбираются из реальных элементов модели: событий, политик, саг, проекций и правил обновления полей проекций.

7.2. DSL бизнес-действий

Бизнес-действия хранятся в виртуальном DSL-файле `use_cases.ddd`. Например:

```
use_case CreateUser {
  description: "Регистрация нового пользователя"
  actor: "Посетитель"
  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` означает, что объявленный результат должен быть достижим из начальной точки. `forbidden` означает, что результат не должен появиться в трассировке. Ссылки должны указывать на существующие элементы модели; произвольные строки не заменяют объектную связь.

Блок `input` объявляет типизированные параметры прикладного сценария. Для `start command` поля должны совпадать с входом целевой команды по имени, типу, порядку и признаку обязательности; несовпадение отмечается диагностикой `D053`. Для обратной совместимости блок можно не указывать: тогда генератор выводит вход из целевой команды.

Бизнес-действие описывает намерение и критерии проверки, а не копирует последовательность команд и событий. Если доменный поток изменился, сценарий продолжает ссылаться на те же объекты модели и показывает, какие ожидания нужно пересмотреть.

7.3. Структурная симуляция

Структурная симуляция позволяет проверить команду или бизнес-действие и увидеть объявленную цепочку:

```
Команда принята
  → событие создано
  → политика сработала
  → следующая команда или событие
  → найден обработчик проекции
  → заявлено обновление проекции
```

В результате отображаются подробная трассировка, созданные доменные события, обновлённые проекции, запущенные саги, ожидаемые события и доступные компенсации. Для бизнес-действия дополнительно показывается вердикт `Пройден` или `Требует доработки` и результат каждого утверждения.

Симуляция является **структурной**, а не интеграционной проверкой работающего приложения. Она проверяет достижимость объявленных связей модели, но не выполняет:

- бизнес-код агрегатов и обработчиков;
- условия политик и инварианты;
- выражения обновления проекций;
- запросы к базе данных и состояние модели чтения;
- реальные очереди, брокеры и внешние системы.

Поэтому пройденная симуляция означает, что модель описывает ожидаемый путь согласованно. Она не заменяет модульные, интеграционные и сквозные тесты приложения.

7.5. Холст бизнес-действия

При выборе бизнес-действия открывается read-only холст, построенный на той же структурной симуляции, что и проверка сценария. Граф начинается с объявленной команды и проходит по связям модели: `emits` команды, триггеры и действия Policy, шаги Saga и обработчики Projection. Событие не прикрепляется к сценарию только по совпадению имени.

Холст показывает все достижимые ветки. Утверждения `expected` и `forbidden` остаются ручной бизнес-разметкой поверх вычисленного графа. Перемещение узлов изменяет только сохранённую раскладку конкретного бизнес-действия; причинные связи на этом холсте не рисуются повторно. Условия и инварианты отображаются как неисполненные, поскольку структурная симуляция не выполняет runtime-код.

Холст отличается от Event Flows и Saga Designer: те представления редактируют архитектуру, а Business Action Canvas объясняет один выбранный сценарий в текущей модели.

7.4. Сгенерированные бизнес-действия

Каждый результат генерации содержит один управляемый файл `GENERATED.md`. В нём указаны версия и язык генерации, каждый owned- и scaffold-файл с понятным описанием его назначения, callable и типизированные входные параметры каждого Business Action, а также настроенные HTTP-метод, путь, режим авторизации и путь адаптера.

Часть VIII. Интерфейс dddesigner: порядок работы

8.1. Основные панели

- **Проект** — дерево для навигации и создания контекстов, агрегатов, саг и проекций. При выборе проекта панель **Properties** показывает настройки генерации кода и подключения Git.
- **Холст** отображает структуру и связи текущего представления. Каждый вид отвечает за свой уровень модели: например, команда редактируется в **Aggregate**, а **Event Flows** служит для обзора потоков.
- **Properties** — контекстная панель для имён, ссылок, атрибутов, инвариантов, предупреждений и этапов работы. Связи выбираются из списков, а не вводятся вручную.
- **Редактор DSL** — текстовое представление той же модели. По умолчанию он отображается узкой панелью внизу центральной колонки и может показывать стратегическую модель или текущий агрегат.
- **Model** — сворачиваемый каталог объектов и связей с быстрым переходом к нужному элементу на холсте.
- **Generated Code / Reviews** — просмотр результата генерации и обсуждение модели. Сама модель здесь не редактируется.
- **Строка состояния** — состояние соединения и синхронизации, конфигурация генерации, предупреждения, версия модели, а также команды отмены и повтора.
- **Панель инструментов** — создание и связывание объектов текущего холста.

8.2. Основной сценарий

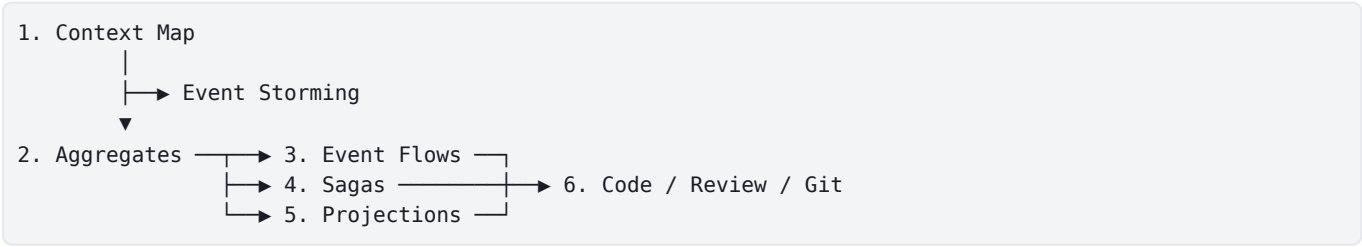

Схема показывает рекомендуемый порядок работы. Когда на холсте ничего не выбрано, в панели **Properties** отображается шестишаговый список от **Context Map** до **Code / Review / Git**; Event Storming открывается отдельно через меню **View** или палитру команд. К нему удобно обращаться до детализации агрегатов и при исследовании новых процессов.

Топология развёртывания подключается только по желанию и не меняет DDD-модель; по умолчанию её нет, поэтому существующие проекты сохраняют monolith output. В дополнительном представлении **Service Map** выберите topology, единый внутренний transport, Compose project и service boundaries; оно остаётся вне обязательного шестишагового процесса. Без явных boundaries отображается выводимое правило «один service на context», а также межсервисные связи контекстов, codegen diagnostics и deployment-only preview. Изменения сохраняются через каноническую Design Model, а preview-файлы становятся сохранёнными артефактами только после **Generate**.

8.3. Проверка качества модели

dddesigner выявляет структурные и семантические проблемы через `GET /api/v1/projects/:id/diagnostics`. Диагностика отображается в панели **Properties**; выбор сообщения переводит к соответствующему объекту.

Код	Уровень	Что проверяется
D001	Ошибка	повторяющееся имя агрегата
D002–D003	Ошибка	у агрегата нет корня или у корня нет поля `id`
D004	Ошибка	команда публикует необъявленное событие
D005	Ошибка	ссылка указывает на отсутствующий агрегат
D006	Ошибка	в последовательности версий события есть пропуск
D007	Ошибка	действие `emit` политики ссылается на неразрешимое событие
D008	Ошибка	действие `send` политики ссылается на неразрешимую команду
D009	Ошибка	у действия политики неизвестный вид, нет обязательной цели или заданы обе цели
D010	Ошибка	имя команды повторяется внутри агрегата
D011	Ошибка	имя Policy повторяется внутри агрегата
D012	Ошибка	имя саги повторяется в модели
D013	Ошибка	имя проекции повторяется в модели
D014	Ошибка	пара «имя и версия доменного события» повторяется внутри агрегата
D050–D052	Ошибка	у бизнес-действия некорректное имя, начало или утверждение
D053	Ошибка	вход бизнес-действия не совпадает с начальной командой
W001	Предупреждение	агрегат содержит более семи сущностей
W002	Предупреждение	у команды нет инвариантов
W003	Предупреждение	у интеграционного события нет описания
W004	Предупреждение	межконтекстная ссылка не защищена ACL
W005	Предупреждение	у события не более одного поля
W006	Предупреждение	команда не публикует событий
W007	Предупреждение	у Policy нет запускающего события
W008	Предупреждение	запускающее событие Policy не найдено в агрегате
W009–W011	Предупреждение	проекция не связана с существующими событиями
W012	Предупреждение	у саги не указано запускающее событие
W013	Предупреждение	у Policy нет действий
W014	Предупреждение	Policy публикует то же событие, которое её запускает
W015	Предупреждение	действие Policy с тем же видом и целью повторяется
W016	Предупреждение	запускающее событие саги не найдено в модели
W017	Предупреждение	у саги нет шагов
W018	Предупреждение	у события нет пары коррекции при включённом event_sourcing; Add correction pair или add_event_correction

Панель **View → Model Health** дополняет эти проверки анализом крупных агрегатов, изолированных контекстов и циклов на карте контекстов. Из отчёта можно перейти непосредственно к проблемному объекту.

8.4. DDD Guide — встроенный помощник

Панель **View → DDD Guide** сочетает два источника рекомендаций:

1. **Правила модели** находят недостающие элементы и предлагают готовое действие, которое можно применить одним нажатием.
2. **Ответы на вопросы** сначала используют встроенные правила и справочные материалы. Если их недостаточно и языковая модель доступна, она объясняет понятие или рекомендацию, не добавляя в проект несуществующие объекты.

Помощник замечает, например, слишком крупный агрегат, команду без ожидаемого результата или незавершённую сагу. Он помогает проверить решение, но не заменяет архитектурных решений команды.

8.5. Генерация кода и Git

Панель **Generated Code** поддерживает Go, C#, Java, Kotlin, TypeScript/Node и Python. Язык, HTTP-фреймворк, адаптер базы данных, брокер сообщений и механизм внедрения зависимостей задаются в свойствах проекта. Перед использованием в производственной системе обращайте внимание на указанную в интерфейсе степень зрелости выбранного стека.

Опционально — топология развёртывания

Проект может подключить `monolith`, `modular_monolith` или `microservices`; для внутреннего транспорта проекта выбирается `https` либо `grpc`. В дополнительном Service Map явные service boundaries назначают каждый Bounded Context ровно одной boundary; без них codegen выводит по одному service на context. Preview в этом представлении выведен codegen и не обещает сохранённых артефактов до **Generate**. Для valid non-monolith deployment генератор создаёт owned-контракты OpenAPI/proto, deployment README, metadata topology/preview/dependencies, а также scaffold Dockerfiles и root Compose file. Полные language-specific server/client stubs, Kubernetes, Helm, service mesh и mTLS не предоставляются; не каждое поле deployment schema полностью меняет renderer. Metadata зависимостей пакетов нужно перенести в пользовательский project descriptor.

Сгенерированные файлы делятся на две группы:

- **Управляемые генератором** (`owned`, значок ↻, `DO NOT EDIT`) — доменная модель, контракты команд и Use Case, порты сервисов, определения саг, модели чтения и HTTP-маршруты. При повторной генерации они заменяются.
- **Заготовки** (`scaffold`, значок ✎) — обработчики команд, исполнители саг, обработчики проекций, миграции и адаптеры репозиториев. Они создаются только при отсутствии файла, поэтому ручные изменения сохраняются.

Генерация persistence теперь доступна для всех целевых языков на уровне repository adapter. Go сохраняет существующие адаптеры (`pgx`, `sqlc`, `ent`, `gorm`). Для C# генерируются EF Core bridge-заготовки (`efcore`), для Java — JPA/Spring Data bridge-заготовки (`jpa`), для Kotlin — Exposed bridge-заготовки (`exposed`), для TypeScript — Prisma bridge-заготовки (`prisma`), для Python — SQLAlchemy bridge-заготовки (`sqlalchemy`). Эти файлы намеренно не создают полную схему БД и не заставляют generated domain objects становиться ORM-сущностями. Вместо этого они задают небольшой store boundary, который команда подключает к `DbContext`, `EntityManager`, транзакциям Exposed, Prisma Client или SQLAlchemy Session и связывает с доменной моделью через собственный mapping.

Генератор также записывает подсказки по ORM-зависимостям в `.dddesigner/generated-dependencies.json`. Этот JSON стоит воспринимать как metadata для сборки: перечисленные пакеты нужно перенести в пользовательский project descriptor, а в scaffold store добавить конкретную runtime-конфигурацию, транзакции, миграции и индексы приложения.

Результат можно скачать как ZIP-архив или отправить в подключённый Git-репозиторий вместе с DSL-файлами из `design/`.

8.6. Дополнительные инструменты

- **Tools → Model releases** — именованные снимки модели и сравнение объектов, добавленных, удалённых или переименованных между релизами.
- **Tools → Export** — PNG и SVG для презентаций, Markdown, Mermaid, PlantUML и C4/Structurizr для документации в репозитории.
- **Палитра команд** (`Ctrl+K` или `Ctrl+P`) — поиск агрегатов, событий, саг, проекций, шлюзов и DSL-файлов, а также быстрый запуск проверок, релизов и экспорта.
- **Инструменты → Участники** — приглашение по электронной почте с ролями архитектора, разработчика и наблюдателя; роль наблюдателя разрешает только просмотр. В той же модалке создаются **ключи агента** для работы с внешним искусственным интеллектом (см. § 8.7). Активные участники отображаются в строке заголовка.
- **Отмена и повтор** — `Ctrl+Z` и `Ctrl+Shift+Z` восстанавливают предыдущие состояния модели. История хранится на сервере и остаётся доступной после перезагрузки страницы.

8.7. Проектирование вместе с внешним искусственным интеллектом

DDDesigner позволяет вести диалог о модели не только внутри веб-интерфейса, но и во внешних средах, где вы уже привыкли работать с помощником: в редакторе кода, в отдельном чате или в автоматических сценариях. Помощник не подменяет редактор: он предлагает изменения, а DDDesigner проверяет их по правилам модели и сохраняет только после явного подтверждения через безопасный цикл «сначала проверить, потом записать».

Зачем это нужно

Предметную модель удобно уточнять в разговоре: «добавь контекст заказов», «свяжи команду с событием», «проверь, нет ли дыр в саге». Внешний помощник хорошо формулирует такие шаги, но не должен бесконтрольно менять чужие проекты. Поэтому доступ строится на **личном ключе агента**,

привязанном к **одному проекту** и к **вашей учётной записи**. Права помощника не шире ваших прав в этом проекте.

Как получить ключ

1. Откройте нужный проект (диаграмму) в рабочей области.
2. Выберите **Инструменты → Участники**.
3. Пролистайте модалку до раздела **«Ключи агента»**.
4. Укажите понятное имя (например, «рабочий ноутбук») и уровень доступа: **только чтение** или **чтение и запись**.
5. Нажмите **«Создать ключ агента»**. Полная секретная строка показывается **один раз**. Сразу скопируйте её в надёжное место.
6. Закройте окно. Повторно увидеть секрет нельзя — только отозвать ключ и создать новый.

Создавать ключ может участник с ролью не ниже разработчика. Наблюдатель этот раздел не видит. Архитектор может отзывать чужие ключи проекта, если нужно срочно закрыть доступ.

Ключ действует от вашего имени: система применяет те же правила изоляции между организациями и проектами, что и для обычного входа. Запросы к другому проекту с этим ключом отклоняются.

Куда вставить ключ

Скопированную строку указывают **вне** DDDesigner — в настройках среды, где работает помощник:

- в конфигурации облачного MCP для Cursor, Claude, Codex или Gemini (`https://app.dddesigner.com/mcp` и заголовок `X-Agent-Key`);
- в настройках действий Custom GPT у ChatGPT (секрет заголовка авторизации);
- в окружении командной строки утилит DDDesigner (адрес, ключ и идентификатор проекта).

В разделе **Участники → Ключи агента** показаны URL MCP и шаблон конфигурации для копирования (в шаблоне — плейсхолдер секрета). Для клиентов MCP идентификатор проекта **не нужен**: он уже зашит в ключ.

Не сохраняйте ключ в репозитории кода и не пересылайте его в открытых чатах. При подозрении на утечку откройте **Участники → Ключи агента → Отозвать** и создайте новый.

Как должен идти диалог с помощником

Рекомендуемый порядок один и тот же во всех внешних инструментах:

1. **Снимок модели** — помощник запрашивает краткое описание текущего состояния и номер версии (`whoami` / `get_summary`).
2. **Каталог операций** — при планировании правок (`ops_catalog`); не выдумывать имена операций.
3. **Предложение** — набор точечных операций или правка текстового описания модели.
4. **Предварительный просмотр** — изменения прогоняются без сохранения (`preview_ops`); система возвращает ошибки и предупреждения.
5. **Применение** — только если просмотр прошёл успешно и вы явно согласились (`apply_ops`).
6. **Диагностика** — повторная проверка качества модели (`get_diagnostics`).

Так источник истины остаётся в DDDesigner: холст и текстовое описание синхронизируются, история и совместная работа продолжают действовать как при ручном редактировании.

Чем отличаются точечные операции и текстовое описание

- **Точечные операции** удобны для малых шагов: добавить команду, связать её с событием, переименовать поле.
- **Текстовое описание модели** (язык проектирования DDDesigner) удобнее для крупных фрагментов: целый агрегат, сага, проекция (`list_dsl_files` / `get_dsl` / `put_dsl`).

Кодогенерация: UI и помощник

Каркас кода можно получить двумя способами — пайплайн один и тот же:

1. В рабочей области: панель **Generated Code** → кнопка **Generate** (просмотр, ZIP, запись в Git).
2. Через облачный MCP: `codegen_preview` → `list_codegen_files` → `get_codegen_file` для выбранных путей.

Перед генерацией модель должна быть без ошибок. Через MCP не выгружайте всё дерево сразу — берите нужные файлы по одному. Запись в Git и ZIP из MCP в текущей версии недоступны; для этого используйте кнопки в UI.

Встроенный помощник и внешний

Внутри приложения уже есть **DDD Guide**: он подсказывает пробелы в модели по правилам и отвечает на вопросы. Внешний искусственный интеллект подключается **дополнительно**, когда нужен свободный диалог в привычном редакторе. Оба пути сходятся к одной модели; внешний путь требует ключа агента.

Подробная инструкция (Cursor, Claude, ChatGPT, Codex, Gemini, CLI) и полный список инструментов MCP: Работа с искусственным интеллектом. Краткий справочник для самого помощника: HANDBOOK.md.

Часть IX. Пример: проектируем интернет-магазин от намерения до проверяемого контракта

Рекомендуемый процесс образует петлю: бизнес-намерение задаёт направление в начале, а в конце возвращается как проверяемый контракт на построенную модель.

Шаг 1. Бизнес-намерение PlaceOrder

В верхней папке **Бизнес-действия** создаём черновик `PlaceOrder`, указываем актора `Customer` и цель: «оформить собранный заказ и получить подтверждение». Бизнес-действие создаём после появления команды `PlaceOrder`, сразу выбираем её как начальную и затем добавляем assertions для существующих элементов модели.

Шаг 2. Context Map

Создаём контексты `ordering`, `catalog`, `payment`, `inventory`, `shipping` и `legacy_billing`. На **Context Map** определяем связи: `ordering` ⇄ `catalog` как **customer_supplier**, `catalog` как **OHS**, а доступ к `legacy_billing` защищаем **ACL**. Границы выводим из ответственности, необходимой для `PlaceOrder` и соседних бизнес-действий.

Шаг 3. Event Storming для PlaceOrder

На вкладке **Event Storming** раскрываем выбранное намерение в хронологическую историю:

`PlaceOrder` (команда) → `OrderPlaced` (событие) → резервирование → оплата → подтверждение либо компенсация.

Event Storming здесь не является отдельным конечным артефактом: он помогает обнаружить элементы и связи, нужные для выполнения бизнес-действия.

Шаг 4. Тактическая модель Order

В представлении **Aggregate** определяем корень `Order`, сущность `OrderLine`, объекты-значения `OrderId` и `Quantity`, `OrderStatus`, команды и версионированные события. Для `PlaceOrder` фиксируем инварианты: заказ содержит строки и находится в допустимом состоянии.

Шаг 5. Процесс и модель чтения

Создаём `PlaceOrderFlow`, запускаемый `ordering.OrderPlaced`: `inventory.ReserveStock` → `payment.ChargeCustomer` → `ordering.ConfirmOrder`. Добавляем компенсации `ReleaseStock` и `RefundCustomer`. В **Projections** строим `OrderSummary` для экрана «Мои заказы».

Шаг 6. Завершаем Business Action

Возвращаемся к тому же `PlaceOrder`: выбираем старт `ordering.Order.PlaceOrder`, затем ожидаемые результаты `OrderPlaced`, `PlaceOrderFlow`, `OrderSummary` и обновления её полей. `OrderCancelled` отмечаем как запрещённый результат успешного сценария.

Запускаем структурную симуляцию. Статус **«Пройден»** означает, что ожидаемые объекты достижимы по объявленным связям. Статус **«Требует доработки»** показывает, какая часть контракта не подтверждается моделью.

Шаг 7. Потоки, генерация и runtime-тесты

В **Event Flows** проверяем сквозную цепочку и источники событий. В **Generated Code** выбираем технологический профиль и генерируем доменную структуру, контракты, application wrapper для `PlaceOrder`, опциональный HTTP-адаптер и `GENERATED.md`. Бизнес-правила, интеграции и runtime-тесты остаются ответственностью разработчиков.

Заключение: проверка качества дизайна

- [ ] У каждого ограниченного контекста есть собственный словарь, а одинаковые термины не смешиваются между контекстами.
- [ ] Для каждой связи на карте контекстов указан её тип, а не только направление.
- [ ] Event Storming проведён до тактического проектирования; ключевые события и их названия согласованы с экспертами предметной области.
- [ ] Агрегаты остаются небольшими, а другие агрегаты обычно упоминаются по идентификатору.

- [ ] Для каждой команды заданы инварианты либо явно зафиксировано, почему они не нужны.
- [ ] Доменные события версионируются без пропусков; преобразование старой схемы в новую описано через `upcast`.
- [ ] Модели чтения спроектированы под конкретный экран или ответ API, а не копируют агрегат.
- [ ] Длительные процессы вынесены в саги; на случай сбоев предусмотрены компенсации.
- [ ] Ключевые бизнес-намерения зафиксированы как command-backed Business Actions после появления начальных команд.
- [ ] После моделирования Business Actions связаны со стартовыми объектами, ожидаемыми и запрещёнными результатами; структурная симуляция пройдена.
- [ ] Предупреждения в панелях **Model Health** и **Properties** устранены либо осознанно приняты.
- [ ] До генерации кода выбраны язык, HTTP-стек, база данных и брокер сообщений.
- [ ] Если deployment включён, проверены topology, transport, boundaries, contracts и scaffolds.
- [ ] Если используется внешний помощник, создан личный ключ агента, секрет сохранён вне репозитория, изменения идут через `preview_ops` перед записью, а сгенерированный код при необходимости читается через `codegen_*` или панель **Generated Code**.

Глоссарий

Термин	Значение
Ubiquitous Language	единый язык экспертов и разработчиков внутри контекста
Bounded Context	граница, внутри которой модель и её термины имеют однозначный смысл
Context Map	карта отношений между ограниченными контекстами
Aggregate / Aggregate Root	граница транзакционной согласованности и её единственная внешняя точка входа
Entity	объект с устойчивой идентичностью
Value Object	обычно неизменяемый объект без собственной идентичности, сравниваемый по значению
Domain Event	уже произошедший факт предметной области
Command	запрос изменить состояние, который может быть отклонён
Policy	правило, реагирующее на событие и инициирующее следующий шаг процесса
Repository	интерфейс сохранения и загрузки агрегата как единого целого
Domain Service	доменная операция, которую нельзя естественно отнести к одной сущности или объекту-значению
Invariant	бизнес-условие, которое агрегат обязан сохранять после каждой операции
Event Storming	совместное исследование предметной области через последовательность событий
Event Sourcing	хранение состояния в виде последовательности событий
Upcasting	преобразование события старой версии в актуальную схему
CQRS	разделение моделей записи и чтения
Projection / Read Model	модель чтения, построенная обработчиками событий под конкретный запрос
Saga / Process Manager	модель многошагового процесса, затрагивающего несколько агрегатов
Orchestration / Choreography	процесс с центральным координатором либо с распределённой цепочкой реакций на события
Compensation (ON FAIL / STEP FAIL)	действие, компенсирующее всю сагу либо отдельный шаг
ACL (Anti-Corruption Layer)	слой преобразования между внешней и внутренней моделями
OHS (Open Host Service)	стабильный опубликованный интерфейс контекста
Ключ агента	личный секрет для внешнего помощника; привязан к одному проекту и к вашей учётной записи; для MCP: `https://app.dddesigner.com/mcp` + `X-Agent-Key`
Предварительный просмотр изменений	прогон правок модели без сохранения
Применение изменений	запись проверенных правок в модель

Краткая справка по каждому виду холста доступна в приложении по кнопке «?» на панели инструментов. Подробности реализации приведены в документах о процессе дизайна, принципах и языке проектирования. Работа с внешним искусственным интеллектом описана в § 8.7 и в документе «Работа с искусственным интеллектом».

8.4. Статус генерации NATS и Kafka

NATS и Kafka — сгенерированные адаптеры/каркасы, а не runtime-интеграции dddesigner. Сгенерированные порты являются owned-контрактами, а broker-specific файлы — scaffold-адаптерами. Подключите в composition root собственный минимальный интерфейс клиента или callback к официальной библиотеке и передайте его в адаптер. Асинхронная поверхность: `Task` в C#, `CompletionStage` / `CompletableFuture` в Java, `suspend` в Kotlin, `Promise` в TypeScript и `async` / `await` в Python. Имена топиков остаются константами, payload — байтами.

Outbox transport отделён от event store и Event Sourcing. Outbox хранит сообщения для публикации после транзакции, а Event Sourcing хранит authoritative-последовательность доменных событий и восстанавливает агрегаты. Сгенерированный broker-контракт не создаёт consumer groups, offsets, schema registry, replay и управление жизненным циклом соединения.

Используйте `.dddesigner/generated-dependencies.json` только как metadata для сборки. Официальные библиотеки и их адаптацию добавляйте в пользовательском composition code: `NATS.Client.Core` или `NATS.Client` и `Confluent.Kafka` для C#, `io.nats/jnats` и `org.apache.kafka/kafka-clients` для Java/Kotlin, `nats` и `kafkajs` для TypeScript, `nats-py` и `aiokafka` для Python.