Domain-Driven Design

Программная инженерия
Лекция 3

Михаил Пожидаев

Институт прикладной математики и компьютерных наук ТГУ

9 ноября 2026 г.

Что такое Domain-Driven Design

Domain-Driven Design, или DDD, — это подход к разработке программного обеспечения, в котором центральное место занимает модель предметной области. Предметная область — это часть реального мира, процессы и правила которой автоматизирует система. DDD помогает команде работать со сложностью не только на уровне кода, но и на уровне смысла, терминов, процессов и границ ответственности. Подход был систематизирован Эриком Эвансом в книге «Domain-Driven Design: Tackling Complexity in the Heart of Software».

Главная идея!

1. Программная модель должна отражать бизнес-модель настолько точно, насколько это необходимо для решения конкретной задачи.
2. Код, архитектура, тесты и разговоры команды должны использовать одни и те же понятия.
3. Сложность предметной области необходимо изолировать, исследовать и выражать явно, а не скрывать в наборе технических процедур.

Когда DDD оправдан

DDD особенно полезен

1. Когда предметная область содержит много изменяющихся правил и исключений.
2. Когда разработчикам необходимо постоянно взаимодействовать с экспертами бизнеса.
3. Когда система состоит из нескольких команд, модулей или интегрируемых подсистем.
4. Когда ошибки в бизнес-логике дорого обходятся компании.
5. Когда необходимо развивать систему годами без потери управляемости.

DDD может быть избыточен

1. Когда приложение является простым CRUD-интерфейсом без сложного поведения.
2. Когда предметная область хорошо стандартизирована и не даёт конкурентного преимущества.
3. Когда стоимость сложного моделирования превышает ценность, которую оно создаёт.
4. Когда команда ещё не обладает достаточным пониманием процессов и должна начать с исследования требований.

- DDD не является обязательным набором классов и паттернов.
- DDD является способом управлять сложностью через язык, модели, границы и явные бизнес-правила.

1. Предметная область отвечает на вопрос: «О чём эта система с точки зрения бизнеса?».
2. Для маркетплейса предметной областью может быть электронная коммерция.
3. Для банка предметной областью может быть управление финансовыми продуктами, платежами, рисками и отношениями с клиентами.
4. Для университета предметной областью может быть организация учебного процесса.

Поддомены

1. Поддомен — это относительно самостоятельная часть предметной области.
2. Поддомены часто выделяются по бизнес-функциям, знаниям экспертов или организационной структуре.
3. Например, в маркетплейсе можно выделить каталог товаров, ценообразование, корзину, оформление заказа, доставку, возвраты и расчёты с продавцами.

Типы поддоменов

1. Core Domain — ключевой поддомен, создающий конкурентное преимущество компании.
2. Supporting Subdomain — вспомогательный поддомен, необходимый для бизнеса, но не уникальный.
3. Generic Subdomain — типовой поддомен, для которого допустимо использовать готовое решение или внешний сервис.

CC BY-NC 4.0 | Институт прикладной математики и компьютерных наук ТГУ

Классический подход

Почему классическое объектное моделирование не всегда справляется

Классический подход

В классическом объектно-ориентированном проектировании разработчик часто начинает с таблиц базы данных, экранов пользовательского интерфейса или набора CRUD-операций. Затем сущности базы данных превращаются в классы, а бизнес-правила распределяются между контроллерами, сервисами и обработчиками событий. Такой подход может быть эффективен для простых информационных систем с небольшим количеством правил.

Проблема сложной области

- В сложной предметной области бизнес-правила взаимосвязаны, изменяются со временем и зависят от контекста;
- одно и то же слово может иметь разные значения для разных подразделений компании;
- простая схема «одна таблица — один класс» перестаёт отражать реальную бизнес-логику;
- модель становится технической проекцией хранения данных, а не моделью поведения бизнеса.

Анемичная доменная модель

Несфокусированное управление данными

Признаки анемичной модели

1. Объекты содержат только поля, свойства и простые методы доступа к данным. 1 1.Поведение размещается в сервисах, менеджерах, контроллерах или больших классах с названием, оканчивающимся на `Service`.
2. Сущности не защищают собственные инварианты.
3. Любая часть приложения может изменить состояние объекта в обход бизнес-правил.

Пример проблемы

- Объект заказа содержит поля `status`, `totalPrice` и `items`.
- Отдельный сервис самостоятельно меняет статус заказа на «оплачен».
- Другой сервис удаляет позиции уже оплаченного заказа.
- Третий сервис меняет цену позиции без пересчёта общей суммы.

Последствие

- Правила оказываются размазаны по коду.
- Разработчикам трудно понять, где находится истинная логика предметной области.
- Изменение бизнес-требования требует поиска множества неявных зависимостей.

CC BY-NC 4.0 | Институт прикладной математики и компьютерных наук ТГУ

Ориентация только на данные

Модель, построенная вокруг таблиц

Реляционная база данных удобна для хранения и выборки данных. Однако структура таблиц не всегда совпадает со структурой бизнес-понятий. Нормализация базы данных может разделить единый бизнес-объект на несколько таблиц. Денормализация может, наоборот, объединить данные, которые в бизнесе относятся к разным процессам.

Ошибочный вопрос

- Вопрос «Какие таблицы нам нужны?» часто задаётся слишком рано.
- Такой вопрос подталкивает команду проектировать хранение до понимания поведения и правил.

Более полезный вопрос

- Следует начинать с вопроса «Какие решения принимает бизнес и какие правила он обязан соблюдать?»
- После этого можно определить, какие объекты, операции и хранилища необходимы для реализации модели.

Проекция понятия на разные компоненты

Смешение разных смыслов одного понятия

Пример понятия «клиент»

- Для отдела маркетинга клиент — это человек, которому можно отправить рекламное предложение.
- Для биллинга клиент — это сторона договора, имеющая платёжные обязательства.
- Для службы поддержки клиент — это автор обращения в поддержку.
- Для логистики клиент — это получатель посылки.

Ошибка универсальной модели

- Попытка создать единую глобальную сущность `Customer` для всех подразделений часто приводит к огромному классу с десятками несвязанных полей.
- Такой класс объединяет разные жизненные циклы, правила и уровни доступа.
- Изменение требований одного подразделения начинает влиять на другие части системы.

Вывод

1. Одинаковое слово не гарантирует одинаковый смысл.
2. Модель должна учитывать контекст, в котором используется понятие.

Подмена архитектурного восприятия

Техническая архитектура вместо бизнес-архитектуры

Систему делят на контроллеры, сервисы, репозитории и таблицы. Такое деление полезно, но оно не объясняет, какие бизнес-границы существуют в системе. В результате модуль `OrderService` может начать управлять оплатой, складом, доставкой, скидками и уведомлениями.

Последствия

- Формируются крупные модули с высокой связностью.
- Команды мешают друг другу при параллельной разработке.
- Развёртывание и изменение одного процесса требуют проверки несвязанных функций.
- Система постепенно превращается в монолит с неявными границами.

Подход DDD

- DDD предлагает сначала найти бизнес-границы и смысловые контексты.
- Технические слои и модули строятся уже вокруг этих границ.

CC BY-NC 4.0 | Институт прикладной математики и компьютерных наук ТГУ

Два уровня DDD

Стратегическое и тактическое проектирование

- Стратегическое проектирование отвечает на вопрос: «Как разделить большую и сложную предметную область на понятные части?».
- Тактическое проектирование отвечает на вопрос: «Как выразить бизнес-правила внутри конкретной части модели в коде?».

Стратегическое проектирование включает:

1. Выделение поддоменов.
2. Определение Core Domain.
3. Формирование Bounded Context.
4. Построение Context Map.
5. Выбор способов интеграции между контекстами.

Тактическое проектирование включает разработку паттернов:

1. Entity.
2. Value Object.
3. Aggregate.
4. Domain Event.
5. Repository.
6. Domain Service.
7. Factory.

Core Domain

Зачем выделять Core Domain?

1. Не все части системы одинаково важны для конкурентоспособности бизнеса.
2. Компания должна вкладывать наиболее сильную инженерную и аналитическую экспертизу в ключевой поддомен.
3. Типовые функции часто выгоднее купить, интегрировать как SaaS или реализовать проще.

Пример для сервиса доставки еды

- Управление меню ресторанов может быть важным, но типовым поддоменом.
- Приём онлайн-платежей обычно является generic subdomain и может быть передан внешнему провайдеру.
- Алгоритм распределения заказов между курьерами может быть core domain, если скорость доставки является конкурентным преимуществом компании.

Практический эффект

- Команда осознанно распределяет бюджет, время и сильных специалистов.
- Архитектурная сложность применяется там, где она действительно окупается.

Ubiquitous Language

Единый язык проекта

Ubiquitous Language — это общий язык, на котором говорят разработчики, аналитики, тестировщики, менеджеры и эксперты предметной области. Этот язык используется в устном обсуждении, документации, названиях классов, API, тестов и событий. Термины языка должны быть точными, согласованными и привязанными к конкретному контексту.

Цель

- Устранить разрыв между тем, как бизнес описывает процессы, и тем, как разработчики реализуют систему.
- Сделать модель понятной не только авторам кода, но и экспертам предметной области.
- Раньше обнаруживать неправильное понимание требований.

Ubiquitous Language

Пример формирования

Неоднозначная формулировка

- Требование «пользователь оформляет заказ» может быть недостаточно точным.
- Неясно, является ли заказ созданным, подтверждённым, оплаченным или переданным в доставку.
- Неясно, можно ли изменить состав заказа после оформления.

Уточнённый язык

- Покупатель создаёт черновик заказа.
- Покупатель подтверждает заказ.
- Система резервирует товар после подтверждения заказа.
- Платёжный провайдер подтверждает оплату.
- Склад передаёт заказ в комплектацию.
- Курьерская служба принимает

Результат

- Каждый термин описывает наблюдаемое бизнес-событие или действие.
- Для каждого состояния можно определить правила перехода.
- Термины естественным образом превращаются в имена методов, событий и тестовых сценариев.

Ubiquitous Language

Правила хорошего языка

1. Один термин должен иметь одно значение внутри одного Bounded Context.
2. Разные термины не должны без причины обозначать одно и то же понятие.
3. Следует избегать технических названий, если они скрывают бизнес-смысл.
4. Следует избегать абстрактных слов вроде «обработать», «выполнить» и «менеджер», когда можно назвать конкретное действие.
5. Язык должен эволюционировать вместе с пониманием предметной области.
6. Спор о названии часто сигнализирует о том, что команда по-разному понимает бизнес-процесс.

Полезные источники языка

- Интервью с экспертами предметной области.
- Анализ регламентов, договоров и бизнес-документов.
- Event Storming и другие совместные моделирующие сессии.
- Реальные пользовательские сценарии и инциденты.

Bounded Context

Bounded Context — это явная граница, внутри которой термины, правила и модель имеют согласованное значение. Внутри контекста модель должна быть непротиворечивой. За пределами контекста одноимённые понятия могут иметь другое устройство и другой смысл.

Что может быть границей контекста?

- Отдельный модуль в модульном монолите.
- Отдельный микросервис.
- Отдельная команда с собственным жизненным циклом разработки.
- Отдельная схема данных или база данных.
- Отдельный набор API и событий.

Уточнение!

- Bounded Context не равен классу, таблице или обязательно отдельному микросервису.
- Один сервис может содержать несколько контекстов, хотя такое решение требует аккуратного разделения модулей.
- Один контекст может временно состоять из нескольких технических компонентов.

Bounded Context

Пример в интернет-магазине

Контекст каталога

- Каталог управляет карточками товаров, характеристиками, категориями и медиафайлами.
- Товар в этом контексте описывается как коммерческое предложение для просмотра покупателем.

Контекст склада

- Склад управляет физическими остатками, ячейками хранения, партиями и операциями перемещения.
- Товар в этом контексте описывается как физическая единица или запас конкретного SKU.

Контекст заказов

- Заказы управляют намерением покупателя приобрести набор позиций.
- Товар в этом контексте может быть зафиксирован как снимок названия, цены и ставки налога на момент покупки.

Все три контекста могут использовать слово «товар». Они не обязаны совместно использовать одну сущность, одну таблицу или одну модель данных.

Context Map

Карта Bounded Context и отношений между ними.

Context map показывает, какие контексты существуют, кто владеет данными и как передаются изменения. Она помогает обсуждать не только технические интеграции, но и организационные зависимости между командами.

Вопросы, на которые отвечает карта

1. Какой контекст является источником истины для конкретных данных.
2. Кто поставляет данные, а кто их потребляет.
3. Может ли изменение поставщика сломать потребителя.
4. Какая команда определяет контракт интеграции.
5. Где требуется трансляция моделей и защита от чужой терминологии.

Context Map

Основные отношения на карте

Upstream и Downstream

Upstream-контекст поставляет данные или возможности другим контекстам. Downstream-контекст использует эти данные или возможности. Например, каталог может быть upstream для контекста заказов, если заказ получает сведения о товаре из каталога.

Conformist

Downstream-контекст принимает модель upstream-контекста почти без изменений. Этот вариант прост в реализации, но повышает зависимость потребителя от внешней модели. Подход допустим, если модель поставщика стабильна и подходит потребителю.

Customer-Supplier

Команда downstream рассматривается как заказчик команды upstream. Поставщик учитывает потребности потребителя при развитии API и контрактов. Такой вариант требует регулярной коммуникации и согласованного планирования.

Shared Kernel

Несколько контекстов совместно используют небольшой общий модуль модели. Общий код требует строгого контроля изменений, потому что влияет на всех участников. Shared Kernel должен быть малым и стабильным.

Anti-Corruption Layer

Проблема внешней модели

Внешняя система может использовать неудобные термины, устаревшие форматы и правила, не подходящие для внутренней модели. Прямое использование её объектов внутри домена загрязняет собственный Ubiquitous Language. Изменения внешнего API начинают распространяться по всему приложению.

Решение

- Anti-Corruption Layer, или ACL, — это слой адаптации между моделями разных контекстов.
- ACL переводит внешние DTO, события и коды статусов в понятия внутреннего домена.
- Внутренний код зависит от собственного интерфейса, а не от деталей внешнего поставщика.

Пример

- Внешний платёжный провайдер возвращает статус `PAYMENT_CAPTURED`.
- ACL преобразует этот статус в доменное событие «Платёж подтверждён».
- Домен заказов не обязан знать терминологию конкретного провайдера.

Entity, или сущность, — это объект, который определяется прежде всего своей идентичностью, а не набором значений. Сущность сохраняет идентичность при изменении своих атрибутов. Две сущности с одинаковыми данными могут быть разными, если у них разные идентификаторы.

Примеры

- Заказ с номером 1523 остаётся тем же заказом, даже если изменился адрес доставки.
- Пользователь остаётся тем же пользователем, даже если сменил имя или номер телефона.
- Банковский счёт сохраняет идентичность при изменении баланса.

Ответственность Entity

- Сущность должна инкапсулировать поведение, относящееся к её жизненному циклу.
- Сущность должна защищать свои инварианты.
- Внешний код не должен напрямую переводить сущность в недопустимое состояние.

CC BY-NC 4.0 | Институт прикладной математики и компьютерных наук ТГУ

Value Object, или объект-значение, определяется своими значениями, а не идентичностью. Два объекта-значения считаются равными, если равны все значимые компоненты. Объекты-значения рекомендуется делать неизменяемыми.

Примеры

1. Деньги состоят из суммы и валюты.
2. Адрес доставки может состоять из страны, города, улицы, дома и квартиры.
3. Период бронирования состоит из даты начала и даты окончания.
4. Координаты состоят из широты и долготы.

Преимущества неизменяемости

1. Неизменяемые объекты проще безопасно передавать между компонентами.
2. Их легче сравнивать и тестировать.
3. Они уменьшают число ошибок, связанных со скрытым изменением состояния.
4. Вместо изменения значения создаётся новый объект-значение.

CC BY-NC 4.0 | Институт прикладной математики и компьютерных наук ТГУ

Aggregate и Aggregate Root

Aggregate

Aggregate, или агрегат, — это группа связанных объектов доменной модели, рассматриваемая как единое целое с точки зрения согласованности данных. Агрегат задаёт границу транзакционной согласованности. Внутри агрегата должны соблюдаться инварианты, важные сразу после завершения команды.

Aggregate Root

Aggregate Root — это единственный объект агрегата, доступный внешнему коду напрямую. Внешние объекты могут ссылаться только на корень агрегата по идентификатору. Корень управляет изменениями внутренних объектов и проверяет инварианты.

Пример

- Заказ может быть Aggregate Root.
- Позиции заказа могут быть внутренними сущностями агрегата.
- Внешний код добавляет позицию через метод заказа, а не изменяет коллекцию позиций напрямую.

Инварианты и границы агрегата

Инвариант

Инвариант — это правило, которое должно оставаться истинным после любой успешно выполненной доменной операции.

- Например, заказ нельзя подтвердить без хотя бы одной позиции.
- Например, количество позиции заказа должно быть положительным.
- Например, сумма заказа должна соответствовать сумме его позиций и скидок.

Зачем нужна небольшая граница агрегата?

- Большой агрегат требует блокировать и загружать слишком много данных в одной транзакции.
- Большой агрегат снижает производительность и увеличивает вероятность конфликтов параллельного изменения.
- Связанные, но не требующие мгновенной согласованности объекты лучше разделять на разные агрегаты.

Практическое правило

- В одну транзакцию следует включать только те данные, которые необходимы для защиты одного инварианта.
- Между разными агрегатами часто используется eventual consistency через доменные события.

CC BY-NC 4.0 | Институт прикладной математики и компьютерных наук ТГУ

Domain Event

Domain Event, или доменное событие, фиксирует значимый факт, произошедший в предметной области. Событие формулируется в прошедшем времени, потому что оно описывает уже состоявшееся изменение. Событие содержит данные, важные для реакции других частей системы.

Примеры

- Заказ подтверждён.
- Платёж принят.
- Товар зарезервирован.
- Подписка отменена.
- Лимит кредита превышен.

Зачем нужны доменные события

1. Они делают важные бизнес-факты явными.
2. Они уменьшают прямую связанность между агрегатами и контекстами.
3. Они позволяют запускать побочные процессы после успешного изменения состояния.
4. Они формируют удобную основу для аудита, интеграций и аналитики.

События

Пример обработки Domain Event

Сценарий

1. Покупатель подтверждает заказ.
2. Агрегат заказа проверяет инварианты и меняет состояние.
3. Агрегат публикует событие «Заказ подтверждён».
4. Обработчик события инициирует резервирование товаров.
5. Другой обработчик отправляет уведомление покупателю.
6. Ещё один обработчик передаёт данные в систему аналитики.

Важное правило

- Публиковать интеграционное сообщение следует только после успешного сохранения изменения в основной транзакции.
- Для надёжной доставки часто применяется паттерн Transactional Outbox.
- События должны быть идемпотентными или обрабатываться с учётом повторной доставки.

Repository

Определение

Repository, или репозиторий, — это абстракция доступа к агрегатам. Репозиторий предоставляет домену коллекционноподобный интерфейс для загрузки и сохранения агрегатов. Доменная модель не должна зависеть от SQL, ORM, HTTP-клиентов или формата хранения.

Типичные операции

- Получить агрегат по идентификатору.
- Сохранить изменённый агрегат.
- Найти агрегаты по критериям, необходимым для доменного сценария.

Ограничения

1. Репозиторий обычно проектируется для Aggregate Root, а не для каждой внутренней сущности.
2. Интерфейс репозитория должен отражать потребности домена, а не все возможности конкретной базы данных.
3. Сложные отчёты и поисковые запросы могут использовать отдельную read-модель вместо перегрузки репозитория.

Domain Service и Application Service

Domain Service

Domain Service содержит доменную операцию, которая неестественно принадлежит одной сущности или одному объекту-значению.

1. Выражает правило предметной области и использует термины Ubiquitous Language.
2. Не должен быть контейнером для всей бизнес-логики.

Application Service

Application Service реализует сценарий использования системы.

1. Координирует загрузку агрегатов, вызов доменных методов, сохранение результатов и публикацию событий.
2. Обычно не содержит сложных бизнес-правил.
3. Управляет транзакциями, авторизацией на уровне сценария и взаимодействием с портами приложения.

Пример

- Сценарий «подтвердить заказ» может быть реализован прикладным сервисом.
- Проверка допустимости перехода заказа в статус подтверждения должна находиться в самом агрегате заказа.

Слои архитектуры в DDD

Четыре уровня определения архитектуры

Доменный слой

Доменный слой содержит модель предметной области и бизнес-правила. В него входят Entity, Value Object, Aggregate, Domain Event и Domain Service. Не должен зависеть от фреймворков, базы данных и транспорта API.

Прикладной слой

Прикладной слой реализует use cases системы. Он принимает команды, координирует доменные объекты и возвращает результаты сценариев. Зависит от домена, но не подменяет его правила.

Инфраструктурный слой

Инфраструктурный слой реализует технические детали. В него входят адаптеры базы данных, брокеры сообщений, HTTP-клиенты, файловые хранилища и реализации репозиториев. Зависит от интерфейсов, определённых внутренними слоями.

Интерфейсный слой

Интерфейсный слой принимает HTTP-запросы, сообщения очередей, CLI-команды или события. Он преобразует внешние данные в команды прикладного слоя. Он не должен содержать бизнес-правила.

CC BY-NC 4.0 | Институт прикладной математики и компьютерных наук ТГУ

Спасибо за внимание!

Всё о курсе:
https://marigostra.ru/materials/engineering-adv.html

Электронная почта:
msp@luwrain.org

Telegram:
@Marigostra



Материал можно распространять с соблюдением требований Creative Commons Attribution-NonCommercial 4.0 International License. При обнаружении любых ошибок и неточностей в материале просьба сообщить о них автору.