merge Иннополис 17-18 апреля 2026

DDD и современная архитектура: как проектировать модель и отражать ее в код

Максим Цепков

IT-архитектор и бизнес-аналитик

Навигатор и эксперт по миру Agile, методов самоуправления и моделей soft skill

mtsepkov mtsepkov.org

Немного обо мне

Создание и внедрение больших корпоративных систем (30 лет)

- Знание практик операционного управления и ведения проектов в коммерческих и государственных организациях и банках
- Опыт управления проектами в IT: от инженерного подхода и PMBOK – к современным Agile-методам (с 2007 года)
- Опыт перестройки организаций при внедрении систем

Навигация в менеджменте цифрового мира

- Agile и самоуправление: бирюзовые организации, холакратия и социократия (**книга**, статьи и выступления)
- Модель спиральной динамики (с 2013) и другие модели soft skills, **модели личности** (**книга**), и **самоопределения** (**книга**)

Китай как лидер новой волны технологических революций

На моем сайте mtsepkov.org – выступления, блог, статьи, заметки с конференций и много других материалов

О чем будет рассказ?

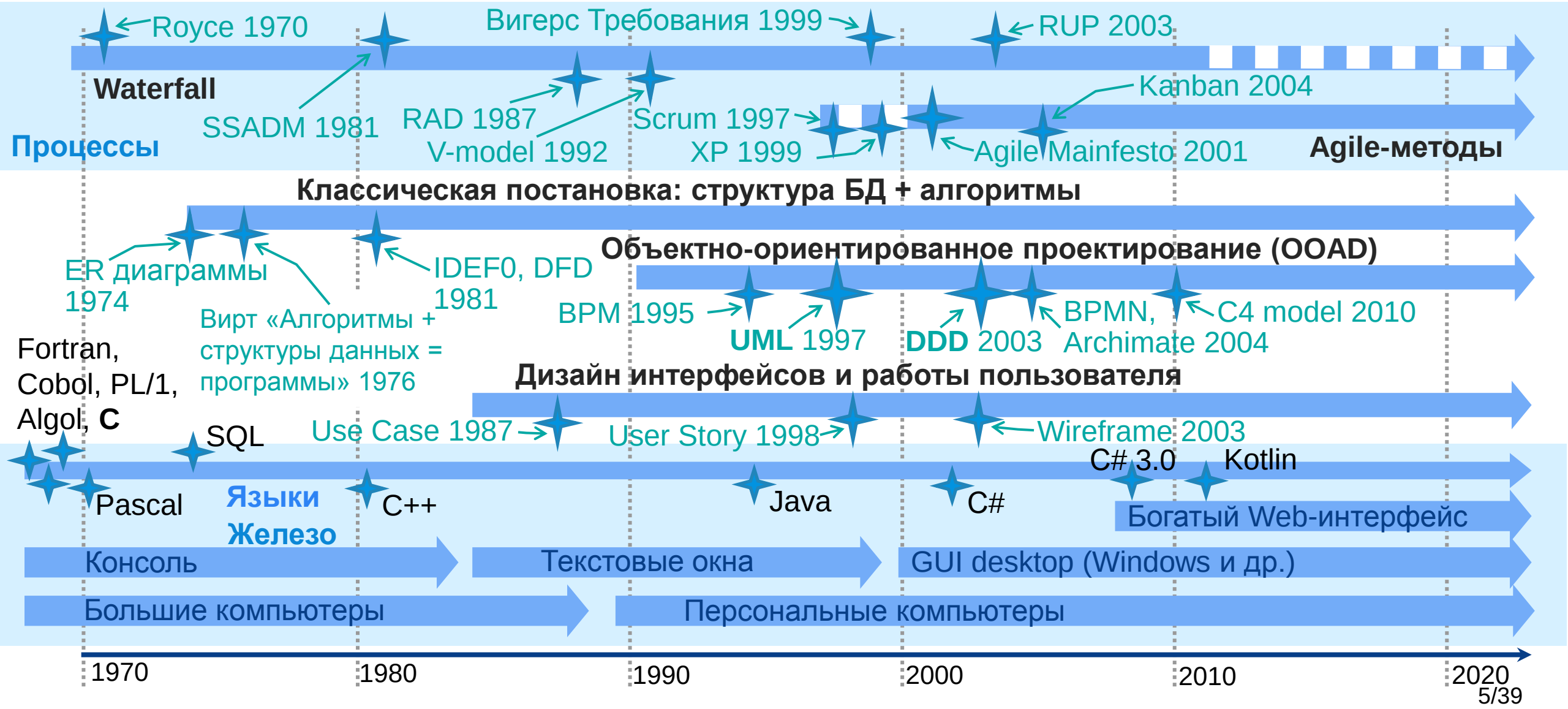
- **Концепция DDD**
  - DDD был создан в 2003, чтобы снять барьеры коммуникации бизнеса с разработкой
  - **Решение: единый язык и единая модель с прозрачным отражением в код**
- **Отражение модели в код**
  - Способы отражения в код были проработаны для архитектуры монолитнов
  - Как эффективно применять DDD в современной архитектуре?

Мои выступления по этой и смежным темам:

- DDD: модели вместо требований 9 лет спустя (ЛАФ-2023)
- Визуальное проектирование масштабируемых приложений (TechLead-2021)
- От монолитных моделей предметной области – к модульным (WIAD-2017)
- Архитектура софта и бизнеса в сложном ИТ-ландшафте (AnalystDays-2025)
- Постановка от модели бизнеса до детального дизайна требований: как делать и кому (13.03.2025)
- Системное мышление и его место в работе аналитика (AnalystDays-2024a)

Domain Driven Design – что это и зачем он нужен?

Проектирование и его окружение

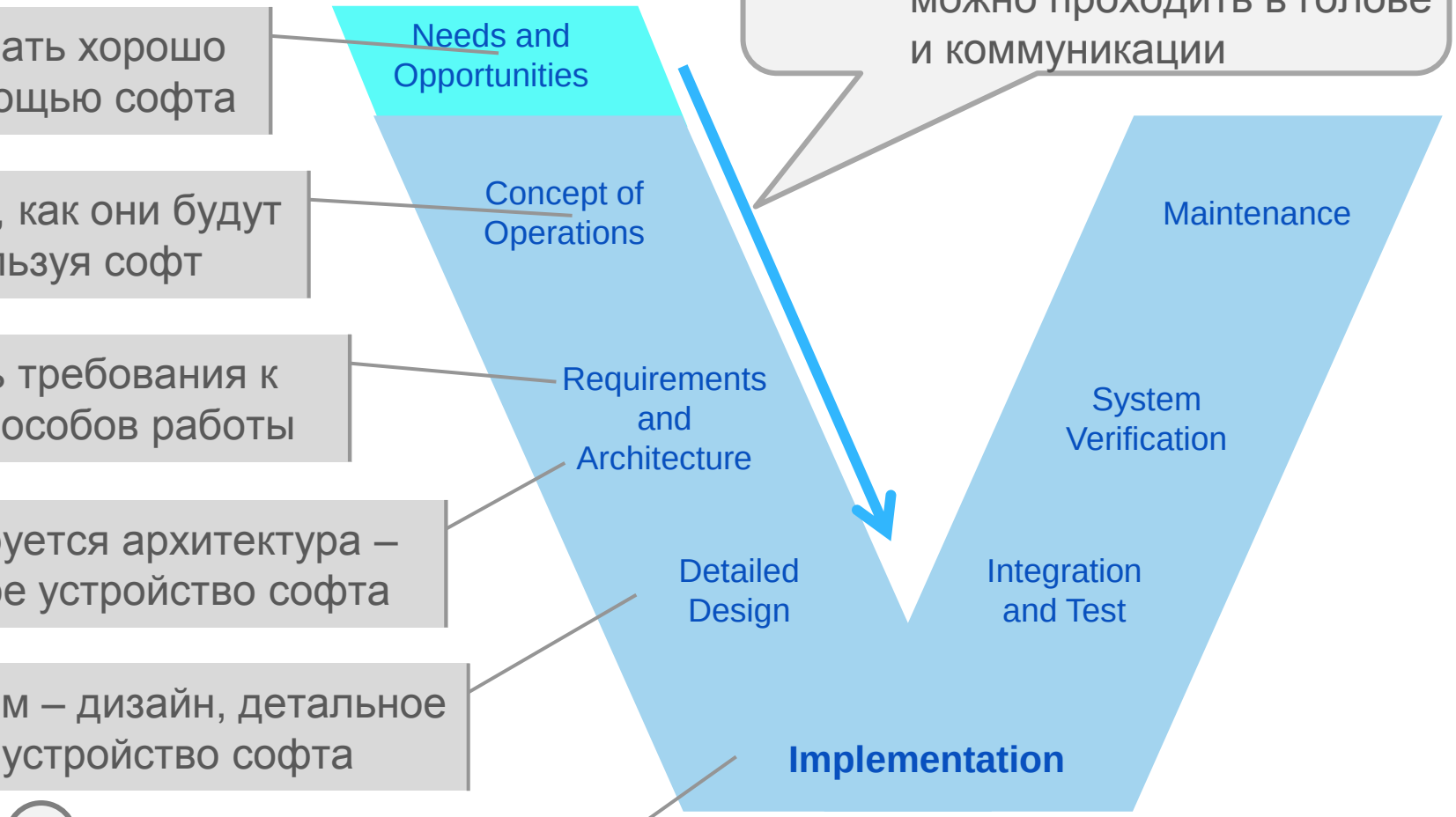


Путь проекта – V-модель

Этот путь не обязательно требует документов, его можно проходить в голове и коммуникации

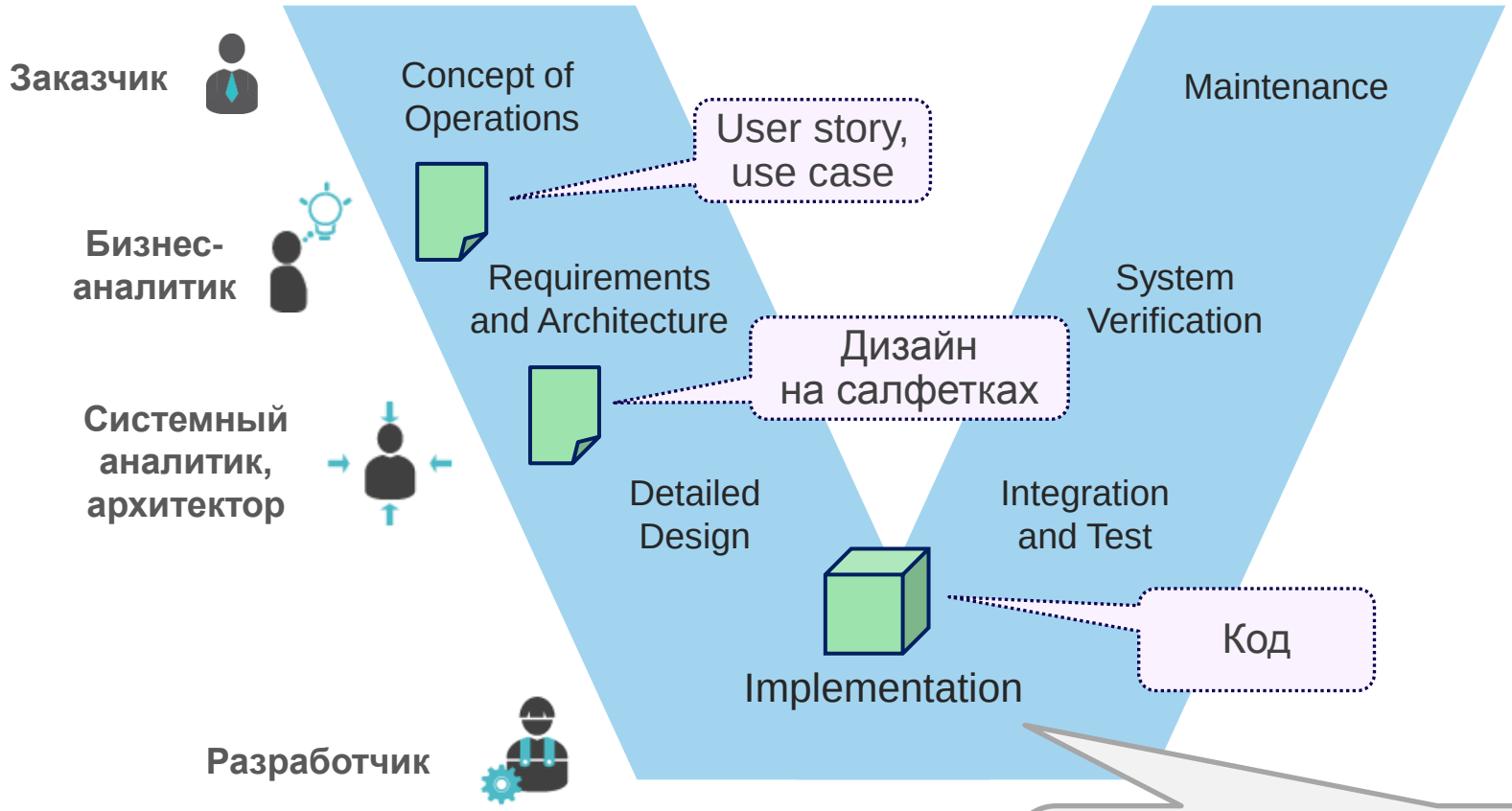
1. Начинается с идеи сделать хорошо некоторым людям с помощью софта
2. Далее надо придумать, как они будут действовать, используя софт
3. И сформулировать требования к софту, исходя из способов работы
4. Затем проектируется архитектура – принципиальное устройство софта
5. Затем – дизайн, детальное устройство софта
6. И реализация в коде

Различный уровень заказа



[V-Model (Wikipedia)]

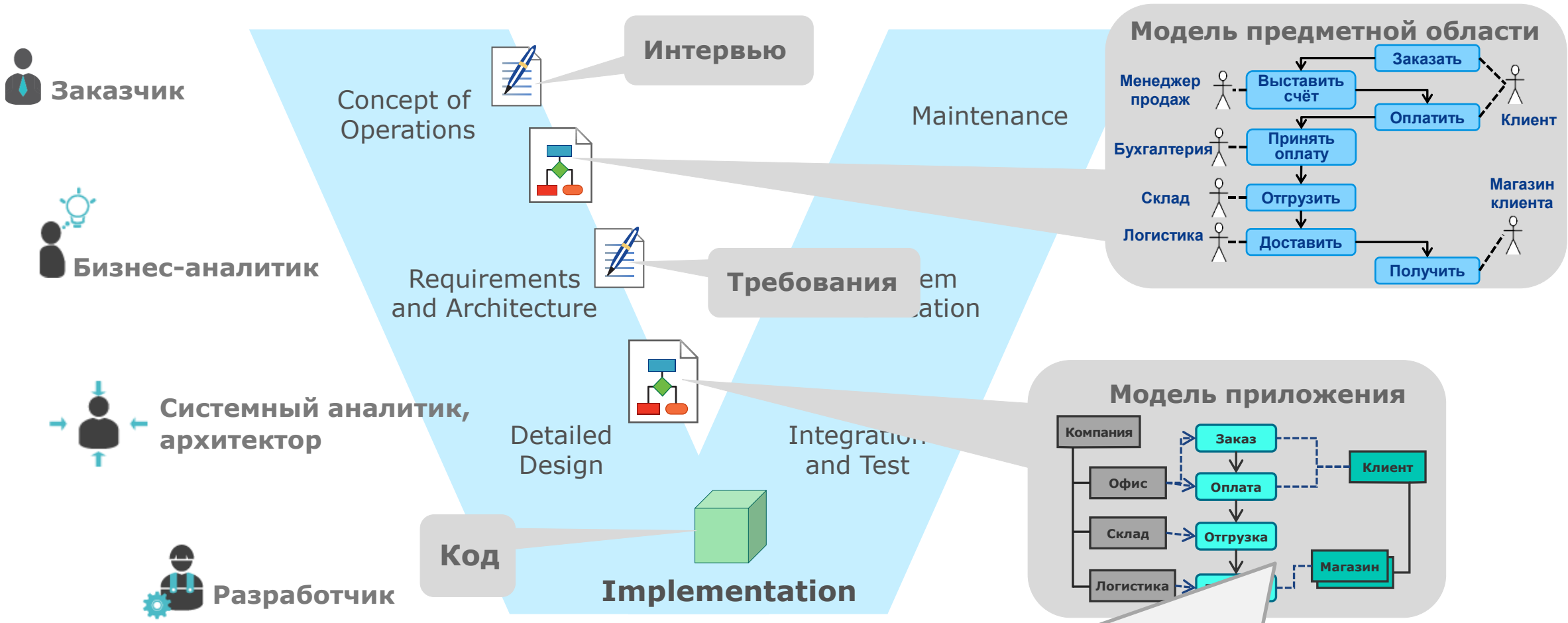
Можно и без моделей



Проблемы:
- нет целостного представления
- реализуемость не понятна

Хорошо подходит для небольших проектов и сервисов

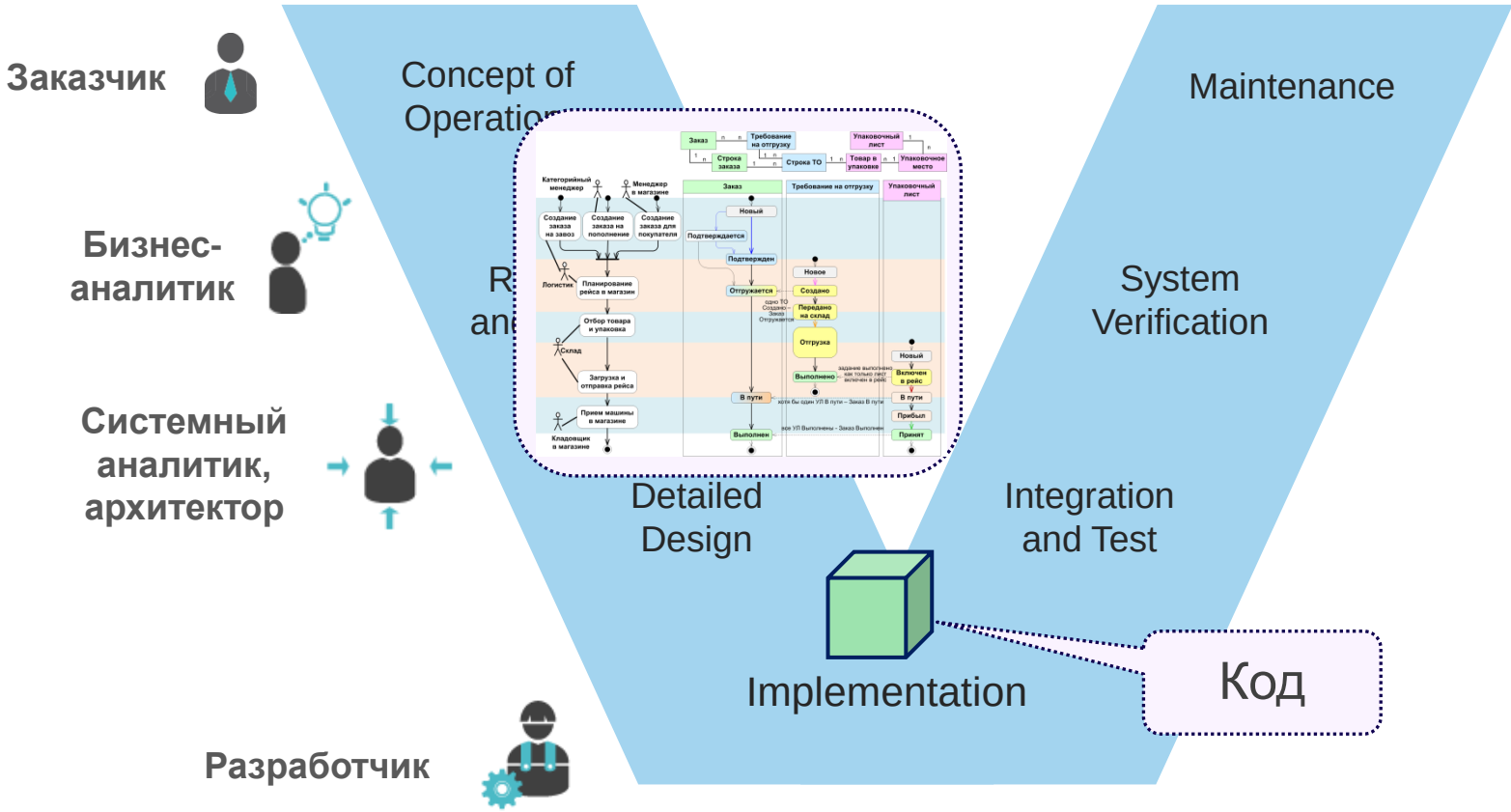
Модели в классических постановках



Проблема: каждый уровень отдельно – сложно изменять и поддерживать соответствие

Domain Driven Design

- **Единый язык**
  - На основе терминов предметной области
  - Понятен всем участникам проекта
- **Единая модель**, приложения и его встройки в бизнес
- Прозрачное отражение модели в код



Основные преимущества единой модели проявляются **в ходе развития системы, а также на внедрении**, но строить ее необходима с начала разработки

Три категории постановок

1. Разработка новой системы поверх фрагментарной и малой автоматизации:
   - выясняем модель бизнес-процессов
   - создаем модель системы
   - работаем над ее встройкой в процессы, изменяя их
2. Доработка существующей системы: проектируем изменения модели существующей системы и ее встройки в процессы
3. Разработка новой системы, заменяющей существующую:
   - существующие процессы несут отпечаток старой системы, его надо снять
   - проектируем новую систему и процессы
   - проектируем работу на переходном этапе, обеспечивая мягкую замену

Большинство методов анализа и проектирования создавались, когда бизнес-процессы были слабо автоматизированы, в расчете **на первую ситуацию.** Сейчас мы имеем дело **со второй и третьей.**

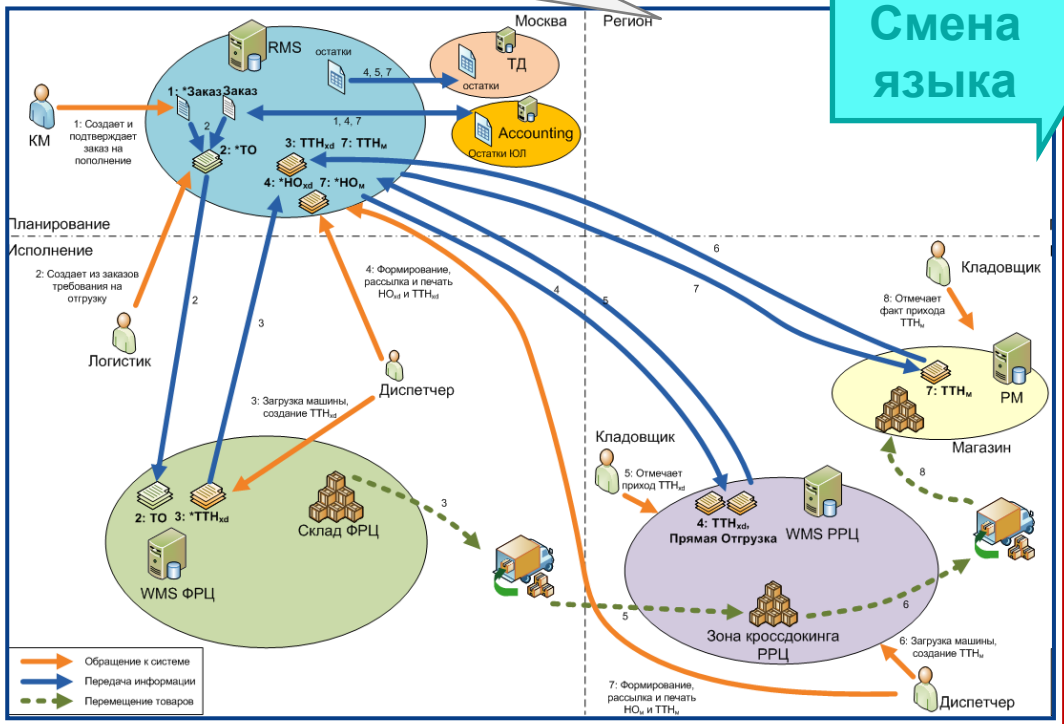
От неформальных схем – к workflow

Неформальная схема деятельности и ее отражение в существующих системах

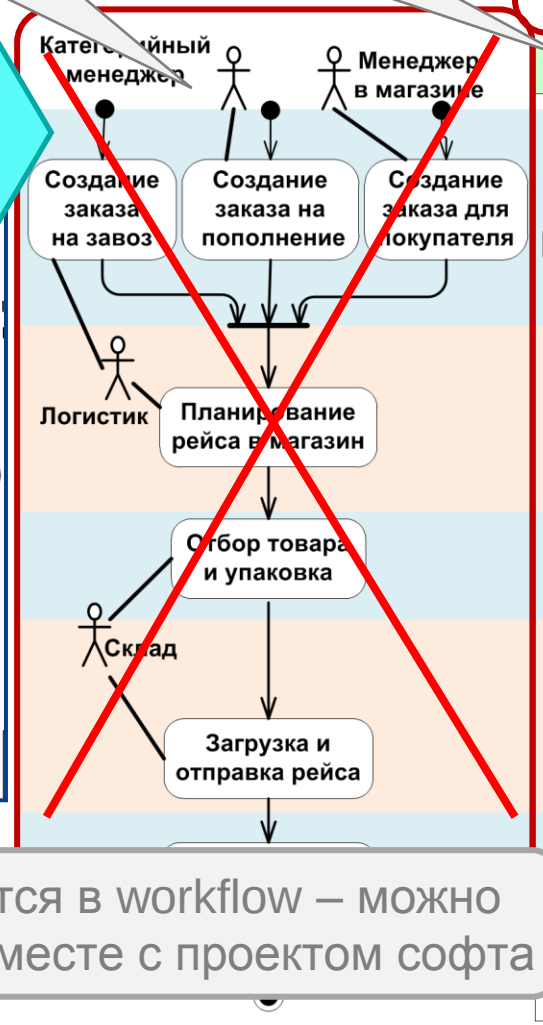
Бизнес-процесс – activity diagram

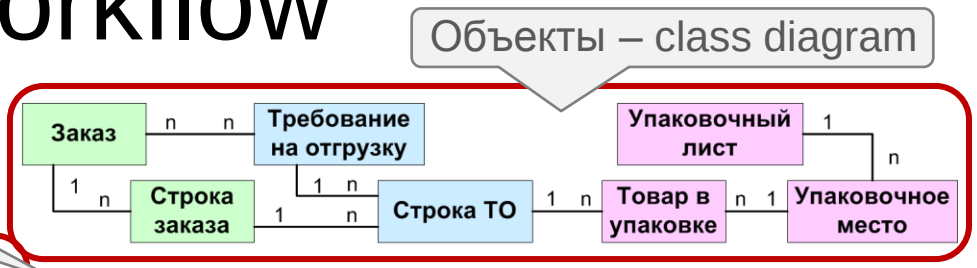
Состояния документов – state diagram

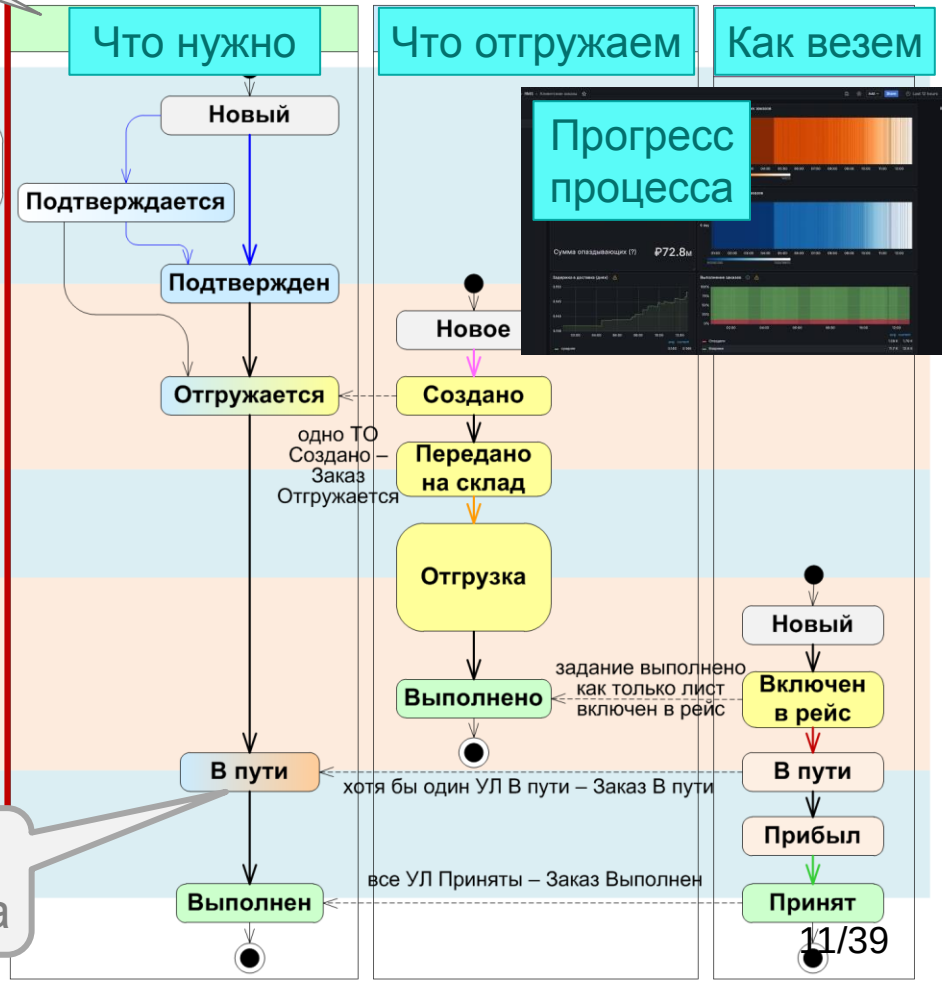
Объекты – class diagram



Смена языка





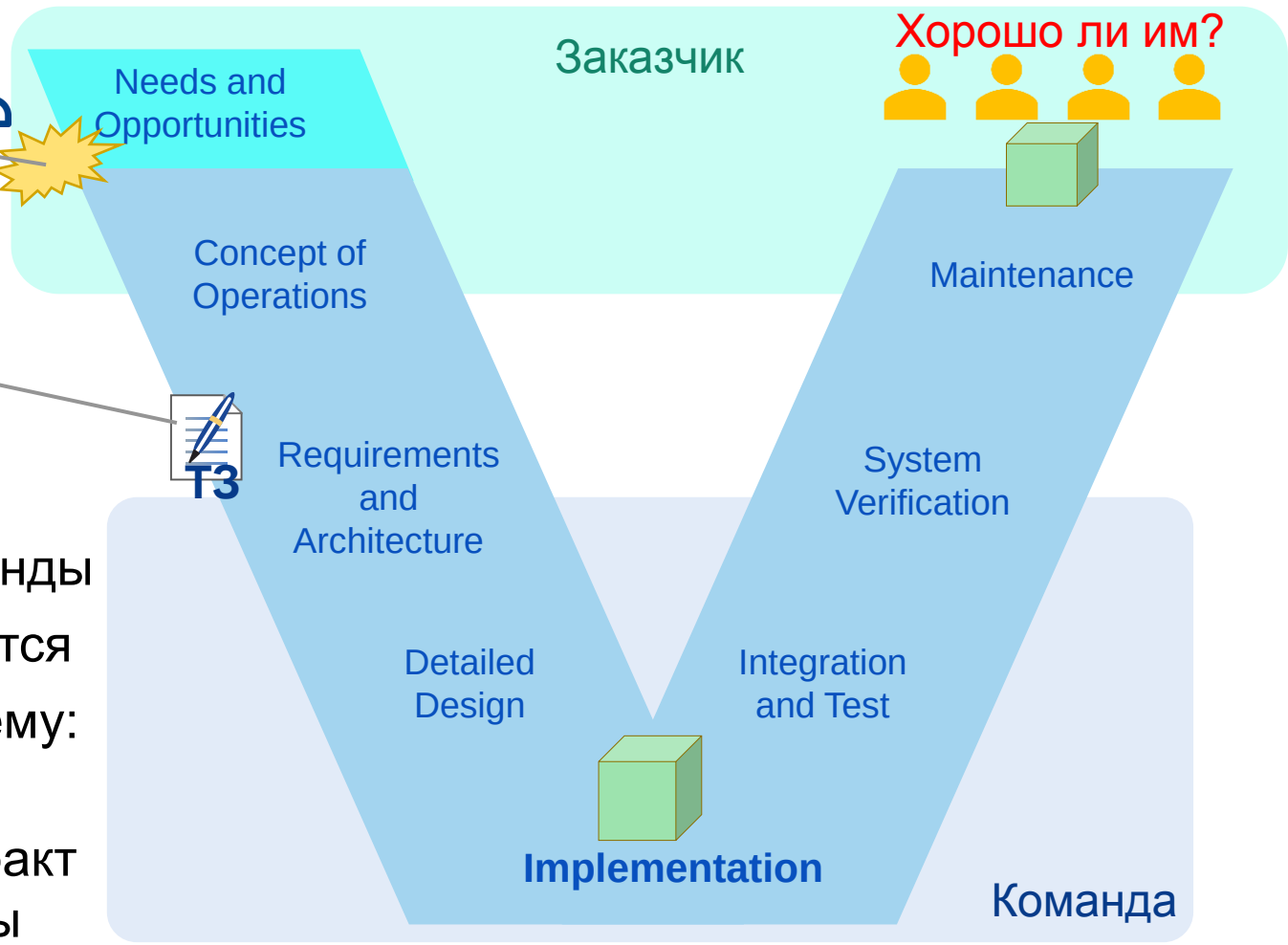


Бизнес-процессы прозрачно отражаются в workflow – можно проектировать их на схеме workflow вместе с проектом софта

ТЗ — способ сменить ответственность

Идея: сделать хорошо людям

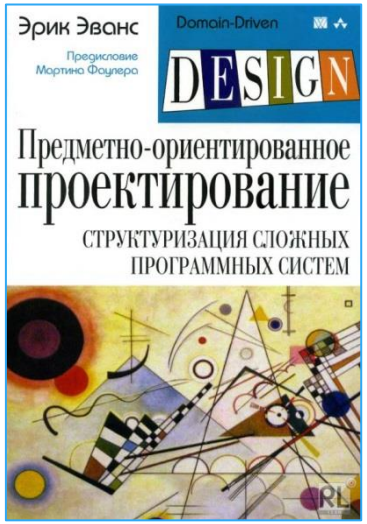
Задача: напишите софт, удовлетворяющий требованиям



- Ответственность за достижение целей проекта вынесена за пределы ИТ-команды
- Как следствие, цели часто не достигаются
- Выделение этапа ТЗ не решает проблему: ТЗ утверждают, а гарантий оно не даёт
- Но требования могут определять контракт с заказчиком проекта – тогда они нужны

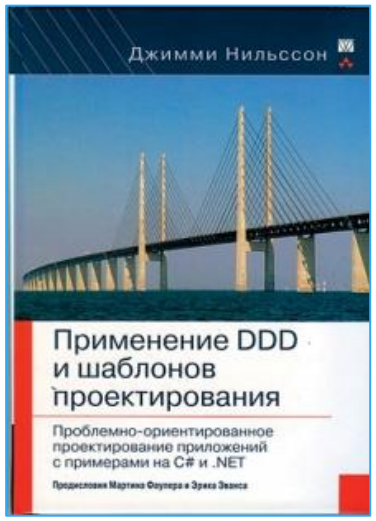
Если требования – важная часть модели разработки, **преимущества DDD сильно уменьшаются** – вам не надо общаться с бизнесом

DDD – источники



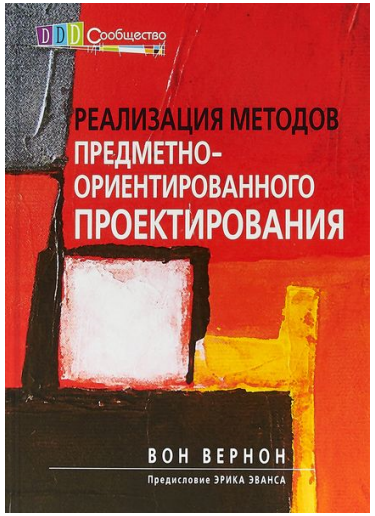
Концептуальная книга Эрика Эванса

- на английском – в 2003 г.
- на русском – только в 2010 г.



Практическая книга Джимми Нильссона

- на английском – в 2006 г.
- на русском – в 2007 г. (почти сразу!)



Обновление от Вона Вернона

- на английском – в 2013 г.
- на русском – в 2016 г.

Что изменяется в проекте при использовании DDD

- Вся команда владеет единым языком, и работает с единой моделью
- Разработчики должны разбираться в предметной области, понимать цели и задачи пользователей при работе с системой – нужно погружение
- Меняется роль аналитиков: они не ограждают разработку от предметной области, а помогают ее понять и проектировать адекватный софт
- Меняется роль тестировщика: он не формально проверяет софт на соответствие требованиям, а моделирует работу пользователя

Именование объектов

- Бизнес-объекты, их атрибуты и методы называются по-русски
- Для них в модели должны быть зафиксированы английские названия
- Политика: перечисление и/или простые справочники
- Все названия в коде — на основе английских названий в модели
- Надписи в интерфейсе — на основе русских названий в модели
- Технологичная поддержка для переименования объектов и атрибутов
- Смена русского названия (полу)автоматически отражается в интерфейсе
- Смена английского названия через (полу)автоматический рефакторинг
- Регламент для изменения — преобразование перечисления в справочник
- Однородное решение для получения пользовательских названий экземпляров объектов, используемых в сообщениях и логах

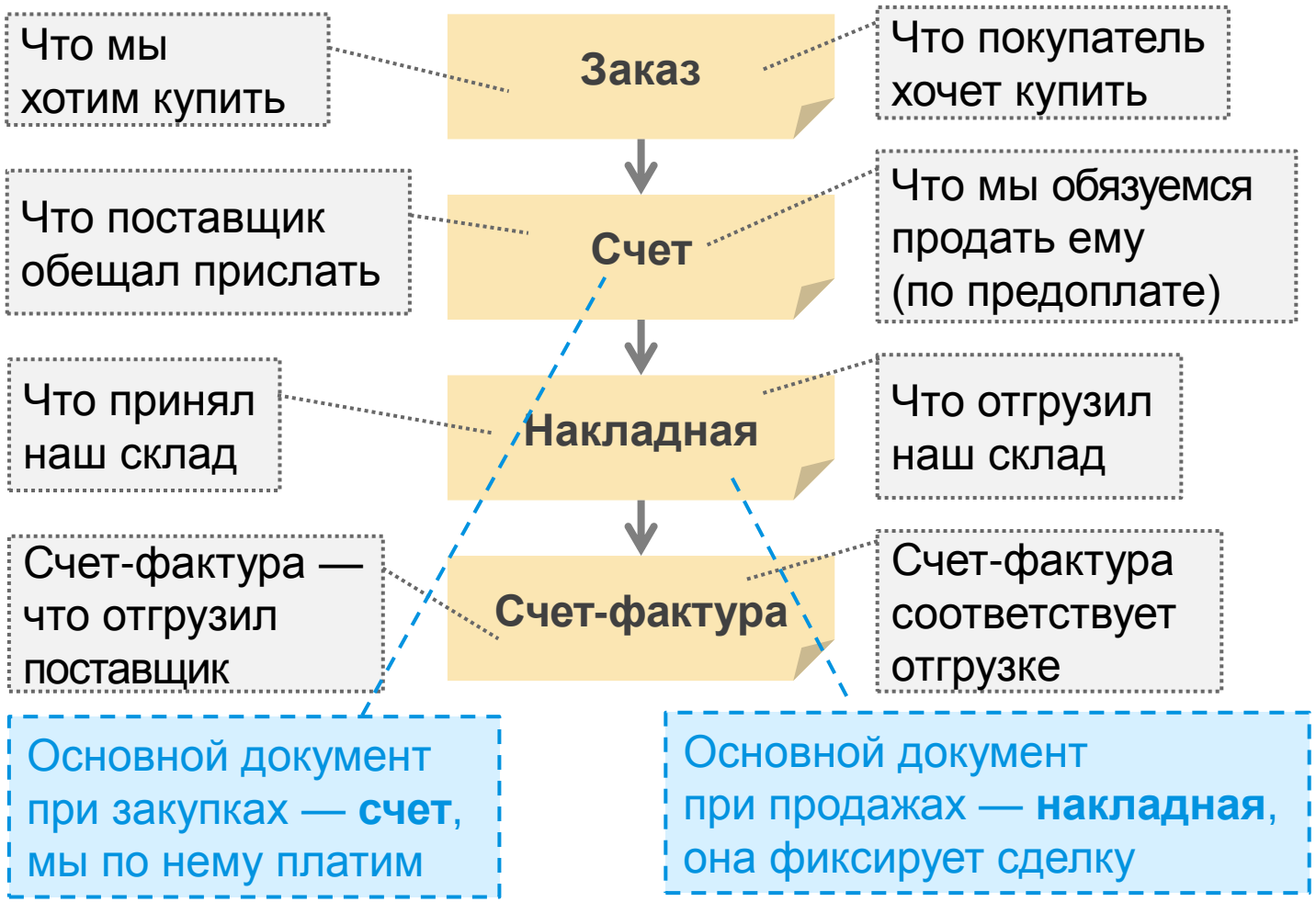
Модель предметной области — модульная: делим предметную область на ограниченные контексты

В сервисной архитектуре контекст обычно включает несколько сервисов

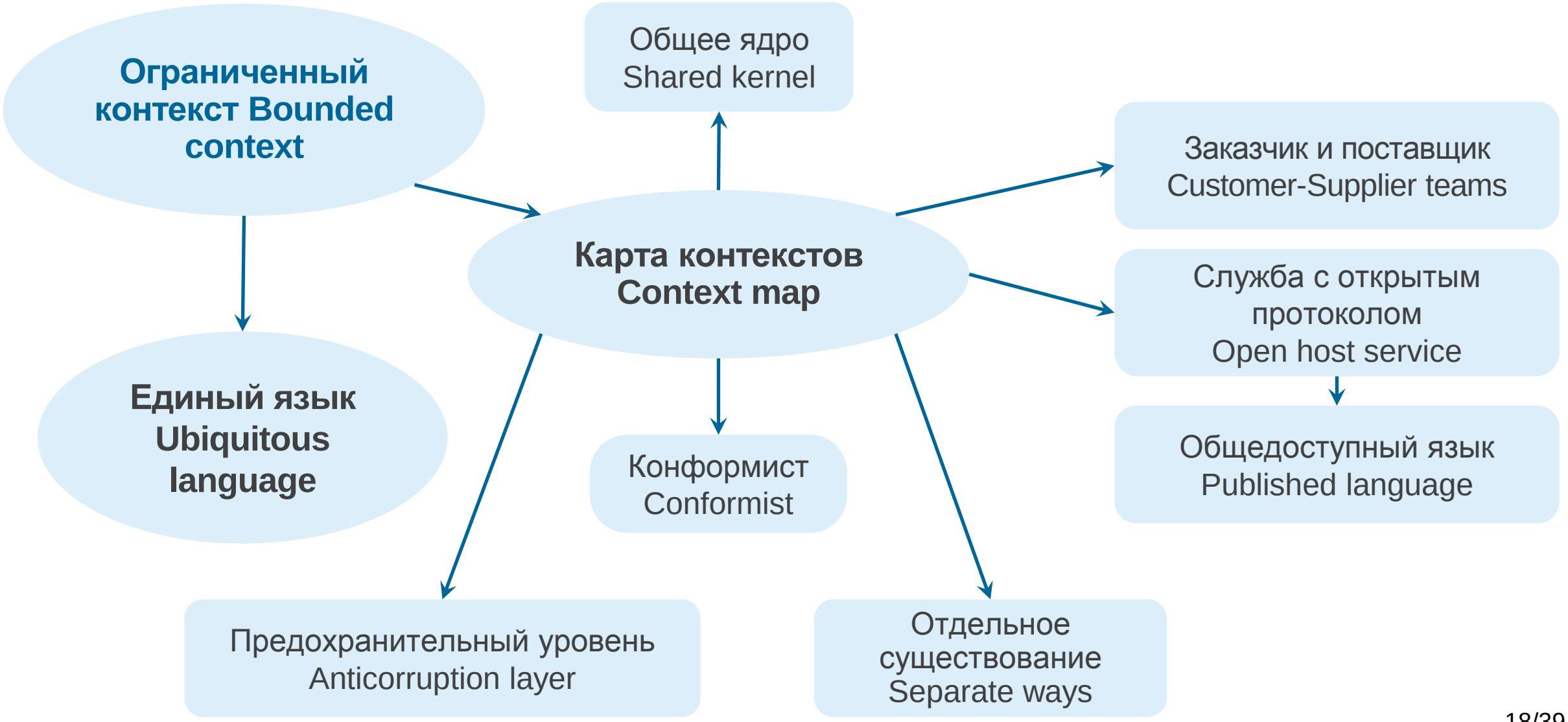
Разница контекстов сделки

Оптовая закупка

Оптовая продажа



Шаблоны работы с контекстами



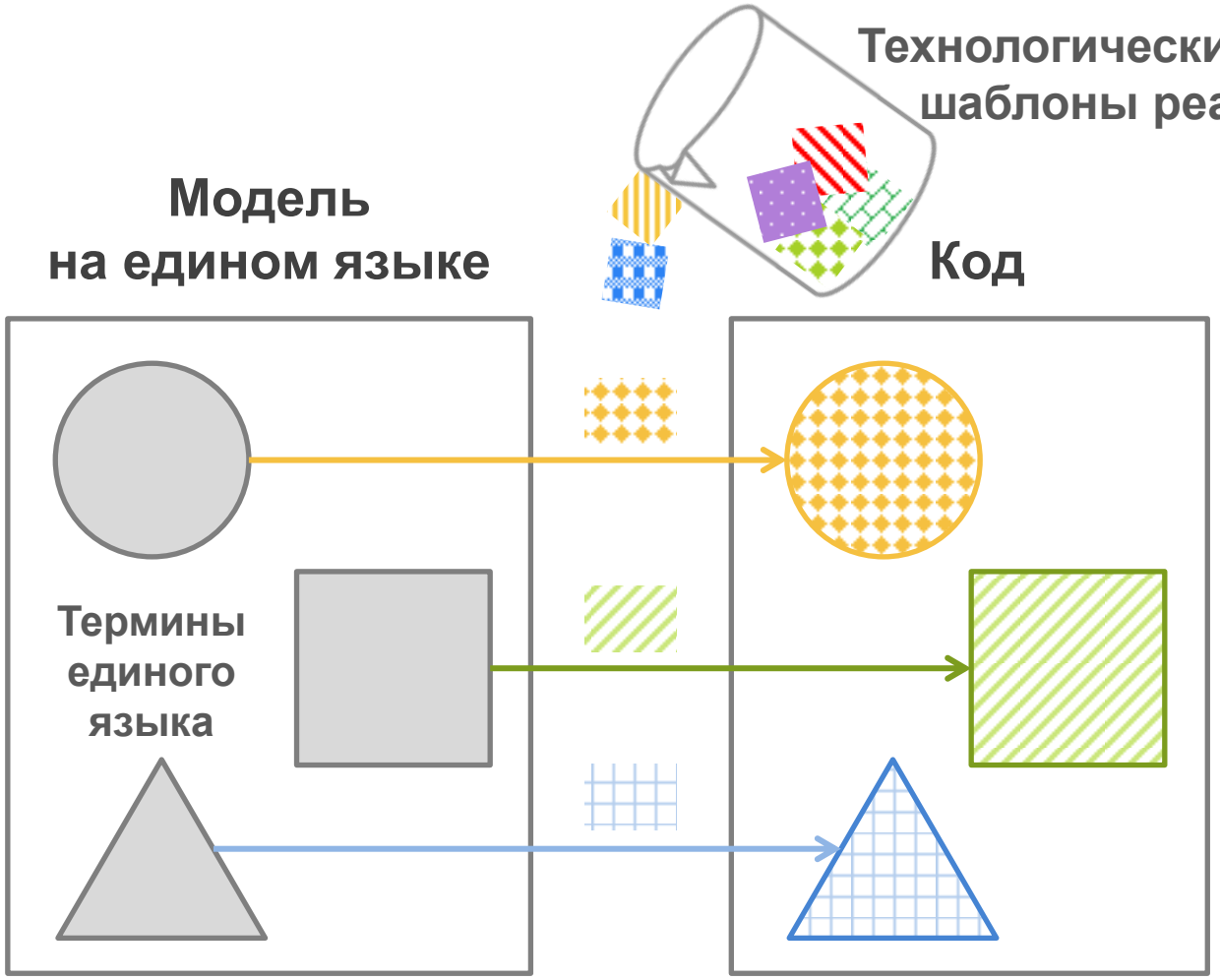
Отражение модели в код

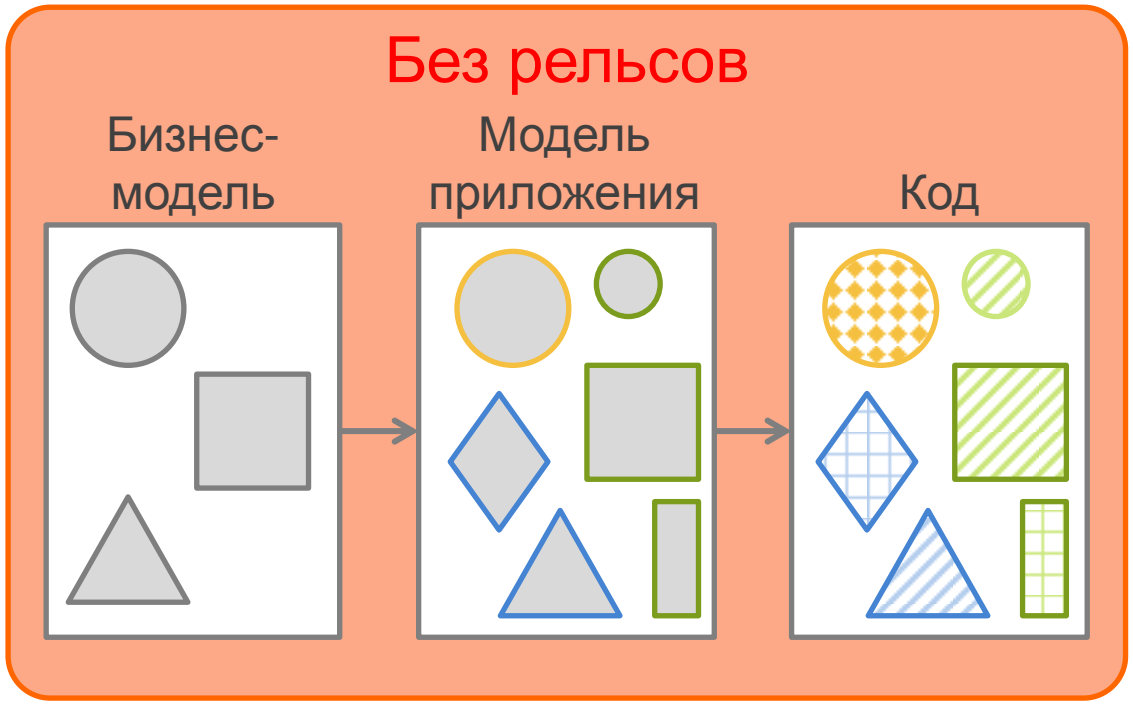
Основной паттерн в книгах – RichObject: он простой

- Необходимость понимания модели заказчиком ограничивает модель: технические подробности и сложные методы недопустимы
- Бизнес-объект включает все аспекты деятельности и связан с другими
  - Клиент – партнер и субъект права, юридические и управленческие аспекты вместе
  - Заказ – полный цикл от переговоров до исполнения, включая юридические аспектыое
  - ☹ При отражении в IT-объекты получается очень похоже на антипаттерн Big Object
  - ☹ Сложно понимать, развивать, тестировать
- Принципы ООП побуждают к применению шаблонов (стратегии, фабрики) и мелкой декомпозиции, есть ограничния со стороны фреймворков
  - ☹ **Сложная модель не понимается заказчиком, простая – не отражает реализацию**

Единственная модель на едином языке –
и преимущество DDD, и источник проблем

Решение: технологические рельсы – типовой способ перевода модели на едином языке в код





Workflow на основе шаблона state entity

- Принцип: единственный атрибут документа отражает его состояние в workflow, возможные действия над ним и права на их совершение
- Граф переходов отражает возможные изменения состояния
- Граф переходов отражает реализацию документом бизнес-процесса, согласован и понятен бизнесу
- Реализация собственным движком или библиотекой типа Activity
- Интерфейсы единообразно опираются на состояния документов, включая проверку на интерфейсах и логику доступных действий

Джимми Нильссон разбирает четыре способа реализации workflow. Я считаю state entity наиболее технологичным, смотри мое выступление «Domain Driven Design — от требований до кода»

Отражение бизнес-объектов

- Анемичный транспортный объект или несколько идентичных
- Прозрачное отражение транспортного объекта в БД (с ORM или без)
- Класс-контроллер бизнес-логики на бэкенд или несколько — для одного объекта, фабрика для создания, для обработки коллекций
- Классы-контроллеры логики для фронтенда и для клиентов

Технологические рельсы

- Генерация всех объектов данных и БД по одному описанию (JSON, XML)
- Решение для workflow на основе state entity
- Библиотека или базовый класс для naked-objects интерфейсов
- API, уведомления, печать и другая инфраструктура — на описаниях

Единство кода для бэк, фронт и интеграции

Логика, которая присутствует в нескольких слоях приложения:

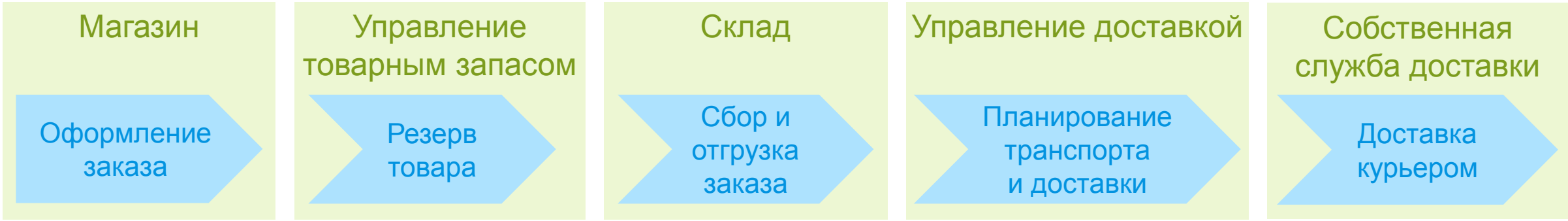
- Бизнес-логика доступности действий на формах и проверка прав
- Бизнес-логика проверки значений для атрибутов и удобного ввода

Проблемы, требующие технологичного решения:

- Как обеспечивается соответствие логики в разных слоях — вручную, одинаковым языком реализации, генерацией на основе метаописаний или конфигураций, как-то иначе?
- Как поддерживается работа бэка с многими версиями фронта и API?

Гибкость инфраструктуры

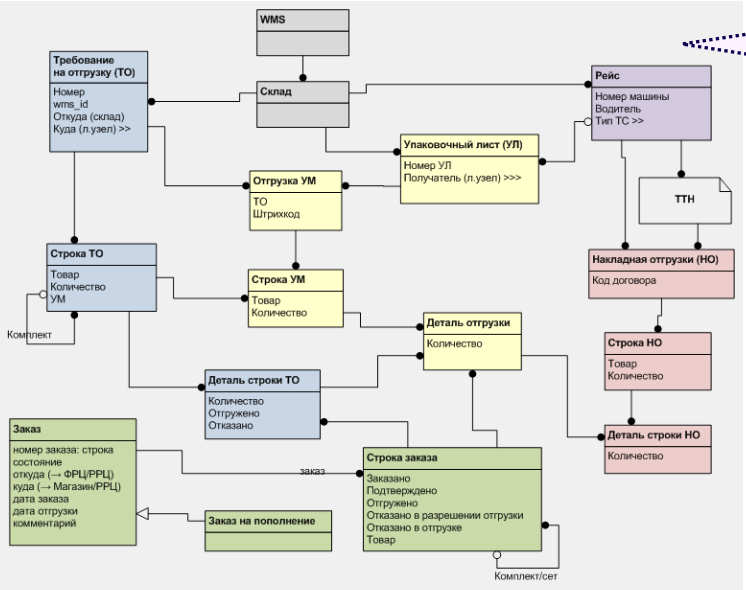
Пример: продажа с доставкой в розничной торговой сети



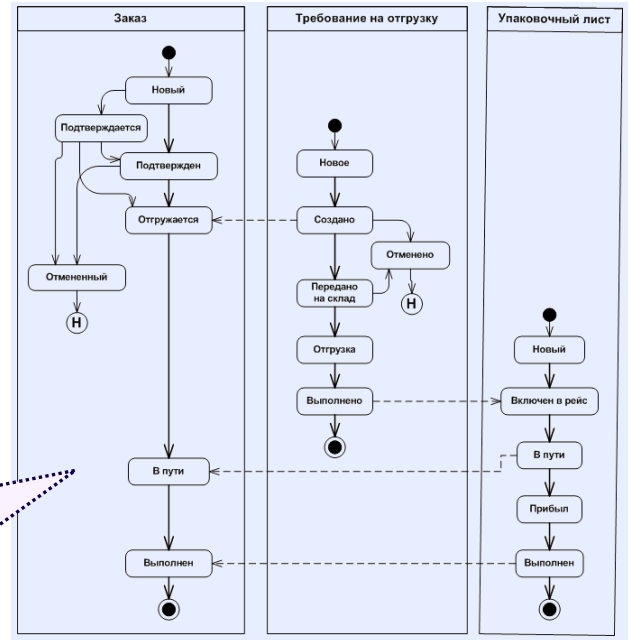
Инфраструктура интеграции поддерживает движение объекта по бизнес-процессу

- В одних системах может быть общий анемичный объект «Заказ покупателя», в других он преобразуется на входе и выходе из собственного представления
- Универсальное представление в шине дает узкое горло
- Надо уметь добавить к заказу «план проезда» — скан рисунка покупателя, который нужен только курьеру

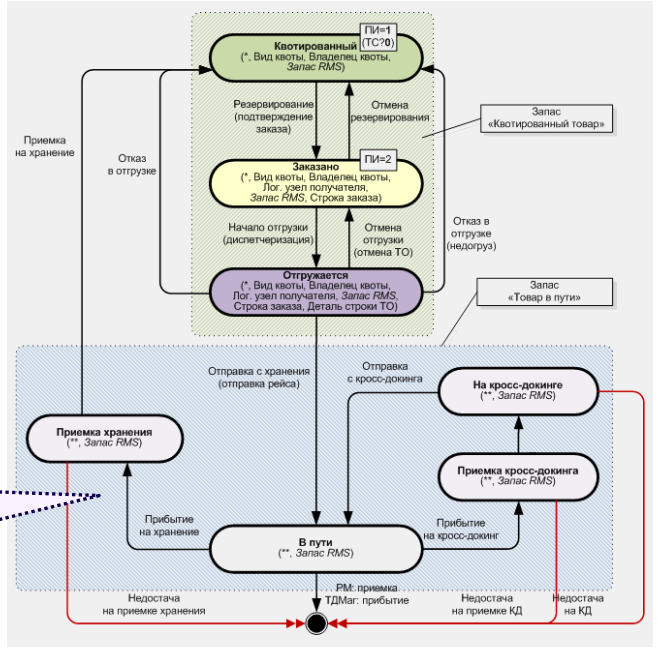
Пример: шаблон для корпоративного приложения



Пользователи создают документы, по необходимости заполняют справочники – используем **объектную модель**



Потом документы исполняют – используем переходы документов между состояниями (State Entity)



При этом меняются учетные данные, которые влияют на исполнение документов и отражаются в отчетах – используем **учетную модель**

Выделение базовых абстракций

- Для разработчика логично выделение базовых абстракций
  - Справочники — объекты без workflow, но с выделением черновиков и устаревших
  - Документы — объекты с workflow, реализующие бизнес-процесс
  - Классы документов: «договор», «документ с товарной спецификацией» и т. п.
  - Обобщенная печать для любых документов
  - Обобщенная интеграция для любых объектов и т. п.
- Для бизнеса эти абстракции не существуют, он мыслит реализацию объектов как не зависящих друг от друга

☺ Базовые абстракции можно выделять как инфраструктурные объекты

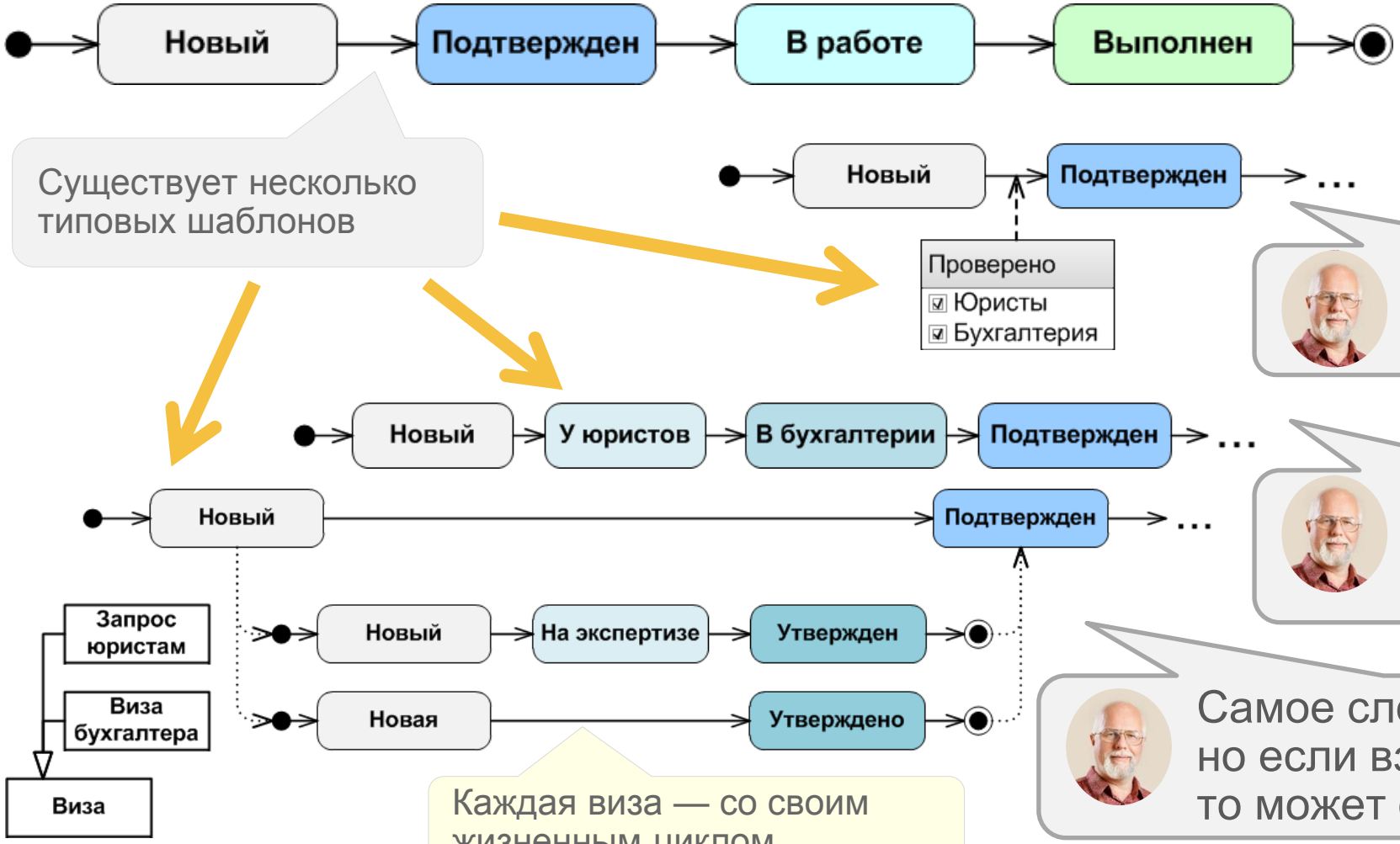
☹ Проблема: вынесенное в базовый объект ограничивает изменения

Пример: если печать ТОРГ-12 или счета-фактуры реализовать в классе «Документ с товарной спецификацией», то контексты закупок и продаж оказываются связаны в инфраструктурном объекте

Обобщения возможны, если бизнес их понимает

- Виза на документы: при обработки сделки, договора, контракта обеспечить проверку и одобрение ответственными сотрудниками или отделами
  - В банках: выдача кредитов, ипотеки и сложные сделки (лизинг, факторинг)
  - В торговле: отгрузка без предоплаты, договора с подрядчиками и т.п.
- Если кейсов много, привлекательно обобщенное решение с настройкой: требования по вариативности неясны и появляются на поздних этапах
- **Классический подход**: в требованиях аналитики формулируют задачу для конкретного документа, его проектируют под конкретную ситуацию
- **Технологические рельсы**: предложение разных шаблонов, которые затем тиражируются в рамках модулях, используя общую библиотеку

Выбор проектного решения



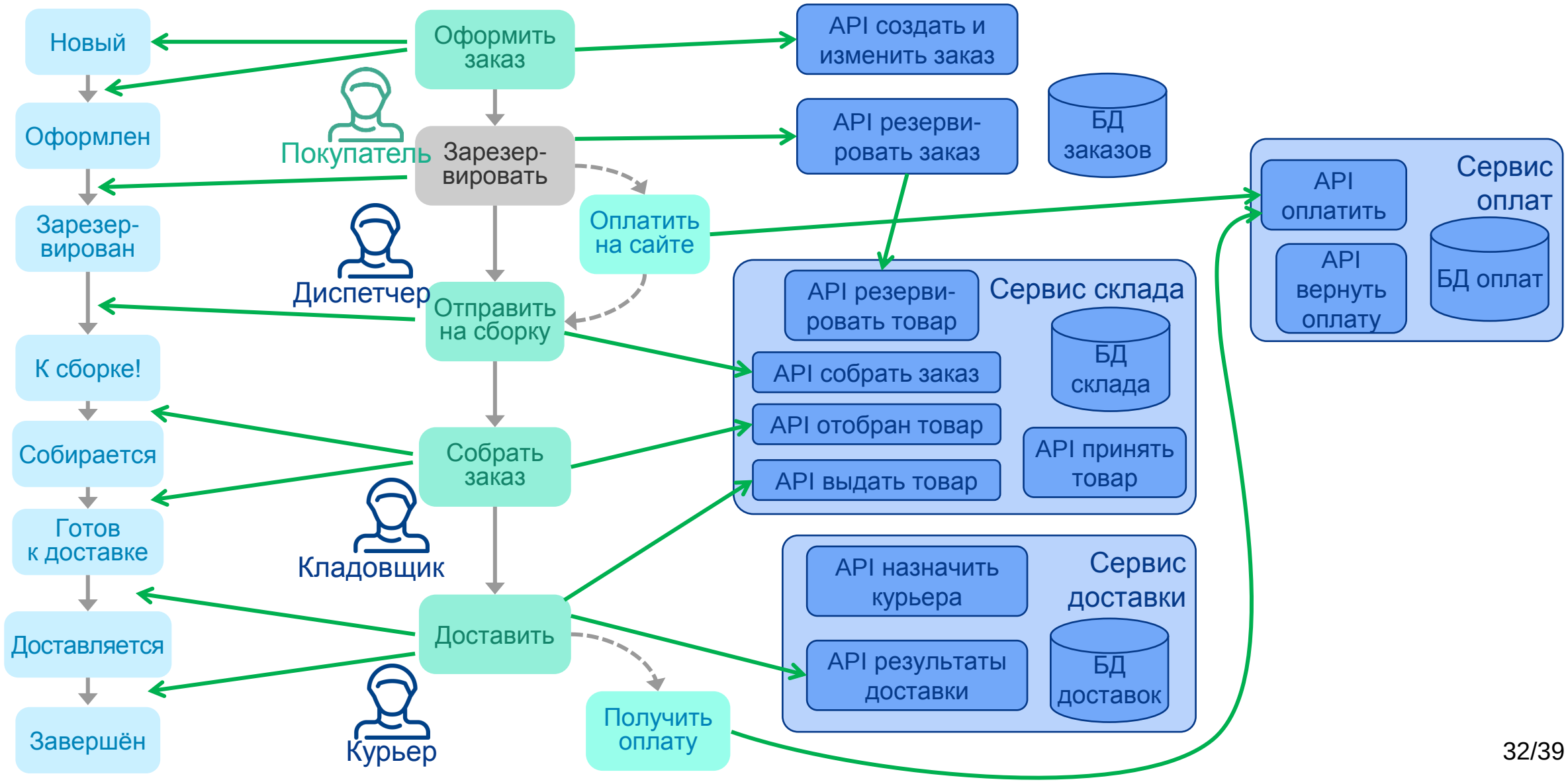
DDD в (микро)сервисной архитектуре

DDD и (микро)сервисная архитектура приложения

- Предметная область разбивается на ограниченные контексты
- Каждый контекст реализуется одним или несколькими сервисами, возможен композит — несколько сервисов и общая БД
  - Ведение покупателей может быть включено в ведение заказов или отдельно
- Не все сервисы основаны на объектной модели, примеры: учетные сервисы ведения остатков, сервисы работы с расписаниями

? Как ведем единую модель, обсуждаемую с бизнесом: для ограниченного контекста или для каждого сервиса, если его границы понятны бизнесу?

Переход от workflow к сервисам



Проектируем API вместо workflow

- Заказ передаётся между сервисами Заказов, Оплат, Склада и Доставки (хореография), или сервис Заказов оркестрирует исполнение, или гибрид: основная цепочка Заказ — Склад— Доставка, а Оплаты — вызываются?
- Транспортный объект: полная структура заказа или то, что нужно сервису? Например, цены и полный адрес со схемой проезда не нужны складу
- Транспорт должен быть устойчив к расширению возможностей систем!
- Какое API предоставляют справочники?
  - Денормализация: названия товаров храним в заказе или каждый раз запрашиваем?
  - Какие у нас сервисы ведения цен, скидок и рекламных акций? Кто вычисляет цену позиции заказа с учётом скидок и акций, учитывая что возможны комплекты и сложные условия (бесплатная доставка, для заказов больше 1000 рублей и крупных товаров)?
- Планирование доставки и её исполнение — один сервис, или два с общей базой, или два с разными базами, плюс сервис база адресов и геоданных?

Нужны не частные решения, а политика и шаблоны типовых решений!

C4 Model: декомпозиция для разработки, а не DDD

- Основные диаграммы – четыре уровня декомпозиции:
  - System context diagram – работа системы в окружении
  - Container diagram – декомпозиция системы на контейнеры, размещаемые на одной ноде
  - Component diagram – декомпозиция контейнера на сервисы
  - Code diagram – **диаграмма классов** для отдельного сервиса
- Дополнительные диаграммы
  - System landscape diagram – взаимодействие многих систем в ландшафте
  - Dynamic diagram – динамика обработки: sequence или collaboration
  - Deployment diagram – физическое размещение контейнеров

Проблемы

- Диаграммы для разработчика: **смешаны бизнес и технические аспекты, их нельзя обсуждать с бизнесом**
- Диаграммы показывают способ декомпозиции, соответствующий сервисной архитектуре, **но не отражают аспектов масштабирования**

Что изменилось в (микро)сервисной архитектуре

Микросервисы: масштабируемость и быстрые доработки приложений

- Каждый бизнес-запрос обрабатывает много сервисов, нет общей транзакции
- Много экземпляров одного сервиса для масштабирования, каждый может упасть по ошибкам или блокировкам, а система должна работать устойчиво
- Асинхронные сообщения, очереди выравнивают производительность

Отказ от единой реляцинной СУБД – она не справляется

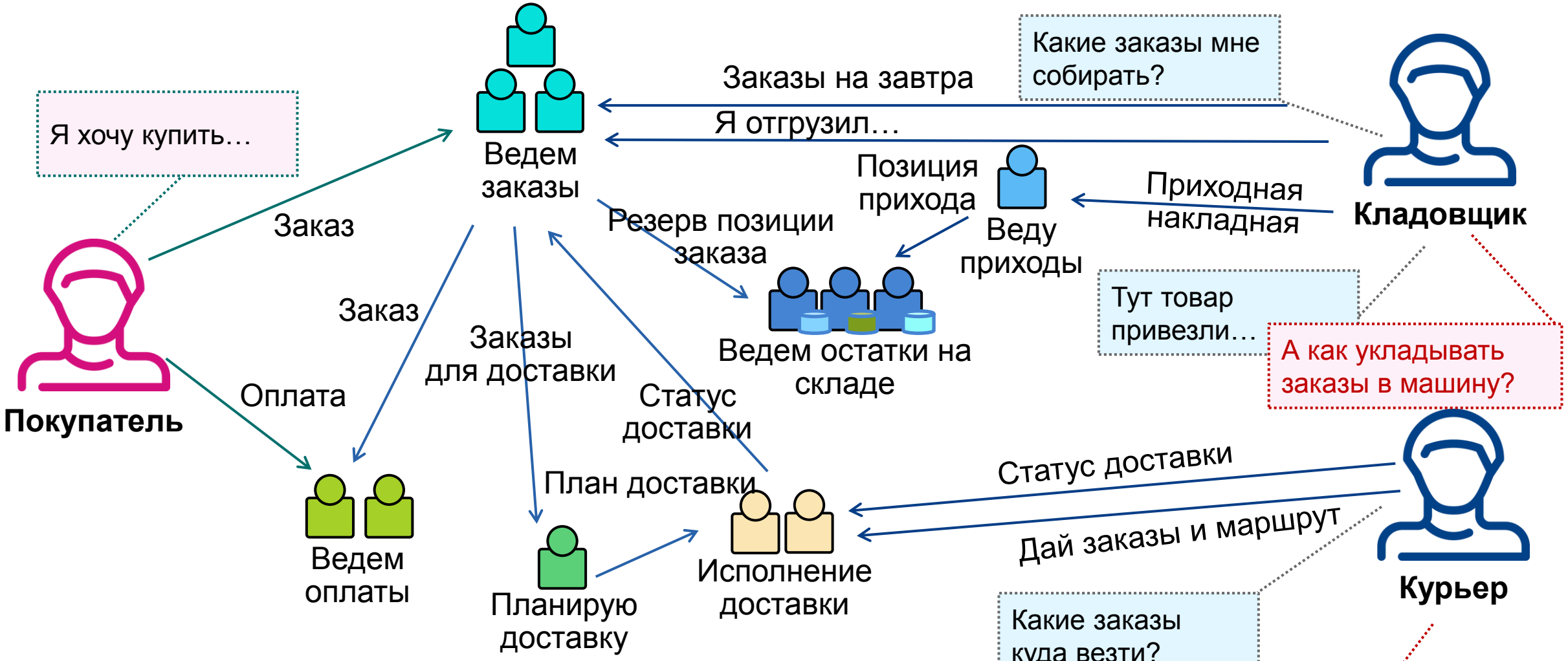
- NoSQL-базы данных, использование многих БД одним приложением, In-memory хранение в БД и очередях, отложенный сброс в хранилища
- Кластерное развертывание с независимым хранением на узлах
- Транзакционность и консистентность обеспечивает приложение, а не БД
- При восстановлении узла кластера данные неконсистентны

Нужна новая модель проектирования

- Показывать способы масштабирования для отдельных сервисов
- Отражать способы взаимодействия между множествами сервисов
- Моделировать поведение при падении экземпляров сервисов, проектировать устойчивость системы в целом
- Рассматривать восстановление при сбоях узлов кластера и дата-центров – техника и базовый софт не обеспечивают консистентного восстановления

Раньше мы проектировали системы, в которых падение одного элемента было форс-мажором.
Теперь надо проектировать системы, в которых падение элемента-сервиса – норма.
Система должна восстанавливать упавшие элементы и устойчиво работать.

Модель для сервисной архитектуры



Из выступления «Визуальное проектирование масштабируемых приложений»

Что должны обеспечить технологические рельсы?

- Как масштабируются типовые сервисы под нагрузкой
- Где используется общая БД, а где – отдельные?
- Как работаем с блокировками БД обработки запросов пользователей?
- Принципы взаимодействия сервисов: где и какие очереди, что происходит с записями в очереди, когда экземпляры сервисов и ноды падают?
- Как обеспечивается устойчивость при падении экземпляров сервисов?
- Как обеспечивается устойчивость при падении нод и дата-центров?
- Как обеспечивается работа, когда пользователь едет в «Сапсане» или в другом месте с плохой связью, соединение рвется, страницы перегружают?
- Какой мусор остается, если пользователь ушел, и как его чистят?

Что же достигает DDD?

Единый язык + единая модель:

- Надежная основа коммуникации всех участников проекта при принятии решений
- Успешно заменяют мелкую россыпь требований
- Позволяют эффективно развивать сложную систему

DDD в современной архитектуре требует более сложного отражения в код, чем для монолитов, но технологические рельсы решают задачу

Максим Цепков
http://mtsepkov.org
maks.tsepkov@ya.ru

На сайте много материалов по анализу и архитектуре, ведению проектов и agile, управлению знаниями, мои выступления, статьи и конспекты книг.