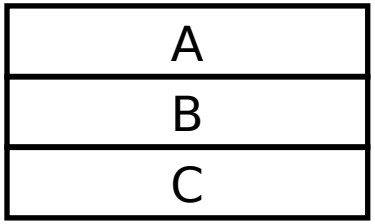
Объектная парадигма

- Ключевые идеи:
  - Программа - это совокупность объектов, способных взаимодействовать друг с другом посредством сообщений;
  - Каждый объект является экземпляром определенного класса;
  - Классы образуют иерархию наследования.
- ОО-программа – это работающая модель.
- Как правильно построить эту модель?

Многослойная архитекура

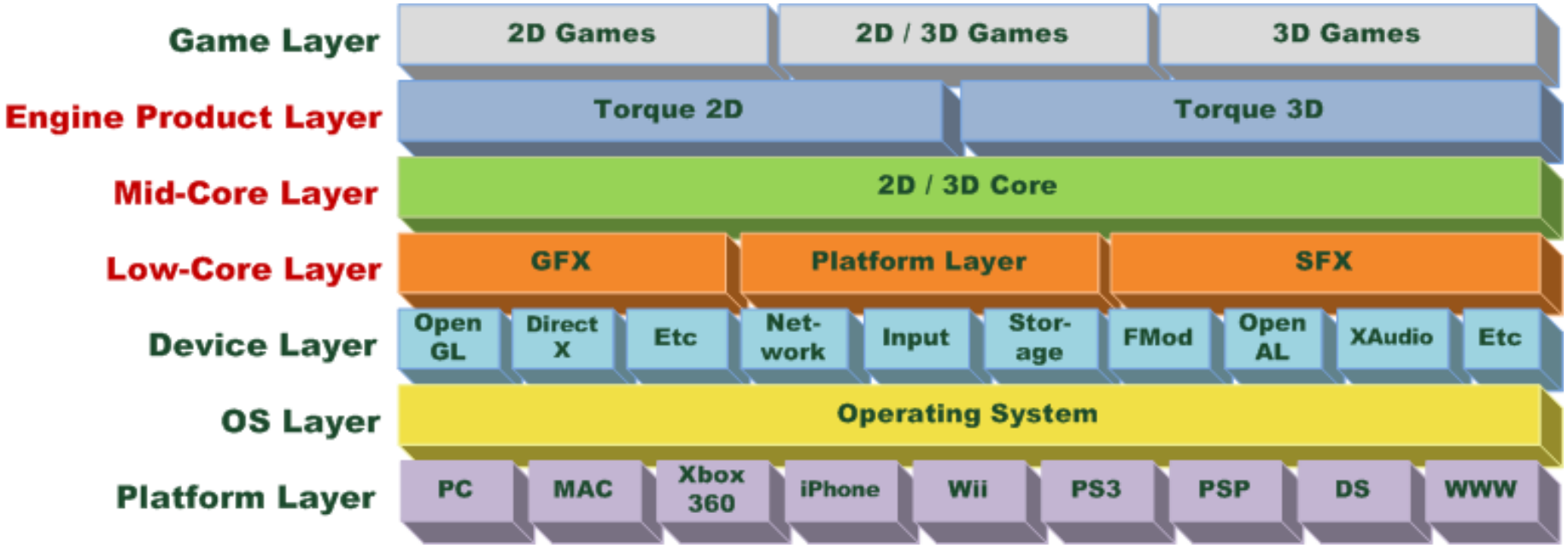
- Контекст
  - Все сложные системы испытывают необходимость разрабатывать и развивать отдельные части этих систем независимо. Для этого разработчики должны четко и однозначно понимать разделение обязанностей между модулями.
- Задача
  - ПО должно быть сегментировано так, чтобы модули могли разрабатываться и развиваться независимо, с минимальным взаимодействием между модулями, этим обеспечивая портируемость, модифицируемость и повторное использование кода.
- Решение
  - Предлагается разделить ПО на логические единицы, именуемые слоями. Каждый слой – это группа модулей, обеспечивающих взаимосвязанные сервисы.
  - На связи между слоями должны быть односторонним, т.е. если слой А использует слой Б, то слой Б не должен использовать слой А.
  - Слои не должны взаимодействовать друг с другом в обход промежуточных слоев, т.е. если слой А -> В -> С, то прямая связь А -> С – это нарушение принципа (не смертельно, но лучше не надо).

Многослойная архитекура (Layers)

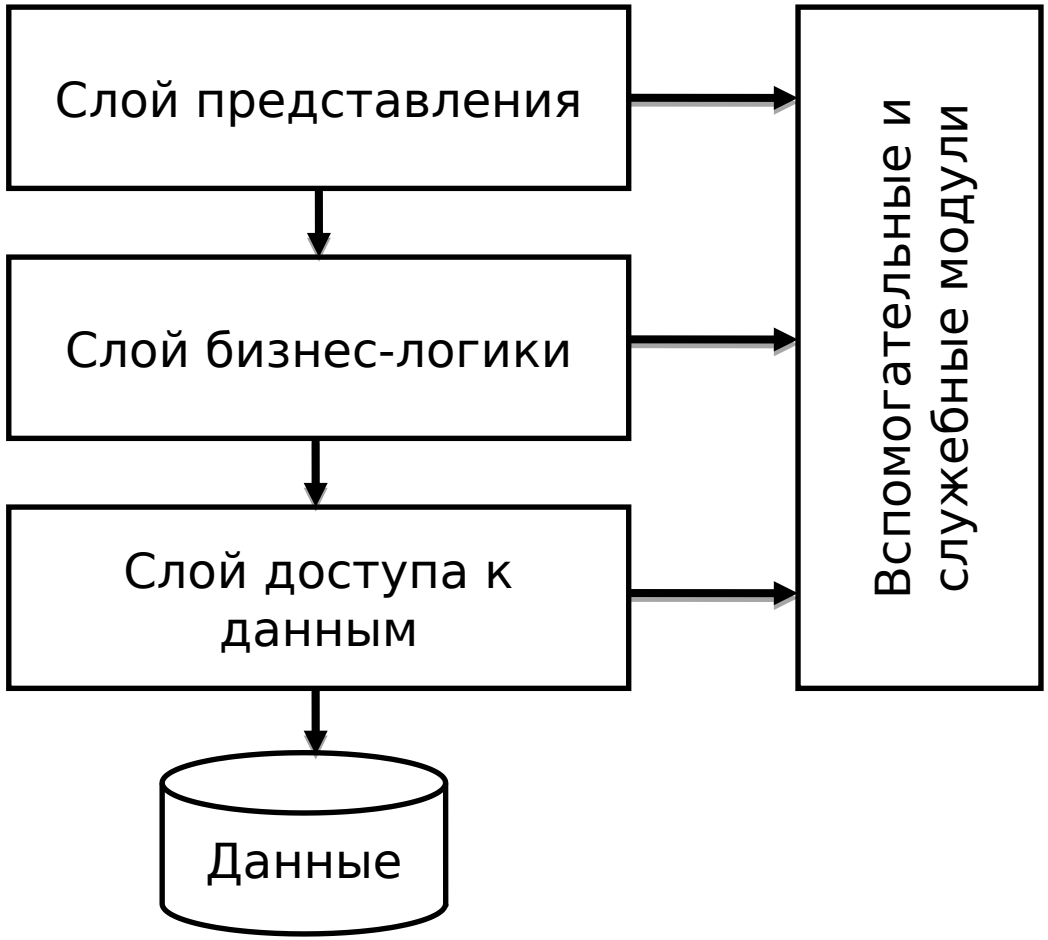


- Элементы
  - Слои обычно отображаются стеком прямоугольников
- Отношения
  - «использует», либо рисуется явно стрелками, либо подразумевается, что верхний слой использует нижний.
- Ограничения
  - Любой модуль ПО относится к какому-либо слою
  - Должно быть как минимум два слоя (обычно 3+)
  - Не должно быть циркулярных отношений (снизу вверх)
- Трудности
  - Добавление слоев несколько усложняет проектирование и систему в целом
  - Проведение вызова через слои сказывается на производительности

Многослойная архитекура (Layers)



Многослойная архитектура (Layers)



Вспоминаем ваш первый КП

- Приложение, осуществляющее работу с данными
  - Создание записей
  - Просмотр записей
  - Редактирование записей
  - Обработка по заданному вариантом алгоритму
  - Загрузка данных из файла
  - Сохранение данных в файл
- Использование процедурного подхода
  - Насколько Ваш код был далек от понятия «спагетти-кода»?

Многослойная архитектура (Layers)

Взаимодействие объектов

- Приложение (main)
  - Создает объект меню, который обрабатывает ввод пользователя
  - Передает управление этому объекту и завершается, когда тот завершает работу (выбран «Выход»).
- Меню
  - Если пользователь выбрал «Просмотр» - создается объект Просмотрщик
  - Если пользователь выбрал «Редактирование» - создается объект Редактор
  - Если пользователь выбрал «Обработка» - запускается логика обработки списка объектов из слоя БЛ.

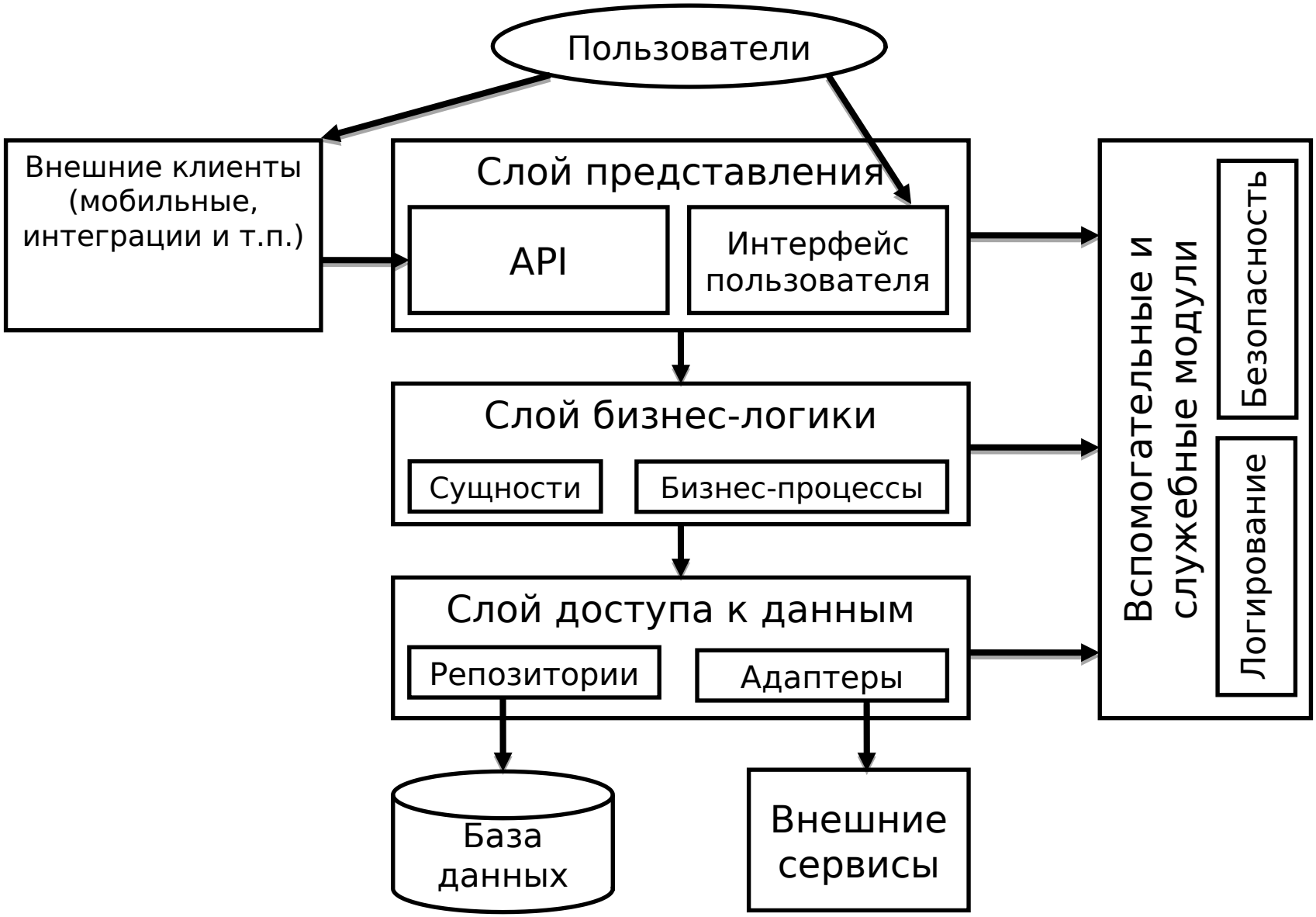
Взаимодействие объектов

- Редактор
  - Запрашивает у пользователя ID записи для редактирования
  - Вызывает объект Репозиторий слоя БЛ для загрузки Записи.
  - Отображает поля Записи и дает их отредактировать
  - Вызывает объект Репозиторий слоя БЛ для сохранения Записи.
- Просмотрщик
  - Вызывает объект Репозиторий слоя БЛ для загрузки списка Записей.
  - Отображает список записей (их ID и часть полей)

Взаимодействие объектов

- Запись
  - Хранит данные. (Анемичный объект/DTO).
- Репозиторий
  - Хранит в памяти набор Записей
  - Организует доступ на чтение и изменение
  - Загружает и сохраняет Записи в файл через Контроллер работы с файлом.
- Контроллер обработки
  - Реализует логику обработки данных в Репозитории
- Контроллер работы с файлом
  - Осуществляет файловое чтение/запись

Многослойная архитектура (Layers)



Модель и бизнес-логика

- ОО-программа представляет собой **модель** предметной области, составленную из набора объектов
  - Каждый объект призван моделировать реальную **сущность** моделируемого мира, ее свойства и поведение
  - Модель объединяет объекты и организует их взаимодействие
- Классический ОО-подход говорит о том, что данные и логика работы с ними должны быть объединены (инкапсулированы) в одном объекте
- На практике это не всегда удобно:
  - часть логики работы с данными может сильно видоизменяться в течении жизненного цикла системы
    - Вспоминаем мотивацию паттернов Стратегия, Посетитель и т.д.
  - часть логики может оказаться невозможным привязать к какой-либо моделируемой сущности (актор может оказаться вне границ моделируемой системы)
    - Вспоминаем принцип GRASP «Чистая выдумка»

Модель и бизнес-логика

- Выход – чёткое разделение логики работы с сущностью на
  - То , что сущность делает всегда и самостоятельно – это методы сущности
    - Клонирование, сравнение, валидация, вычислимые элементы состояния, сериализация/десериализация
    - Обычно тесно связаны со структурой данный сущности, в том числе приватной
  - То, что можно сделать с сущностью в рамках системы – это методы бизнес-логики системы
    - Реализация специфичных для функциональности разрабатываемой системы сценариев работы с сущностями
    - Работают только с публично доступными данными сущности
- Итого:
  - **Сущности**, формирующие **Модель**, содержат данные и самую базовую логику работы с этими данными
  - **Бизнес-логика** вынесена в отдельные **Контроллеры**, отделенные от **Модели** и способные развиваться независимо.
- Дьявол в деталях
  - DDD как один из полюсов, Анемичная модель как другой

Жизненный цикл сущности

- Типичный ЖЦ сущности:
  - Создание сущности
    - Создание объекта в памяти
    - Сохранение в персистентном хранилище
  - Использование сущности
    - Поднятие сущности из хранилища в объект
    - Участие объекта в бизнес-процессах
      - Модификация объекта
    - Сохранение измененного объекта в хранилище
  - Уничтожение сущности
    - Уничтожение данных в хранилище
- В типичном сценарии сущность постоянно существует лишь в персистентном хранилище
  - В память она обычно поднимается для участия в бизнес-процессах
  - Это не исключает возможности временного кеширования сущностей в памяти приложения

Анемичные сущности (DTO)

- **Анемичная сущность**
  - объект, содержащий данные, описывающие некую сущность предметной области, но не содержащий никакой бизнес-логики, связанной с сущностью (валидации данных, вычислений, бизнес-правил и т.п.).
  - Впервые описан Фаулером в качестве **антипаттерна** (нарушение принципов ООП).
- На практике же используется довольно широко, играя роль объекта, передающего данные о сущности между модулями приложения.
  - **DTO** – Data Transfer Object. Легкость передачи.
  - Стандарт именования – **ИмяСущности**.
    - Student, Account, GameObject, etc.

Анемичные сущности (DTO)

Анемичная сущность:

```
Class Student
{
   public int Number { get; set; }
   public string Name { get; set; }
}
```

Полноценная сущность

```
class Student
{
   public int Number { get; set; }
   public string Name { get; set; }

   public Student(int number , string name, Date dob)
   {
      if (number <= 0) {throw new ArgumentOutOfRangeException(nameof(number));
}
      if (name <= “”) { throw new ArgumentOutOfRangeException(nameof(name)); }
      Name = name;
      Number = number;
   }

   public override string ToString(){ return Number.ToString()+Name;  }
}
```

Шаблон Контроллер (из GRASP)

- Проблема
  - Кто должен отвечать за получение и координацию выполнения системных операций, поступающих от уровня интерфейса пользователя?
- Решение
  - Присвоить эту обязанность классу, удовлетворяющему одному из следующих условий.
    - Класс представляет всю систему в целом, корневой объект, устройство или важную подсистему (фасадный контроллер).
    - Класс представляет сценарий некоторого варианта использования, в рамках которого выполняется обработка этой системной операции (контроллер варианта использования или контроллер сеанса).

Контроллеры в приложении

- Слой бизнес-логики обычно формируется как набор контроллеров, реализующих сценарии бизнес-логики приложения
  - Стандарт именования контроллеров – **ИмяСущностиController** или **ИмяСущностиManager**
- Контроллер содержит набор методов для работы с сущностью, это могут быть
  - базовые CRUD-операции
  - реализация бизнес-процесса в целом
- Методы контроллеров используют в качестве параметров и возвращаемых значений **сущности** (или контейнеры, содержащие **сущности**)

Шаблон CRUD

- Контекст
  - Любое приложение в конечном итоге работает с какими-либо данными. Данные обычно моделируют какие либо сущности реального мира.
- Задача
  - Как организовать простой, понятный и непротиворечивый интерфейс (как пользовательский, так и межкомпонентный) для работы с информационными моделями сущностей так, чтобы на его основе можно было собрать реализацию всех необходимых сценариев?
- Решение
  - CRUD – сокращение от CREATE, READ, UPDATE, DELETE
  - Набор базовый операций над сущностями, который должна реализовывать система и становится интерфейсом этой системы.

CRUD

- Элементы
  - Сущность – объект, моделирующий какую-либо сущность предметной области
  - Контроллер/менеджер – объект, реализующий набор CRUD-операций над сущностью
- Ограничения
  - Бизнес-логика и используемые протоколы могут накладывать ограничения на набор операций (например, может быть запрещено или нереализуемо удаление сущностей)
- Трудности
  - Не всегда оправдано выставление наружу полного набора сущностей и примитивных операций над ними

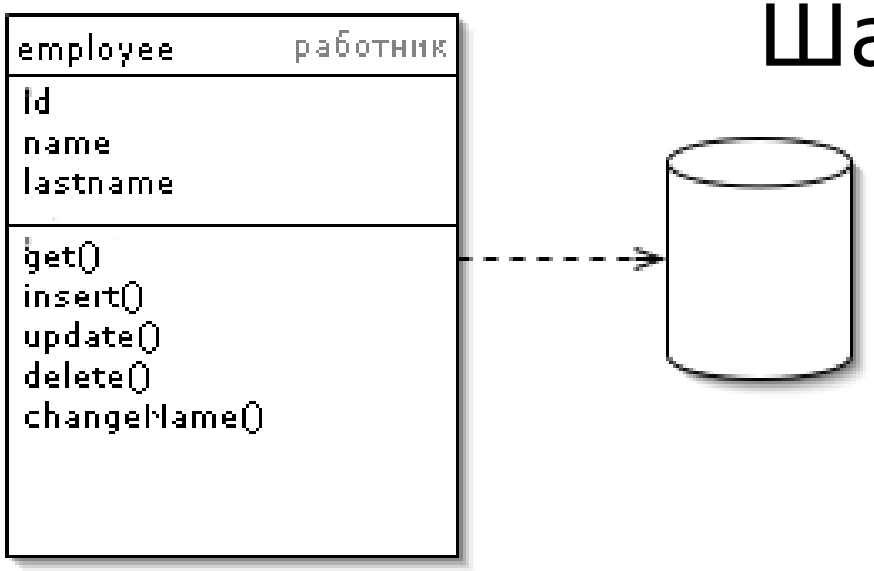
Пример контроллера

- class StudentController
  - CRUD-операции
    - void Create(Student s);
    - Student Read(int number);
    - void Update(Student s);
    - void Delete(Student s);
  - Сценарии бизнес-логики
    - void Expel(Student s, string reason);
    - void Expel(int studNumber, string reason);
    - void Enroll(Student s, Group g);
    - void Enroll(int studNumber, int groupNumber);

Слой доступа к данным

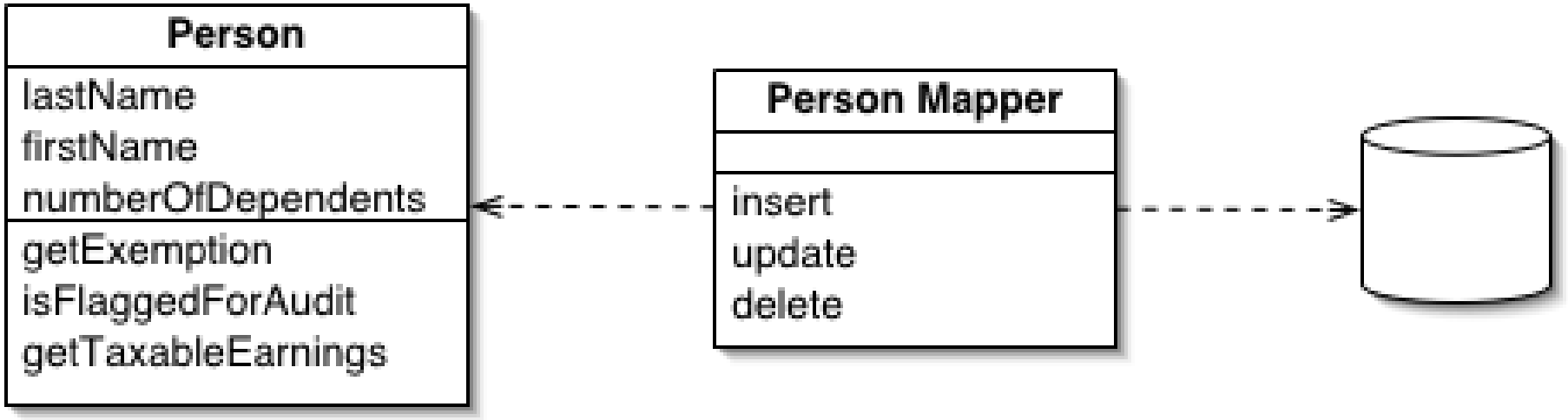
- Итак, мы уже выяснили, что между слоями приложения данные передаются в виде сущностей и хранятся в контейнерах, но постоянно хранятся в персистентном (постоянном) хранилище.
  - Обычно этого какого-то рода БД (SQL, NoSQL, какая угодно)
- Для работы с хранилищем существует слой доступа к данным и различные шаблоны доступа к данным. Шаблонов таких довольно много, но на практике чаще всего используются 2.5:
  - **Active Record** – популярный, но спорный шаблон, нарушающий изоляцию слоев и другие принципы проектирования
  - **Data Mapper** – классический шаблон связи с хранилищем
  - **Repository** – усложненный вариант Data Mapper
    - Чаще всего используется некий гибрид последних двух, гордо носящий имя Репозитория. Отсюда и 2.5 шаблона ϑ

Шаблон Active Record



- Один объект управляет и данными, и поведением.
  - Большинство данных объекта постоянны и их надо хранить в БД.
  - Этот паттерн использует наиболее примитивный подход – реализацию логики доступа к данным в объекте сущности.
- Объект является "обёрткой" одной строки из БД и включает в себя доступ к БД и логику обращения с данными.
  - Пример: сущность "Студент" может содержать как данные о студенте, так и методы CRUD, работающие напрямую с БД
- Некоторыми рассматривается как **антипаттерн**
  - Нарушает SRP (Принцип единой ответственности из SOLID)
  - Мешает тестированию кода
  - В целом – применим только в простых системах типа «форма для отображения/редактирования данных в БД», где пользуется популярностью.

Шаблон Data Mapper

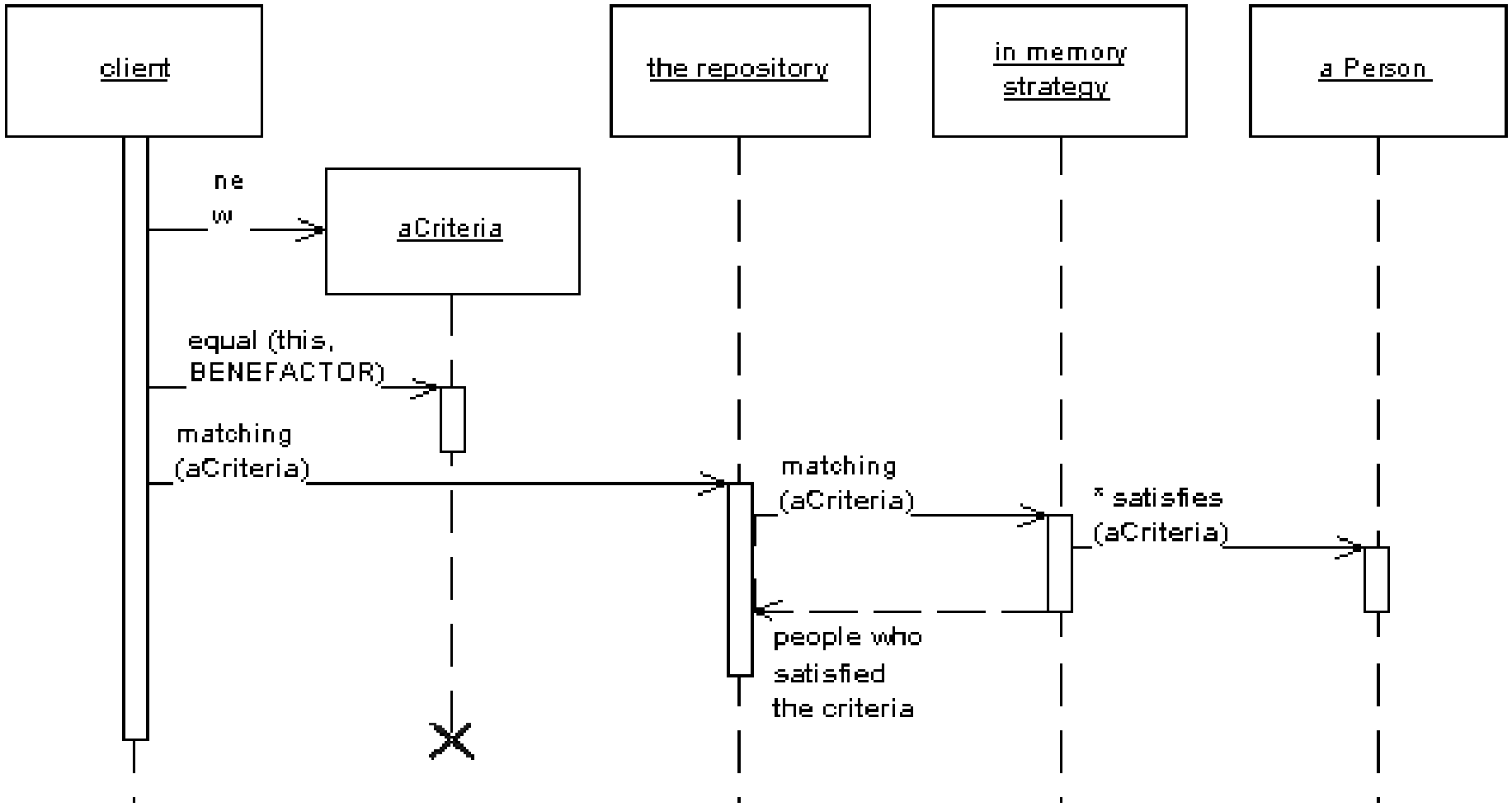


- Объектные и реляционные БД используют разные способы структурирования данных.
  - Указатели, контейнеры, наследование – моделируются в БД сложнее
- Так что объектная и реляционная схемы не идентичны, тем не менее, необходимость в обмене данными между ними – необходима.
  - Если же объект знает о реляционной структуре — изменения в одной из структур приведёт к проблемам в другой.
- Data Mapper — это программная прослойка, разделяющая объект и БД.
  - Его обязанность — пересылать данные между ними и изолировать их друг от друга.
  - При использовании Data Mapper'а объекты не нуждаются в знании о существовании БД. Они не нуждаются в SQL-коде, и (естественно) в информации о структуре БД.
  - Сам объект-Mapper неизвестен объекту.

Шаблон Repository

- Репозиторий – Data Mapper «на стероидах»
  - Добавляет слой абстракции, экспонирующий интерфейс обычного контейнера, реализация которого работает с хранилищем
    - Объект помещается внешним клиентом в репозиторий, как в контейнер, а Репозиторий записывает его в БД
    - Аналогично происходит чтение, итерирование, удаление и т.п.
  - Репозиторий может реализовывать дополнительное кеширование данных в памяти
  - Репозиторий может получать на вход сложное декларативное описание критериев поиска объектов, конструировать по нему сложные запросы к хранилищу, и исполнять их, возвращая подходящие данные.

Шаблон Repository



Слой доступа к данным

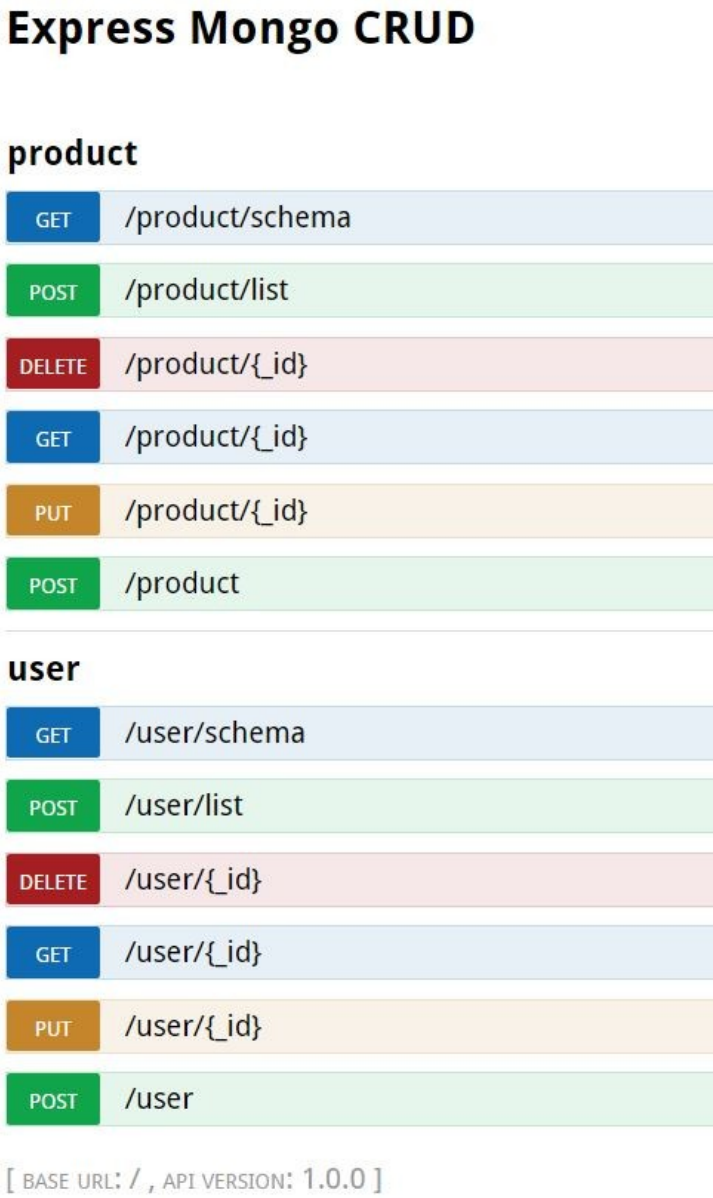
- Обычно используются шаблоны Data Mapper или Репозиторий (той или иной сложности)
  - class StudentRepository
    - Базовые CRUD-операции
      - void Create(Student s);
      - Student Read(int number);
      - void Update(Student s);
      - void Delete(Student s);
    - Более сложные операции
      - List<Student> FindByName(string name);
      - List<Student> FindBySearchCriteria(List<Criterion> criteria);
- Есть готовые решения – библиотеки ORM (Object-Relational Mapping)
  - Entity Framework (EF) для .NET
  - Hibernate/Nhibernate для Java/.NET
  - И другие (их немало)

Слой представления

- Задача слоя представления – представить данные модели в удобном для клиента виде
- Клиент может быть разный:
  - Человек – для него строится UI (User Interface)
    - Диалоговый (CUI, боты)
    - Графический (GUI)
    - Простой Веб-интерфейс (server-side WebUI)
  - Другие системы – для них строится API (Applicative Programming Interface)
    - Интеграция с внешними системами
    - Мобильные приложения, котоыре уже взаимодействуют с пользователем
    - Продвинутый веб-интерфейс (SPA WebUI)
- Принцип действия же слоя представления остается тем же самым:
  - передавать данные модели клиенту, а ввод клиента – модели.

CRUD и REST API

- Часто CRUD-подход используется в рамках REST API
- REST = REpresentational State Transfer
  - Архитектурный стиль построения веб-сервисов на базе протокола HTTP
  - Объект (или контейнер, набор объектов), с которым производятся действия, идентифицируется через URL HTTP-запроса.
  - CRUD-методы реализуются в виде методов протокола HTTP
    - POST = CREATE
    - GET = READ
    - PUT/PATCH = UPDATE
    - DELETE = DELETE
  - Сам объект при передаче может быть представлен как угодно (обычно – JSON, XML или другие протоколы сериализации)



Диалоговый интерфейс пользователя

- Характерен для консольного интерфейса или ботов
- Основное действующее лицо – контроллер диалога
  - Может быть несколько, управление между которым передается в результате команд пользователя, например
    - контроллер главного меню,
    - контроллер просмотра списка студентов
    - Контроллер редактирования студента
- Принцип действия контроллера – в цикле, до получения команды на окончание работы (передачу управления):
  - Задает вопрос о следующем действии пользователю
  - Распознает его ответ и вызывает соответствующий метод контроллера бизнес-логики (который, в свою очередь, вызывает слой доступа к данным), возвращающий сущность(-ти).
  - Выдает результат пользователю в удобном для пользователя виде
  - Выдает пользователю список дальнейших доступных команд

Графический интерфейс пользователя

- Характерен для десктопных и мобильных приложений
  - Есть и веб-реализации, не слишком удачные (WebForms)
- Основное действующее лицо – элемент интерфейса.
  - примитив графического интерфейса пользователя, имеющий стандартный внешний вид и выполняющий стандартные действия.
  - Другие названия: элемент управления, виджет (widget), контрол (control)
  - Каждый **тип элемента управления** (форма, поле ввода, кнопка и т.п.) моделируется отдельным **классом** (Form, TextBox, Button etc.)
    - Реализует функциональность отрисовки и приема ввода от пользователя.
    - Обычно используются стандартные библиотечные реализации
  - Каждый конкретный **экземпляр элемента управления** (например, форма редактирования студента, поле ввода имени и кнопка «OK») моделируются отдельным экземпляром **объекта**.
    - Реализует хранение специфичных данных экземпляра элемента управления – расположение, размер, состояние и т.п.
  - *«Если класс один на все кнопки, как мне реализовать индивидуальную реакцию на нажатие каждой кнопки?»*

Обработка ввода пользователя

- Чтобы развязать общий класс и конкретное поведение каждого его экземпляра используются либо шаблон «Наблюдатель» (Observer)
  - Класс элемента управления при получении действия от пользователя генерирует соответствующее событие
  - При создании каждого экземпляра элемента управления на нужное событие подписывается индивидуальный обработчик, реализующий логику реакции на событие
- Либо шаблон «Шаблонный метод» (Template Method)
  - В методе базового , библиотечного класса элемента управления, обрабатывающем действие пользователя, предусматривается вызов метода-зацепки (hook, хук), имеющего в этом классе пустую реализацию
  - Экземпляр класса элемента управления создается не от базового, библиотечного класса (*Form*), а от производного, пользовательского класса (*Form1*), который переопределяет нужный метод-зацепку так, чтобы реализовывать в нем нужную логику реакции на действие пользователя

Функционирование элемента управления

- При создании экземпляра элемента управления:
  - В поля записывается его начальное состояние
  - Определяется логика реакции на действие пользователя
    - через подписку на событие или метод-зацепку
  - Вызывается отрисовка
    - Проходя по всему дереву дочерних элемента управления
  - При действии пользователя срабатывает логика его обработки, обычно включающая в себя:
    - Вызов соответствующего контроллера бизнес-логики с передачей ему данных введенных пользователем в этот или другие элементы управления
      - Контроллер в свою очередь, вызывает слой доступа к данным), возвращающий сущность(-ти) из хранилища
    - Получение сущности(-тей) от контроллера и запись их свойств в соответствующие свойства себя или других элементов управления.
    - Изменение свойств элементов управления вызывает их перерисовку.

Собираем вместе

//слой представления

StudentEditor.ButtonFind.OnClick()

```
string name = StudentEditor.TextBotStudName.Text;
Student s = StudentController.FindFistByName(name);
```

//слой бизнес-логики

```
return StudentRepository.FindFistByName(name).FirstOrDefault;
```

//слой доступа к данным
// используя нужную библиотеку работы с БД выполнить

```
SELECT Number, Name from STUDENT
WHERE Name = %name%
```

StudentEditor.TextBotStudName.Text = s.Name;

StudentEditor.TextBotStudNumber.Text = s.Number;

Проблемы классической модели элементов управления

- Нарушение SRP (принцип единой ответственности SOLID)
  - элемент управления занимается как взаимодействием с пользователем (отрисовка данных и обработка ввода пользователя), так и реализацией логики работы интерфейса (реакция на конкретное действие над конкретным элементом управления, специфичное для данной системы)
- Сильное зацепление
  - В силу необходимости отрисовки элемент управления должен знать специфичные для этого данные – координаты отрисовки, размеры родительского и дочерних элементов, режимы относительного выравнивания, отступы, шрифтовую и цветовую схемы и т.п.
  - В силу необходимости реализации конкретного действия, элемент управления должен знать об объектах нижележащих слоев (БЛ, сущности).
- Следствие
  - Затрудняется модификация кодовой базы – изменения в визуальном расположении элементов управления затрагивают файлы с классами и наоборот.
  - Затрудняется тестирование кода – логика жестко привязана к элементам пользовательского интерфейса и фреймворку обработки действий пользователя.

MVC (Model-View-Controller, Модель-Представление-Контроллер)

- Контекст
  - UI – наиболее часто меняющаяся часть интерактивного ПО, поэтому необходимо обеспечить возможность изменения UI затрагивания других компонентов системы.
  - Пользователи часто хотят рассматривать одни и те же данные с разных перспектив (таблицы, графики, и т.п.), и такие разные представления должны отражать текущее состояние данных
- Задача
  - Как отделить UI от функциональности приложения, при этом обеспечив его реакцию как на действия пользователя, так и на внешние изменения данных?
  - Как обеспечить тестируемость логики сценариев работы пользователя?
- Решение
  - Паттерн MVC делит функциональность приложения на 3 компонента:
    - Модель, содержащая данные приложения
    - Представление (View), которое отображает пользователю нужный аспект данных модели, и взаимодействующее с пользователем
    - Контроллер, который является посредником между Моделью и Представлением и управляет уведомлениями об изменениях состояния

MVC

- Элементы
  - Модель – представление данных или состояния приложения, непосредственно содержит бизнес-логику или предоставляет интерфейсы для использования себя ею.
  - Представление (View) – UI-компонент, отображает данные пользователю и организует возможность приема ввода от пользователя.
  - Контроллер – управляет сценариями взаимодействия между Моделью и Представлением, переводя действия пользователя в Представлении в операции над Моделью и обновляя представление, когда это необходимо.
- Отношения
  - Уведомление – все компоненты обмениваются уведомлениями об изменениях своего состояния
- Ограничения
  - Должны существовать не менее 1 экземпляра каждого компонента
  - Модель не должна инициировать взаимодействие с контроллером
- Трудности
  - Может вносить лишнюю сложность для простых интерфейсов
  - Не всегда однозначно и чисто реализуется фреймворками.

Классический MVC

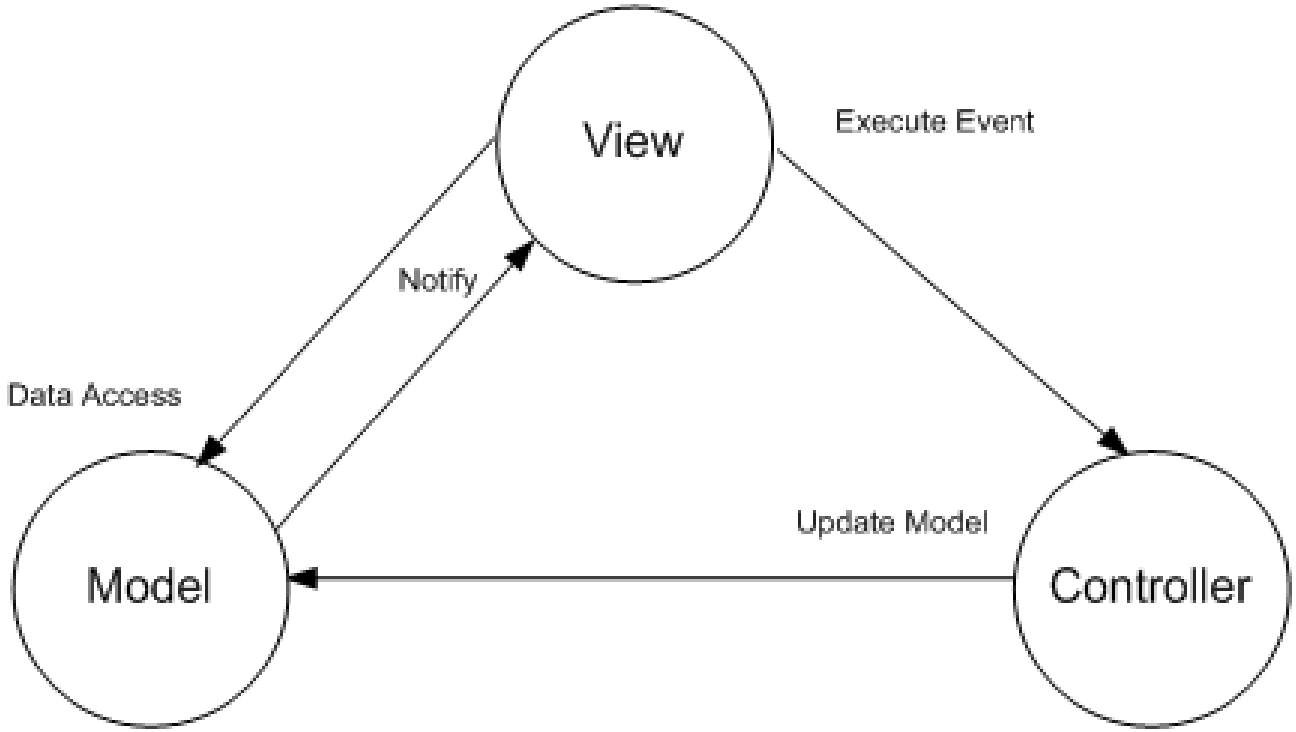


Figure 4: Simplistic MVC

Классический MVC

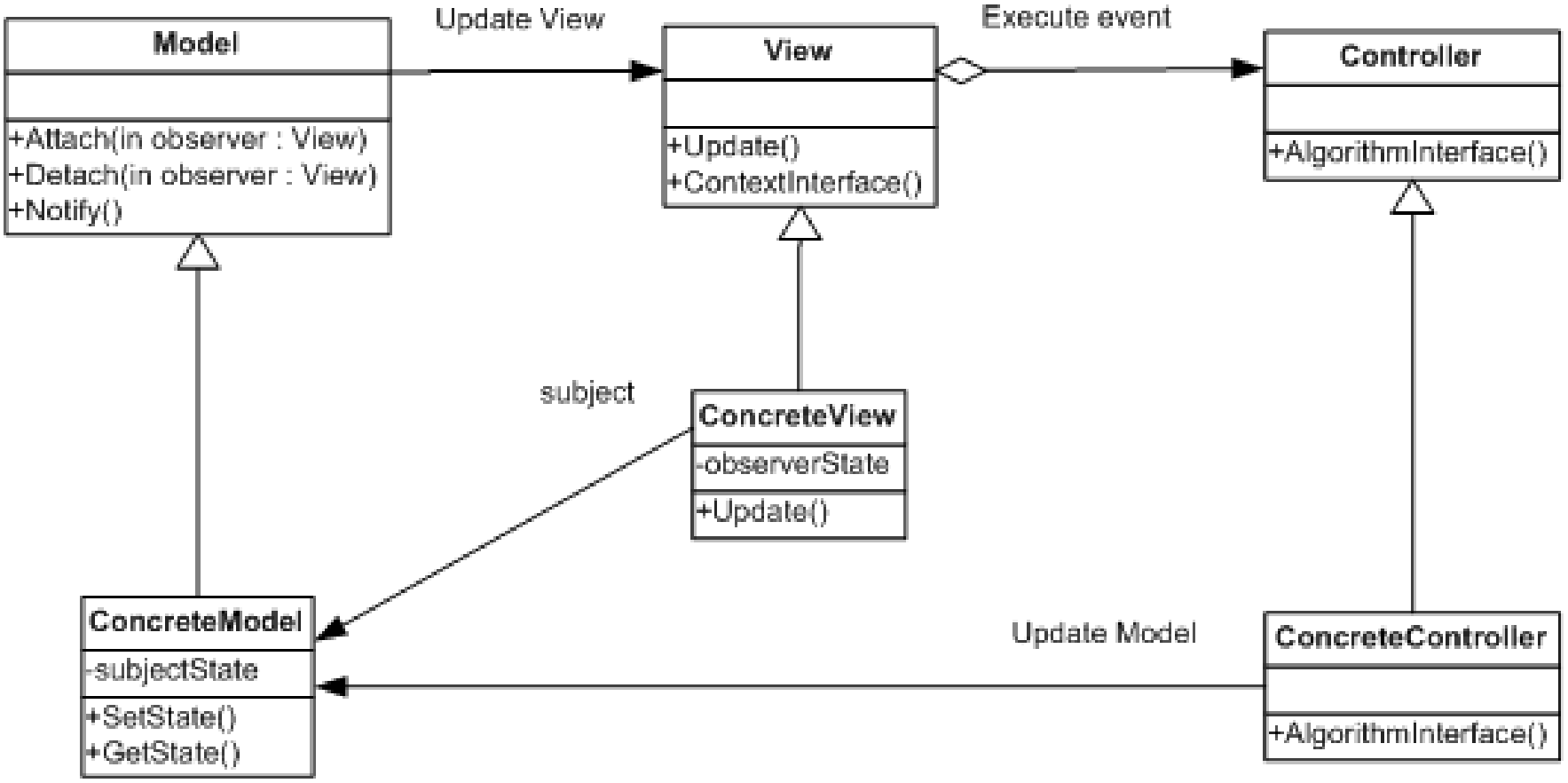
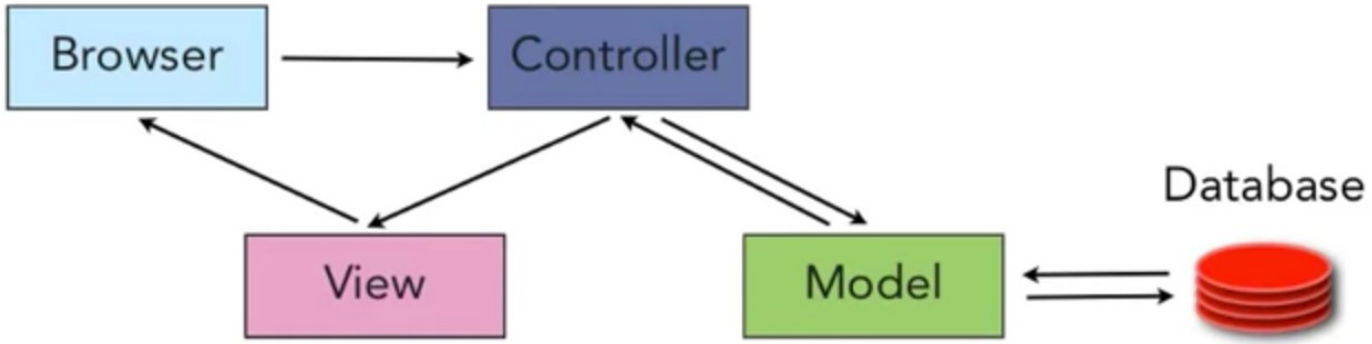


Figure 3: MVC

MVC в вебе

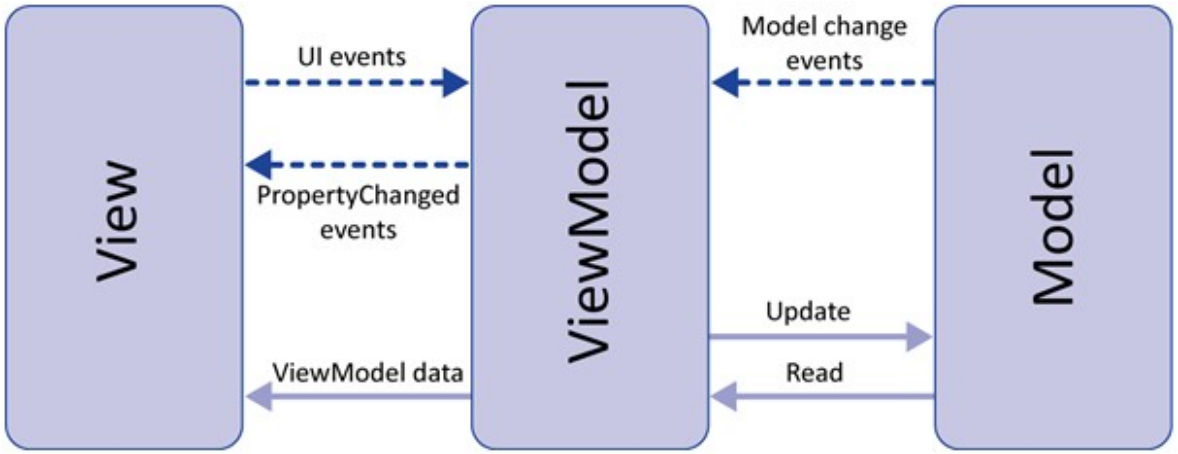


- Модель
  - Работает на стороне сервера
  - Хранит и обрабатывает объекты предметной области
- Контроллер:
  - Работает на стороне сервера
  - Генерирует веб-страницу Представления, используя данные из Модели
  - Обрабатывает HTTP- запросы со страницы Представления, инициируя изменения в модели
- Представление:
  - Работает на стороне клиента
  - Отображает данные, выданные контроллером и обрабатывает пользовательский ввод, отсылая HTTP-запросы на контроллер.

MVVM – (Model-View-ViewModel)
Модель-Преставление-МодельПредставления

- Исходный вопрос – как и где хранить состояние Представления?
  - Вышеописанные проблемы классической модели, объединяющей в одном классе данные о визуальном образе элемента интерфейса и его состояние/поведение.
- Новая декомпозиция:
  - Представление не содержит кода вообще и строится дизайнером, а не разработчиком.
  - Новая сущность **Модель Представления** хранит данные состояния Представления.
    - **Привязка** визуальных элементов Представления к этим данным осуществляется фреймворком.
    - Тут же определяются **команды**, меняющие данные в Модели.

MVVM



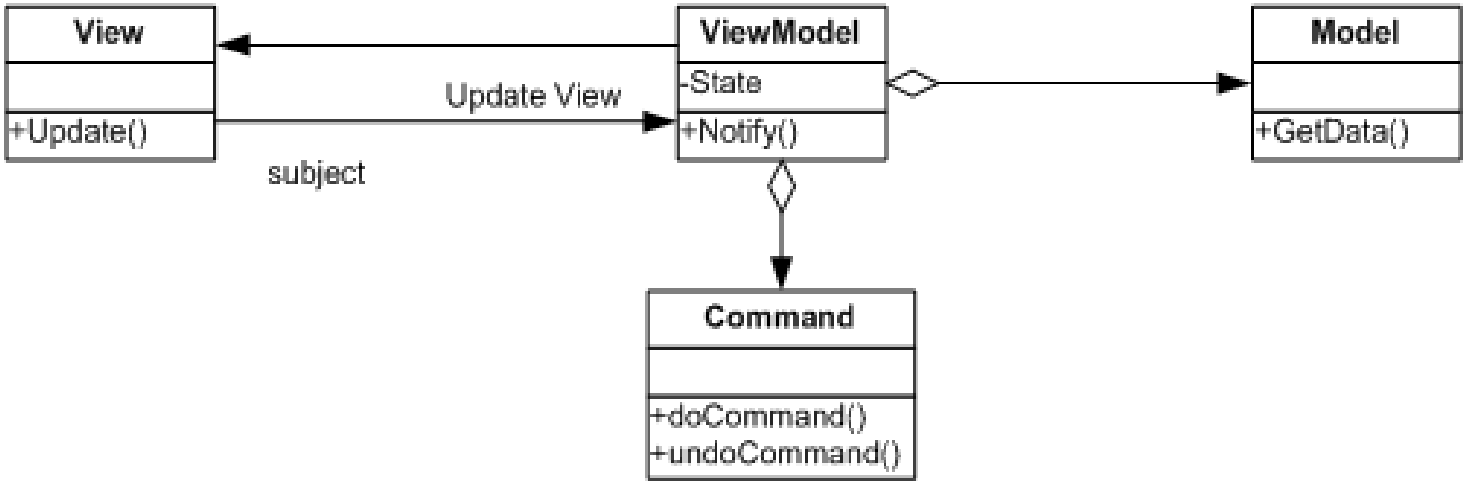
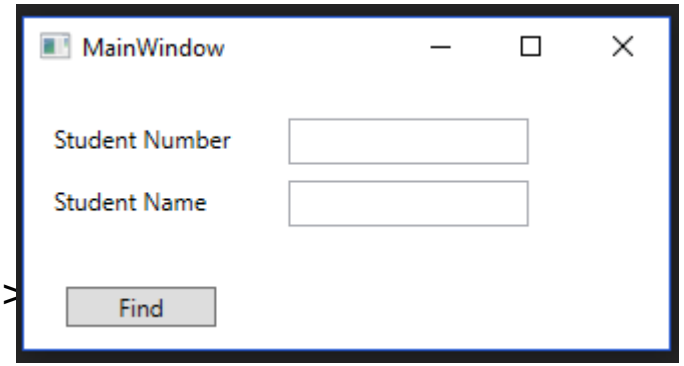


Figure 8: MVVM

Представление WPF (MVVM)




```
<Window x:Class="WpfApp1.MainWindow"
        //неймспейсы убраны для краткости
        Title="MainWindow" Height="176.559" Width="286.62">
    <Grid>
        <TextBox x:Name="tbNumber" Text="{Binding Path=Number}"
HorizontalAlignment="Left" Height="23" Margin="127,20,0,0" TextWrapping="Wrap"
VerticalAlignment="Top" Width="120"/>
        <Label Content="Student Number" HorizontalAlignment="Left" Margin="10,17,0,0"
VerticalAlignment="Top" Grid.ColumnSpan="2"/>
        <Label Content="Student Name" HorizontalAlignment="Left" Margin="10,48,0,0"
VerticalAlignment="Top" Grid.ColumnSpan="2"/>
        <TextBox x:Name="tbName" Text="{Binding Path=Name}"
HorizontalAlignment="Left" Height="23" Margin="127,59,0,0" TextWrapping="Wrap"
VerticalAlignment="Top" Width="120"/>
        <Button x:Name="btnFind" Command="{Binding Path=FindStudent}"
Content="Find" HorizontalAlignment="Left" Margin="21,104,0,0" VerticalAlignment="Top"
Width="75" Grid.ColumnSpan="2" />


    </Grid>
</Window>
```

Модель представления WPF

```
class StudentViewModel : INotifyPropertyChanged
  {
     public string Name { get; set; }
     public int Number { get; set; }

     private readonly DelegateCommand findCommand;
     public ICommand FindStudent => findCommand;
     public StudentViewModel()  { findCommand = new
DelegateCommand(SearchByNumber);}

     private void SearchByNumber(object commandParameter)
     {
        Name = "Student#" + Number; //здесь мог быть вызов Контроллера
        OnPropertyChanged(nameof(Name));
     }
     public event PropertyChangedEventHandler PropertyChanged;
     public void OnPropertyChanged(string prName) =>
  PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(prName));
  }
```

Практическое применение MVC/MVVM

- MVC
  - Начинался в эру десктопов (Smalltalk), особой популярности не получил из-за трудности стыковки с имевшимися фреймворками/библиотеками виджетов
  - «Выстрелил» в 2000х годах в привязке к Вебу – где эта архитектура очень красиво легла на HTTP-протокол (stateless)
- MVVM
  - Начинался в десктопе, и продолжает там существовать (WPF)
  - Получил «второе дыхание» в 2010х в привязке к сложным веб-приложениям с развитой логикой на клиентской стороне (SPA, Single-Page Applications)

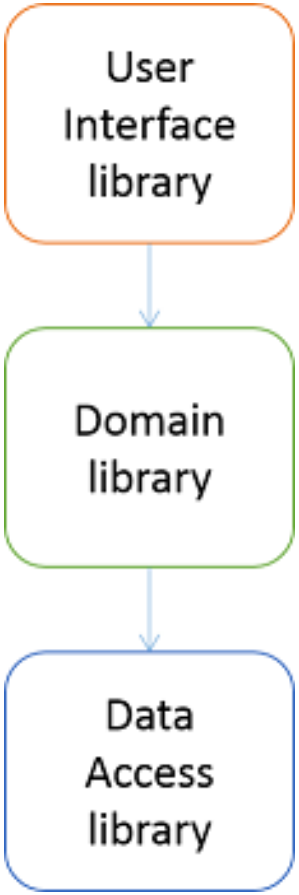
Классическая трехуровневая архитектура - резюме

- Модель предметной области, состоящая из набора сущностей (DTO), хранящихся в контейнерах в памяти – проходит через все слои.
  - Слой представления – реализует интерфейс системы, позволяет пользователю (человеку или другой системе) получать информацию о состоянии модели и отдавать команды на модификацию этого состояния контроллерам бизнес-логики.
  - Слой бизнес-логики – содержит контроллеры, по команде от слоя представления модифицирующие состояние модели.
  - Слой логики доступа к данным – обеспечивает персистентность сущностей (загрузку/сохранение в постоянное хранилище)

Классическая трехуровневая архитектура - сценарий

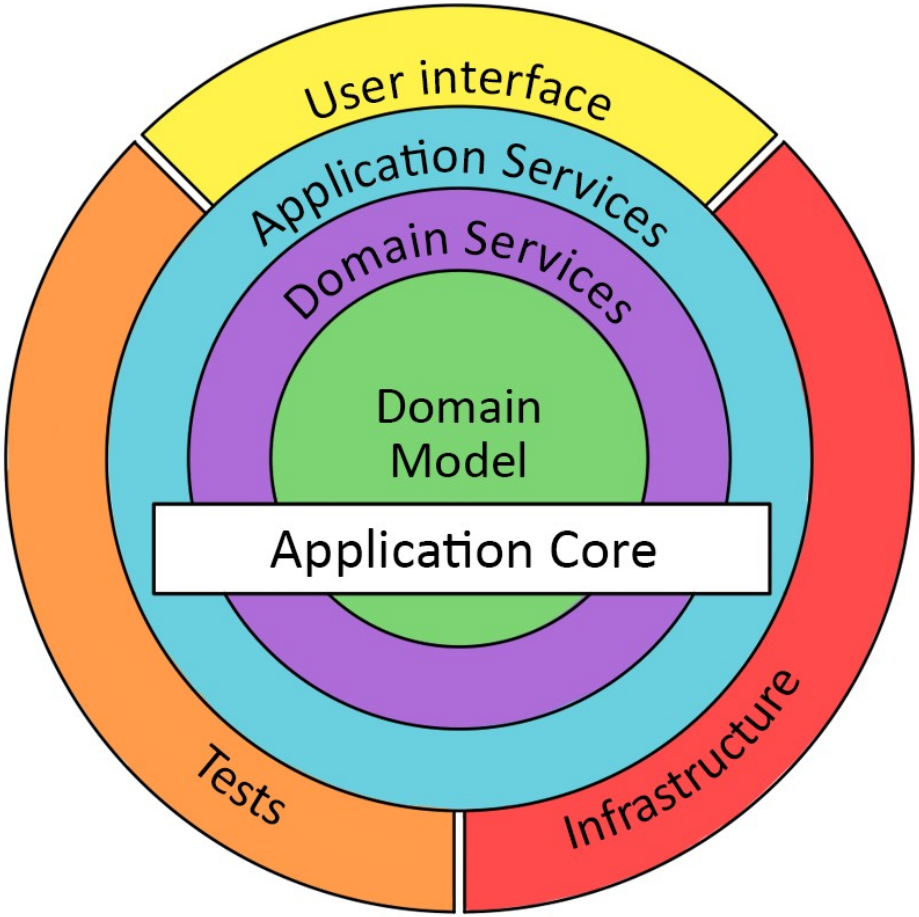
- Классическая архитектура подразумевает работу с сущностью как единым целым в CRUD-стиле
- Типовой сценарий от лица слоя бизнес-логики:
  - Получить от пользователя через слой представления команду на загрузку сущности
  - Найди и загрузить ее из базы через слой доступа к данным
  - Поместить ее во временное хранилище в модели (контейнер с DTO)
  - Передать на слой представления для выдачи пользователю («отрисовки»)
  - Получить от пользователя через слой представления новые (отредактированные) данные сущности, изменить модель в соответствии с ними
  - Получить от пользователя через слой представления команду сохранения изменений в постоянном хранилище
  - Передать измененную сущность слою доступа к данным для сохранения

Развитие слоистой архитектуры



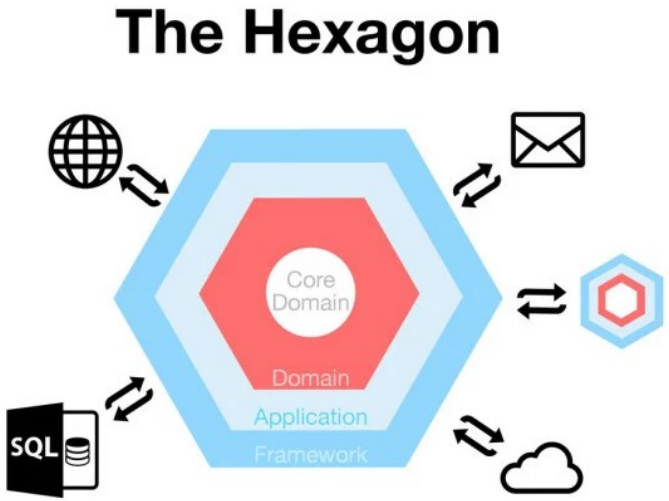
- Данная архитектура **датацентрична**
  - В основе – единое хранилище данных
  - Все слои зависят от структуры данных
  - Наследие ранней **клиент-серверной** архитектуры, где сервер БД, зачастую, был единственной точкой объединения системы («толстые клиенты»)
- Для современных систем более характерен **доменоцентричный** подход
  - В основе – доменная модель (модель предметной области).
  - Все слои зависят от модели, в т.ч. хранение данных – такой же обслуживающий процесс как и их визуализация.
  - Характерная черта **микросервисной** архитектуры – отдельные хранилища для каждого микросервиса.
- Принцип инверсии зависимостей SOLID
  - Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций

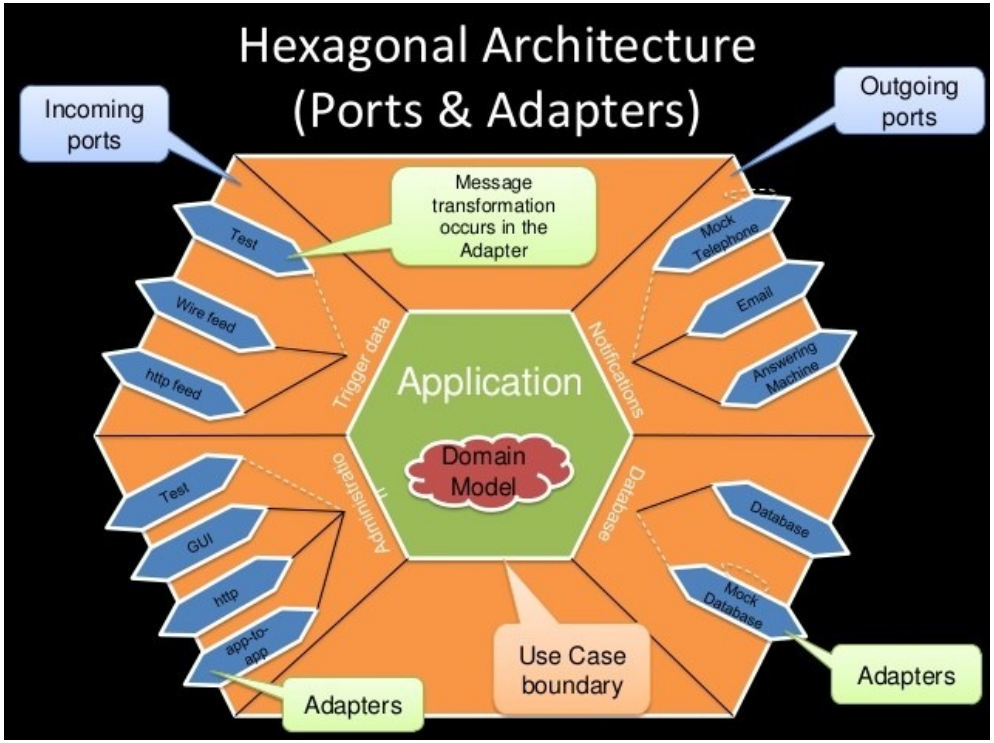
Луковая архитектура

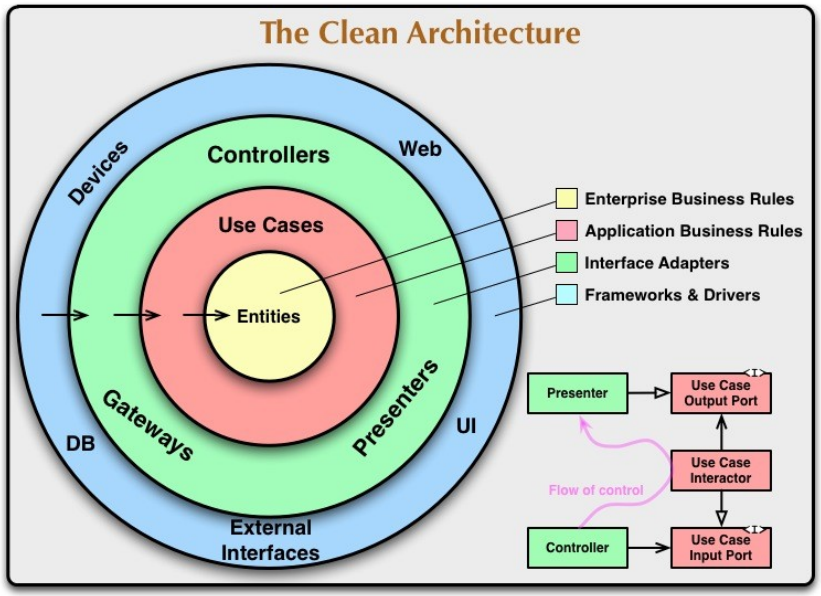


- Ядро системы:
  - **Доменная модель** – сущности предметной области
  - **Доменные сервисы** – общая для всей предметной области логика
  - **Сервисы приложения** – специфичная для данного приложения логика
- Внешние модули:
  - **Интерфейс пользователя**
  - **Тесты**
  - **Инфраструктура** – хранение данных, передача по сети другим системам и т.п.

Разновидности







Плюсы и минусы доменного подхода (DDD и т.д.)

- Плюсы
  - Не всегда оправдано выставление наружу полного набора сущностей и примитивных операций (CRUD) над ними
    - Сущности могут быть чересчур сложными или не все их свойства могут быть сделаны доступными внешнему клиенту
    - Бизнес-сценарий может требовать сложного транзакционного взаимодействия над множеством различных сущностей, и выносить его на какой-то внешний уровень было бы вредно.
  - Адекватная доменная модель может позволить легко и гибко реализовывать любые бизнес сценарии прозрачным и понятным способом. В теории
- Минусы
  - Сложно, дорого, долго. ОЧЕНЬ. Экономическая эффективность под большим вопросом.
  - Масштабирование – требует дополнительных усилий.
  - Попытка загнать все сценарии использования в одну доменную модель обычно терпит фиаско
    - Пример из жизни – построение отчетности, используя классический репозиторий –> Java OOM.

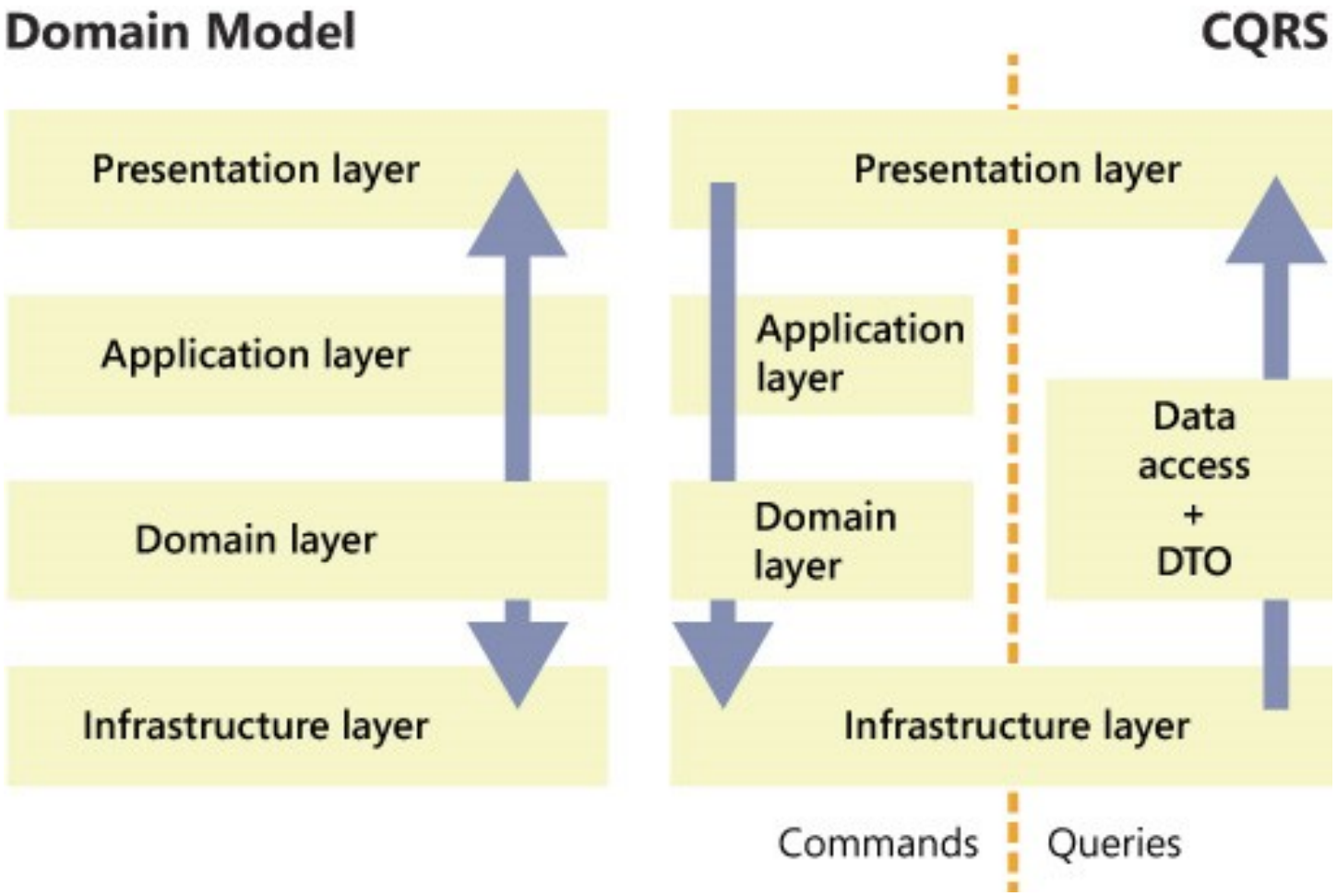
CQRS (Command and Query Responsibility Segregation)

- Контекст
  - Необходимо реализовать работу с сущностями системы не напрямую, а через механизм доменной модели и сервисов, реализующих сложную бизнес-логику, при этом не ухудшая производительность простых сценариев (поиск, отчетность) и сохраняя легкость масштабирования.
- Задача
  - Как организовать простой, понятный и непротиворечивый интерфейс (как пользовательский, так и межкомпонентный) для работы со сложными сценариями использования системы?
- Решение
  - Паттерн CQRS предоставляет **отдельные интерфейсы** для реализации операций **чтения** и **записи**
  - Это позволяет разделить сущности для этих операций – не обязательно везде использовать полные сущности
  - Команды позволяют реализовать сложные операции для реализации целых бизнес-сценариев

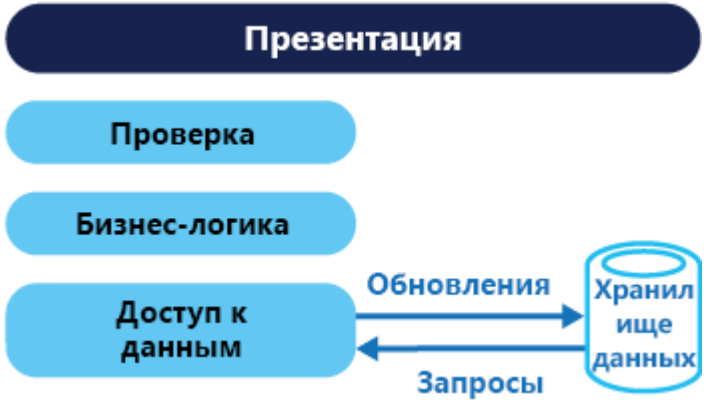
CQRS (Command and Query Responsibility Segregation)

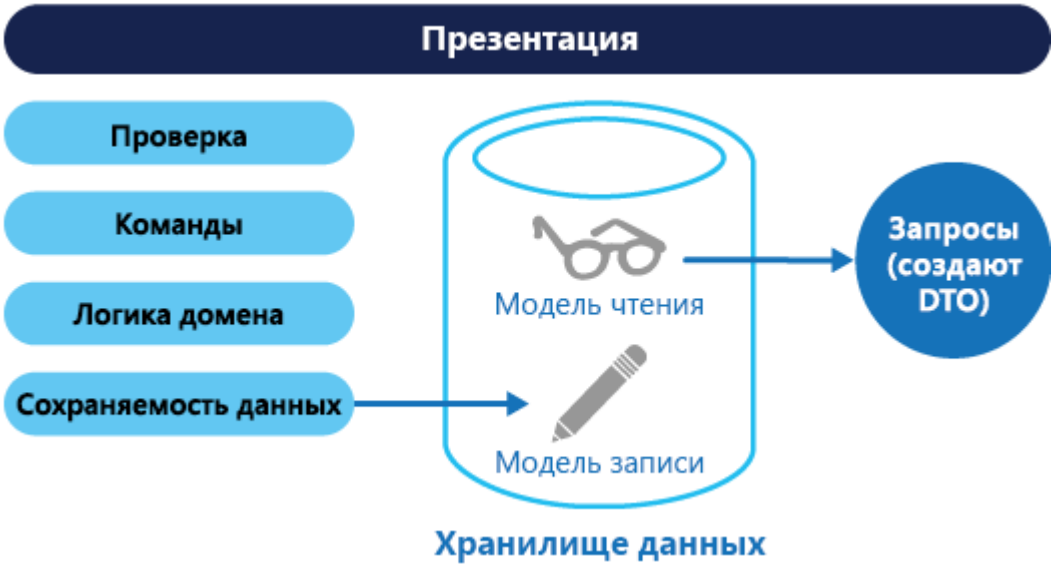
- Элементы
  - **Команды** – операции обновления данных
  - **Запросы** – операции считывания данных
- Ограничения
  - Запросы не могут менять данные, а лишь считывают состояние
  - Команды не получают информацию о состоянии, а только меняют данные
- Трудности
  - Гораздо сложнее в реализации, чем классический CRUD
  - Есть ряд (успешно решаемых) сложностей с распараллеливанием
  - При разделении хранилищ чтения и записи возникает задача обеспечения их консистентности

Разделение ответственностей

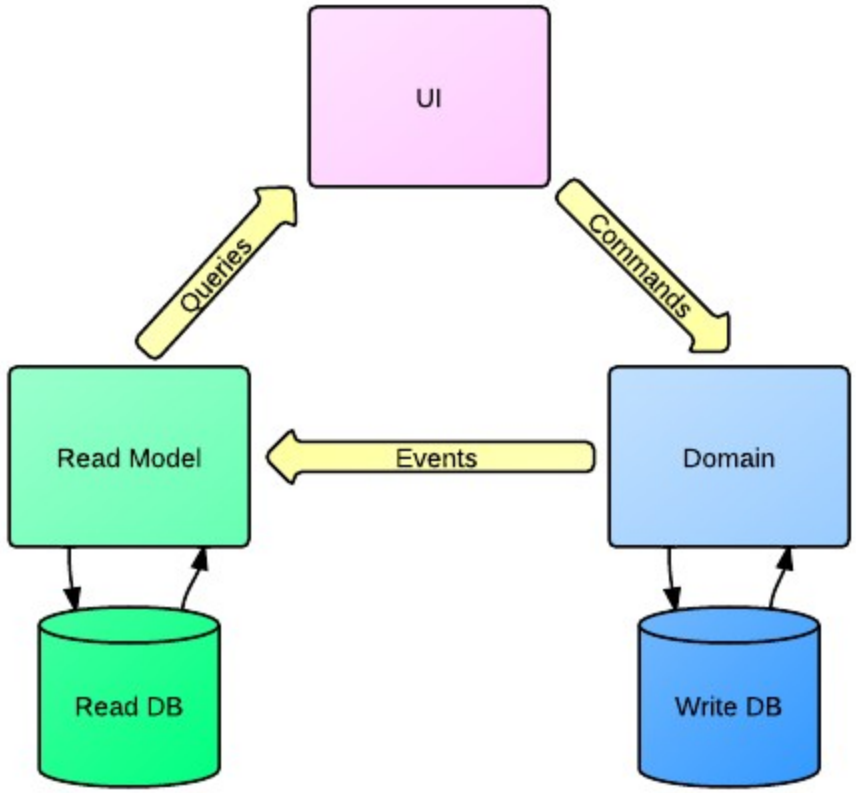


Классика vs CQRS





CQRS с раздельными хранилищами



Разница между CRUD и CQRS на примере

- Сценарий:
  - Система управления кадрами
  - Нужно увеличить оклад Василия Пупкина на 5 000 р.
- CRUD-подход
  - Найти по имени ID записи Василия Пупкина
  - Считать запись по ID
  - Модифицировать запись
  - Сохранить запись
- CQRS-подход
  - Найти по имени ID записи Василия Пупкина
  - Вызвать специальный метод изменения оклада для ID и передать ему параметром величину изменения, и с какой даты внести изменения.
    - Данный метод не просто поменяет величину оклада в БД, но и выполнит все связанные в этим изменением перерасчеты, требуемые законодательством и правилами бухучета.

Проблема счетчика лайков

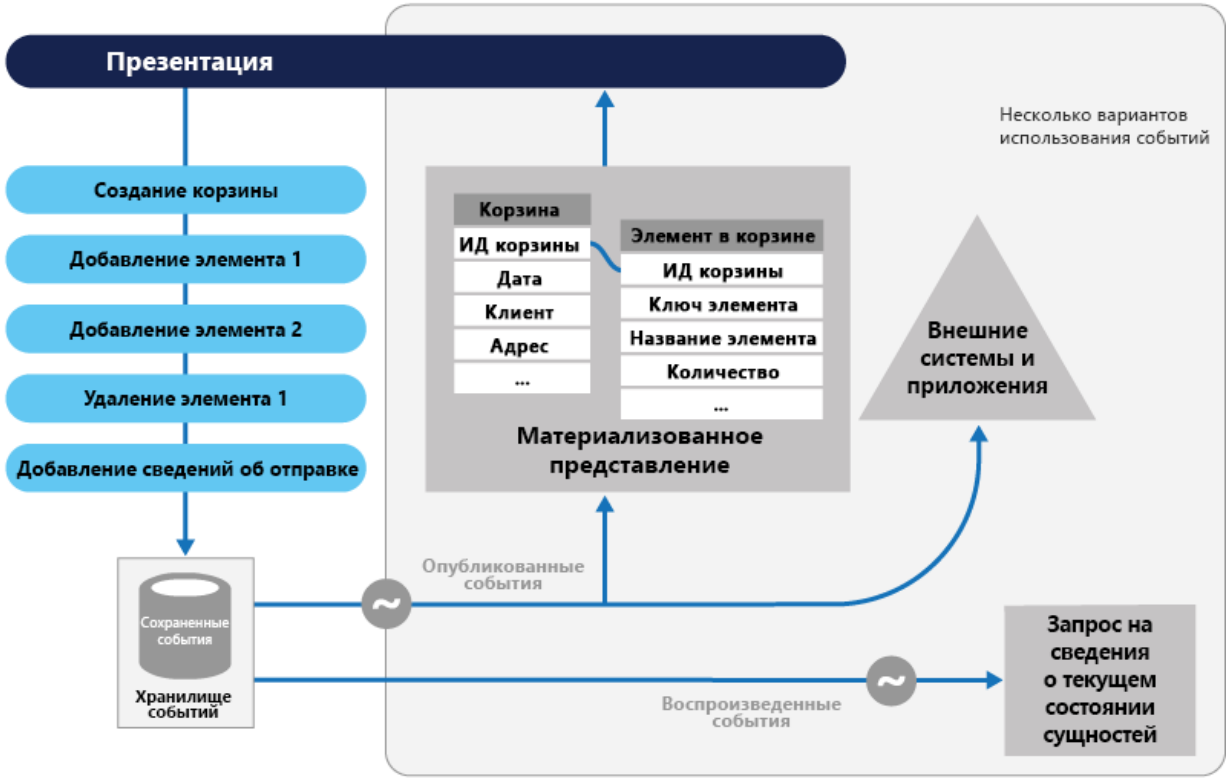
- Счетчик лайков в соцсети
  - Миллионы пользователей онлайн, работающие через десятки АПИ-серверов одновременно и генерирующие сотни лайков в секунду для популярных вирусных постов.
  - Объект Пост – довольно крупный агрегат, включающий в себя ссылку на объект Пользователь, контейнер объектов Лайк, объекты Медиа и т.п.
- Проблема - как записать данные в доменную базу и как одновременно читать их?
  - Поднять весь агрегат объекта Пост в память, со всеми связанными объектами, обновить с учетом всех бизнес-правил и сохранить в БД?
    - Удачи.
  - И даже прямым SQL - сложно
    - Проблемы конкурентности записи – блокировки в БД
    - Эскалация блокировок
    - Чистота чтения (и связанные с ней блокировки)

Изобретаем Event Sourcing

- Решение
  - Вместо того, чтобы толпой ломиться на запись в одну единственную строку большой общей БД – пишем множество отдельных записей-фактов о произошедших событиях вида:
    - **{действие: лайк; юзер: №xxxx; пост: №yyyyy;}**
  - Обновляем отдельный счетчик (минимального объема, лишь одно число) в максимально быстрой и небольшой in-memory БД ключ-значение для последующего чтения (**частичная модель чтения** – для отображения счетчика в ленте - достаточно)
  - Обновление полного агрегата объекта Пост в доменной модели (подвязка объектов Лайк, обновление поля-счетчика в объекте) в БД происходит в **пакетном** режиме **в фоне**, отдельным процессом. Вместе с ним обновляется и модель чтения.

CQRS и Event Sourcing

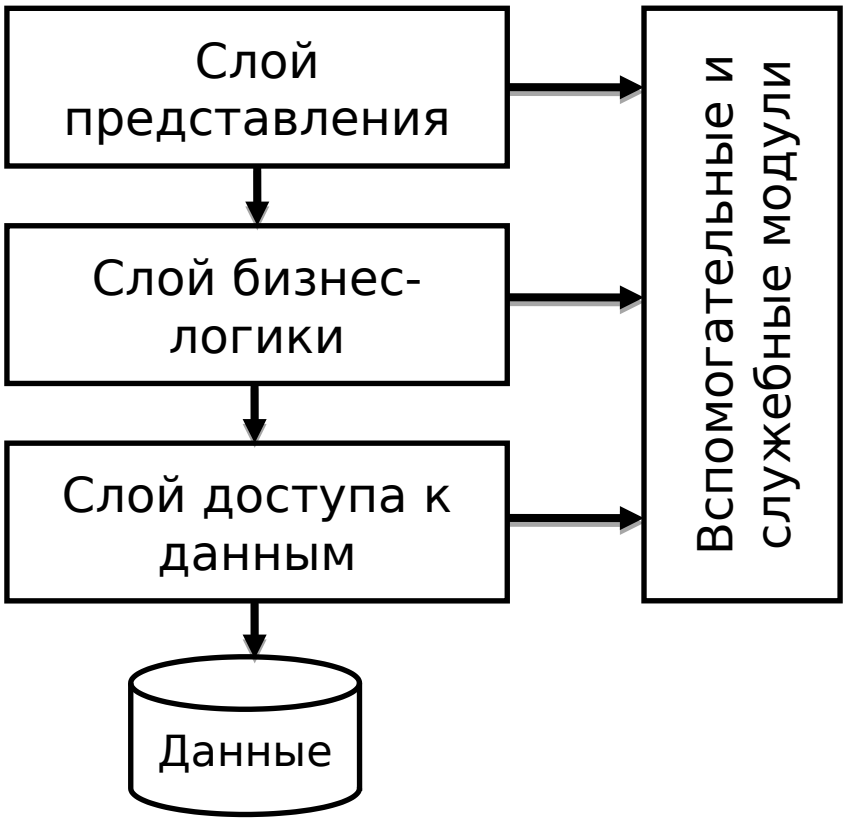
- Часто CQRS используется вместе с паттерном Event Sourcing (источник событий)
- Все действия пользователя приводят к созданию команды, которая помещается в хранилище событий
- Команды могут
  - Менять данные, которые затем считываются запросами
  - Напрямую транслироваться заинтересованным (подписанным) клиентом, и они сами будут на них реагировать
  - Отправлять внешним системам на обработку



Масштабирование

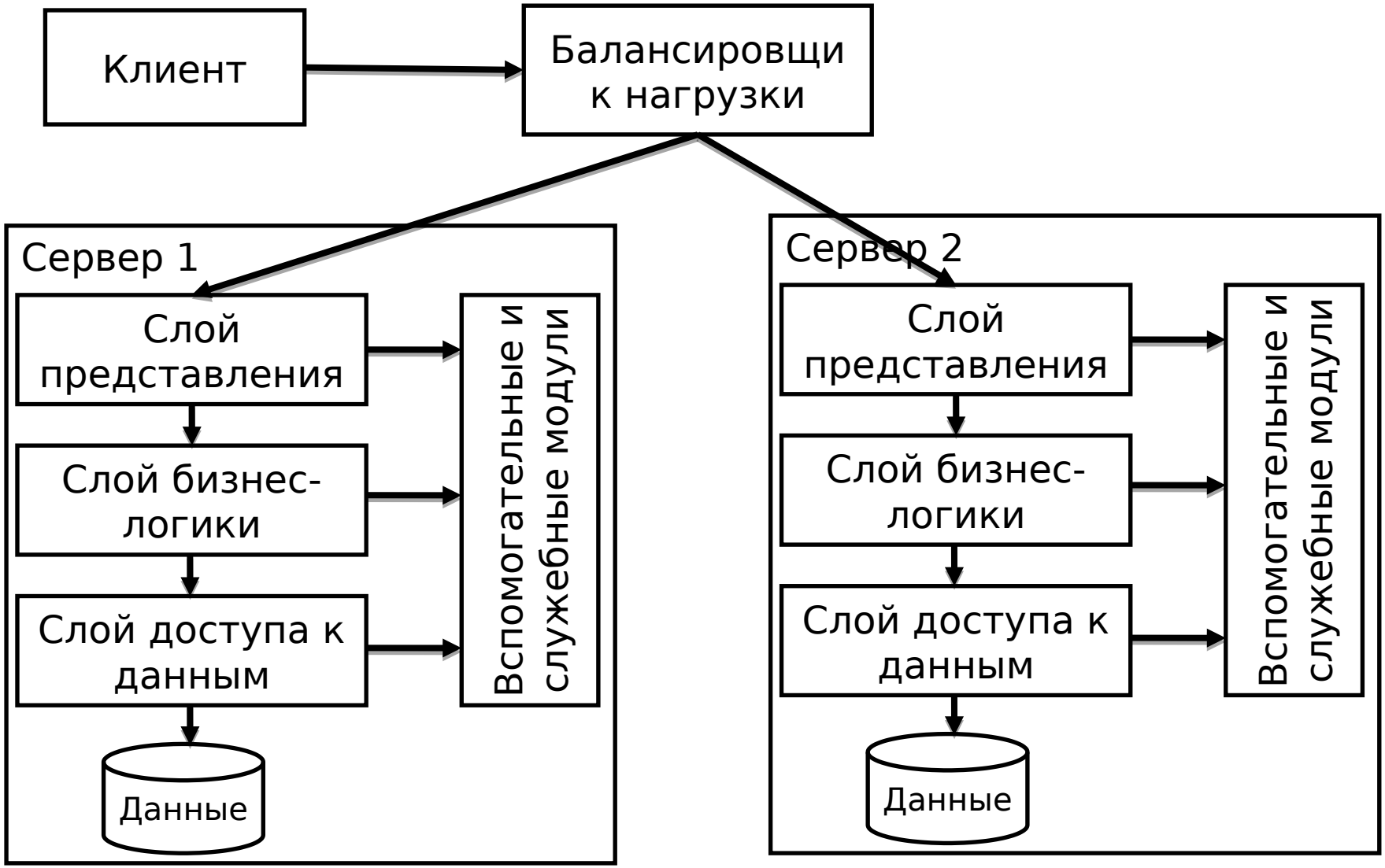
- Раз уж заговорили о высоконагруженных приложениях, рассмотрим модели масштабирования
  - **Масштабирование** – способность системы справляться с увеличением рабочей нагрузки (увеличивать свою производительность) при добавлении ресурсов.
- **Вертикальное** масштабирование
  - Не хватает производительности – возьмем более мощную машину
  - Тупиковый путь – мощность машин ограничена, потребности – нет. **Нелинейный рост стоимости.**
  - Иногда может стать единственной моделью (например, единый сервер БД в датацентричной системе)
- **Горизонтальное** масштабирование
  - Не хватает производительности – возьмем больше машин
  - Эффективный путь – машин всегда можно подключить побольше. **Линейный рост стоимости.**
  - Требует архитектурной подготовки решения.

Пример: Масштабирование веб-приложения

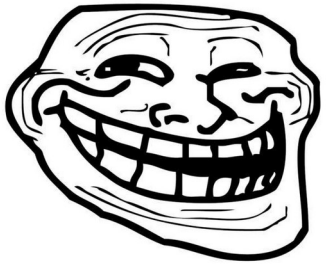


- Дано
  - Есть веб-приложение, написанное в классическом трехуровневом стиле
  - Поток пользователей вырос, сервер не справляется, запросы обрабатываются неприемлемо долго
- Решение
  - Попробуем горизонтально масштабировать систему

Пример: Масштабирование веб-приложения «в лоб»

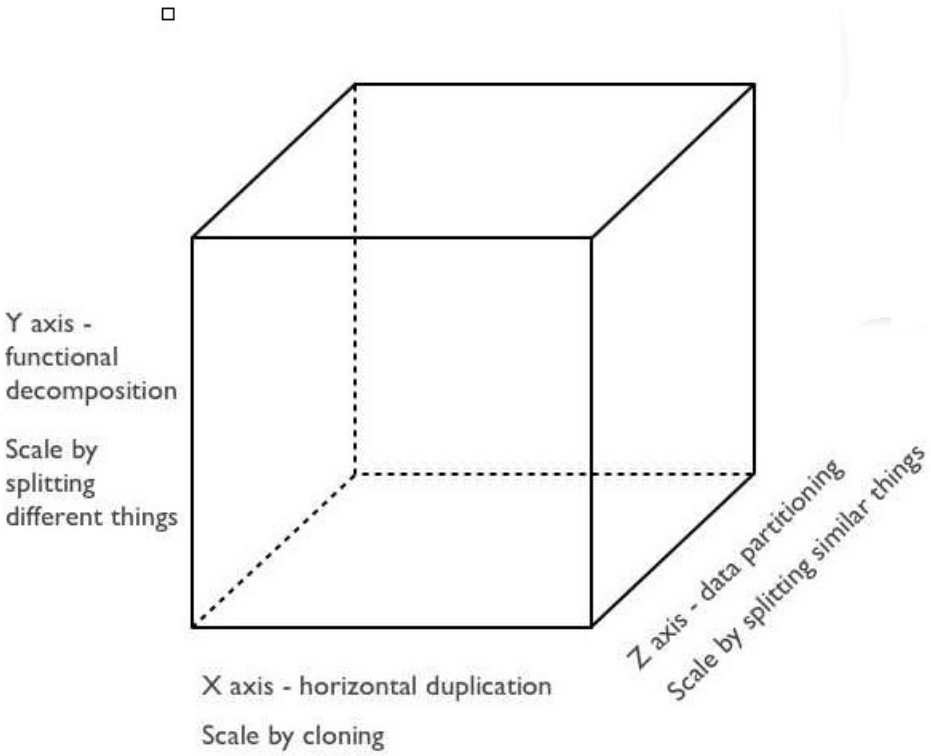


Пример: Масштабирование веб-приложения «в лоб»



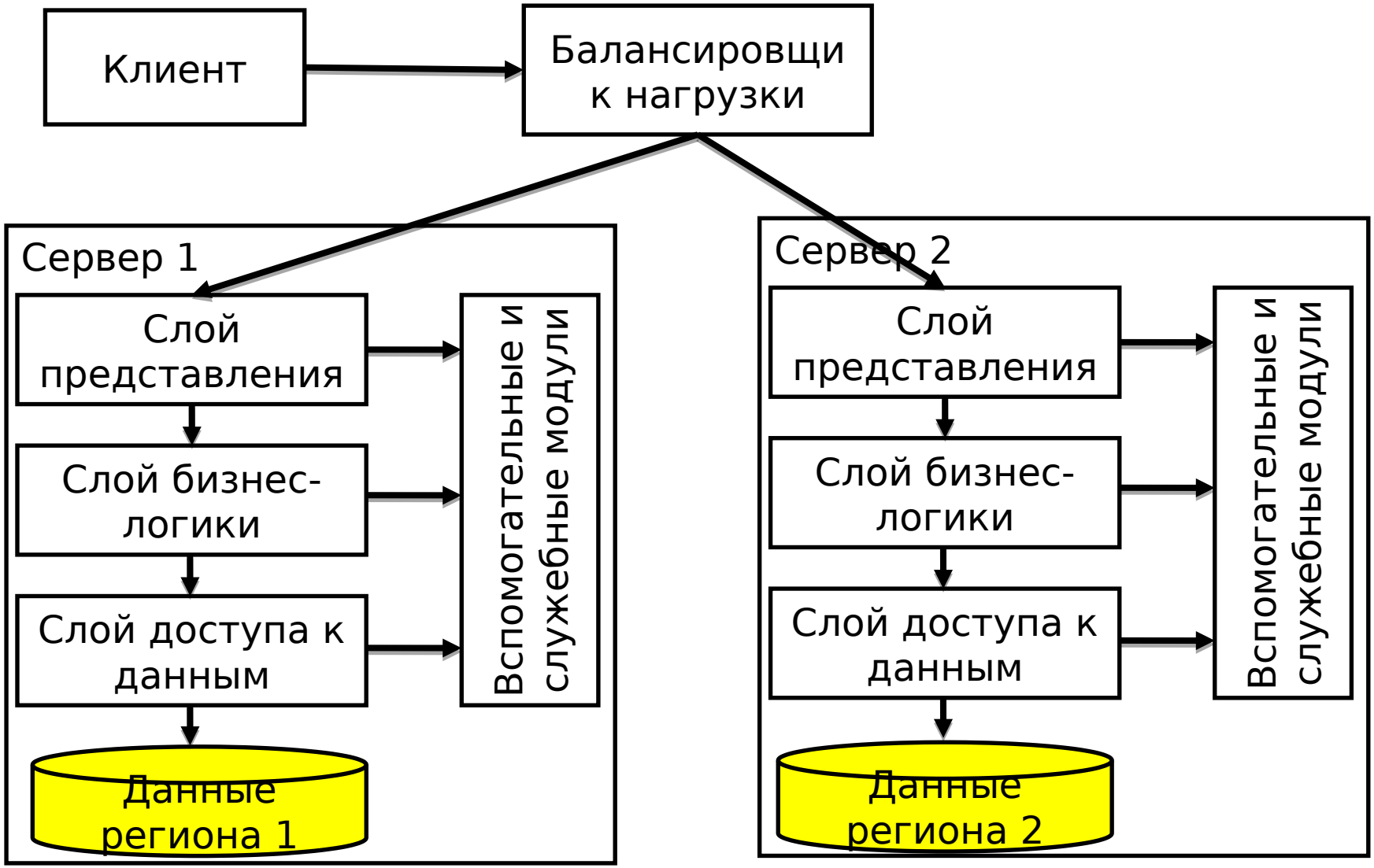
Пример: Масштабирование веб-приложения «в лоб»

Куб масштабируемости

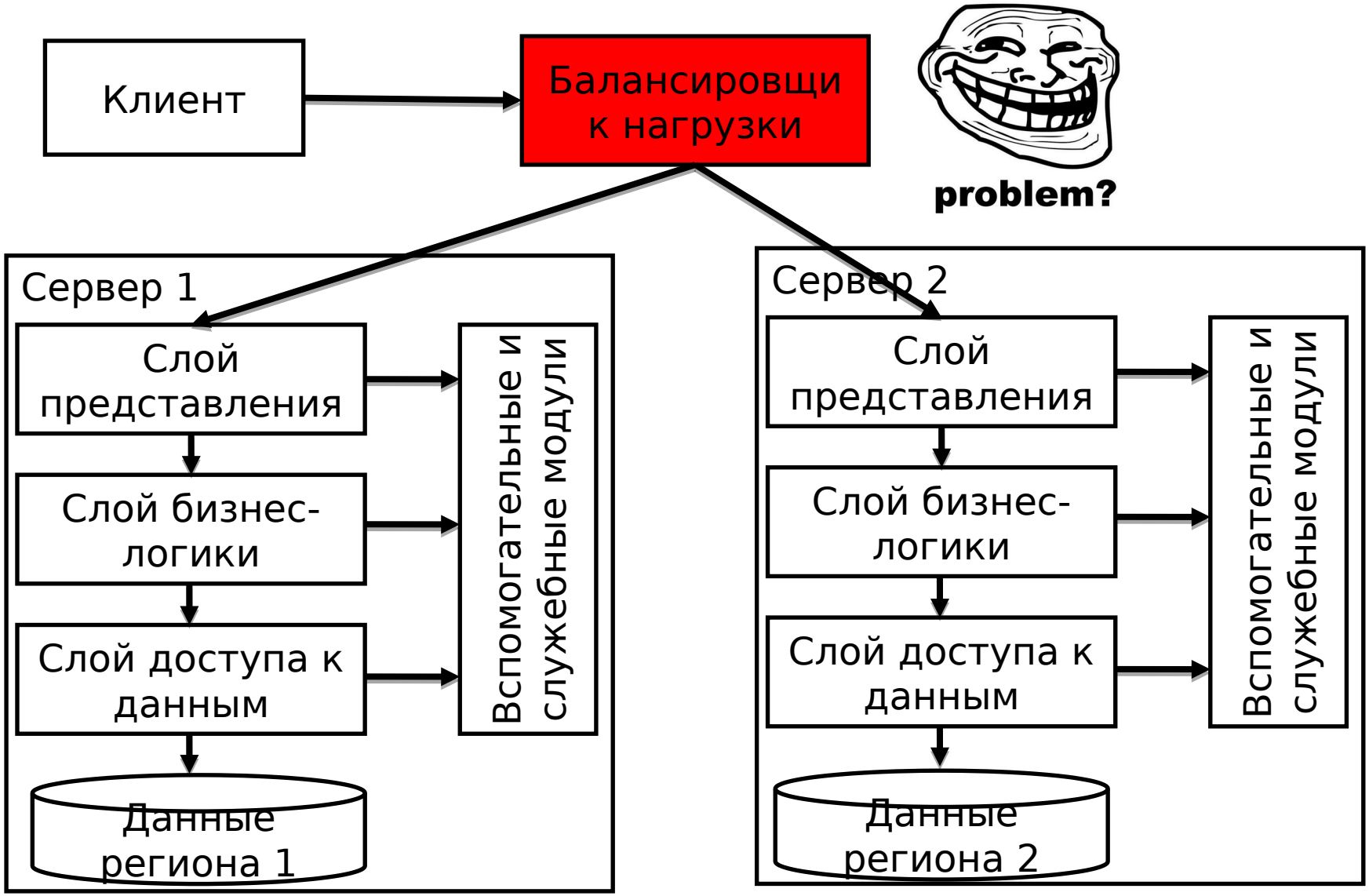


- Ось X – горизонтальное масштабирование
  - Масштабирование клонированием
- Ось Y - функциональная декомпозиция
  - Масштабирование разделением разных вещей
- Ось Z – партиционирование данных (шардинг)
  - Масштабирование разделением схожих вещей

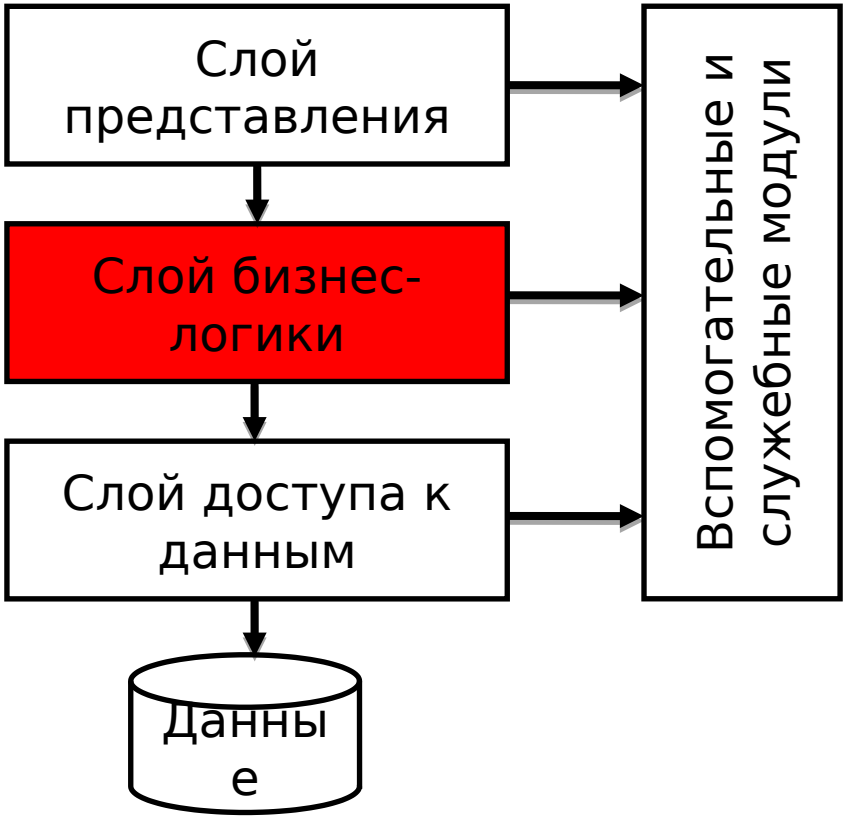
Масштабирование веб-приложения с партиционированием данных



Масштабирование веб-приложения с партиционированием данных

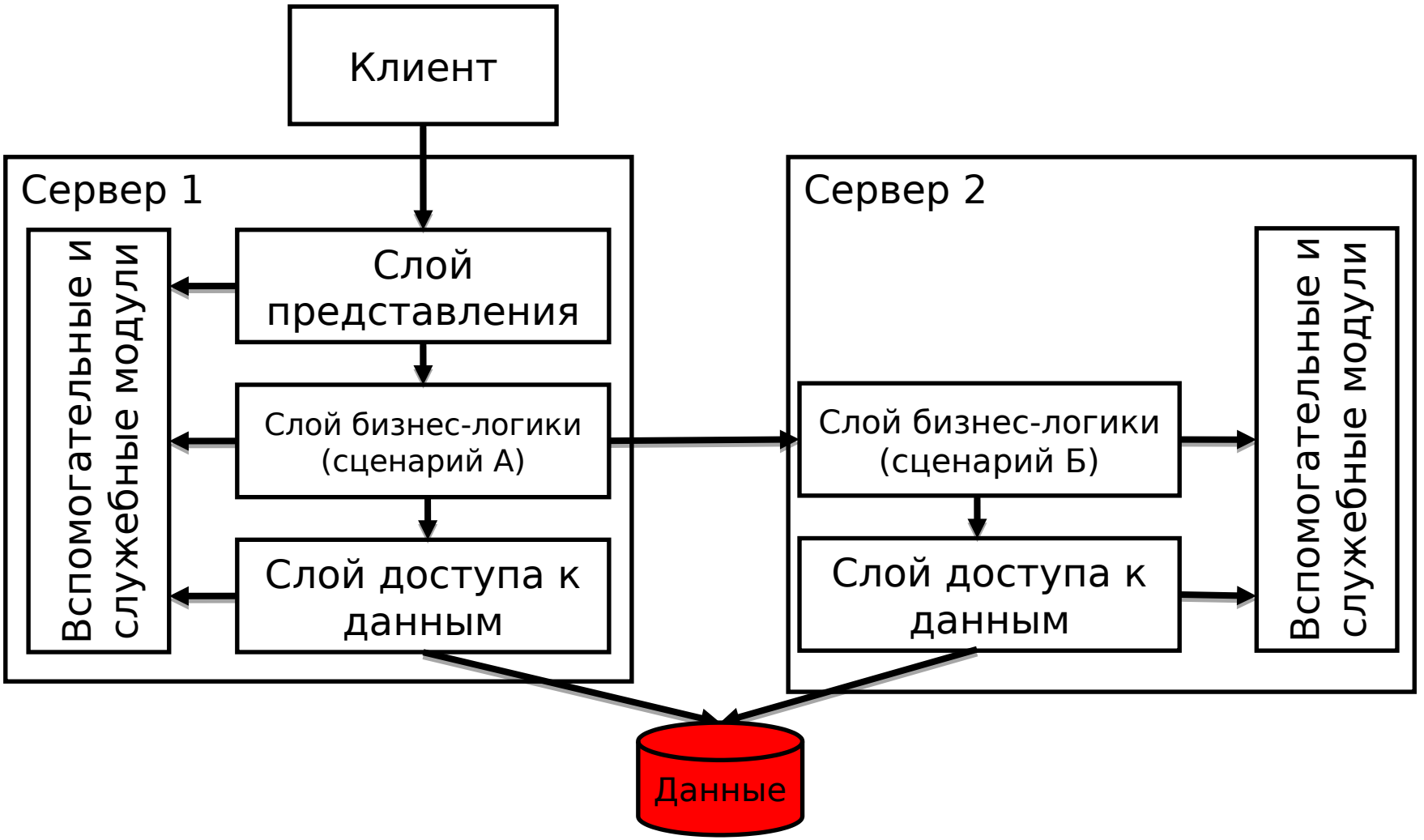


Пример: Масштабирование веб-приложения

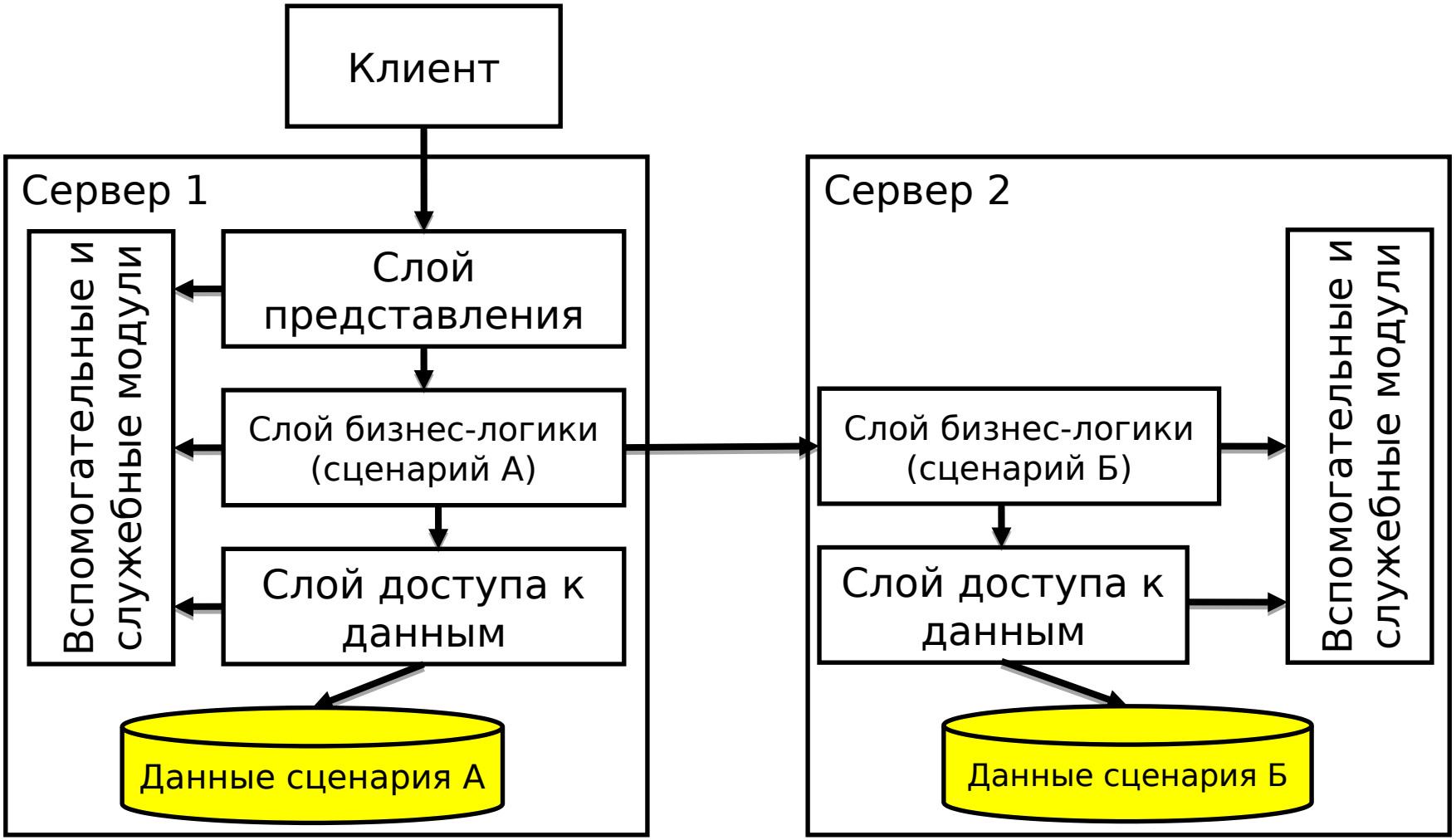


- Редко проблемы производительности касаются всего приложения в целом
  - «Скорость эскадры равняется скорости самого медленного её корабля» (с)
  - Чаще всего от нагрузки страдает какая-то одна определенная подсистема
  - Зачем дублировать все приложение в целом, если можно продублировать одну подсистему?
  - Вспоминаем ось Y куба масштабирования – **функциональную декомпозицию**
- Необходимо вынести на отдельный вычислительный ресурс те функции, производительность которых страдает больше всего
  - Приложение должно быть архитектурно готово к этому

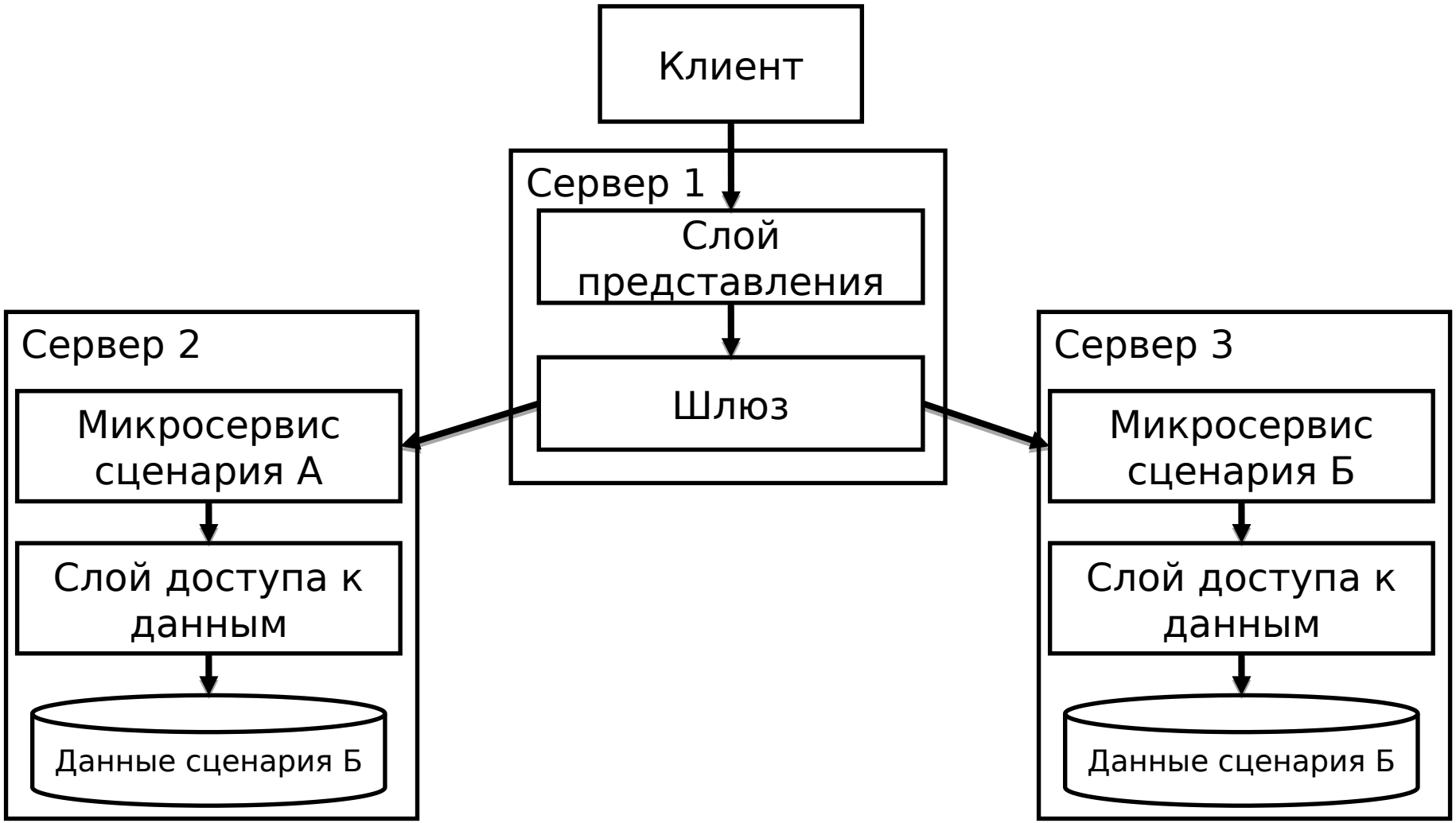
Пример: Масштабирование веб-приложения – распределенные вычисления



Пример: Масштабирование веб-приложения – объединяем идеи



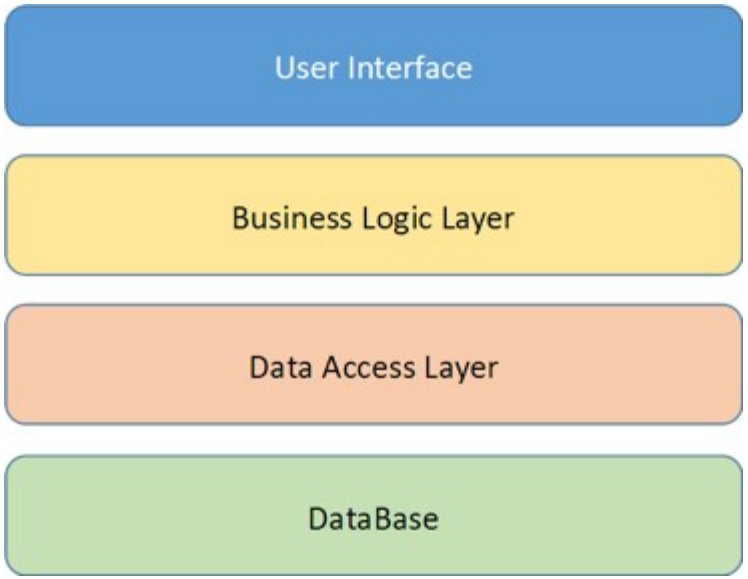
Пример: Масштабирование веб-приложения – микросервисы

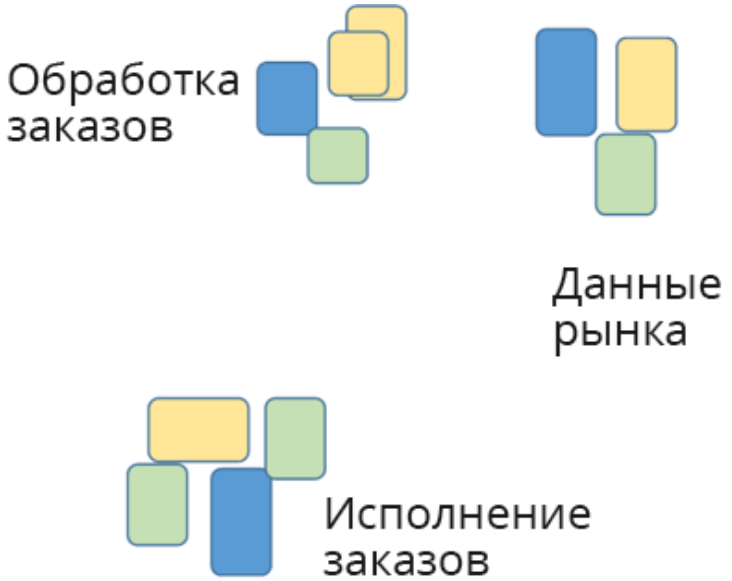


Микросервисная архитектура

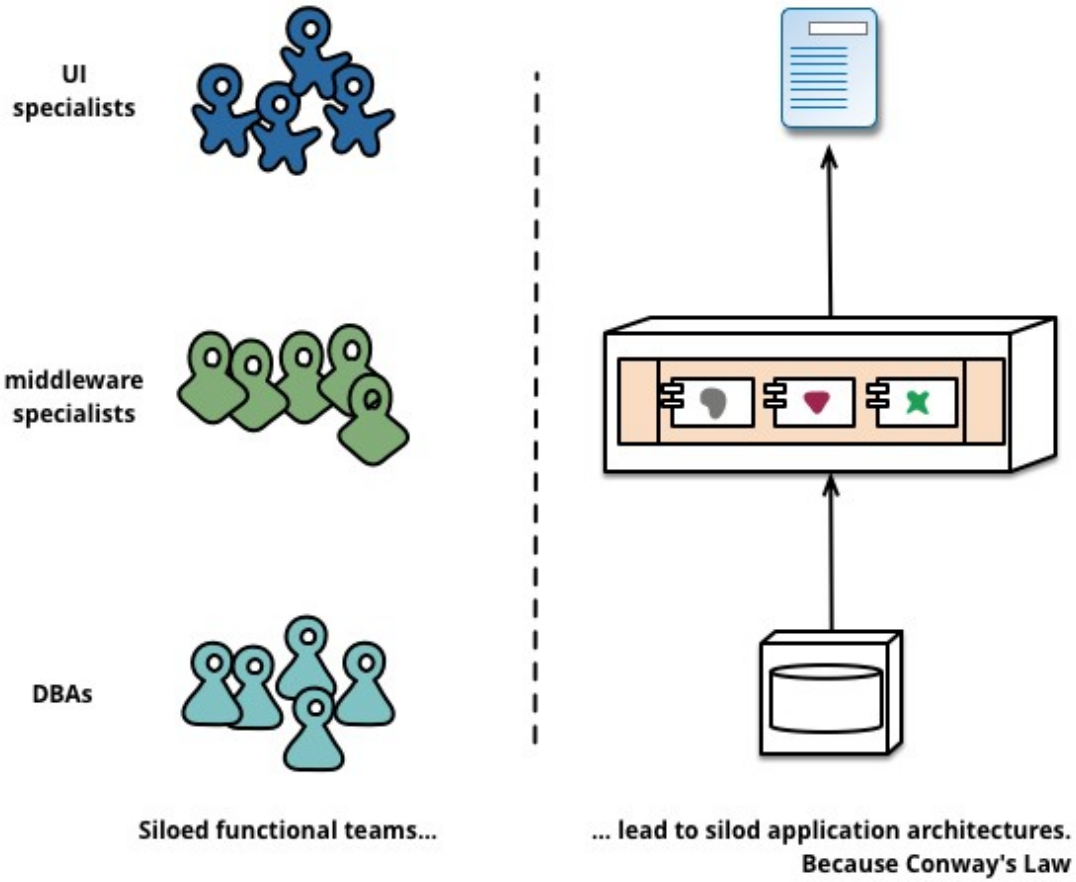
- **Микросервисы** - это способ разбиения большого проекта на небольшие, независимые и слабо связные модули.
  - Независимые модули отвечают за четко определенные и дискретные задачи и общаются друг с другом посредством простого и доступного API.
- Свойства микросервисов:
  - **Маленькие**
    - Сервис не должен требовать множества людей для разработки. Одна команда может разрабатывать несколько сервисов.
  - **Сфокусированные**
    - Один сервис – одна задача.
  - **Слабосвязанные**
    - Изменения в одном сервисе не влияют на другой.
  - **Высокосогласованные**
    - Компонент или класс создаются с учетом всех методов решения бизнес-задачи.

Монолит vs Микросервисы





Организация вокруг структуры команды

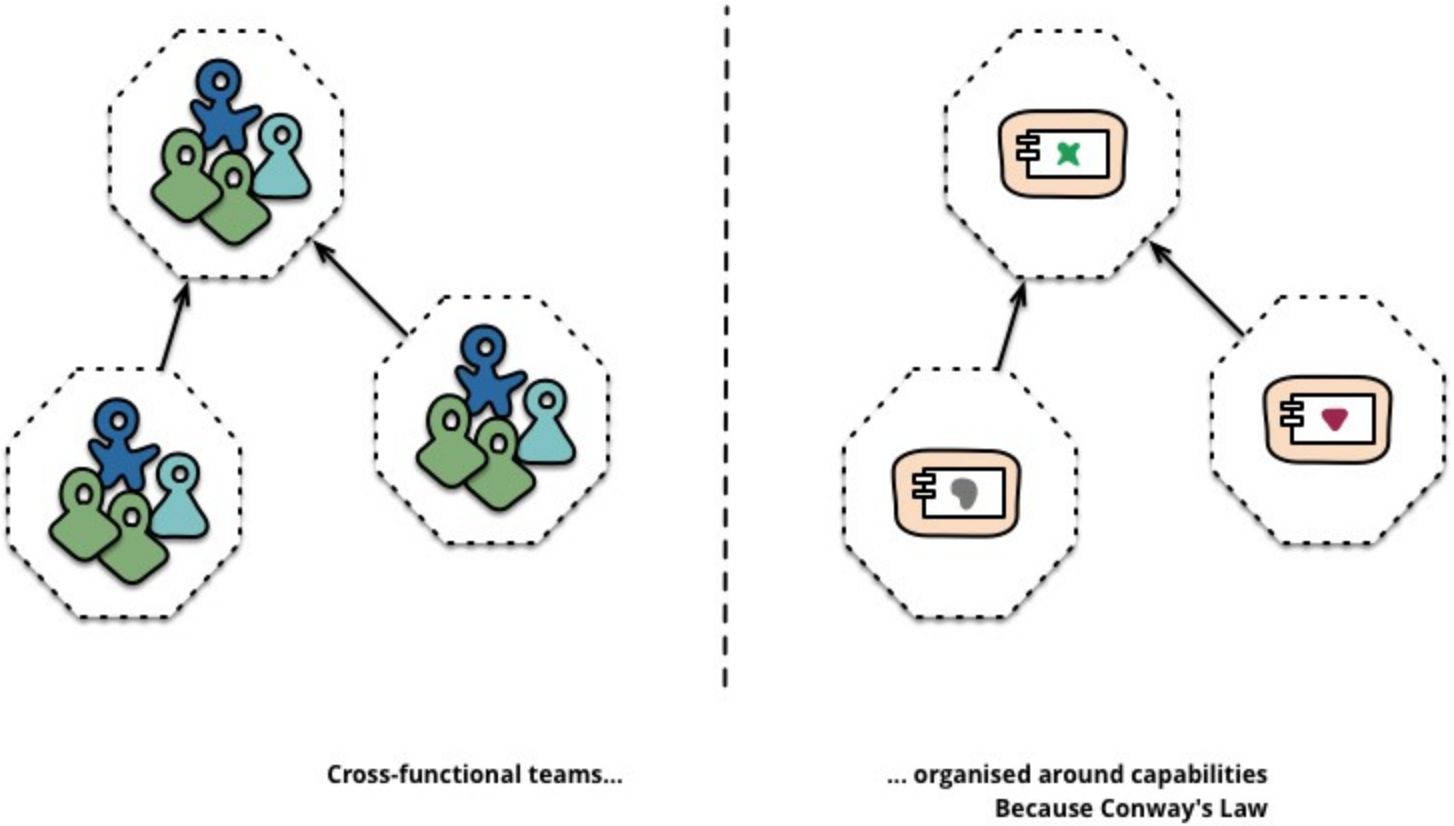


Закон Конвея:

«Любая организация, которая проектирует какую-то систему (в широком смысле) получит дизайн, чья структура копирует структуру команд в этой организации»

— Melvyn Conway, 1967

Организация вокруг потребностей бизнеса



Cross-functional teams...

... organised around capabilities
Because Conway's Law

Микросервисы vs SOA

- Service Oriented Architecture (SOA) – прародитель идеи микросервисов
  - «Глупые оконечные точки и умные каналы передачи данных»
    - общая шина, соединяющая отдельные сервисы
    - существенная часть бизнес-логики системы реализуется заданием правил маршрутизации, оркестровке и трансформации сообщений шиной
    - сами сервисы реализуют лишь ограниченную, атомарную часть функциональности и могут рекомбинирвоаться для получения различных результатов
    - добавление новых экземпляров сервисов требует перенастройки правил коммуникации
- Микросервисы
  - «Умные оконечные точки и глупые каналы передачи данных»
    - обычные веб-протоколы или легковесная шина сообщений для общения между сервисами
    - вся бизнес-логика реализуется в самих сервисах
    - сервисы максимально самодостаточны
    - добавление новых экземпляров сервисов требует лишь перенастройки балансировщика нагрузки

Рабочая среда. DevOps

- Популярность микросервисной архитектуры обусловлена развитием среды исполнения
  - Аппаратные серверы – слабая автоматизация, долгое время разворачивания. Монолитные приложения.
  - Виртуальные машины (ВМ) – уже лучше, есть возможности автоматизации, но все же разворачивание идет довольно длительно. Даже в облаках.
  - Контейнеризация приложений (максимально облегченные ВМ) – позволяет запустить каждый процесс в независимом вычислительном контейнере, и подымать столько экземпляров каждого процесса, сколько необходимо.
- DevOps – культура разработки, когда роли разработчика (Developer, Dev) и системного администратора (Operations, Ops) сливаются.
  - Автоматизация развертывания виртуальных машин (IaaC - Infrastructure as a Code) и ПО на них (рецепты chef, puppet)
  - Автоматизация мониторинга и управления контейнерами на кластере (Docker Swarm , Kubernetes)
  - «*Пишите код так, как будто сопровождать его будет склонный к насилию психопат, который знает, где вы живёте*» - ничто так не мотивирует писать код хорошо, как осознание факта, что все проблемы с системой решать тебе же.
    - Проектирование с учетом выхода из строя (Design for failure)

Характеристики микросервисов (подведение итогов)

- Разделение на компоненты (сервисы).
  - если вы можете взять что-то и спокойно заменить на новую версию, — это компонент.
- Группировка по бизнес-задачам.
  - сервисы имеют бизнес-смысл.
- Умные сервисы и простые коммуникации.
  - если вы будете все складывать в среду передачи, у вас получится умный монолит и тупые сервисы-обертки баз данных.
- Децентрализованное управление данными.
  - у каждого сервиса свое хранилище данных
- Автоматизация развертывания и мониторинга.
  - сервисы должны автоматически развертываться и останавливаться по мере необходимости, чтобы держать нагрузку и не потреблять лишних ресурсов
- Проектирование с учетом выхода из строя (Design for failure).
  - с самого первого этапа, начиная строить микросервисную архитектуру, вы должны исходить из предположения, что ваши сервисы не работают, т.е. каждый сервис должен понимать, что ему могут не ответить никогда, если он ожидает каких-то данных.

Плюсы и минусы микросервисов

- Плюсы
  - Четкое деление по модулям.
    - Всегда будет понятно, как работает та или иная часть кода. Просто добавлять новые функции.
  - Высокая доступность.
    - Если какая-то часть монолита сломается – сломается все приложение. С микросервисами иначе: сервисы могут работать не все (не критические, вроде авторизации), но приложение при этом останется доступным.
  - Разнообразные технологии в одном решении.
- Минусы
  - Сложность разработки
    - За доступность и модульность приходится платить скоростью разработки.
    - Множество баз данных и управление транзакциями может быть реальной болью.
  - Сложность развертывания
    - Нужно развернуть не один продукт, а комплекс разных сервисов нужных версий.
  - Сложность поддержки.
    - Каждый микросервис нуждается в отдельном обслуживании, поэтому нужен постоянный автоматический мониторинг. Оркестраторы.

Эпилог

- Шаблоны – не догма
  - Смешивайте шаблоны. Дорабатывайте шаблоны. Нарушайте шаблоны, если понимаете, что делаете ☺
- Серебряной пули нет.
  - Для каждой задачи есть свое решение.
    - Client-Server + 3-layers – классическое веб-приложение
    - CQRS + Event Sourcing + Microservices - отлично для высоконагруженных систем, но абсолютно не надо для интернет-магазина с посещаемостью в 100 уников в день.
- Важны не только технические, но и экономические факторы.
  - Посредственное решение вовремя – всегда лучше идеального невовремя.