

Системный дизайн

Немного обо мне

Дзюба Дмитрий Владимирович

ст. преп. каф 806

опыт разработки 20+ лет

участвовал в проектировании и разработки сложных
высоконагруженных телеком-платформ, классических
учетных систем, геоинформационных систем,
аналитических систем



О чем этот курс

- Архитектура и процессы проектирования
- Документирование и презентация архитектуры
- Контейнеризация
- Микросервисы
- Масштабирование statefull сервисов
- Производительность программных систем
- Событийно-ориентированная архитектура

Мы будем
кодировать

GE's CEO Wants Every New Hire to Learn This Skill

"IF YOU ARE JOINING THE COMPANY IN YOUR 20'S, UNLIKE WHEN I JOINED, YOU'RE GOING TO LEARN TO CODE. IT DOESN'T MATTER WHETHER YOU ARE IN SALES, FINANCE OR OPERATIONS. YOU MAY NOT END UP BEING A PROGRAMMER, BUT YOU WILL KNOW HOW TO CODE."

-Jeff Immelt

<https://fortune.com/videos/watch/ge%E2%80%99s-ceo-wants-every-new-hire-to-learn-this-skill/ed265904-92cb-4a32-9cbb-4d9a17032dd3>

Что нам потребуется

- Знание основ программирования на C++ (python – опционально) и умение написать простую программу;
- Базовое знание работы в bash по ОС Linux;
- Физическая или виртуальная машина под управлением ОС Linux Ubuntu Server 22 или более свежих версий;

Что вам понадобится

- Компьютер с 8Gb оперативной памяти (или больше) и операционной системой Windows / MacOS / Linux
- Среда разработки Visual Studio Code
<https://code.visualstudio.com/download>
- Знание python на базовом уровне
- Среда виртуализации Microsoft HyperV/ Oracle VirtualBox/ VMWare
- На некоторых занятиях нам понадобится дополнительное ПО, которое нужно будет скачать.
- Практические работы мы делаем под Linux;


Общая информация по курсу

- Практическое задание (курсовая работа), разбитая на части (лабораторная работа):
 1. Документирование архитектуры
 2. REST сервисы
 3. Реляционные базы данных
 4. NoSQL базы данных
 5. in-memory базы данных
 6. Event-driven architecture
- Зачет по практическому заданию (теоретические вопросы)

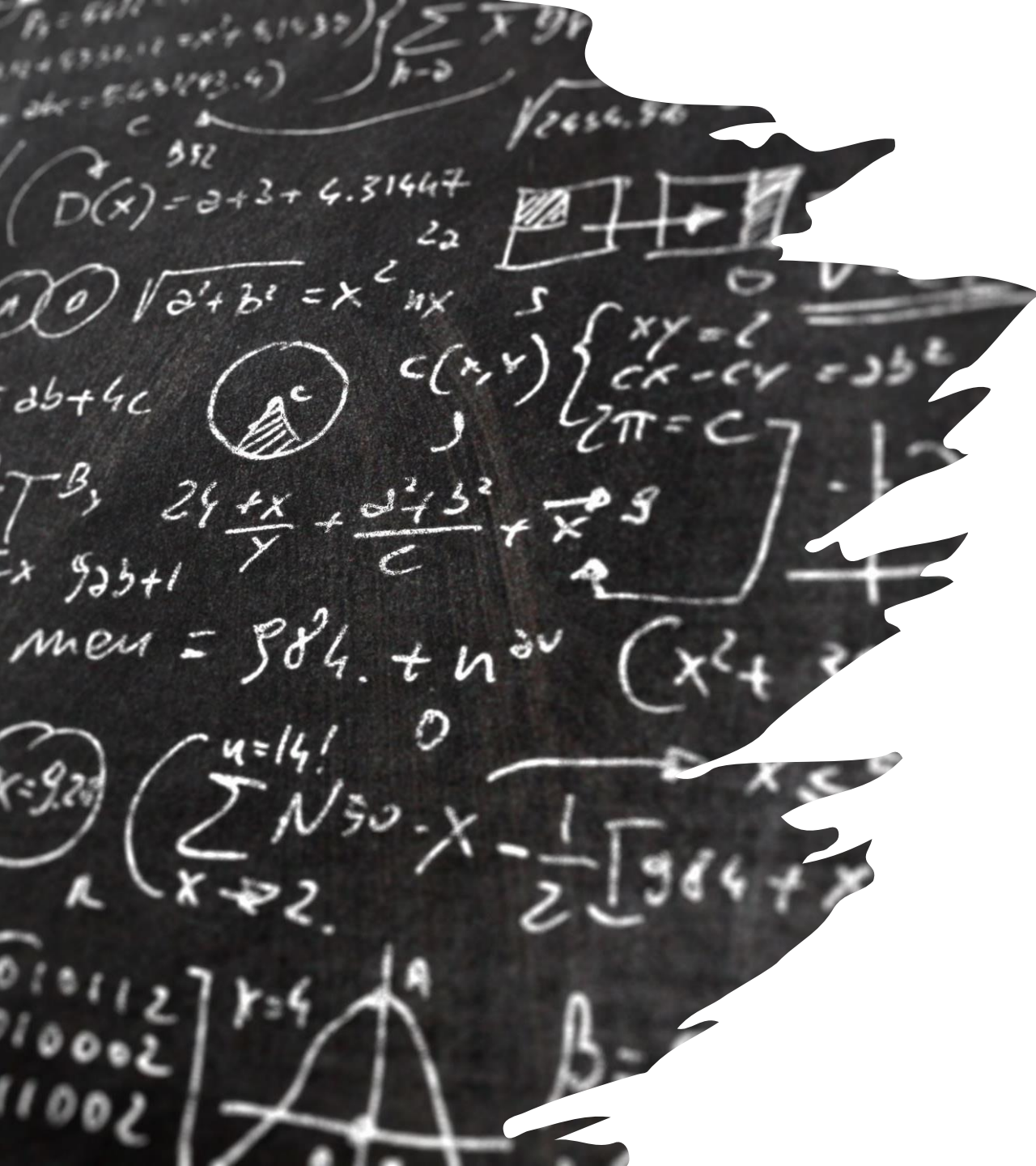
Отчетность по курсу

- Лабораторная работа – до 15 баллов.
- Время на сдачу ЛР – две недели (позже максимальный балл снижается до 10)
- Теоретический зачет - до 20 баллов

5-бальная система	Рейтинговая система	Европейская система
5 - Отлично	90-100	A
4 – Хорошо	82-89	B
	75-81	C
3 - Удовлетворительно	67-74	D
	60-66	E
2 - Неудовлетворительно	Менее 60	F



организационные
вопросы?



1. Общие понятия
2. Процесс проектирования
3. Архитектурные стили

Общие ПОНЯТИЯ



Цитата

Существует два вида сложности при создании систем*:

1. Первичный - **сложность разработки** как таковой, она зависит от сложности решаемой задачи и сложности выбранного решения и инструментов. **С этой сложностью борется Архитектура.**
2. Вторичный - **это сложность, связанная со взаимодействием людей** и специфики процессов которые влияют на создание системы. С ней боятся разные управленческие фреймворки наборы лучших практик, в том числе и Agile.

* "Анализ и проектирование ИТ-систем с помощью UML 2.0",
Лешек А. Мацяшек.

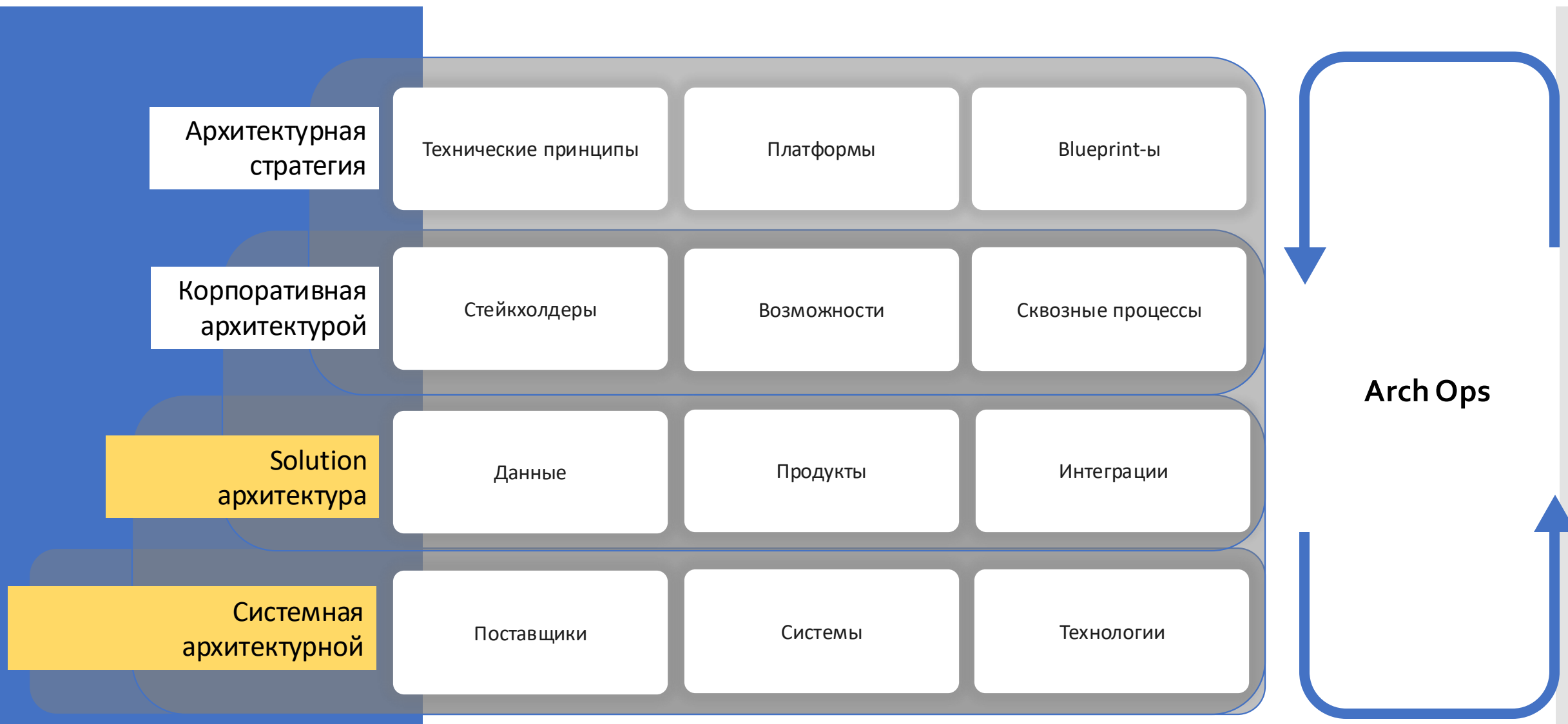
Архитектура

Определение ANSI / IEEE Std 1471-2000

Основные принципы организации системы, воплощенной в его компонентах, их взаимоотношениях друг с другом и окружающей средой и принципах, регулирующих дизайн и эволюцию системы

Архитектура - описание структуры системы, состоящей из элементов ПО, описания видимых свойств ПО и связей между ними. Архитектура строится на основе небольшого числа приоритетных требований.

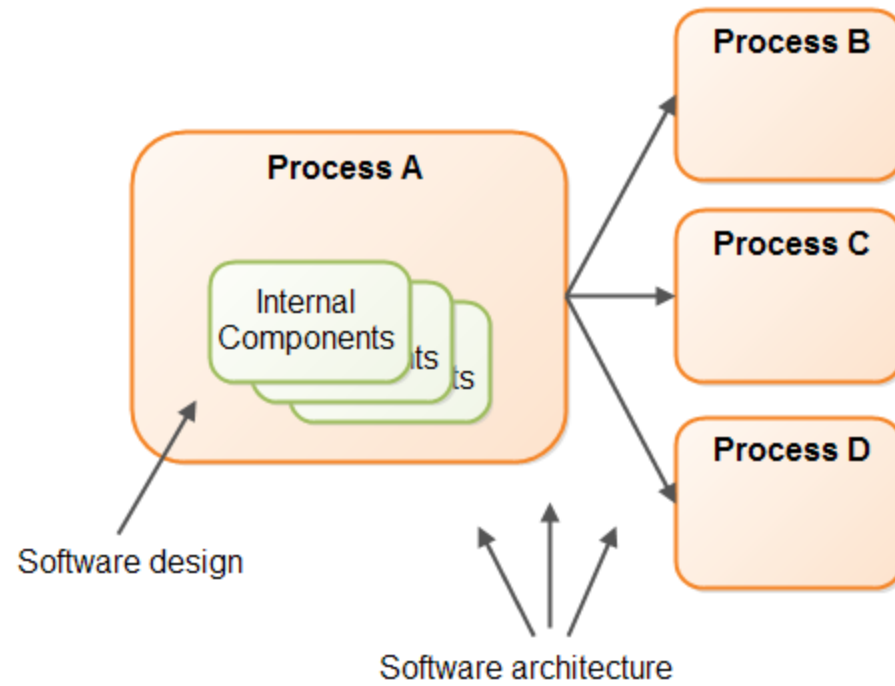
Виды архитектур



Проектирование архитектуры

- Проектирование архитектуры программного обеспечения включает принятие решений для удовлетворения **функциональных требований, качественных атрибутов и ограничений**.
- Программный дизайн архитектуры включает в себя **определение структур, составляющих решение, и их документирование**.
- Структуры системы программного обеспечения состоят из **элементов и взаимосвязей между элементами**.

Архитектура





Атрибуты качества



Требования

- Функциональные (**что** нужно сделать)
- Нефункциональные (**как** нужно сделать)

Атрибуты качества

- Это свойства программной системы и подмножество ее **нефункциональных требований**.
- Должны быть **измеримыми** и **проверяемыми**.
- Могут оказать **существенное влияние** на проектирование архитектуры, поэтому они представляют большой интерес для архитекторы программного обеспечения.

Пример:

«Система должна быть доступна все время в режиме 24/7»

Взаимное влияние атрибутов качества

- При проектировании вашей архитектуры необходимо понимать, что атрибуты качества программного обеспечения **могут влиять друг на друга**. Важно определить потенциальные конфликты между качественными атрибутами.
- **Приоритет каждого атрибута качества** будет фактором в вашем общем дизайне.



Атрибуты качества могут быть
внутренними или внешними.

Внутренние атрибуты качества

- Могут быть измерены самой системой программного обеспечения и видны команде разработчиков.
- Примерами внутренних атрибутов качества являются аспекты системы программного обеспечения, такие как **уровень согласованности классов, удобочитаемость кода** и **степень связи между модулями**.
- Эти атрибуты отражают **сложность системы** программного обеспечения.
- Хотя **внутренние атрибуты** качества не видны пользователям напрямую, они влияют на **внешние атрибуты качества**. Более высокий уровень внутреннего качества часто приводит к большему уровню внешнего качества.

Внешние атрибуты качества

- Это свойства, которые **видны снаружи**; следовательно, они заметны для конечных пользователей.
- Что бы их померить – нужна рабочая версия системы.
- Примерами внешних атрибутов качества являются **производительность, надежность, доступность и удобство использования системы.**

А какие архитектурные качества вообще бывают?

accessibility	accountability	accuracy	adaptability	administrability
affordability	agility	auditability	autonomy	availability
compatibility	composability	configurability	correctness	credibility
customizability	debugability	degradability	determinability	demonstrability
dependability	deployability	discoverability	distributability	durability
effectiveness	efficiency	usability	extensibility	failure transparency
fault tolerance	fidelity	flexibility	inspectability	installability
integrity	interoperability	learnability	maintainability	manageability
mobility	modifiability	modularity	operability	orthogonality
portability	precision	predictability	process capabilities	producibility
provability	recoverability	relevance	reliability	repeatability
reproducibility	resilience	responsiveness	reusability	robustness
safety	scalability	seamlessness	self-sustainability	serviceability
securability	simplicity	stability	standards compliance	survivability
sustainability	tailorability	testability	timeliness	traceability

Процесс проектирования

Два способа проектирования архитектуры

1. Сверху вниз **top-down**
2. Снизу вверх **bottom-up**

An aerial, top-down view of a roundabout in a city. The roundabout has a central green area with a white star-shaped pattern. Six roads radiate from the center, and the surrounding area is filled with dense urban buildings and streets. The text "Top Down" is overlaid on the left side of the image.

Top Down

Проектирование сверху-вниз (top-down)

- **Начинается со всей системы** на самом высоком уровне, а затем процесс декомпозиции начинает работать вниз к более детальным деталям.
- Отправной точкой является **высший уровень абстракции**. По мере разложения дизайн становится более детальным, пока не будет достигнут уровень компонента.
- Одной из целей такого дизайна является определение **интерфейсов компонентов**.
- Выполняется **итеративно**.

Этапы создания архитектуры

1. Определение архитектурных целей
2. Архитектурно-важные сценарии
3. Создания описания приложения
4. Определение основных вопросов
5. Выбор решения – кандидата

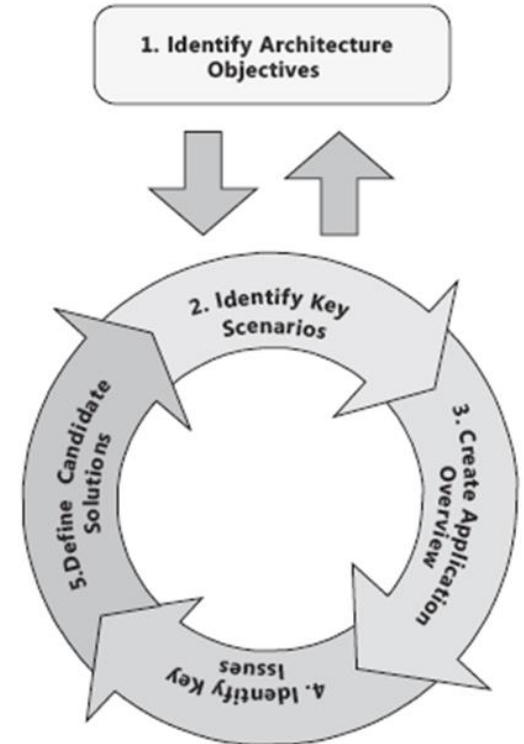


Figure 1
The iterative steps for core architecture design activities

1

Определение архитектурных целей

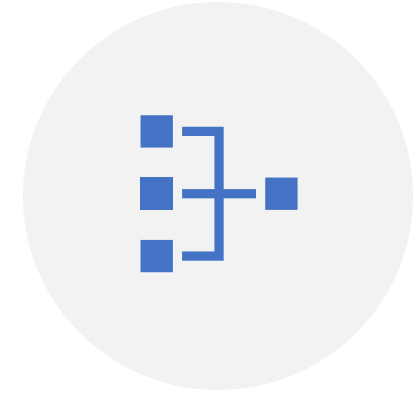
- **Определяем цели данной итерации**
Возможны разные уровни полноты/детализации
 - Создание полного дизайна приложения
 - Создание прототипа
 - Идентификация ключевых технических рисков
 - Тестирование потенциальных решений
 - Создание модели для понимания системы
- **Определяем, кто будет читателем архитектуры**
Архитектура может использоваться другими архитекторами, разработчиками, тестировщиками, руководителями.
- **Определяем ограничения**



СПЕЦИФИКА КОМАНДЫ
(ЗНАНИЯ ЛЮДЕЙ)



СПЕЦИФИКА ПРОДУКТА
(ОГРАНИЧЕНИЯ ПО СРОКАМ И
РЕСУРСАМ)



СПЕЦИФИКА ОКРУЖЕНИЯ
(СТАНДАРТНЫЕ ИНТЕГРАЦИИ)

Помним про ограничения

2 Архитектурно- важный сценарий

- **Вариант использования должен:**
 - Описывать важную не исследованную область или область с большими рисками
 - Описывает использование атрибутов качества системы
 - Описывает пересечение атрибутов качества системы
- **Архитектурно-важный сценарий**
 - Критический для бизнеса
Т.е. имеет важность как в достижении основных целей пользователей или спонсоров проекта.
 - Имеет большое влияние на архитектуру
Сценарии которые задействованы как в реализации важной бизнес-функциональности, так и в реализации атрибутов качества.

3

Создания описания приложения

- Определяем **архитектурный стиль** (см. дальше)
 - Layered-application
 - Microkernel
 - Microservices
 -
- Определение подходящих технологий
- Создаем описание с использованием принятых в компании инструментов
 - Archimate (Archi, Sparx Enterprise Architect)
 - UML (Sparx Enterprise Architect, PlantUML)
 - BPMN (Sparx Enterprise Architect, Camunda)
 - C4 (Draw.io, Camunda)

4

Формулировка вопросов к решению

Определяем основные области в которых сконцентрированы риски для разрабатываемой архитектуры. В основном они связаны с новыми технологиями и критическими бизнес-требованиями.

- Атрибуты качества

- Производительность
- Надежность
- Безопасность
- Тестируемость
- Модифицируемость
- Usability
- ...

- Инфраструктурные варианты использования

- Аутентификация и авторизация
- Кэширование
- Коммуникации
- Управление конфигурациями
- Обработка исключительных ситуаций
- Логирование и диагностика
- Валидация данных
- ..

Пример проблемных областей для безопасности

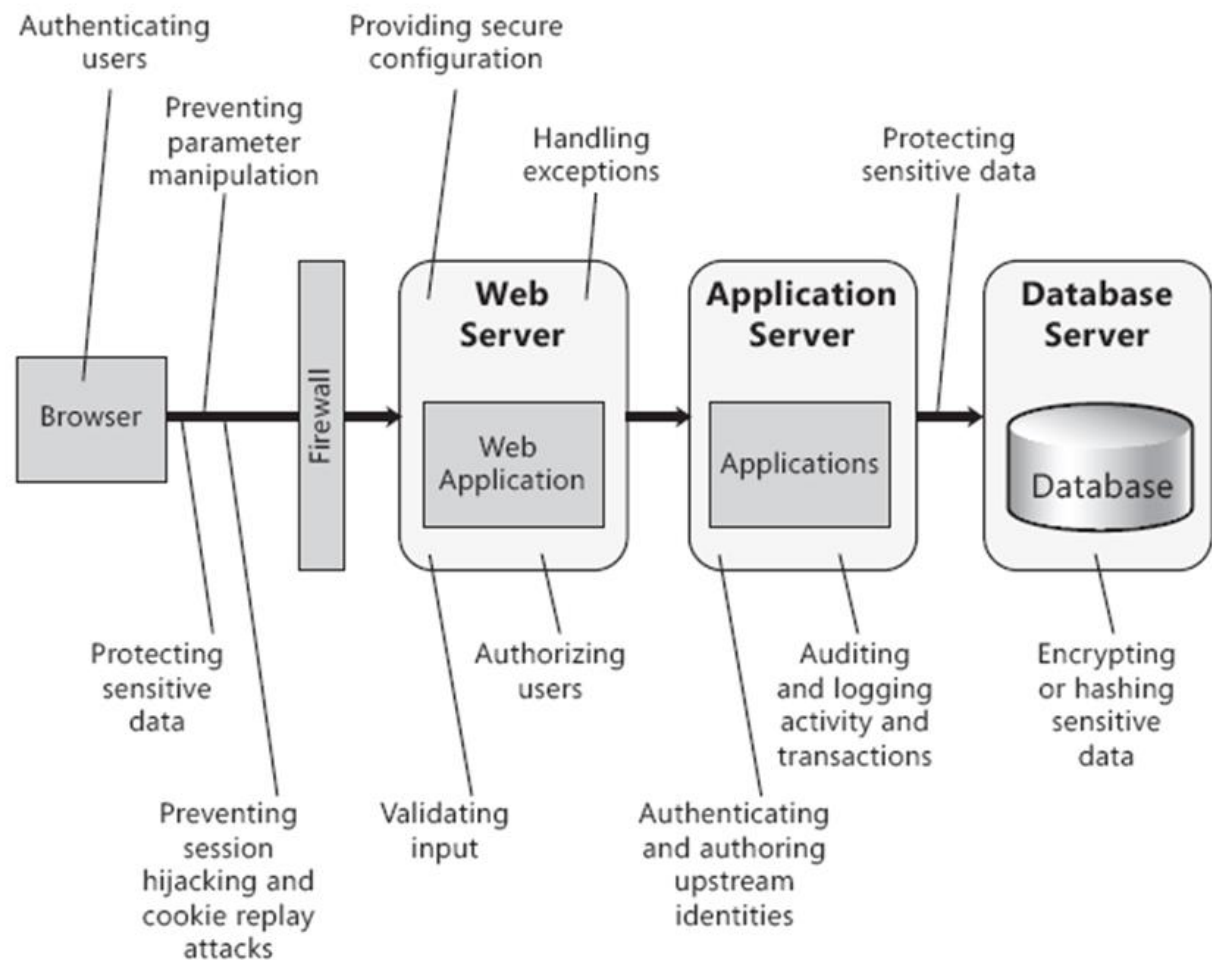


Figure 3

Security issues identified in a typical Web application architecture.

5 Выбор решения – кандидата

Для тестирования кандидата, нужно найти ответы на следующие вопросы:

- Стало ли меньше рисков?
- Новая архитектура покрывает больше требований чем предыдущая?
- Новая архитектура поддерживает архитектурно-важные варианты использования?
- Новая архитектура решает поставленные задачи по достижению атрибутов качества системы?
- Новая архитектура реализовывает дополнительные инфраструктурные варианты использования?

Недостатки top-down подхода

- Строго нисходящий подход сопряжен с риском создания **слишком больших сложных проектов**, которые могут не учитывать сложных деталей.
- Программное обеспечение является сложным, и может быть **сложно создать всю архитектуру заранее**.
- Недостатки проекта или отсутствующие функциональные возможности в архитектуре могут быть не обнаружены до тех пор, **пока проект не будет закодирован**.
- Если над проектом работают несколько групп, каждая из которых отвечает за определенную подсистему или модуль, обмен и **повторное использование знаний может быть затруднено** при таком подходе.
- Если вы используете подход сверху вниз, будьте осторожны, чтобы не стать архитектором **башни из слоновой кости**. Если вы спроектируете более высокие уровни архитектуры, а затем передадите их разработчикам для выполнения детального проектирования более низкого уровня, архитектуры могут разойтись.



Bottom-Up

Проектирование bottom-up

- В отличие от подхода «сверху вниз», подход «снизу вверх» **начинается с компонент**, которые необходимы для решения, а затем проект работает на более высоких уровнях абстракции.
- Различные компоненты могут затем использоваться вместе, как **строительные блоки**, для создания других компонентов и, в конечном итоге, более крупных структур.
- Процесс продолжается до тех пор, пока не будут выполнены все требования
- В отличие от нисходящего подхода, который начинается с высокоуровневой структуры, нет предварительного проектирования архитектуры с нисходящим подходом. **Архитектура появляется по мере завершения работы.**

Преимущества подхода «снизу-вверх»

- Одним из преимуществ подхода «снизу вверх» является **большой уровень простоты**. Команда должна сосредоточиться только на отдельных частях и создавать только то, что нужно для конкретной итерации.
- Этот подход хорошо **работает с методологиями гибкой разработки**. Благодаря итеративному подходу, который обрабатывает изменения, может осуществляться рефакторинг для добавления новых функциональных возможностей или для изменения существующих функциональных возможностей.
- Каждая итерация заканчивается рабочей версией программного обеспечения до тех пор, пока не будет построена вся система.
- Такой подход **облегчает повторное использование кода**. Поскольку в каждый конкретный момент команда сосредоточена на ограниченном количестве компонентов, распознавание возможностей для повторного использования становится проще.

Недостатки подхода «снизу-вверх»

- Подход снизу вверх предполагает, что изменения дешевы. Однако, в зависимости от характера изменений, рефакторинг архитектуры программного обеспечения может быть очень **дорогостоящим**.
- Подход «снизу вверх» без исходной архитектуры может привести к **снижению качества** архитектуры системы в целом.
- Такой подход **затрудняет планирование** и оценку всего проекта, что может быть неприемлемым для корпоративного программного обеспечения.
- Одним из недостатков нисходящего подхода является то, что **недостатки проекта могут быть обнаружены только позже**, что приведет к дорогостоящему рефакторингу.

Top-down

- ✓ Проект большой/сложный
- ✓ Заказное программное обеспечение
- ✓ Команда большая, или есть несколько команд, которые будут работать над проектом
- ✓ Домен понятен

Bottom-up

- ✓ Проект небольшой по размеру/не сложный
- ✓ Команда небольшая, или есть только одна команда
- ✓ Домен не совсем понятен или требования нестабильны/меняются

Что выбрать?

Стили описания архитектуры



Architecture Style

«A coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style.»

Architectural Styles and the Design of Network-Based Software Architectures, by Roy Fielding; published by the University of California; refer to: www.ics.uci.edu/fielding/pubs/dissertation/fielding_dissertation.pdf



- Монолитные архитектуры
- Распределенные архитектуры

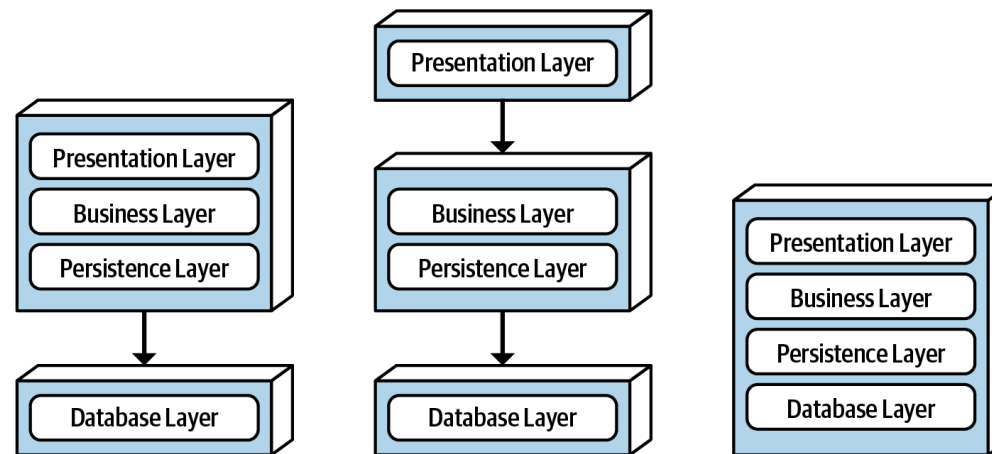


Монолитные архитектуры

- Layered architecture
- Pipeline architecture
- Microkernel architecture

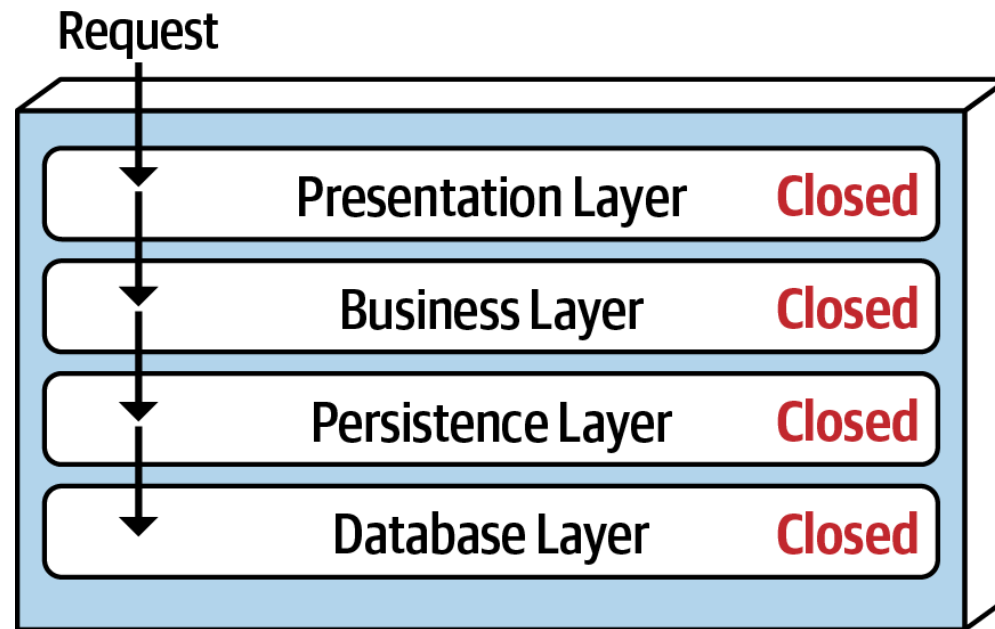
Layered architecture

- Компоненты разделяются на абстрактные слои
- Устанавливаются правила взаимодействия между слоями: в общем случае взаимодействуют только смежные слои
- Подходит для небольших приложений

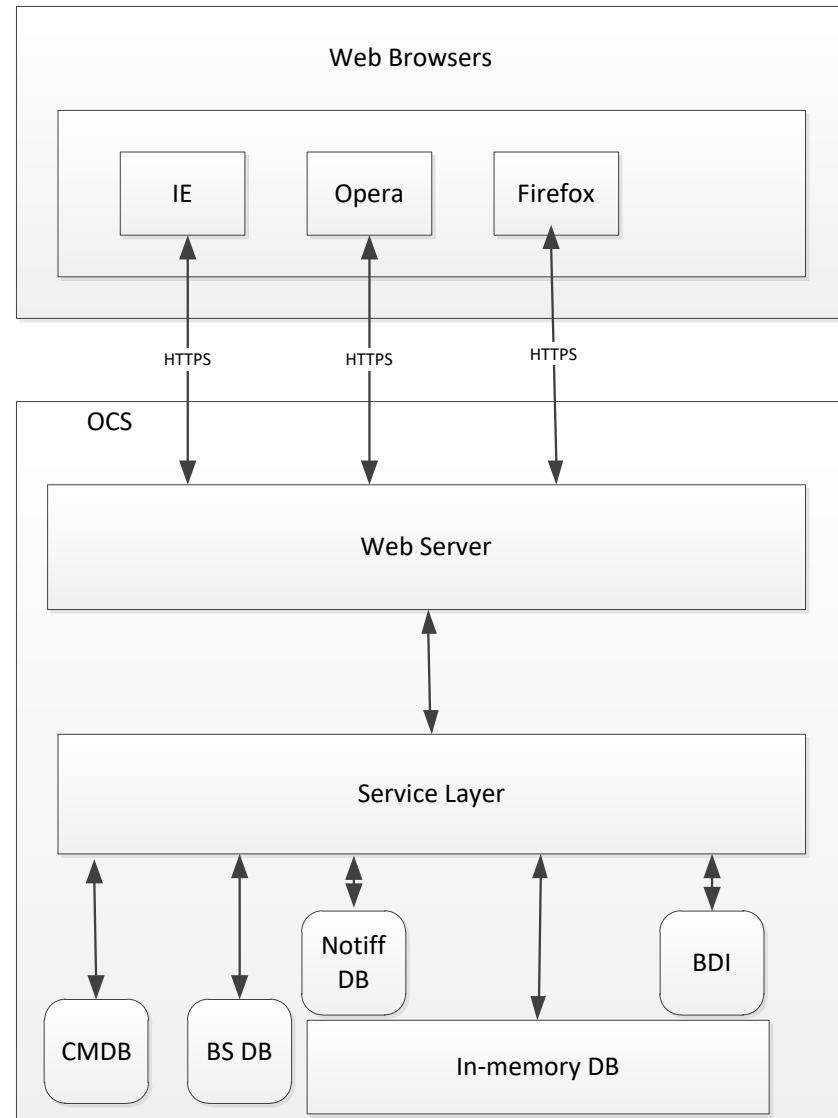


Layered architecture

- Слои могут быть либо «закрытыми» либо «открытыми»
- К закрытым слоям может быть обращение только из соседних.
- Что делать если мне на уровне представления понадобится отобразить большой объем данных?

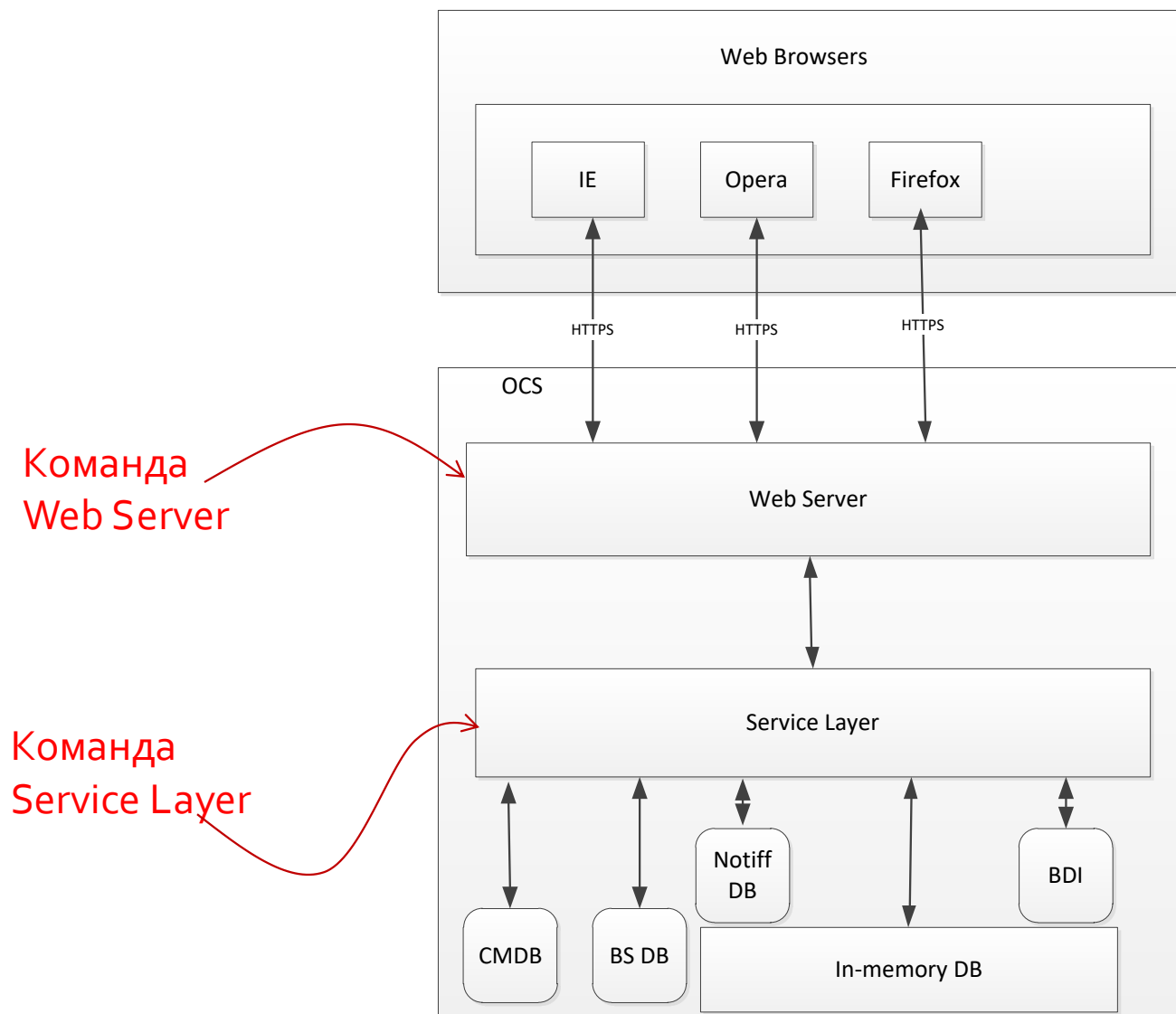


пример



Проблема:

разделение зон
ответственности
и по функциям
а не по
ценности

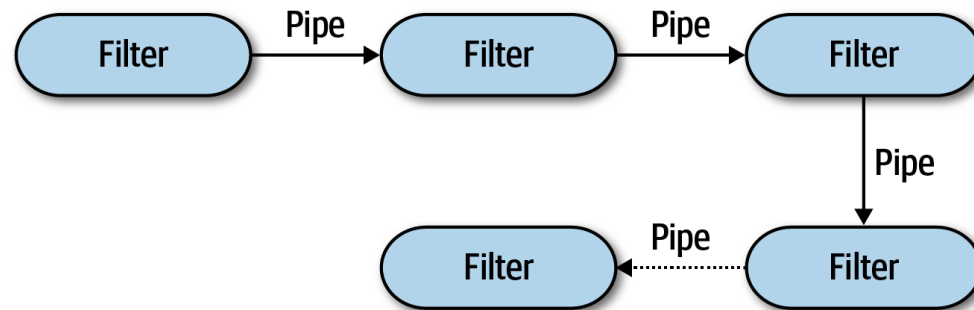


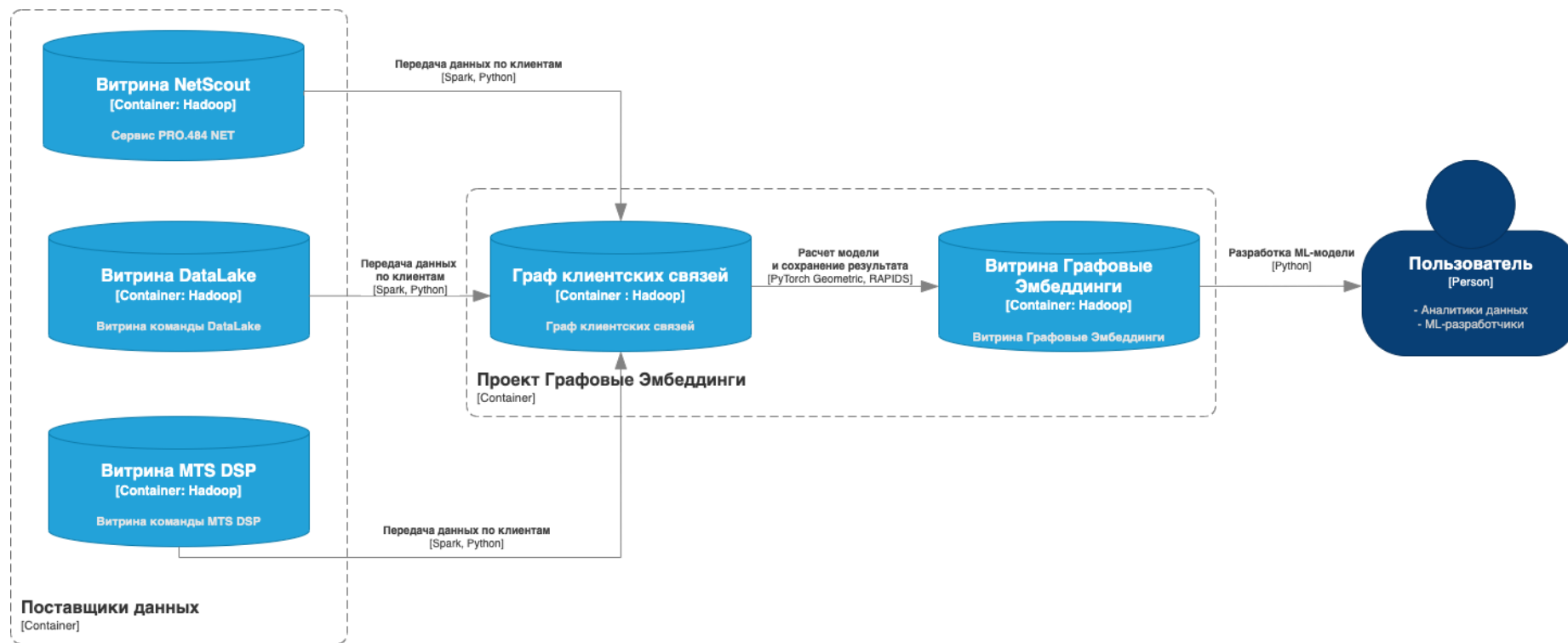
Когда
применять?

Layered architecture хороша для небольших приложений или при старте разработке больших приложений (с последующим рефакторингом)

Pipeline architecture

- Статическая конфигурация элементов
- Интеграция точка-точка
- Управление передается вместе с данными
- Filter – обработчики, Pipe – каналы передачи





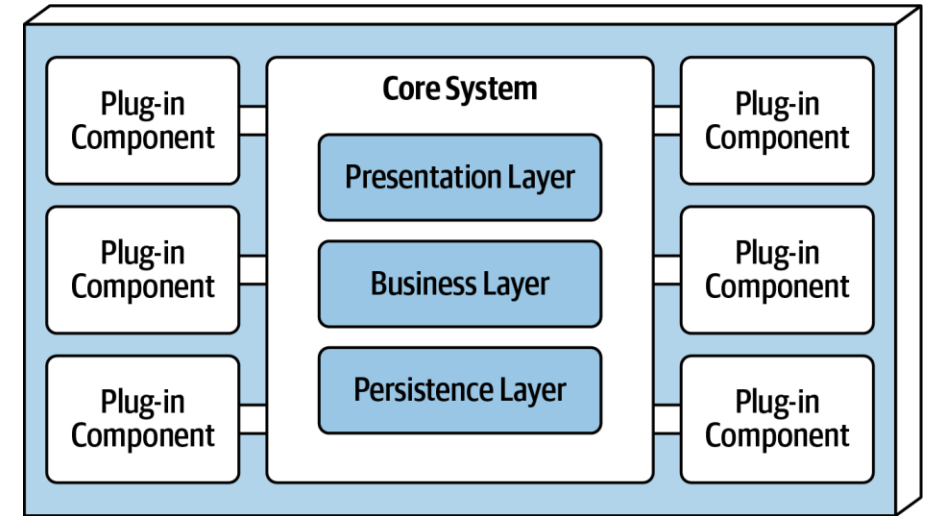
Пример

Когда применять?

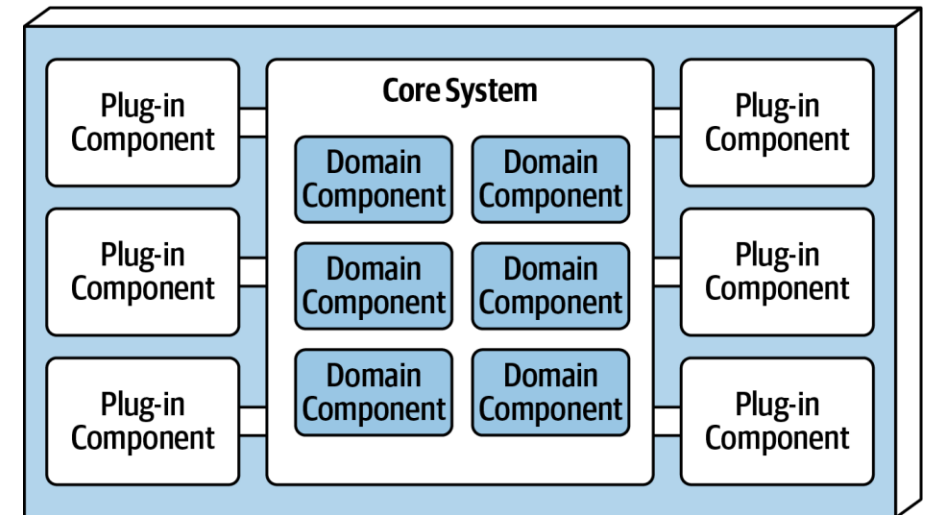
- Есть predetermined алгоритм трансформации и передачи данных между системами.
- Система предназначена для конвейерной обработки данных (map reduce, сеть приема платежей, mediation, fraud management system....)

Microkernel architecture

- Монолитная архитектура
- Два вида компонент:
 - Ядро
 - Расширение
- Позволяет сделать расширяемый функционал

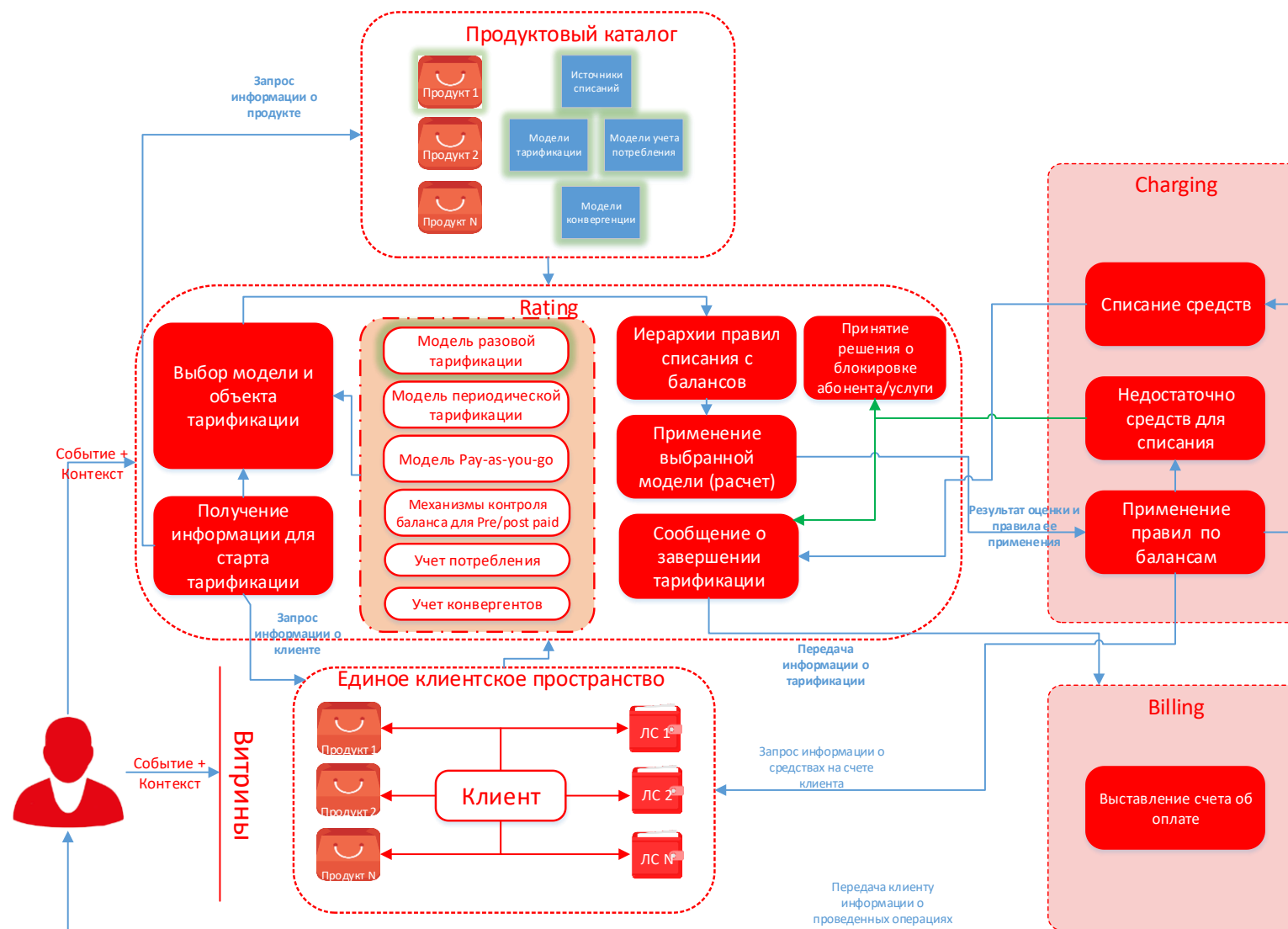


Layered Core System (Technically Partioned)




Modular Core System (Domain Partioned)

Пример



Когда применять?

- В системе есть хорошо выделяемая «общая часть» и «расширение». При этом между ними можно определить стабильный интерфейс.
- Примером таких систем является Adobe Photoshop, Eclipse IDE

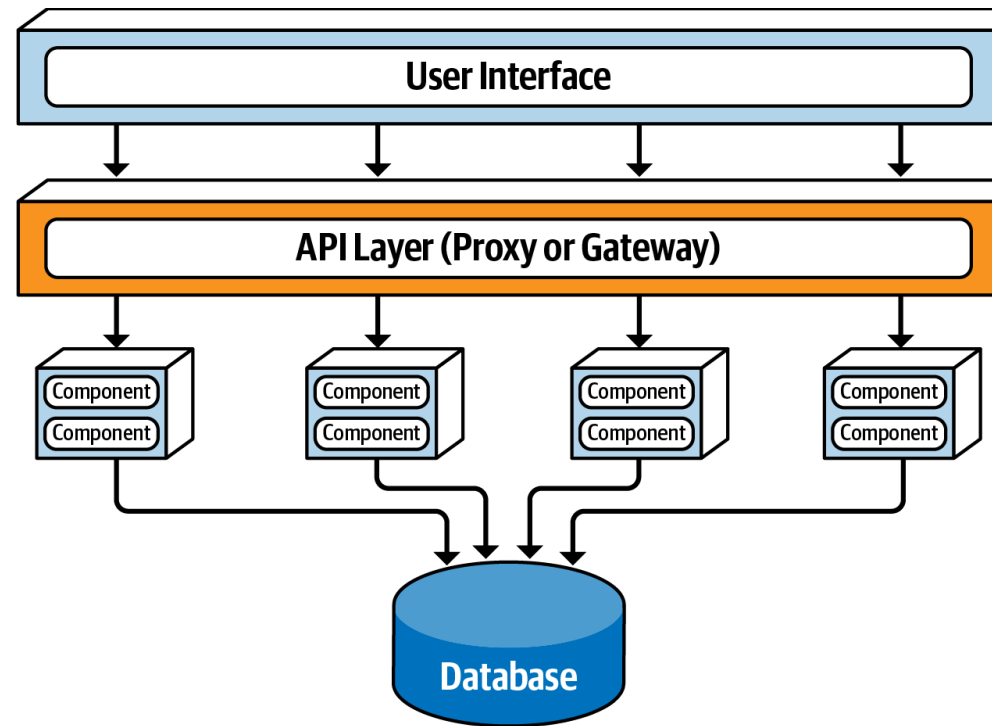


Распределенные архитектуры

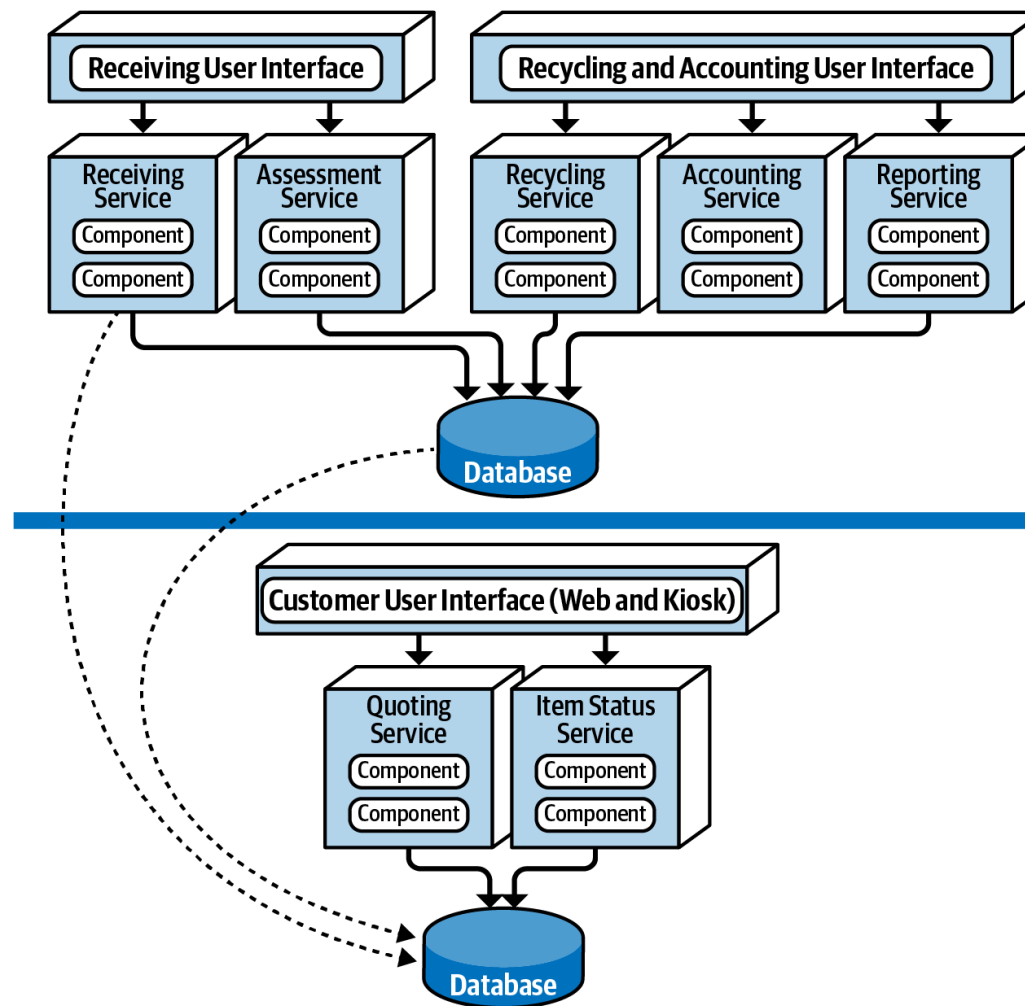
- Service-based architecture
- Event-driven architecture
- Space-based architecture
- Service-oriented architecture
- Microservices architecture

Service-based architecture

- Гибрид между монолитной и микросервисной архитектурой
- Приложение разбивается на автономные сервисы, выполняющие законченные бизнес-функции
- Единая база данных. Однако может разбиваться на части, ориентируясь на домены

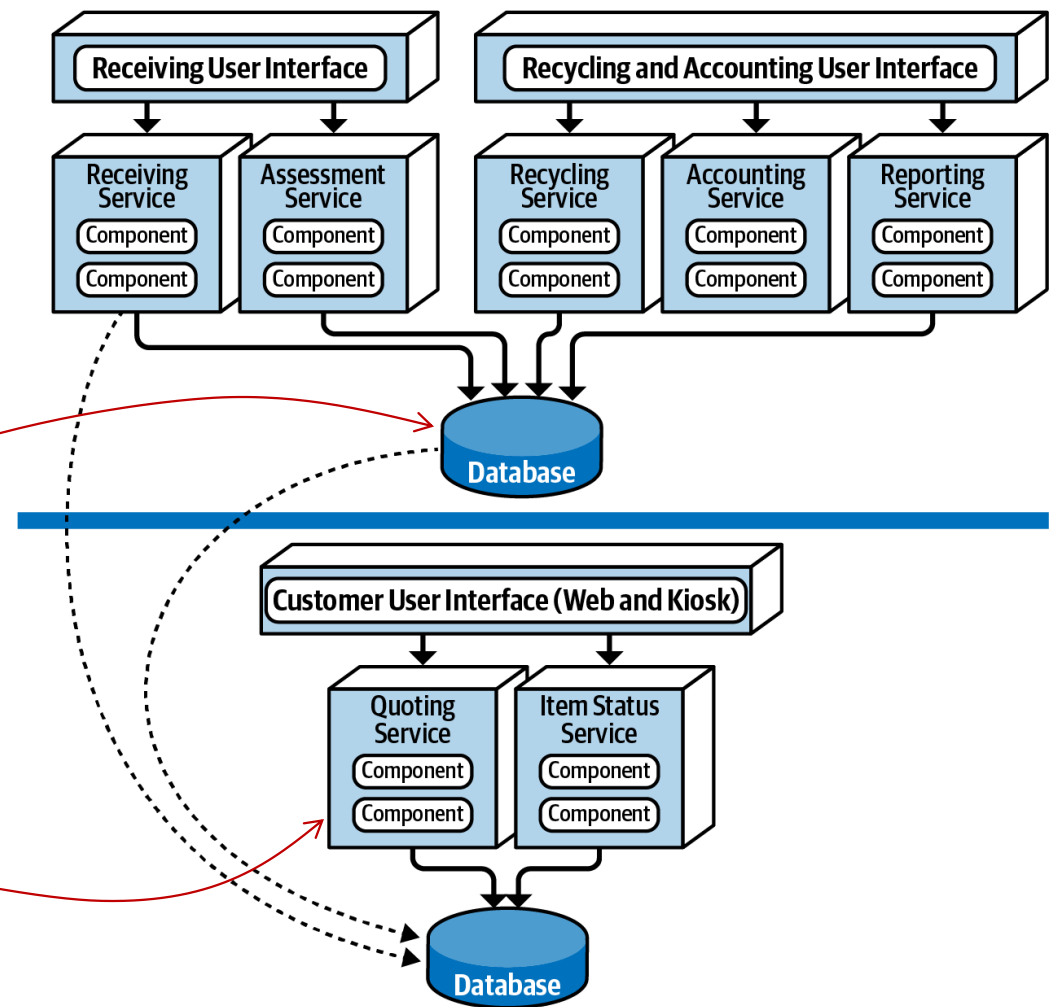


Пример



Проблемы

- Масштабирование (сложность масштабирования СУБД)
- Гранулярность сервисов (сервисы могут быть большие и сложные в доработке)

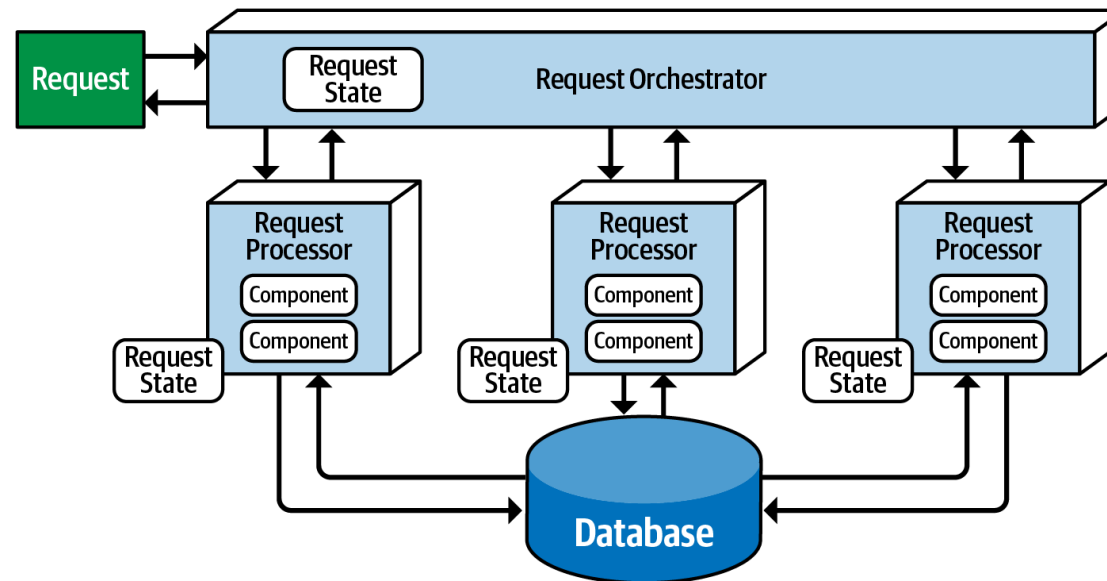


Когда применять?

- Хорошо подходит на начальных стадиях для большинства систем
- Корпоративные учетные системы, веб-порталы

Event-driven architecture

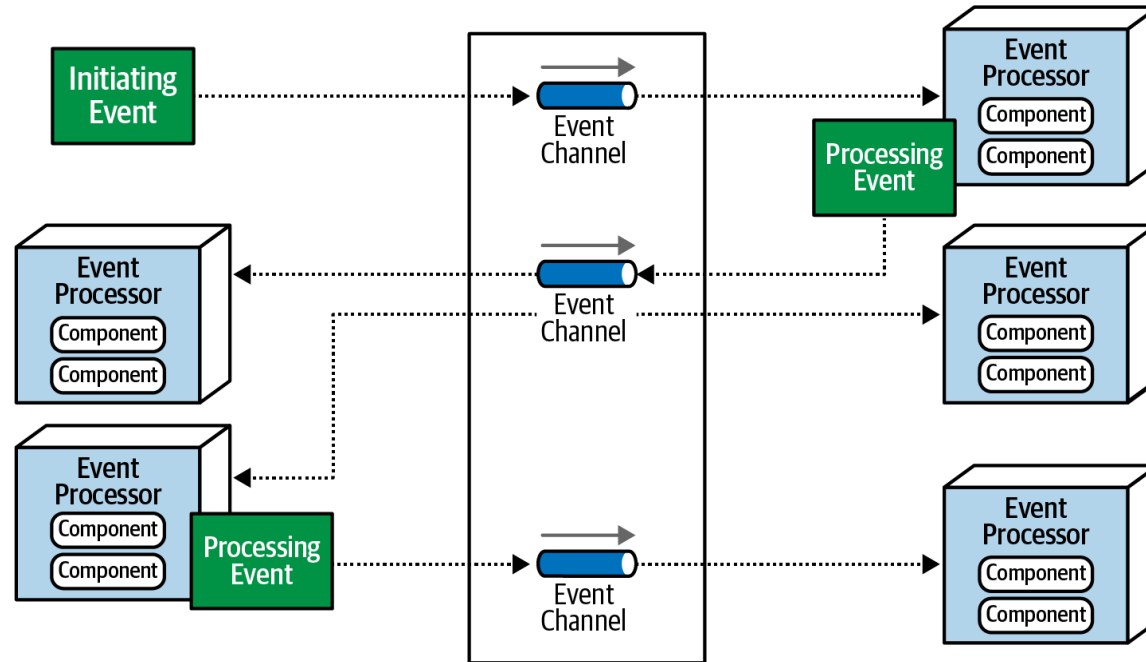
- Хорошо масштабируемая архитектура, основанная на обработке асинхронных событий
- Используется для построения высокопроизводительных приложений



Event driven architecture

1. Брокер
2. Медиатор

Event Driven Architecture: broker topology (нет единого управления)



Event Driven Architecture: broker topology

Плюсы

Слабосвязанные компоненты

Высокая масштабируемость

Высокая скорость отклика

Высокая
производительность

Высокая надежность

Минусы

Сложность контроля сквозных процессов

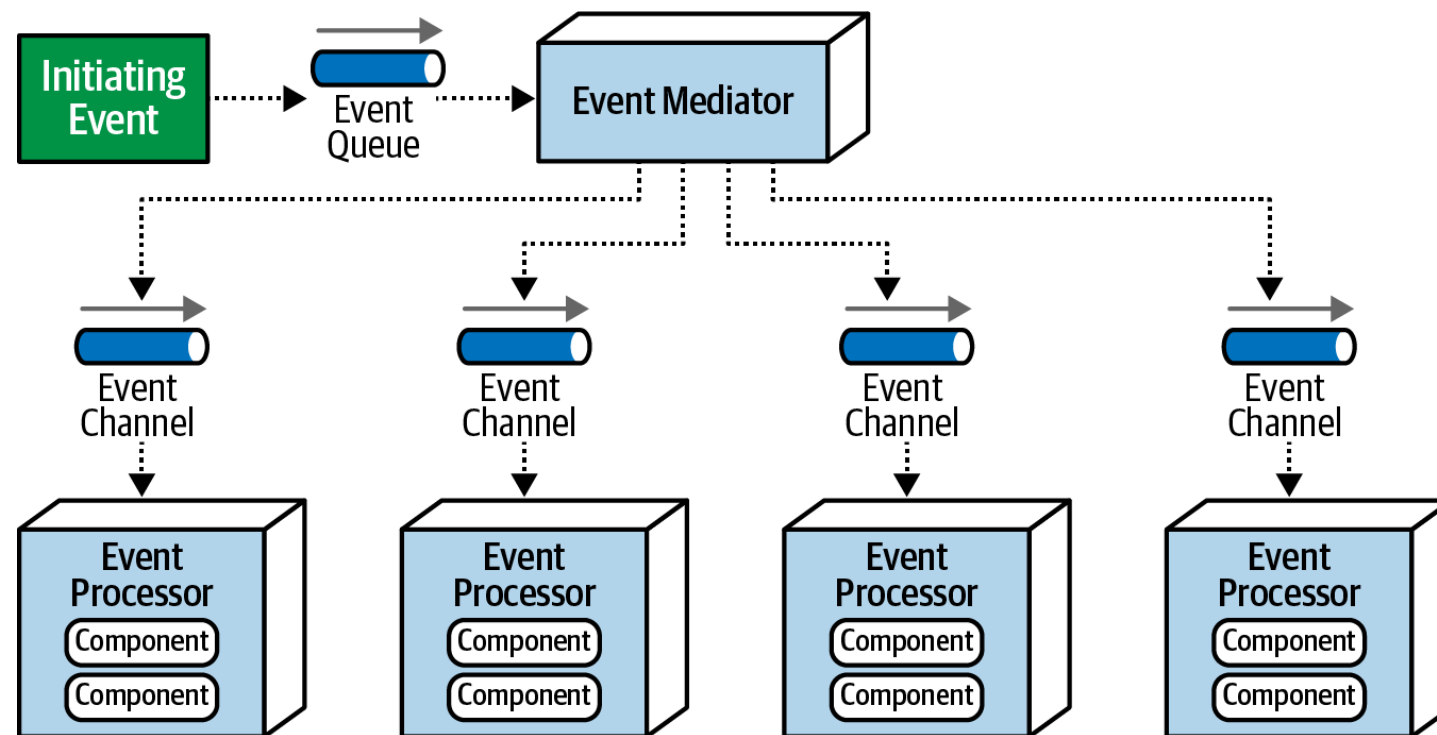
Сложность централизованной обработки
событий

Сложности в восстановлении системы

Сложность контроля состояния бизнес
транзакции и ее отката/рестарта

Сложность поддержки консистентных
данных

Event Driven Architecture медиатор



Наличие медиатора, который управляет взаимодействием
(например, <https://camel.apache.org/>)

Event Driven Architecture медиатор

Плюсы

Контроль сквозных процессов

Возможность обрабатывать ошибки

Возможность восстанавливать состояние

Возможность перезапуска бизнес-транзакций

Улучшенная целостность данных

Минусы

Связанность между обработчиками событий

Уменьшение масштабируемости

Меньшая производительность

Меньшая устойчивость к падениям

Сложность моделирования больших процессов

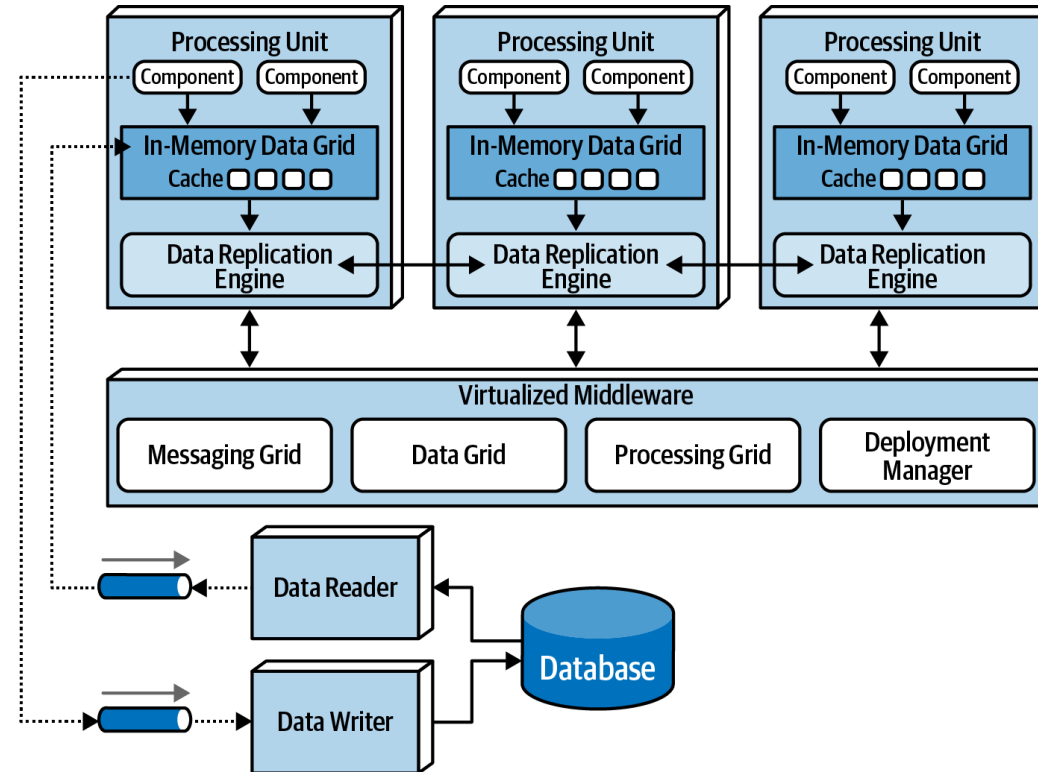
Пример



Когда применять?

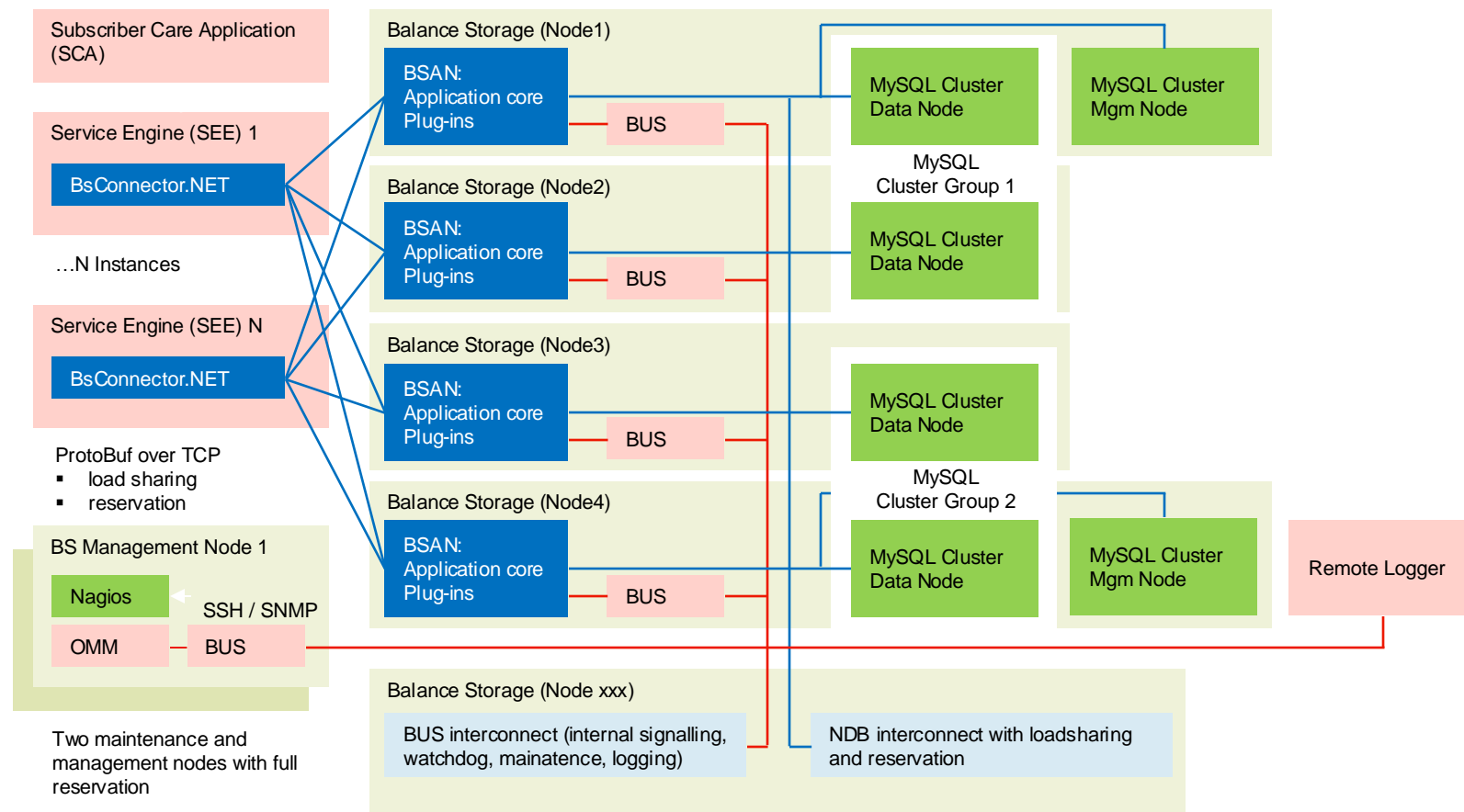
- В системах, которые завязаны на обработку внешних событий
- В системах, которые должны хорошо масштабироваться

Space-based architecture



Построена на параллельной архитектуре обработки данных с разделяемым пространством памяти

Пример: система управления балансами клиентов

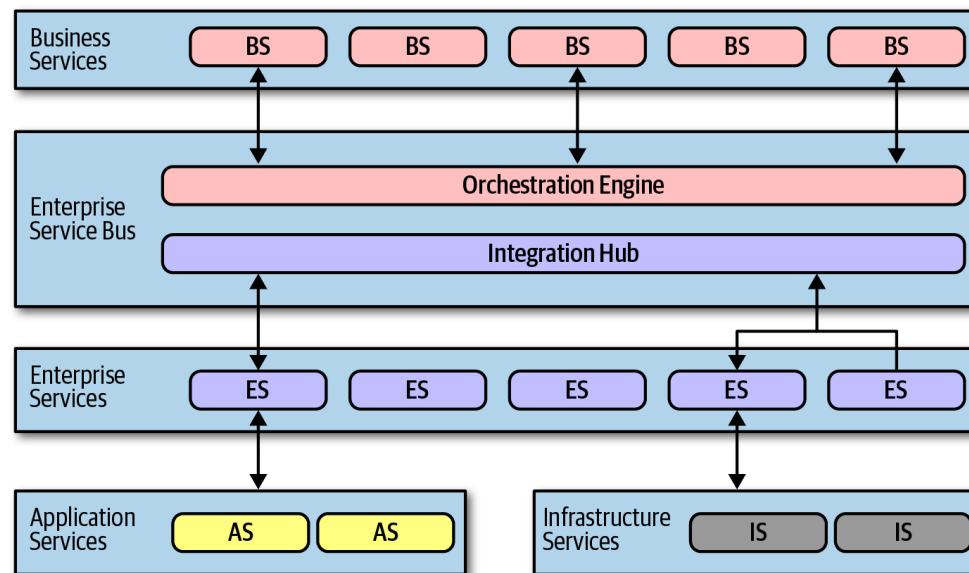


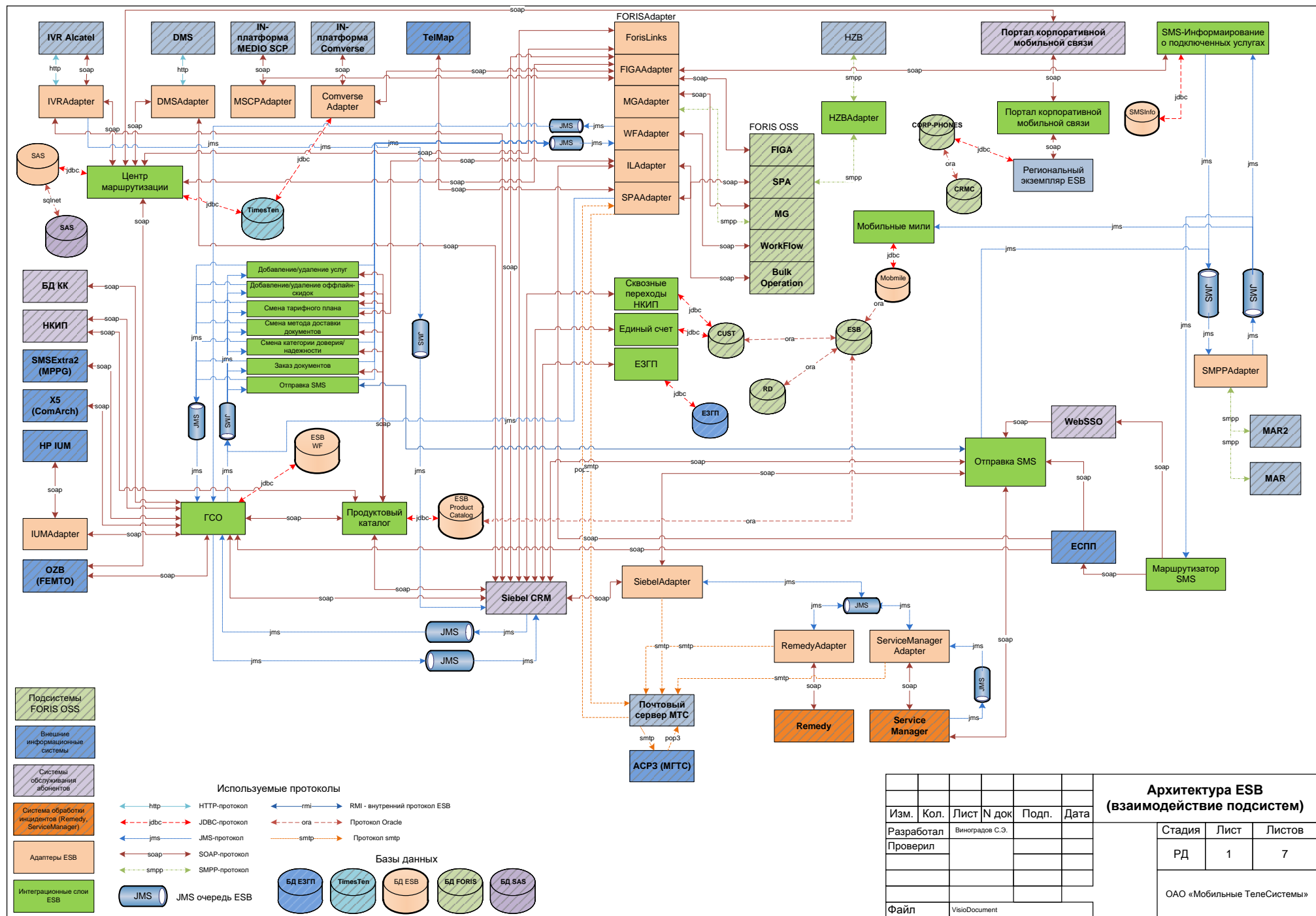
Когда применять?

- Когда требуется хранение и обработка больших массивов связанных данных

Service-oriented architecture

- Разделение сервисов на слои: бизнес-сервисы, сервисы компании и сервисы приложений
- Идея состоит в максимизации переиспользования созданной функциональности
- Выделенный «оркестратор» для сервисов – Enterprise Service Bus



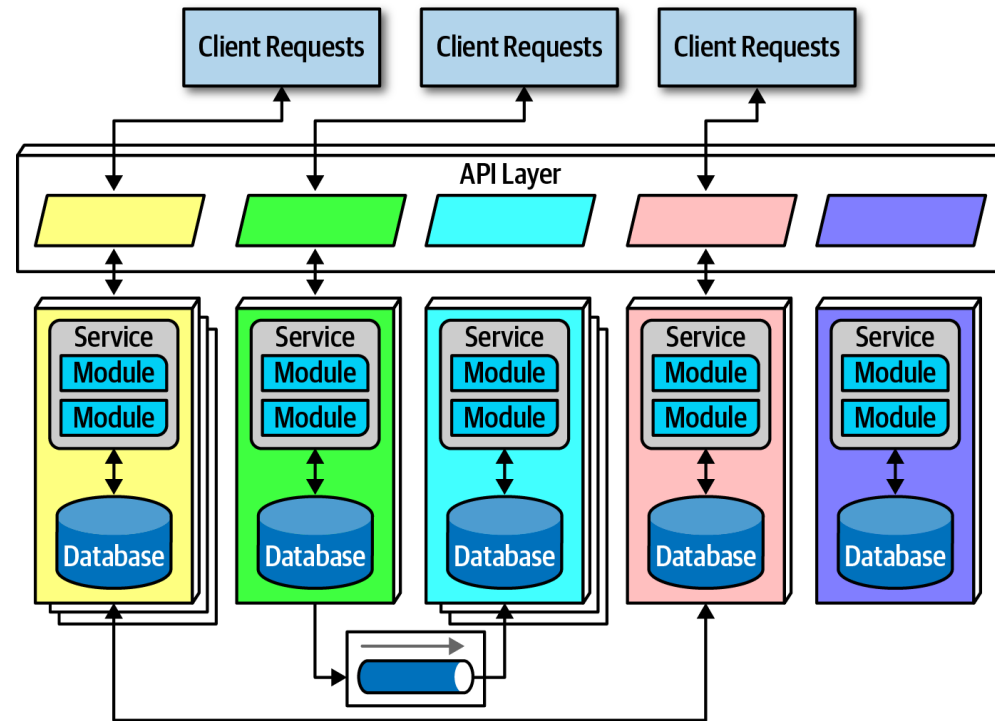


Когда применять?

- При интеграции корпоративных систем, когда требуется многократное переиспользование различных «доменных» сервисов

Microservices architecture

это способ проектирования сложного приложения, как набора небольших и независимых сервисов, которые работают, разрабатываются и деплоятся независимо друг от друга



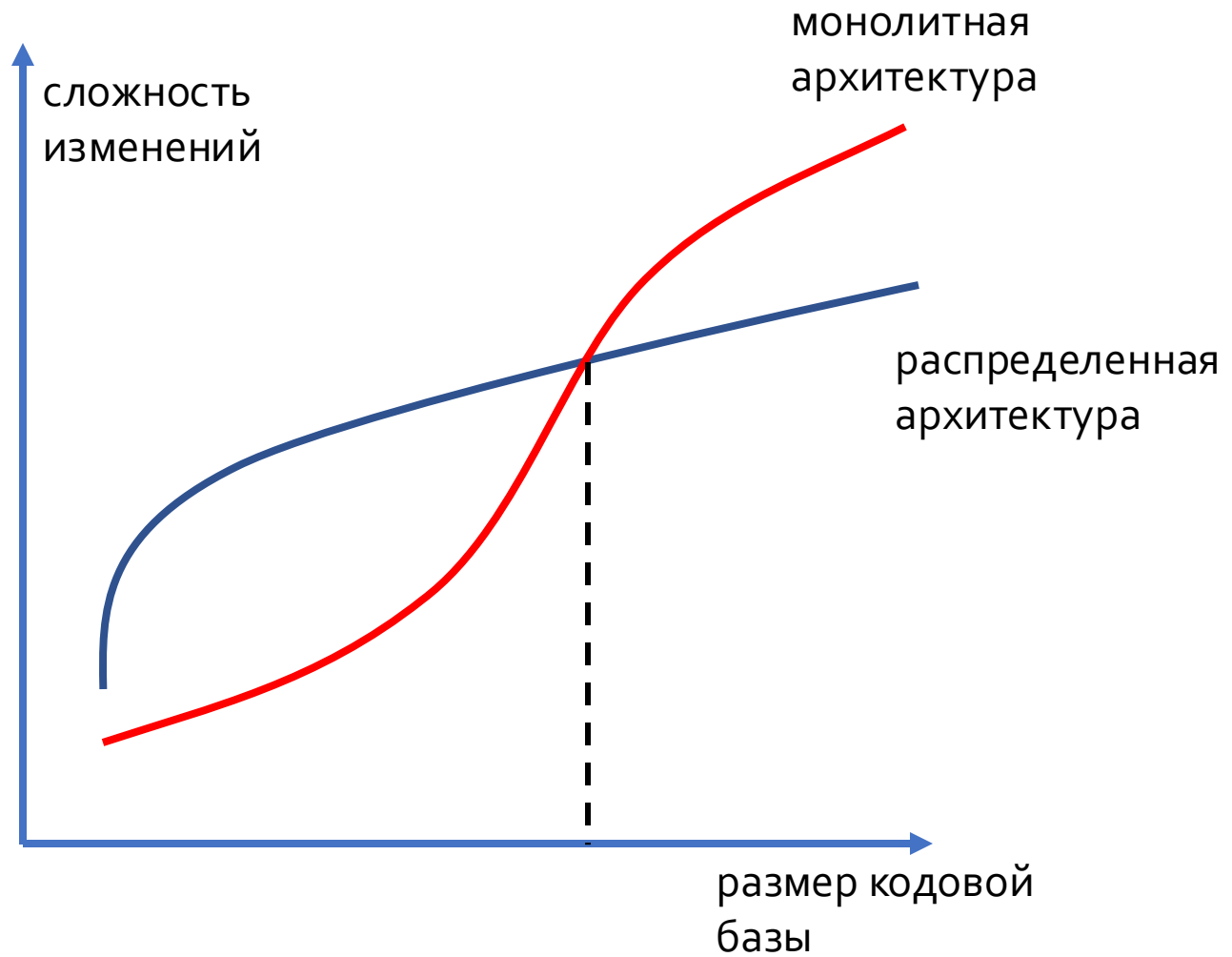
Тема отдельной лекции



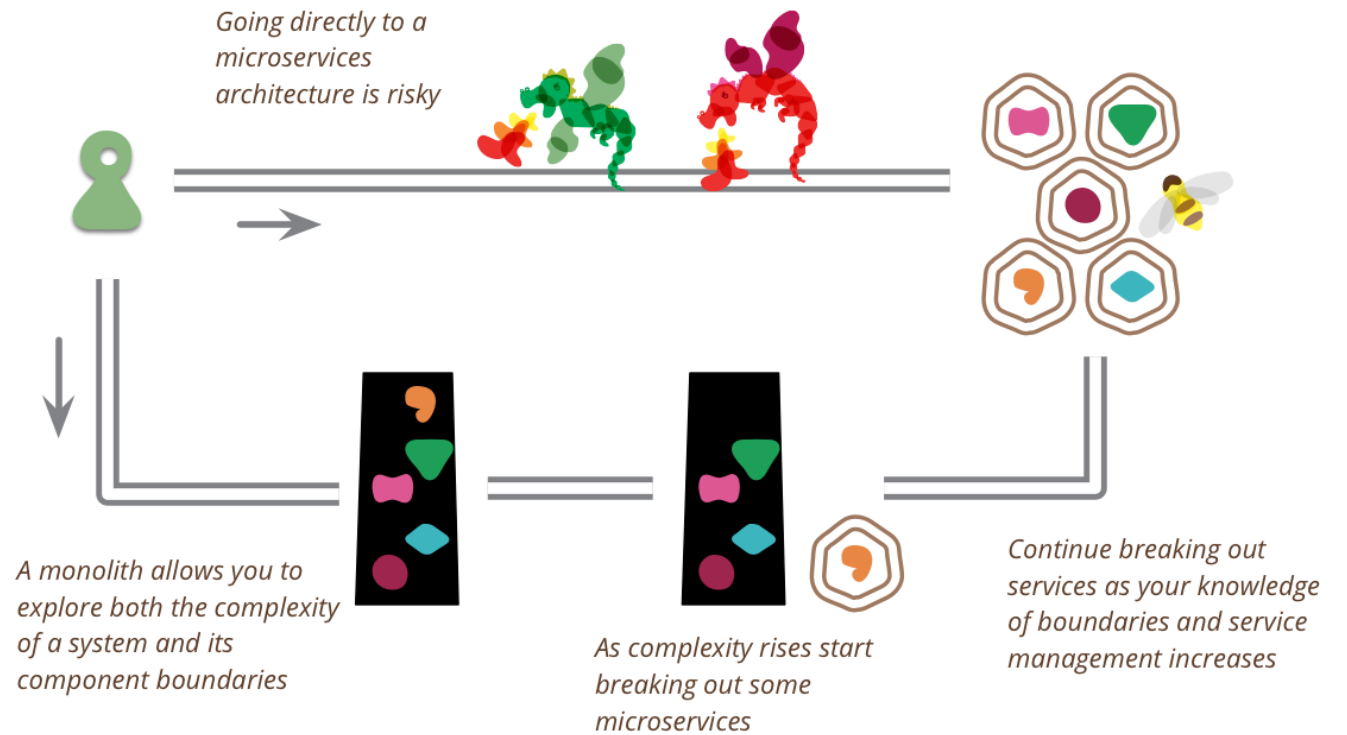
Какой стиль выбрать?

монолитная архитектура vs распределенная архитектура

Стоимость изменений



Сначала сделайте
монолит, потом
его делите его на
микросервисы



O'REILLY®

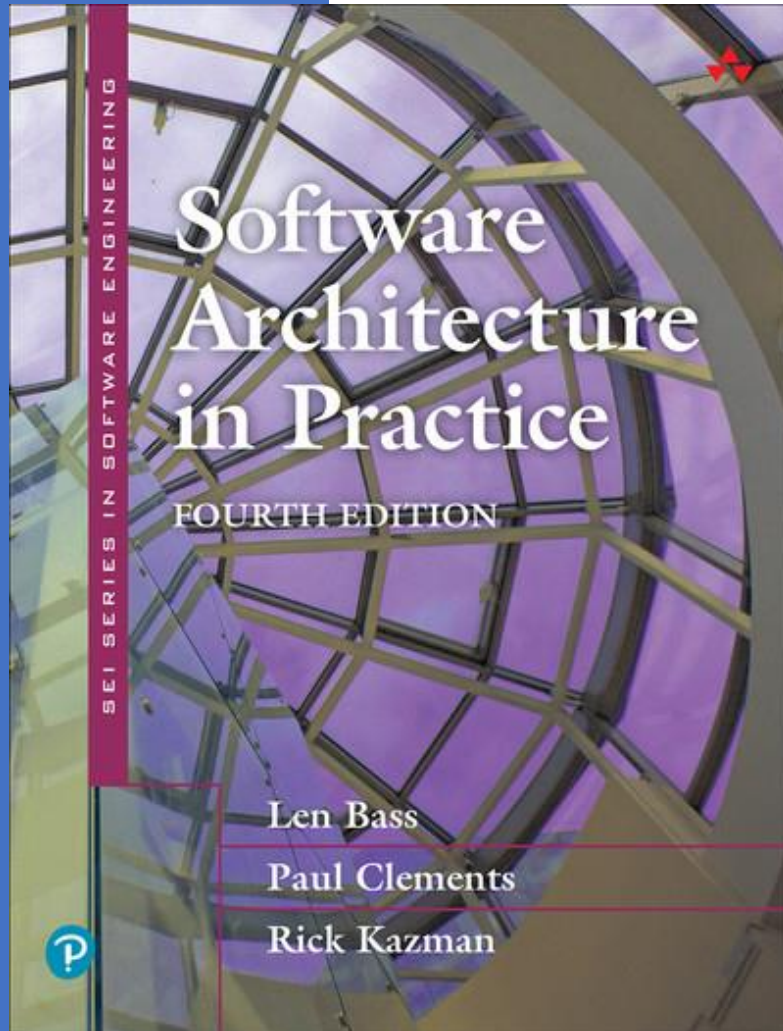


Fundamentals of Software Architecture

An Engineering Approach

Mark Richards & Neal Ford

- **Architecture patterns:** The technical basis for many architectural decisions
- **Components:** Identification, coupling, cohesion, partitioning, and granularity
- **Soft skills:** Effective team management, meetings, negotiation, presentations, and more
- **Modernity:** Engineering practices and operational approaches that have changed radically in the past few years
- **Architecture as an engineering discipline:** Repeatable results, metrics, and concrete valuations that add rigor to software architecture



- Discover how architecture influences (and is influenced by) technical environments, project lifecycles, business profiles, and your own practices
- Leverage proven patterns, interfaces, and practices for optimizing quality through architecture
- Architect for mobility, the cloud, machine learning, and quantum computing
- Design for increasingly crucial attributes such as energy efficiency and safety
- Scale systems by discovering architecturally significant influences, using DevOps and deployment pipelines, and managing architecture debt
- Understand architecture's role in the organization, so you can deliver more value

O'REILLY®

Building Evolutionary Architectures

SUPPORT CONSTANT CHANGE



Neal Ford, Rebecca Parsons & Patrick Kua

The software development ecosystem is constantly changing, providing a constant stream of new tools, frameworks, techniques, and paradigms. Over the past few years, incremental developments in core engineering practices for software development have created the foundations for rethinking how architecture changes over time, along with ways to protect important architectural characteristics as it evolves. This practical guide ties those parts together with a new way to think about architecture and time

На сегодня все