Анализ архитектурных стилей ПО

**Оглавление**

[Сводная таблица анализа архитектурных стилей 3](#_Toc149908775)

[Правила выставления оценок 4](#_Toc149908776)

[Описание характеристик 5](#_Toc149908777)

[Caliber 5](#_Toc149908778)

[Agility 5](#_Toc149908779)

[Abstraction Level 5](#_Toc149908780)

[Configurability 5](#_Toc149908781)

[Domain portioning 5](#_Toc149908782)

[Cost of implementation 6](#_Toc149908783)

[Const of ownershiping 6](#_Toc149908784)

[First Deployability 6](#_Toc149908785)

[Next Deployability 6](#_Toc149908786)

[Main-Structure Simplicity 6](#_Toc149908787)

[Component-Structure Simplicity 6](#_Toc149908788)

[Infrastructure Simplicity 6](#_Toc149908789)

[Hardware fault tolerance 6](#_Toc149908790)

[System-component fault tolerance 6](#_Toc149908791)

[Web-communication fault tolerance 6](#_Toc149908792)

[Performance 6](#_Toc149908793)

[Testability 6](#_Toc149908794)

[Technical Decomposability 7](#_Toc149908795)

[Technical Evolvability 7](#_Toc149908796)

[Elasticity 7](#_Toc149908797)

[Business Decomposability 7](#_Toc149908798)

[Business Evolvability 7](#_Toc149908799)

[Service self-Sufficiency 7](#_Toc149908800)

[Service Data Intersection 7](#_Toc149908801)

[Integration 7](#_Toc149908802)

[Interoperability 7](#_Toc149908803)

[Horizontal Scaling Possibility of Computing-Resources 8](#_Toc149908804)

[Horizontal Scaling Possibility of Functionality 8](#_Toc149908805)

[Scalability 8](#_Toc149908806)

[Подробный разбор архитектурных стилей 9](#_Toc149908807)

[Стили ориентированные на решение технических задач (Caliber 1–5) 9](#_Toc149908808)

[Монолит 9](#_Toc149908809)

[Модульный Монолит 11](#_Toc149908810)

[Microkernel 12](#_Toc149908811)

[Стили ориентированные на решение бизнес-задач (Caliber 6–10) 14](#_Toc149908812)

[Service-Oriented Architecture 14](#_Toc149908813)

[Service-Based 15](#_Toc149908814)

[Space-Based 18](#_Toc149908815)

[Event-Driven 21](#_Toc149908816)

[Microservices 23](#_Toc149908817)

# Сводная таблица анализа архитектурных стилей[[1]](#footnote-1)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| [Caliber](#_Caliber) | [Agility](#_Agility) | [Abstraction Level](#_Abstraction_Level) | [Configurability](#_Configurability) | [Domain Portioning](#_Domain_portioning) | [Const of implementation](#_Cost_of_implementation) | [Const of ownershiping](#_Const_of_ownershiping) | [First Deployability](#_First_Deployability) | [Next Deployability](#_Next_Deployability) | [Main-Structure Simplicity](#_Main-Structure_Simplicity) | [Component-Structure Simplicity](#_Component-Structure_Simplicity) | [Infrastructure Simplicity](#_Infrastructure_Simplicity) | [Hardware fault tolerance](#_Hardware_fault_tolerance) | [System-component fault tolerance](#_System_component_fault) | [Web-communication fault tolerance](#_Web-communication_fault_tolerance) | [Performance](#_Performance) | [Testability](#_Testability) | [Technical Decomposability](#_Technical_Decomposability) | [Technical Evolvability](#_Technical_evolvability) | [Elasticity](#_Elasticity) | [Business Decomposability](#_Business_Decomposability) | [Business Evolvability](#_Business_Evolvability) | [Service self-Sufficiency](#_Service_self-Sufficiency) | [Service Data Intersection](#_Service_Data_Intersection) | [Integration](#_Integration) | [Interoperability](#_Interoperability) | [HSP of Computing-Resources](#_Horizontal_Computing-Resources_Scal) | [HSP of Functionality](#_Horizontal_Functionality_Scaling) |  |
| **1-5** | [**Technical-Task Oriented Architecture Styles**](#_Стили_ориентированные_на) | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1-2 | [Monolithic](#_Монолит) | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|  | 2 | 2 | 2 | 1 | 9 | 9 | 9 | 9 | 9 | 2-9 | 9 | 2 | 2 | 9 | 10 | 9 | 2 | 2 | 2 |  |  |  |  |  |  |  |  |  |
| 1-4 | [Modular Monolithic](#_Модульный_Монолит) | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|  | 3 | 3-4 | 3-4 | 2-4 | 9 | 8-9 | 9 | 9 | 8 | 4-6 | 9 | 2 | 1 | 9 | 9 | 8-9 | 10 | 3 | 2-5 |  |  |  |  |  |  |  |  |  |
| **5-10+** | [**Business-Task Oriented Architecture Styles**](#_Стили_ориентированные_на_1) | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 4-6 | [Microkernel](#_Microkernel) | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|  | 4 | 4-9 | 4-9 | 10 | 5-6 | 7-9 | 7-9 | 7-9 | 4-6 | 4-6 | 8-9 | 1 | 3-9 | 9 | 6-8 | 5-7 | 6-8 | 5-9 | 5-9 |  |  |  |  |  |  |  |  |  |
| **5-10+** | [**Service Oriented Architecture Styles**](#_Service-Oriented_Architecture) | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 4-6 | [Service-Based](#_Service-Based) | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|  | 4 | 4-9 | 4-9 | 10 | 4-7 | 4-7 | 2-6 | 2 | 4-6 | 4-6 | 2-7 | 2-9 | 3-9 | 2-5 | 4-6 | 2 |  |  |  | 4-6 | 2-4 | 2-4 | 1 | 2-4 | 2-4 | 5-9 | 2-4 |  |
| 9-10+ | [Space-Based](#_Space-Based) | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|  | 8 | 4-9 | 10 | 9 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 5-8 | 5-8 | 5-8 | 2-8 | 2-3 |  |  |  | 7 | 7 | 8 | 8 | 5-7 | 3-5 | 6-7 | 7 |  |
| 5-9 | [Event-Driven](#_Event-Driven) | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|  | 5\* | 5\* | 5\* | 5\* | 3-4 | 3-4 | 3-4 | 3-4 | 3-5 | 3-4 | 3 | 5-8 | 5-8 | 2-8 | 3-6 | 6 |  |  |  | 5\* | 5\* | 7 | 9 | 5-6 | 6 | 9 | 7 |  |
| 5-10 | [Microservices](#_Microservices) | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|  | 10 | 9 | 9 | 9 | 4 | 2-4 | 2-4 | 7-8 | 2 | 7-8 | 5 | 9 | 9 | 8 | 1 | 3 |  |  |  | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 |  |

## Правила выставления оценок

* **таблица не содержит абсолютных оценок: все оценки относительные!**
* если по какому-либо из показателей два и более архитектурных стиля имеют одинаковые значения, значит различия незначительны;
* каждый показатель оценивается от 1 до 10;
* если показатель оценен как 9 или 10, то ему обязательно будет дано подробное описание в разделе «ключевые сильные стороны» для соответствующего архитектурного стиля;
* если показатель оценен как 1 или 2, то ему обязательно будет дано подробное описание в разделе «ключевые слабые стороны» для соответствующего архитектурного стиля;
* если показатель оценен в диапазоне от 3 до 8, то подробное описание будет дано только в случае наличия необходимых к раскрытию нюансов;
* каждый показатель может быть оценён в диапазоне (например 3-6): подобное означает, что показатель существенно зависит от области знаний и/или сферы бизнеса в котором применяется соответствующий архитектурный стиль, и/или от конкретной реализации системы:
  + если показатель попадает в диапазон 2-[3; 8], то подробное описание будет дано наихудшей ситуации (2);
  + если показатель попадает в диапазон [3-8]-9, то подробное описание будет дано наилучшей ситуации (9);
  + если показатель оценен как 2-9, то в зависимости от стиля подобное описание будет дано в любой из колонок, но с обязательным объяснением наихудшего и наилучшего случаев;
* каждый архитектурный стиль может и должен иметь оценку 10 только по одному показателю:
  + если архитектурный стиль имеет оценку 10, окрашенную в зелёный цвет, по какому-либо из показателей, значит он лучший по данному показателю в сравнении с любым другим, представленным в таблице, архитектурным стилем и именно для него данное качество является эталонным;
* если архитектурный стиль имеет оценку 9, окрашенную в жёлтый цвет, по какому-либо из показателей, значит именно по данному показателю стиль является наиболее удачным в сравнении с другими своими показателями и лишь немного не дотягивает до первенства в абсолютном сравнении;
* каждый архитектурный стиль может и должен иметь оценку 1 только по одному показателю:
  + если архитектурный стиль имеет оценку 1, окрашенную в красный цвет, по какому-либо из показателей, значит от худший по данному показателю в сравнении с любым другим, представленным в таблице, архитектурным стилем;
* если архитектурный тиль имеет оценку 2, окрашенную в оранжевый цвет, по какому-либо из показателей, то это означает что при разработке следующего по эволюции архитектурного стиля, архитекторы делали акцент на исправлении именно этого показателя;
* если два и более названия архитектурных стиля обрамлены в одинаковую рамку, то подобное означает, что нижележащий стиль идеологически и концептуально базируется на вышележащем;
* отдельным показателем является первая колонка Caliber.

## Описание характеристик

|  |
| --- |
| Заметка! |
| 1. Когда употребляется слово **«качество»** относительно какой-то характеристики рассматриваемого архитектурного стиля, то имеется в виду тот факт, что независимо от специфики/тематики/области знаний/сложности ПО, написанного в концепции рассматриваемого стиля, указанная характеристика не может быть не соблюдена (т.е. нельзя спроектировать ПО таким образом, чтобы данная характеристика была испорчена). 2. Когда употребляется слово **«способность»** относительно какой-то характеристики архитектурного стиля, то имеется в виду тот факт, что сам по себе архитектурный стиль не гарантирует, но лишь способствует (не препятствует) созданию ПО, архитектура которого будет обладать указанной характеристикой (т.е. в условиях реальной разработки ПО, даже при использовании рассматриваемого архитектурного стиля, указанная характеристика может быть испорчена). |

|  |
| --- |
| Определение! |
| 1. **Задача** – что-то, требующее решения с использованием ПО, запущенного на вычислительном оборудовании. 2. **Техническая задача (ТЗ)** – это задача, доступная к определению через строго формулируемые технические требования, и характеризующая следующими факторами:    * контекст её формулировки относится к одной области знаний и/или, одной специфики решения, и/или одному подходу к своему решению;    * скорее важны результаты, а не промежуточные состояния и контекст решения:      + набор входных данных необходимый для её решения, всегда заранее определён;      + набор выходных данных, являющихся результатом решения всегда заранее определён; 3. **Бизнес-задача (БЗ)** – это задача, для комплексного решения которой необходимо решить две и более технические задачи в рамках и по заранее определённой процедуре сорганизованного обмена результатами вычислений, и характеризующаяся следующими факторами:    * контекст её формулировки относился к некоторой абстрактной сфере (человеческих коммуникаций, межкомпонентного взаимодействия и т.д.);    * скорее важен контекст и процесс решения, а не итоговые результаты:      + набор входных данных заранее не определён и может меняться по ходу решения задачи;      + набор выходных данных заранее не фиксирован и может меняться по ходу решения задачи, имея промежуточные результаты. |

### [Caliber](#_Сводная_таблица_анализа)

Масштаб задач, для которых данный архитектурный стиль подходит наиболее хорошо:

* показатели от 1 до 5 свидетельствуют о том, что данный архитектурный стиль ориентирован на создание приложений решающих ТЗ:
  + оценка 1 свидетельствует о том, что архитектурный стиль наиболее подходит для решения строго сформулированной, однозначно понимаемой задачи;
  + оценка 5 свидетельствует о том, что архитектурный стиль наиболее подход для решения комплекса задач из одной области знаний, или одной специфики, или согласованных с одним подходом к своему решению, или общей для всех логики решения;
* показатели от 6 до 10 свидетельствуют о том, что данный архитектурный стиль ориентирован на создание приложений решающий БЗ;
  + чем больше оценка тем масштабнее решаемая бизнес-задача.

### [Agility](#_Сводная_таблица_анализа)

Способность архитектуры ПО быстро адаптироваться под постоянно меняющиеся бизнес-требования, обеспечивая гарантию сохранности удовлетворения функциональных и нефункциональных требований.

### [Abstraction Level](#_Сводная_таблица_анализа)

Способность архитектуры ПО скрывать технические детали своей реализации.

### [Configurability](#_Сводная_таблица_анализа)

Способность архитектуры ПО гибко адаптироваться под конкретные задачи/запросы/действия своих пользователей.

### [Domain portioning](#_Сводная_таблица_анализа)

Способность архитектуры ПО быть хорошо декомпозируемой на изолированные по роли компоненты.

### [Cost of implementation](#_Сводная_таблица_анализа)

Качество архитектуры ПО быть дешёвой на этапе реализации.

### [Const of ownershiping](#_Сводная_таблица_анализа)

Качество архитектуры ПО быть дешёвой на этапе владения.

### [First Deployability](#_Сводная_таблица_анализа)

Качество архитектуры ПО быть максимально простой на этапе ввода в эксплуатацию.

### [Next Deployability](#_Сводная_таблица_анализа)

Качество архитектуры ПО быть максимально простой на этапе сопровождения/обновления/поддержки.

### [Main-Structure Simplicity](#_Сводная_таблица_анализа)

Способность архитектуры ПО быть максимально простой в контексте общей архитектуры.

### [Component-Structure Simplicity](#_Сводная_таблица_анализа)

Способность архитектуры ПО быть максимально простой в контексте архитектуры локальных компонентов.

### [Infrastructure Simplicity](#_Сводная_таблица_анализа)

Качество архитектуры ПО быть максимально простой в контексте требований к обеспечивающей инфраструктуре.

### [Hardware fault tolerance](#_Сводная_таблица_анализа)

Качество архитектуры ПО быть устойчивой к аппаратным сбоям.

### [System-component fault tolerance](#_Сводная_таблица_анализа)

Качество архитектуры ПО быть устойчивой к внутренним/собственным/системным сбоям.

### [Web-communication fault tolerance](#_Сводная_таблица_анализа)

Качество архитектуры ПО быть устойчивой к сбоям в работе сети.

### [Performance](#_Сводная_таблица_анализа)

Качество архитектуры ПО быть максимально производительной насколько это возможно в сравнении с доступными решениями/реализациями.

### [Testability](#_Сводная_таблица_анализа)

Качество архитектуры ПО быть легко поддающейся отладочному тестированию.

### [Technical Decomposability](#_Сводная_таблица_анализа)

Способность архитектуры ПО быть хорошо декомпозируемой на компоненты независимые по технической роле:

1. UI – роль интерфейса пользователя;
2. DB – роль базы данных;
3. Controller – роль логики системы;
4. Security – роль защиты информации;
5. и другие роли, в зависимости от ПО.

### [Technical Evolvability](#_Сводная_таблица_анализа)

Способность архитектуры ПО быть хорошо адаптируемой к будущим изменениям в технических запросах/задачах, решаемых системой.

### [Elasticity](#_Сводная_таблица_анализа)

Способность архитектуры ПО быть хорошо адаптируемой к использованию аппаратных ресурсов в зависимости от реальной моментальной нагрузки без прерывания функционирования.

### [Business Decomposability](#_Сводная_таблица_анализа)

Способность сервис-ориентированной архитектуры ПО быть хорошо декомпозируемой на компоненты независимые по бизнес-роли (Security, Validation, User Orchestration, …).

### [Business Evolvability](#_Сводная_таблица_анализа)

Способность сервис-ориентированной архитектуры ПО быть хорошо адаптируемой к будущим изменениям в бизнес-запросах/задачах, решаемых системой.

### [Service self-Sufficiency](#_Сводная_таблица_анализа)

Способность сервис-ориентированной архитектуры ПО наделять составные компоненты общей системы максимальной независимостью друг от друга.

### [Service Data Intersection](#_Сводная_таблица_анализа)

Способность сервис-ориентированной архитектуры ПО максимально разделять данные необходимые для работы отдельных компонентов.

### [Integration](#_Сводная_таблица_анализа)

Способность сервис-ориентированной архитектуры ПО легко и просто интегрировать новые компоненты, улучшая тем самым свойства системы.

### [Interoperability](#_Сводная_таблица_анализа)

Способность сервис-ориентированной архитектуры ПО быть легко и просто адаптируемой для предоставления и получения данные от других систем в процессе кооперационного взаимодействия.

### [Horizontal Scaling Possibility of Computing-Resources](#_Сводная_таблица_анализа)

Способность сервис-ориентированной архитектуры ПО наращивать вычислительную мощность путём дублирования сервисов.

### [Horizontal Scaling Possibility of Functionality](#_Сводная_таблица_анализа)

Способность сервис-ориентированной архитектуры ПО добавлять новый функционал путём добавления новых сервисов.

# Подробный разбор архитектурных стилей

|  |
| --- |
| Заметка! |
| Рассматривая очередной архитектурный стиль, например монолитный, для краткости будет писаться «монолит», а не «монолит»-стиль и/или «монолит»-архитектура, и/или «монолит»-система, и/или «монолит»-компонента, подразумевая, что целевой постфикс будет определяться в зависимости от контекста. |

## Стили ориентированные на решение технических задач (Caliber 1–5)

Данная категория стилей – это архитектурные стили, отличающиеся локальным масштабом решения проблем, т.е. всех тех, которые могу быть сформулированы в рамках решения конкретной ТЗ.

### Монолит

Общее описание:

Исторически первый архитектурный стиль ПО, – можно сказать «естественный», «который нельзя было не придумать», – главная отличительная черта которого в том, что ПО представлено в виде одного единственного артефакта, являющегося как правило исполняемым файлом.

Ключевые сильные стороны:

* Const of Implementation:

«монолит» не подразумевает каких-либо специфических “приседаний” необходимых для обеспечения возможности реализации (нужен лишь компилятор и блокнот).

* Const of ownershiping

Стоимость владения монолитной системой либо бесплатна, либо ложится на плечи пользователя.

* First/Next Deployability

Первичная и последующие развёртывания системы на мощностях пользователя по факту являются лишь стандартной процедурой установки/переустановки целевого ПО.

* Main-Structure/Infrastructure simplicity

Простейшая верхнеуровневая структура и отсутствие необходимости сетевого взаимодействия, являеся неоспоримым «преимуществом простоты» данного архитектурного стиля.

* Web-communication tolerance

ПО, созданное в монолитном стиле, запускается на единственной машине из-за чего, даже если у данной машины возникнут проблемы с сетевым сообщением, приложение не упадёт, но просто зависнет на время до возобновления связи с внешним миром.

* Performance

Приложения, построенные в монолитном стиле, обладают высочайшей из доступных (потенциально доступных) производительностью.

* Testability

Тестирование монолитных приложений является наиболее простым с технической точки зрения (для этого достаточно отладчика и, желательно, IDE).

Ключевые слабые стороны:

* Agility

«монолит» не подразумевает ситуации постоянно меняющихся бизнес-требований, но подразумевает, что функциональные и нефункциональные требования формулируются единственный раз до начала реализации и не меняются на протяжении всего жизненного цикла.

* Abstraction Level

«монолит» не имеет какого-либо уровня абстракции, а если начинает иметь, то вероятно такая система требует переосмысления в пользу использования другого архитектурного стиля[[2]](#footnote-2).

* Configurability

«монолит» не подразумевает гибкость рода смены конфигурации под запросы пользователя[[3]](#footnote-3).

* Domain portioning

«монолит» – это один компонент и одна роль![[4]](#footnote-4)

* Component-Structure Simplicity

«монолит», будучи состоящим из одно единственного компонента, образующего всю систему целиком, может *дожить* до ситуации, когда сложность такового компонента будет исключительно большой[[5]](#footnote-5).

* Hardware fault tolerance

Монолитное ПО, будучи лишь процессом в ОС, просто не может выполнять своих функций в случае отказа среды исполнения.

* System component fault tolerance

Поскольку монолитное ПО, состоит из одного единственного компонента, то его отказ равносилен отказу всего ПО.

* Technical Decomposition

В монолитном ПО разделение по техническим ролям возможно только на логическом уровне и, частично, на уровне семантической структуры, но не более того.

* Technical Evolvability

идеологически, «монолит» задуман как архитектурный стиль для построения ПО, решающее одну единственную задачу в конкретной, строго определённой области знаний, строго определёнными методами и потому в него изначально не закладывается какая-либо гибкость адаптации к меняющимся техническим условиям.

* Elasticity

Будучи лишь процессом в ОС пользователя, монолитное приложение не имеет какой-либо власти над контролем за потребляемыми ресурсами.

Рекомендации к применению:

Односложные, однопоточные, простые программы, изолированные по данным и процессам их обработки, как правило не требующие сетевого взаимодействия и предназначенные для решения одной ТЗ:

* скрипты настройки окружения;
* утилиты ОС;
* вспомогательные/служебные программы;
* драйверы.

### Модульный Монолит

Общее описание:

«модульный монолит» является естественным развитием стиля «монолит», отличаясь от него тем только, что теперь «монолит» представляется в виде модулей.

|  |
| --- |
| Определение! |
| Модуль – это компонент ОП, обладающий следующими свойствами:   * выполняет в ПО строго определённую функцию/роль/задачу: * скрывает от остальных компонентов детали своей реализации; * представлен в виде подгружаемого артефакта ПО. |

Сущностно, «модульный монолит» не привносит ничего нового, кроме того, что теперь ПО на уровне артефактов выглядит ни как один единственный исполняемый файл, но как исполняемый файл и некоторое количество библиотек (dll), подгружаемых либо сразу, либо по факту запроса соответствующего функционала.

Ключевые сильные стороны:

* Const of Implementation

«модульный монолит» привносит дополнительную сложность по сравнению с «монолитом», выраженную в необходимости иметь адекватную, гибко конфигурируемую систему сборки[[6]](#footnote-6).

* Const of ownershiping (аналогично «монолиту»)
* First/Next Deployability (аналогично «монолиту»)
* Infrastructure simplicity (так же как в «монолите»)
* Web-communication fault tolerance (аналогично «монолиту»)
* Performance

Разделение ПО на компоненты может повлечь накладные расходы на обмен данными между ними, но в общем и целом, «модульный монолит» может вплотную приблизиться по производительности к стилю «монолит».

* Testability

Могут возникнуть некоторые сложности с отладкой на этапе загрузки dll[[7]](#footnote-7).

* Technical Decomposability

«модульный монолит» поощряет архитекторов к тому, чтобы на этапе проектирования ПО, разделять его на компоненты, независимые по техническим ролям.

Ключевые слабые стороны:

* Domain Portioning

«модульный монолит» позволяет разделять ПО на компоненты, независимые по зоне ответственности, но не по роли в полноценном смысле, т.к. каждый из таковых компонентов продолжает является структурной частью целевого ПО.

* Hardware fault tolerance (так же как в «монолите»)
* System component fault tolerance

Если «монолит» состоит из одно компонента, отказ которого равносилен краху всей системы, то система построенная по архитектуре «модульного монолита» падает с N-раз большей вероятностью, где N – это количество модулей.

* Technical Evolvability

«модульный монолит» крайне требователен к тому, чтобы все технические роли были определены и продуманы ещё на этапе проектирования ПО, т.к. внесение изменений в таковые на этапе реализации и поддержки может быть очень и очень сложным.

* Elasticity

«модульный монолит» обладает ограниченными возможностями по контролю над потребляемыми ресурсами, т.к. может управлять процессом загрузки/выгрузки целевых компонентов и исполнять их программный код в параллельных потоках.

Рекомендации к применению:

|  |
| --- |
| Заметка! |
| 1. «модульный монолит» можно считать родоначальником паттернов в разработке ПО, т.к. идея делить ПР на компоненты, дала толчок созданию концепций такого разделения, первыми среди которых были: Model-View-Controller (MVC), Model-View-Presenter (MVP), Model-View-Presenter-ViewModel. 2. Именно с появлением архитектурного стилям «модульный монолит» получили распространение программы, выполняющихся во множестве потоков одновременно; и классическим воплощением этой концепции стала технология OpenMP. |

Многопоточное ПО, локализованного типа, предназначенное для решения конкретных задач, обобщённых единой областью знаний/методологией решения/подходом к решению:

* различные CAD системы (не слишком сложные);
* редакторы документов (без коллаборации);
* САПР системы;
* простые игры.

### Microkernel

Общее описание:

Эволюционное развитие архитектуры монолитных приложений.

«microkernel» делает основной акцент на простоте разработки больших сложных систем.

В основе идеи лежит разделение «модульного монолита» на множество независимых «монолитов», каждый из которых по существу является отдельным процессом ОС, а общение между ними осуществляется через «канал данных».

|  |
| --- |
| Определение! |
| 1. Система – ПО, представленное в ОС в виде множества независимо запускаемых процессов, управляемых ОС по отдельности. 2. Канал данных – область оперативной памяти ОС, разделяемая между всеми процессами одной системы, доступ к которой осуществляется напрямую или по средствам обращения к специальному служебному процессу, называемому в данном контексте медиатором данных. |

Ключевые сильные стороны:

* Abstraction Level

В виду того, что все компоненты «microkernel» архитектуры общаются не на прямую а через канал данных, то уровень сокрытия внутренних технических деталей может варьироваться вплоть до полного сокрытия, когда бы общение происходило посредством обмена сообщениями/индикаторами.

* Configurability

Грамотно спроектированная «microkernel» система может быть универсальным инструментом для решения почти любых проблем пользователя, т.к. может быть инструментарием для создания чего-то специфического под решение конкретной задачи.

* Domain Portioning

«microkernel» создавалась для того, чтобы проектировать с её помощью многокомпонентные приложения, компоненты которого имеют максимальную независимость по исполняемой роли в контексте общей системы[[8]](#footnote-8).

* Infrastructure simplicity

В большинстве случае, запуск «microkernel» систем не вызывает избыточных сложностей по сравнению с «монолитными» или «модульно монолитными» ПО[[9]](#footnote-9).

* System-Component fault tolerance

«microkernel» довольно хорошо решает проблемы связанные с крахом какого-то компонента системы, т.к. у него всегда есть возможность просто взять и перезапустить его заново[[10]](#footnote-10).

* Performance

Несмотря на то, что «microkernel» системы будут сильно уступать по производительность аналогичному ПО, построенному в монолитных стилях, мы всё ещё имеем дело с системой высочайшей производительности, каковой достаточно даже при создании real-time систем/ПО.

* Technical Decomposability

«microkernel» системы могут испытывать определённые трудности разделении системы на компоненты по исполняемым техническим ролям на этапе проектирования в ситуациях как: (1) два и более компонентов системы нуждаются, например, в компоненте ответственно за UI, но спроектировать общий для всех компонентов интерфейс очень сложно, (2) два и более компонента системы нуждаются, например, в компоненте ответственном за DB, однако возникают сложности синхронизации данных и/или предоставления одновременного доступа к данным.

Возможны и другие ситуации, но большинство из таковых могут быть успешно решены на этапе проектирования.

* Technical Evolvability

Грамотно спроектированная «microkernel» система может очень хорошо адаптироваться к ситуации меняющихся технических требований, в виду того подобная ситуация очень редко затрагивает всю систему целиком, но лишь какой-то из её компонентов.

* Elasticity

Несмотря на то, что «microkernel» системы, как правило, потребляют значительно больше аппаратных ресурсов в сравнении с приложениями, построенными по предыдущим архитектурным стилям, именно разделение системы на множество независимых потоков в рамках одной ОС, обеспечивает возможность адаптировать мощности системы под целевые запросы/задачи пользователя.

Ключевые слабые стороны:

* Hardware fault tolerance

«microkernel» – это единственный архитектурный стиль, приложения по которому, можно спроектировать так, что сбой в их работе приведёт к отказу вообще всей системы на которой приложение запущено[[11]](#footnote-11).

Рекомендации к применению:

Большие сложные программы, ограниченные контекстом работы в рамках одной ОС:

* операционные системы;
* большие сложные AAA-игры;
* большие сложные IDE, редакторы и т.д.

## Стили ориентированные на решение бизнес-задач (Caliber 6–10)

Каждый бизнес существует для того, чтобы предоставлять конечному потребителю какой-то продукт или услугу. Услуга или товар могут быть “простыми” в том смысле, что для того, чтобы предоставить их конечному потребителю, бизнесу необходимо решить одну техническую задачу (например, написать ПО для решения дифференциальных уравнений). Когда же услуга или товар, для предоставления которого существует бизнес, разрастается/приобретает такой масштаб, что для своего совокупного решения требует решения некоторого (порой довольно большого) количества технических задач, то мы говорим что бизнес решает бизнес-задачу, для успешного решения каковой нуждается в ПО, построенном по архитектуре, ориентированной для решения бизнес-задач.

Каждая из технических задач, связанная с бизнес-задачей, как правило, должна быть решена в заранее определённой последовательности с требуемым качеством и в определённый временной интервал. Учитывая же тот факт, что в большинстве ситуаций, для решения очередной технической задачи, необходимы результаты, полученные от решения других технических задач, то важным моментом при решении бизнес-задачи становится ситуация обмена данными, т.е. результатами решения технических-задач.

Учитывая это, ПО, созданное/приобретённое бизнесом для решения своей бизнес-задачи, является по сути информационно-вычислительной системой, состоящей из множества различного ПО, предназначенного, (1) как для решения специфических технических задач, требуемых бизнесу, (2) так и вспомогательного ПО, призванного решать задачи всесторонней коммуникации.

## Service-Oriented Architecture

Разговор об архитектуре ориентированной на сервисы начинается тогда, когда компоненты системы, созданной для решения некоторой бизнес-задачи, ввиду различных причин[[12]](#footnote-12) вынуждены общаться через сеть.

|  |
| --- |
| Определение! |
| 1. Сервис-ориентированная архитектура создания ПО – это архитектура, определяющая принципы и правила построения распределённых информационно-вычислительных систем (далее «сервис-систем»), главным образом решающая две следующие задачи, т.е. дающая ответы на два главных вопроса:   **(1) по какому принципу бизнес-задача будет декомпозироваться на технические задачи? = по какому принципу сервис-система будет декомпозировать на сервисы?**  **(2) каким образом различные сервисы будут общаться между собой?**   1. Сервис – это компонент сервис-системы, выполняющий в системе уникальную роль для решения целевой бизнес-задачи, т.е. решающий определённую техническую задачу, результаты решения которой доступны другим сервисам по средствам сетевой коммуникации. |

### Service-Based

|  |
| --- |
| Заметка! |
| При оценки табличных показатель для «service-based» архитектуры большая часть таковых была синхронизирована с аналогичными же для «microkernel» архитектуры и потому описываться не будут, тогда как подробное описание будет дано всем тем показателям по которым два упомянутых архитектурных стиля отличаются |

Общее описание:

Исторически, «service-based»-архитектуры появились в тот момент, когда потребовалось создать ПО, построенное по архитектуре «microkernel», но в условиях сетевого взаимодействия, когда бы различные компоненты системы общались не через *канал связи*, а по средствам сетевых протоколов.

|  |
| --- |
| Определение! |
| Service-Based-сервис-архитектура определяется через следующие ответы:   1. По какому принципу сервис-система будет декомпозироваться на сервисы?   Будучи концептуальной преемницей «microkernel» архитектуры, «service-based» архитектура декомпозирует решаемую бизнес-задачу на уникальные, максимально маленькие, логически не пересекающиеся технические задачи, не делая акцент на не пересечение по данным и сервисную самодостаточность.   1. Каким образом различные компоненты системы (далее «сервисы») будут общаться между собой?   Общение происходит посредствам web-обмена готовыми программными сущностями: структурами, классами, переменными. |

|  |
| --- |
| Заметка! |
| «service-based»-архитектуру можно считать идеальным инструментом, когда требуется создать конвейерную распределённую систему обработки информации. |

Ключевые сильные стороны:

* Hardware fault tolerance

«service-based» крайне толерантна к отказу вычислительных узлов, т.к. при отказе любого из таковых, запущенные на нём сервисы могут быть перезапущены на любом другом доступном вычислительном узле, имеющем доступ в сеть[[13]](#footnote-13).

* Business Decomposability

На этапе первичного проектирования «service-based» хорошо декомпозируется на компоненты, выполняющие отдельные бизнес-роли[[14]](#footnote-14).

* Horizontal Scaling Possibility of Computing-Resources

«service-based» может довольно легко наращивать вычислительные мощности путём размножения запущенных экземпляров сервисов, выполняющих наиболее ресурсоёмкие вычисления на пути обработки запроса.

Ключевые слабые стороны:

* Const of implementation

В лучшем случае «service-based чуть дороже в реализации в сравнении с «microkernel» в виду того, что каждый из компонентов системы, становясь сервисом, теперь имеет коммуникационный модуль в своём составе. В худшем случае для реализации масштабного ПО, созданного в согласии с «service-based»-архитектурой, компании придётся обзавестись собственными серверами.

* Const of ownershiping

«service-based» может быть очень дорогой в случае если реализуемая сервис-система состоит из большого количества серверов либо арендуемых, либо обслуживаемых компанией.

* First Deployability

«service-based» может очень сложной на этапе первичного развёртывания в условиях большого количества серверов и синхронизации их работы.

* Next Deployability

При обновлении «service-based» зачастую придётся с толкнуться с проблемой синхронизированной остановки всех серверов, или даже более худшей – с проблемой переформатировании данных на новый формат использования/интерпретации/хранения.

* Infrastructure Simplicity

«service-based», будучи основанной на сетевом взаимодействии своих компонентов, может быть очень требовательна к обеспечивающей инфраструктуре: могут потребоваться как собственные сервера, так и создание уникального коммуникационного ПО, – однако в базовом случае, всё требуемое для старта функционирования можно либо арендовать (если это оборудование), либо найти на рынке свободного ПО (если это коммуникационное ПО), либо и вовсе обойтись штатными средствами ОС.

* Web-communication fault tolerance

«service-based» очень чувствительна даже к частичным сбоям в сетевых коммуникациях, т.к. подобная ситуация приводит к полной остановки системы на некоторое время.

* Performance

«service-based» будет всегда дольше выполнять операции в сравнении с аналогичной по функционалу «microkernel» системе в виду значительно возрастающих накладных расходов на коммуникацию.

* Testability

Интеграционное «service-based» сервис-системы будет всегда значительно сложнее чем в любой другой системе, т.к. краеугольным камнем будет проблема синхронизации серверов.

* Business Evolvability

«service based» в виду специфики обмена информацией между сервисами крайне чувствительна к изменению бизнес-требований, предъявляемые к уже существующим компонентам.

* Service self-Sufficiency

Сервисы в «service-based»-архитектуре не задуманы как полностью самодостаточные компоненты системы, т.к. коммуникация между ними происходит готовыми программными сущностями, и, по сути, являются полностью бесполезными в отрыве от совокупной сервис-системы.

* Service Data Intersection

Сервисы в «service-based» как правило являются лишь этапами в последовательности обработки одних конкретных данных в связи с чем изначально не задумываются как обособленные по данным компоненты.

* Integrational

«service-based» хорошо относится к ситуации добавления новых сервисов, когда таковые своим появлением не меняют нефункциональные требования существующих сервисов.

* Interoperability

Процесс обработки информации в «service-based» как правило изолирован и замкнут, а единственными точками коммуникации с пользователями/внешним миром является один-два сервиса, выступающие в роли тонкого клиента.

* Horizontal Scaling Possibility of Functionality

«service-based» сервис-системы не задуманы как постоянно расширяющиеся в функциональном смысле, если только новый функционал не влияет на специфику работы уже имеющегося.

Рекомендации к применению:

Данная архитектура, будучи переходной от «microkernel» к крупнокалиберным/полноценным сервис-ориентированным архитектурам (начиная со «space-based»), по сути является сильно отказоустойчивым microkernel-монолитом.

Несмотря на большое количество недостатков, тем не менее, существует ряд условий, наличие которых позволяет делать осознанный выбор в пользу «service based»:

* все данные бизнеса ради которых построена сервис-система не нуждаются в коммуникации с другими системами из внешнего мира;
* процесс обработки данных бизнес-задачи строго определён и очень редко меняется;
* конечный пользователь может взаимодействовать только с одним из сервисов (как правило с последним в последовательности решения бизнес-задачи);
* сервис-система не может работать в случае отказа какого-либо сервиса и вынуждена дожидаться восстановления его работоспособности, чтобы возобновить процедуру решения бизнес-задачи.

Примерами таких систем могут быть:

* медицинские сервис-системы;
* модульные распределённые CAD-системы;
* online игры.

### Space-Based

Общее описание:

Довольно сложная в своей реализации сервис-ориентированная архитектура, хорошо подходящая в основном для больших многопользовательских облачных систем.

В основе идеи данной архитектуры лежит следующий набор концептуальный принципов:

* сервис-система представлена в виде набора компонентов-сервисов (далее *компонент*), совокупно предназначенных для решения обобщённо сформулированной бизнес-задачи;
* для решения конкретной бизнес-задачи создаётся конфигурация из нужных *компонентов*, которые запускаются в изолированном вычислительном пространстве (далее *порция*);
* каждые из таковых *порций* являются изолированными друг от друга по данным, вычислительным ресурсами и процессам обработки информации;
* синхронизация результатов вычислений происходит (если вообще происходит) только на этапе завершения работы *порции*, когда происходит освобождение выделенных для неё ресурсов.

|  |
| --- |
| Определение! |
| Space-Based-сервис-архитектура, определяется через следующие ответы:   1. По какому принципу сервис-система будет декомпозироваться на сервисы?   Каждый сервис проектируется как *компонент*: полностью самодостаточный, изолированный модуль сервис-системы, согласованный с принятыми в сервис-системе принципами приёма/хранения/передачи информации (как правило, результатами функционирования такого сервиса является некоторая БД/таблица/файл с заранее определённой моделью/форматом/структурой к которой любой другой *компонент* имеет доступ), способный выполнять возложенную на него задачу, будучи полностью изолированным от других компонентов в контексте изолированных *порций* (если подобное невозможно, то сервисы объединяются до тех пор пока данное требование не начинает соблюдаться).   1. Каким образом различные компоненты сервисы будут общаться между собой?   Общение между *компонентами* происходит исключительно сообщениями или структурированными данными (никогда не программными сущностями) не имеющими ни адресата ни получателя, но имеющие смысл только в контексте обработки информации внутри некоторой *порции*, конфигурация которой подразумевает помимо *компонентов*, необходимых для решения определённой бизнес-задачи, ещё и различные *компоненты-медиаторы*, занимающиеся оркестрацией потоков данных.  Порции же в строгом смысле вообще не общаются напрямую, т.к. в моменте своего существования ничего не знают о совокупно решаемой бизнес-задаче, для решения части[[15]](#footnote-15) которой были сконфигурированы (в этом случае, уже отдельные **порции** могут рассматриваться в качестве **компонентов** на уровне выше). |

Ключевые сильные стороны:

* Configurability

*Порции* в «space-based» по сути являются идеальным воплощением концепции конфигурирования системы под задачу.

* Domain Portioning

Сама идея конфигурирования системы под задачу, подразумевает, что архитектура таковой способствует и поощряет структуризацию на компоненты не пересекающиеся по исполняемой роли и зоне ответственности.

* Hardware/System-Component/Wem-communication fault tolerance

«space-based» очень отказоустойчива, т.к. сама по себе является конструктором для создания *порций*, решающих определённые задачи, и если что-то отказывает, то в худшем случае отказывает запущенная *порция*, которую можно перезапустить и даже дублировать для надёжности.

* Performance

С точки зрения всей «space-based» сервис-системы, таковая точно не отличается быстродействием, однако в условиях конкретной реализации, конфигуратор *порций* можно реализовать таким образом, что *порции* будут сравнимы по скорости с аналогичной системой построенной по «microkernel» архитектуре.

* Business Decomposability/Evolvability

Грамотно спроектированная «space-based»-сервис-система может послужить основой для решения почти любой бизнес-задачи и быть очень и очень гибко адаптирующейся.

* Service self-Sufficiency/Data Intersection/Integration

Сервисы «space-based» сервис-архитектуры, будучи *компонентами*, в качестве основных принципов своего проектирования и создания держат в фокусе максимизацию этих показателей.

* Integration

«space-based» хорошо адаптирована к интеграции новых сервисов, если таковые удовлетворяют критериям *компонента* и могут быть составляющим компонентом *порции*.

* Horizonal Scaling Possibility of Computing-Resources

Теоретически, «space-based» имеет неограниченный потенциал для масштабирования вычислительных мощностей т.к. позволяет гибко конфигурировать и множить *порции*, требуемые для решения бизнес-задач. Но с точки зрения рациональности использования вычислительных ресурсов отдельными *порциями*, в смысле загруженности *компонентов*, могут возникать значительные сложности.

* Horizonal Scaling Possibility of Functionality

Масштабирование доступного функционала, является вопросом реализации нового *компонента*.

Ключевые слабые стороны:

* Const of implementation

Учитывая реальные масштабы[[16]](#footnote-16) «space-based» сервис-систем, таковые являются очень дорогими на этапе своей реализации[[17]](#footnote-17).

* Const of ownershiping

Будучи следствием специфики и причинности больших затрат на этапе первичной реализации, «space-based» крайне требовательна к своему содержанию.

* First/Next Deployability

Концепция построения обобщённой сервис-системы, может быть очень сложной в развёртывании и обновлении для владельцев бизнеса, но очень простой для клиентов бизнеса[[18]](#footnote-18).

* Main/Component-Structure

«space-based» как правило очень сложна на любом из уровней абстракции. Более того, данная архитектура подразумевает конструирование архитектуры *компонентов* таким образом, чтобы в последствии из них возможно было собрать архитектуру *порций*.

* Infrastructure Simplicity

«space-based» очень требовательна к обеспечивающей инфраструктуре, куда входят большие затраты на серверное оборудование и большое количество специального ПО.

* Testability

Почти любая «space-based» сервис-система, будучи плохо спроектированной, почти не поддаётся адекватной отладки, поэтому, ещё на этапе проектирования таких архитектур, большое внимание уделяется вопросам наблюдаемости в реальном времени.

* Interoperability

Реальные «space-based» сервис-системы, конечно, имеют в своём составе *компоненты*, ответственные за коммуникацию с внешним миром, однако, подобные *компоненты* обычно играют одну из двух ролей: (1) дать доступ внешним клиентам к конфигуратору *порций*, (2) дать внешним клиентам возможность взаимодействовать со сконфигурированными *порциями*. Т.е. «space-based» скорее задумана как самодостаточная, самостоятельная сервис-система, не нуждающаяся в коммуникации изнутри наружу, но дозволяющая коммуникацию снаружи вовнутрь.

Рекомендации к применению:

|  |
| --- |
| Заметка! |
| 1. «service-based»-сервис-архитектура ввиду исключительной сложности своей реализации очень требовательна к компетенции архитекторов и масштабов бизнеса, для решения задач которого предназначается. 2. «space-based»-сервис-архитектуру можно считать идеальным инструментом для создания архитектуры для бизнеса, предоставляющего какие-либо облачные сервисы по подписке. |

Облачные сервисы (SaaS, IaaS, DBaaS и т.д.)

Примерами таких систем могут служить:

* сервисы Microsoft/Amazon.

### Event-Driven

Общее описание:

Event-Driven, это довольно специфический тип архитектуры приложений, получивший свои концептуально-идеологические основы со времён «modular monolithic» архитектуры для которой впервые был создан паттерн программирования MVC, одной из особенностей которого была идея подписки на события.

|  |
| --- |
| Определение! |
| Even-Driven-сервис-архитектура, определяется через следующие ответы:   1. По какому принципу сервис-система будет декомпозироваться на сервисы?   Каждый сервис в «event-driven» архитектуре является полностью независимой структурной частью системы, выполняющей на верхнем уровне абстракции две роли: (1) генерирует события, (2) реагирует на события, сгенерированными другими сервисами.  Таким образом, декомпозиция подобной сервис-системы на сервисы, осуществляется по принципу «событийной самостоятельности», когда бы каждый сервис являлся изолированной средой, независимой по логике и по данным, для обработки определённого события системы.   1. Каким образом различные компоненты сервисы будут общаться между собой?   Сервисы в «event-driven» архитектуре не общаются межу собой, они генерируют события и подписываются на обработку таковых, т.е. коммуникация происходит на уровне обмена сообщениями.  С технической точки зрения, каждое событие – это какое-то сообщение, обладающее идентификационными маркерами; а подписка на событие – это процесс ожидания сообщения с соответствующими идентификационными маркерами.  Реализует и управляет данную концепцию, ПО, называемое брокерами/медиаторами-сообщений (далее *брокер*), которые занимаются классификацией и оркестрацией событий в «event-driven» сервис-системах. |

|  |
| --- |
| Заметка! |
| «event-driven»-архитектура является идеальным инструментом для реализации сервис-систем, ориентированных на обработку событий, но только для них. В виду этого данная архитектура является обособленной: не имеет зелёных и красных показателей в сравнительной таблице, т.к. не поддаётся однозначному прямому сравнению. |

Ключевые сильные стороны:

* Agility + Abstraction Level + Configurability + Domain Portioning + Business Decomposability/Evolvability

Гибкость, уровень абстракции сервисов, конфигурируемость и гибкость разделения на роли ответственности, адаптивность к меняющимся условиям бизнеса… у любой сервис-системы, построенной по «event-driven» архитектуре, приближена к идеальной, но только если рассуждать в терминах обработки событий.

* Testability

В отличии от двух предыдущих сервис-ориентированных архитектур, «event-driven»-архитектура довольно хорошо поддаётся процедуре отлаживания, в виду того, что симуляция сообщений, процедура достаточно простая, особенно в ситуации наличия *брокера*.

* Service self-Sufficiency

Сервисы в «event-driven» самодостаточны, если не считать того факта, что каждый из них неразрывно связан с *брокером*, будучи абсолютно бесполезными в отрыве от него.

* Service Data Intersection

«event-driven» вообще не подразумевает, что в ней существуют какие-то данные: сервисы обмениваются сообщениями, содержащими все необходимые данные внутри[[19]](#footnote-19).

* Integrational

Интеграция нового сервиса в «event-driven» архитектуре, есть вопрос его подключения к брокеру-сообщений, что, как правило, достаточно просто.

* Interoperability

Для «event-driven»-архитектуры в контексте данного свойства существует определённый парадокс:

- с одной стороны «event-driven» системы должны быть полностью изолированы от внешнего мира ввиду замкнутости на концепции событийной обработки,

- но, с другой стороны, реальные сервис-системы, созданные по данной архитектуре, абсолютно бессмысленны в случае отсутствия кооперации с внешними система.

Поэтому, ДА: «event-driven» архитектура довольно хорошо настраивается на коммуникацию с внешним миром; однако НЕТ: и каждая такая коммуникация, требует создания отдельного сервиса, занимающегося интерпретацией полученных из вне данных во внутреннее событие сервис-системы.

Ключевые слабые стороны:

* Infrastructure Simplicity

«event-driven» архитектура, даже будучи не масштабной, обходящейся лишь арендой вычислительной инфраструктуры, всегда требует для своего функционирования *брокера*[[20]](#footnote-20).

* Web-communication fault tolerance

В общем случае «event-driven» довольно хорошо переживают ситуации частичного отказа сети, однако возможна ситуация, когда станет недоступным *брокер*, что приводит к параличу всей системы и/или как минимум к потере части сообщений.

Рекомендации к применению:

Примеры таких систем:

* интернет-магазины;
* сервисы государственных услуг;
* системы электронных очередей.

### Microservices

Общее описание:

«microservices» архитектура, является воплощением идеи построения сложных распределённых вычислительных систем, конструируемых из односложных, изолированных по логике и по данным, не стандартизируемым по протоколам компонентов[[21]](#footnote-21).

|  |
| --- |
| Определение! |
| Microservices архитектура, определяется через следующие ответы:   1. По какому принципу сервис-система будет декомпозироваться на сервисы?   Каждый сервис в «microservices» архитектуре является полностью изолированным по логике и поданным компонентом сервис-системы, выполняющим максимально простую функцию.  Можно утверждать, что каждый сервис, тогда только начинает обретать концептуально задуманное воплощение, когда (1) у исполняемой им функции отсутствует контекст, (2) по факту анализа такового сервиса нельзя сказать что-то конкретное о совокупной сервис-системе.   1. Каким образом различные компоненты сервисы будут общаться между собой?   Простой ответ: как придётся!  Сервисы в «microservices» архитектуре заранее не проектируются под конкретную специфику общения, кроме самой общей – общения должно происходить через сеть по средствам, например, GET/POST-запросов, через HTTP-протокол[[22]](#footnote-22). |

|  |
| --- |
| Заметка! |
| На «microservice»-архитектуру можно смотреть как на коробку инструментов специального или общего назначения, когда каждый инструмент является некоторым сервисом со строго определённым предназначением, контекст и специфика использования которого, никак заранее не определены |

Ключевые сильные стороны:

* Agility

Неоспоримым преимуществом «microservices»-архитектуры является её гибкость и адаптивность к бизнесу[[23]](#footnote-23).

* Abstraction Level

«microservices» полностью скрывает все детали реализации своих отдельных компонентов до такой степени, что единственное, что нужно знать клиентами, как связаться с сервисом, а он уже сам расскажет о том, что умеет делать и как этим пользоваться.

* Configurability + Domain Portioning

«microservices» не накладывает каких-либо ограничений на структуризацию сервисов.

Наличие множества сервисов позволяем создавать логические связи, образующие отдельные зоны ответственности, из каковых уже можно конфигурировать систему под целевую бизнес-задачу.

* Component-Structure Simplicity

Если не требуется контекст выполнения, то, по задумке, структура каждого сервиса «microservices» системы будет чрезвычайно простой: это будет лишь два компонента, (1) какая-то простая функция, (2) и коммуникационный модуль[[24]](#footnote-24).

* Hardware/System-component/Web-communication fault tolerance

«microservices» является стандартом отказоустойчивости к сбоям любого рода, т.к. отказ чего-либо в худшем случае, лишит сервис-систему какой-то единичной функциональности и только тотальный отказ любой коммуникации, может привести полной дисфункциональности (не краху) системы, до времени пока связь не восстановится.

* Business Decomposability/Evolvability

«microservices» не накладывает никаких ограничений на декомпозицию бизнеса: будучи набором элементарных сервисов, которая может конструироваться под бизнес, без необходимости постоянной адаптации существующих бизнес-требований.

* Service self-Sufficiency

Сервисы в «microservices» архитектуре являются исключительно самостоятельными вплоть до того, факта, что даже в отрыве от целевой сервис-системы представляют определённую ценность, продолжая выполнять возложенную на них элементарную функцию.

* Service Data-Intersection

Сервисы в «microservices» архитектуре полностью независимы друг от друга по используемым данным[[25]](#footnote-25).

* Integration

«microservices» архитектура является, де-факто, стандартом гибкости в вопросах интеграции новых сервисов: каждый новый сервис, это просто очередная точка доступа к компоненту системы, не требующей в случае своего появления никаких действий по отношению к другим уже существующим сервисам.

* Interoperability

Сервисы в «microservices» архитектуре изначально проектируются таким образом, чтобы быть доступными для использования кем угодно, а не только составными компонентами сервис-системы.

* Horizontal Scaling Possibility of Computing Resources

«microservices»-архитектура не накладывает никаких ограничений на горизонтальное масштабировании вычислительных мощностей, за исключением того, что, в зависимости от конкретной сервис-системы, для управления подобной экспансивностью потребуется отдельный планировщик нагрузки.

* Horizontal Scaling Possibility of Functionality

«microservices» архитектура является, де-факто, стандартом гибкости в вопросах горизонтального масштабирования функциональности, т.к. любое расширение подобного рода микросервисной архитектуры, это всего лишь вопрос добавления нового сервиса.

Ключевые слабые стороны:

* Performance

В виду того, что у каждого сервиса в «microservices»-архитектуре отсутствует контекст взаимодействия с клиентами, а общение происходит по сети, то совокупное время на выполнение сервис-системой своей задачи становится максимальным в сравнении с выполнением таких же задач аналогичными система, но построенными по другим архитектурным стилям[[26]](#footnote-26).

* Const of Implementation/Ownershiping + First/Next Deployability

Если изначальный масштаб «microservices» системы не большой, то стоимость её реализации и развёртывания может быть сравнительно невелика, однако в виду того, что «microservices» архитектура подразумевает и поощряет неограниченный рост, со временем стоимость содержания такой системы будет всё возрастать и возрастать, что, однако не будет сказываться на сложности обновления.

* Main-Structure Simplicity

«microservices»-архитектура существующая уже достаточно долго, может обрасти таким большим количеством микро-сервисов, что даже самом верхнем уровне абстракции будет необозримо громоздкой.

Рекомендации к применению:

|  |
| --- |
| Заметка! |
| Чистых «microservices»-архитектур не существует и не может существовать, т.к. подобную архитектуру можно рассматривать лишь как универсальный инструмент для решения бизнес-задач. Поэтому, в реальной жизни «microservices»-архитектуры являются лишь какой-то составляющей частью бо́льшей системы, построенной, как правило, по другим архитектурным стилям. |

Примерам таких систем могу служить:

* сервисы Yandex/Google.

1. Все оценки являются относительными и потому при появлении нового архитектурного стиля вся таблица будет пересмотрена. [↑](#footnote-ref-1)
2. Появление в системе компонентов требующих абстрагирования части своей логики/функционала от других компонентов на логическом уровне, как правило, требует абстрагирования/разделения и на уровне технической реализации. [↑](#footnote-ref-2)
3. Исключением, однако, могут быть ситуации, когда монолитное приложение изначально создано как некоторый конструктор компонентов и/или инструмент конструирования, когда бы целевые задачи решались только в случае специфической настройки базисной модели (но по средствам конфигурационных файлов, без вмешательства в исходный код). [↑](#footnote-ref-3)
4. Усложнять эту схему нельзя, либо же необходимо переходить на другие архитектурные стили. [↑](#footnote-ref-4)
5. В большинстве ситуаций, подобное является главной причиной перестройки приложения на новый архитектурный стиль. [↑](#footnote-ref-5)
6. Таковые, однако, обильно представлены в современном мире ОП (CMake, Maven, …). [↑](#footnote-ref-6)
7. Данная проблема хорошо решается использованием современных IDE. [↑](#footnote-ref-7)
8. Яркий пример такой декомпозиции, это современные операционные системы [↑](#footnote-ref-8)
9. Однако, возможна ситуация, когда возможности ОС могут просто не позволить запустить «microkernel»-систему. [↑](#footnote-ref-9)
10. Исключением является ситуация, когда падающий компонент написан с ошибками, из-за чего его падение является стабильно-систематическим, а также в ситуации, когда падает главный компонент-медиатор, когда бы перезапускать упавший компонент было бы уже некому; но последнее решается тем, что функции медиатора распределяются между всеми компонентами системы. [↑](#footnote-ref-10)
11. Да, довольно сложно добиться этой ситуации, т.к. необходимо заниматься разработкой соответствующего ПО, и потому в некотором смысле у подхода «microkernel» нет каких-то явно выраженных слабых сторон, но тем не менее, это возможно. [↑](#footnote-ref-11)
12. Как правило, главной причиной подобного становится ориентация бизнеса на создание услуг доступных по сети (но возможные и другие). [↑](#footnote-ref-12)
13. Исключением с катастрофическими последствиями может быть ситуация выхода из строя вычислительного узла, выполняющего роль БД в сервис-системе без репликации. [↑](#footnote-ref-13)
14. Но в условиях когда подразумевается система постоянного масштаба. [↑](#footnote-ref-14)
15. В реальных системах *порции*, обычно, решают не какую-то часть бизнес-задачи, но полностью уникальную бизнес-задачу. В этом смысле совокупная «space-based» сервис-система решает множество бизнес-задач одновременно в контексте глобального предоставления услуг (каждой такой задачей, например, может быть предоставление возможностей некоторой инфраструктуры в контексте IaaS-сервис-системы). [↑](#footnote-ref-15)
16. Применять данную архитектуру для систем маленького масштаба не имеет смысла в виду того, что сложность её проектирования становится оправданной только в условиях большого бизнеса, способного себе позволить потратить много времени и денег на детальную проработку огромного количества требований, сопутствующих «space-based» архитектуре. [↑](#footnote-ref-16)
17. Подразумевается, что бизнес не арендует, но именно владеет вычислительными серверами и всей коммуникационной инфраструктурой, обеспечивающей функционирование «space-based» сервис-системы. [↑](#footnote-ref-17)
18. Собственно, подобная ситуация и является ключевой бизнес-идеей облачных технологий. [↑](#footnote-ref-18)
19. Единственная связанность на уровне данных может быть осмыслена/сформулирована в рамках того факта, что все сервисы обмениваются через *брокера*, который является единым для всей сервис-системы и без которого её существование невозможно. [↑](#footnote-ref-19)
20. В базовом случае таковое бесплатно и широко представлено на рыке открытого ПО, однако его грамотная настройка и использование требует опыта и времени. [↑](#footnote-ref-20)
21. Т.е. из многочисленного монолитного ПО с модулем сетевой коммуникации. [↑](#footnote-ref-21)
22. В идеале каждый сервис проектируется таким образом, чтобы (1) прежде всего максимально эффективно выполнять свою функцию, (2) предоставить как можно больше способов общения с клиентами, но ничего заранее не подразумевать о своих клиентах. [↑](#footnote-ref-22)
23. Можно сказать, что у таковой архитектуры нет специфики предназначения: она подходит для решения абсолютно любой бизнес-задачи, т.к., будучи представленной в виде набора элементарных инструментов, способна быть адаптированной/настроенной/дополненной для решения любой бизнес-задачи без необходимости менять существующие бизнес-требования. [↑](#footnote-ref-23)
24. Коммуникационный модуль, как правило стандартный и добавляется по шаблону. [↑](#footnote-ref-24)
25. Конечно, в реальный системах, решаемые бизнес-задачи часто требуют определённой интеграции данных между различными сервисами, что, однако, решается созданием дополнительного сервиса, играющего роль медиатора. [↑](#footnote-ref-25)
26. Однако, подобная проблема, может быть решена путём добавления новых сервисов с расширенной зоной ответственности. [↑](#footnote-ref-26)