

Domain-Driven Design: боремся с плохой архитектурой

Вячеслав Чернышов
СберТех

Как дела на **проекте**?



Мой проект **отлично**
себя чувствует

Как дела на **проекте**?



Мой проект **отлично**
себя чувствует



Мой проект
чувствует себя **не очень**

Как дела на **проекте**?



Мой проект **отлично**
себя чувствует



Мой проект
чувствует себя **не очень**



Мой проект –
это **большой ком грязи**



Вячеслав Чернышов

Backend-разработчик
СберТех, Platform V Flow



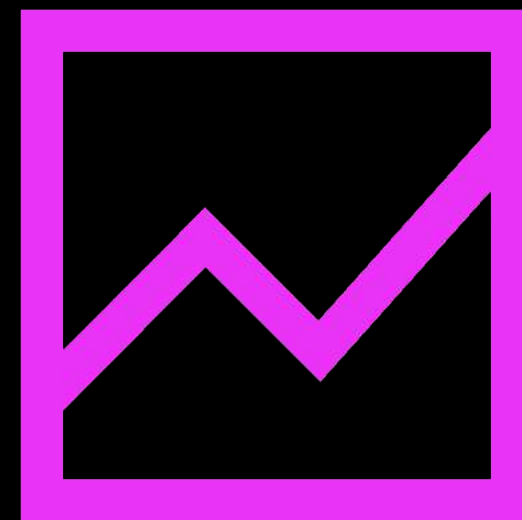
Статья на Хабре



Канал в Telegram

Что такое **плохая архитектура**?

Что такое **плохая архитектура**?



Хрупкость

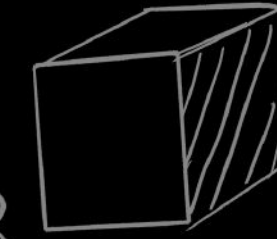
Состояние системы, в которой один компонент отвечает за множество реализаций.

Хрупкость

Есть одна функция.

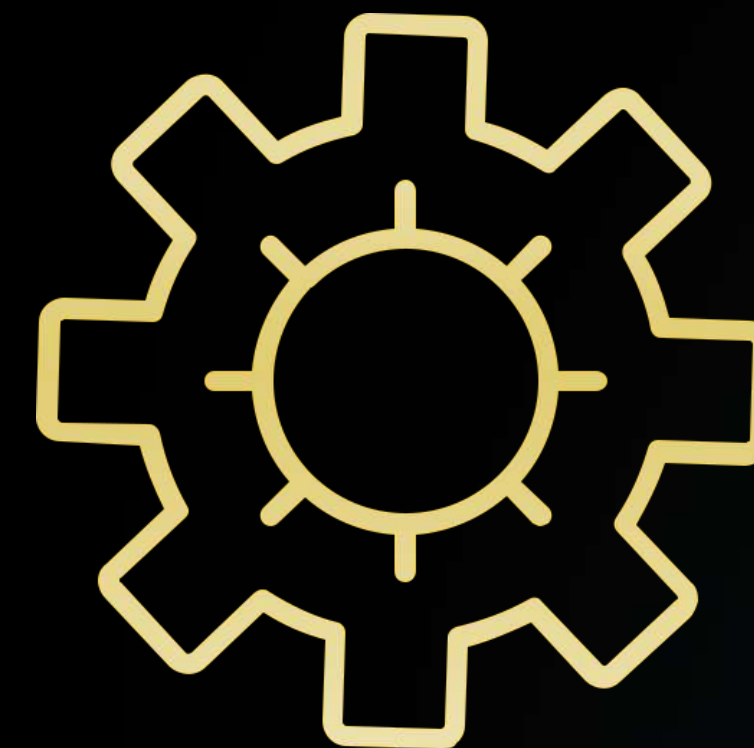
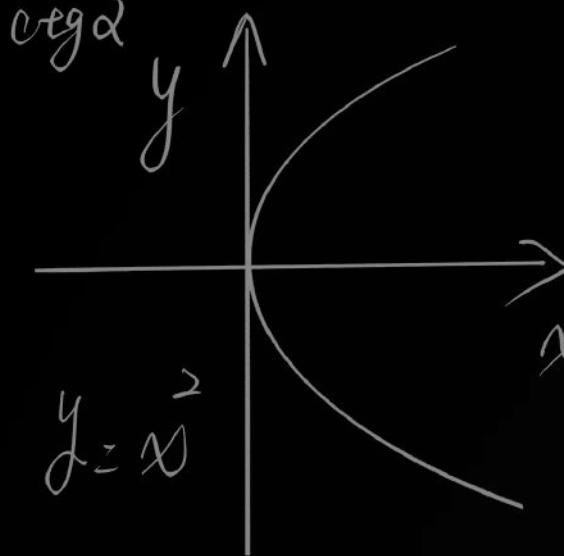
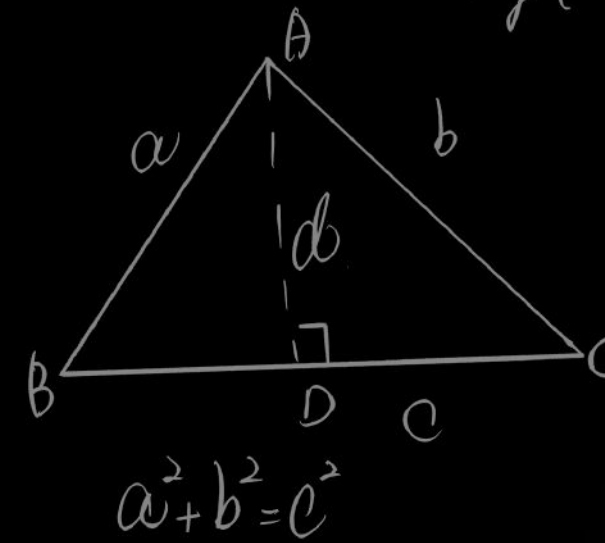
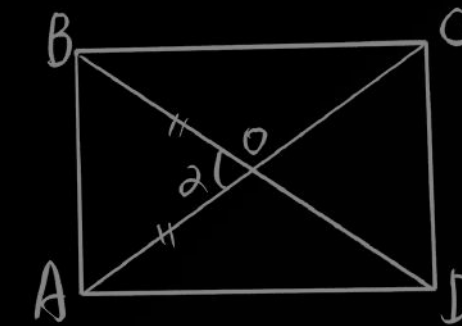
$$\sin(\alpha \pm \beta) = \sin \alpha \cos \beta \pm \cos \alpha \sin \beta$$

$$\cos(\alpha \pm \beta) = \cos \alpha \cos \beta \mp \sin \alpha \sin \beta$$



$$\operatorname{tg}(\alpha \pm \beta) = \frac{\operatorname{tg} \alpha \pm \operatorname{tg} \beta}{1 \mp \operatorname{tg} \alpha \cdot \operatorname{tg} \beta}$$

$$\operatorname{ctg}(\alpha \pm \beta) = \frac{\operatorname{ctg} \alpha \cdot \operatorname{ctg} \beta \mp 1}{\operatorname{ctg} \beta \pm \operatorname{ctg} \alpha}$$



Функция расчёта сверхурочного времени

Хрупкость

Есть одна функция.

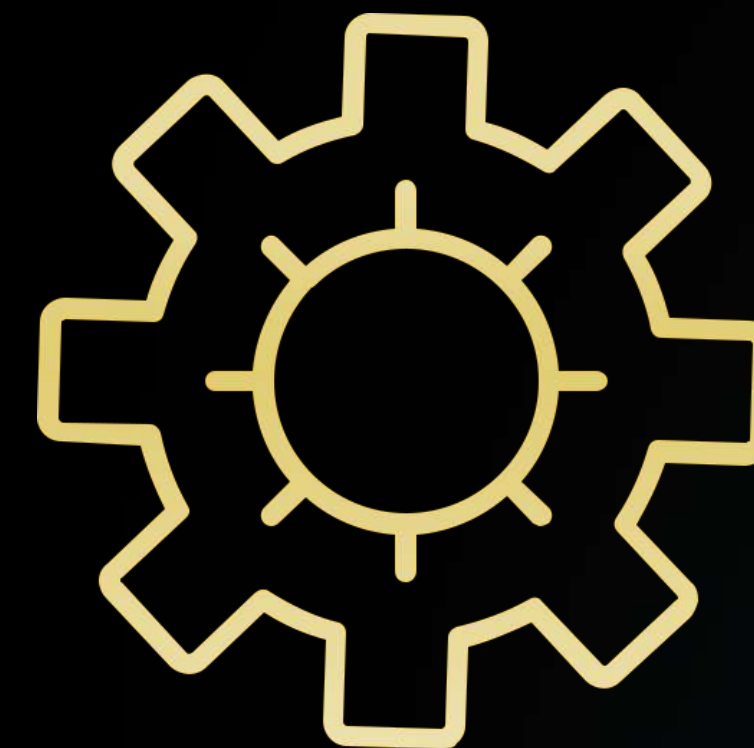
У функции есть два потребителя.



Отдел кадров



Бухгалтерия



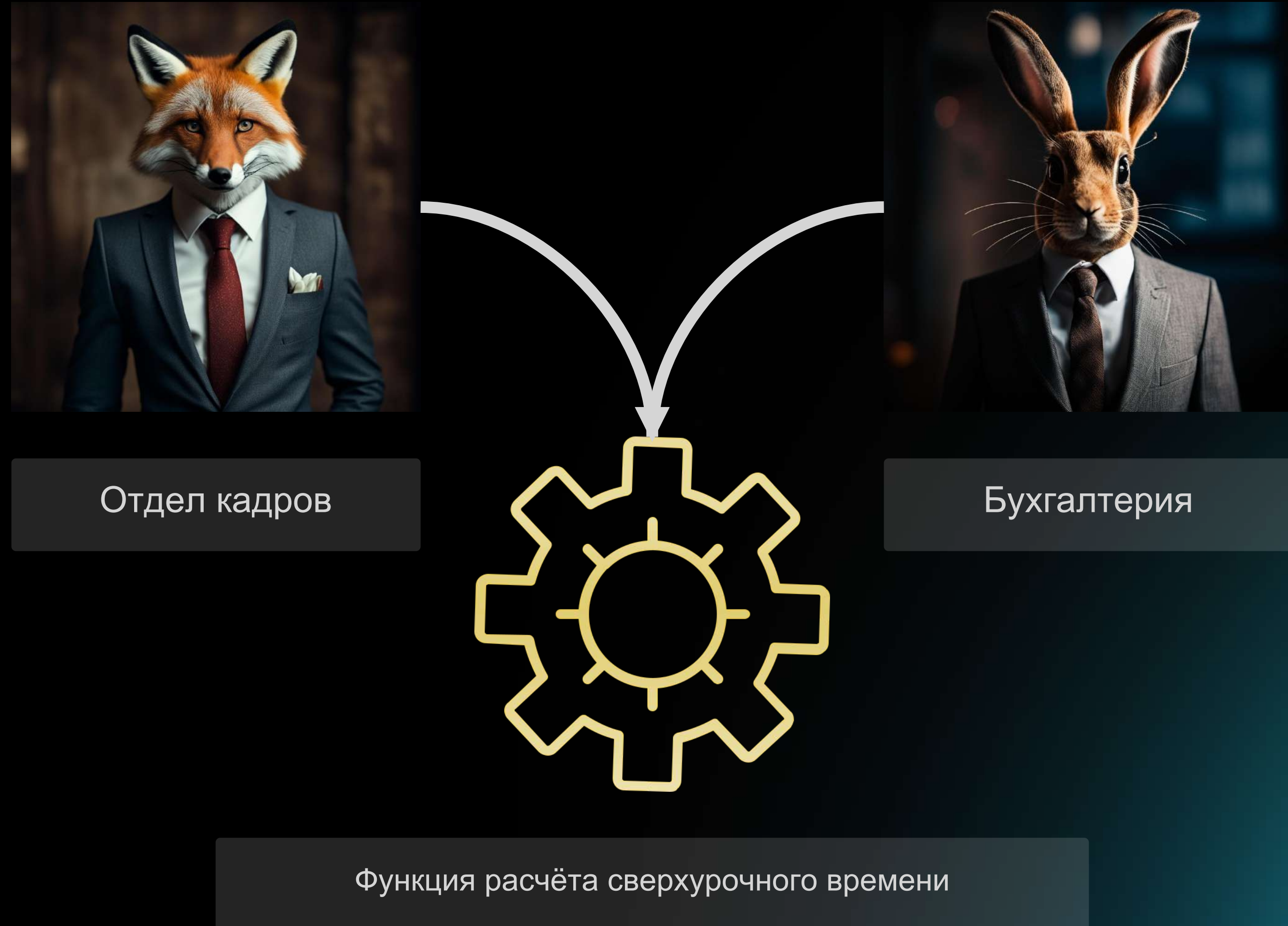
Функция расчёта сверхурочного времени

Хрупкость

Есть одна функция.

У функции есть два потребителя.

Функция отвечает перед двумя потребителями.



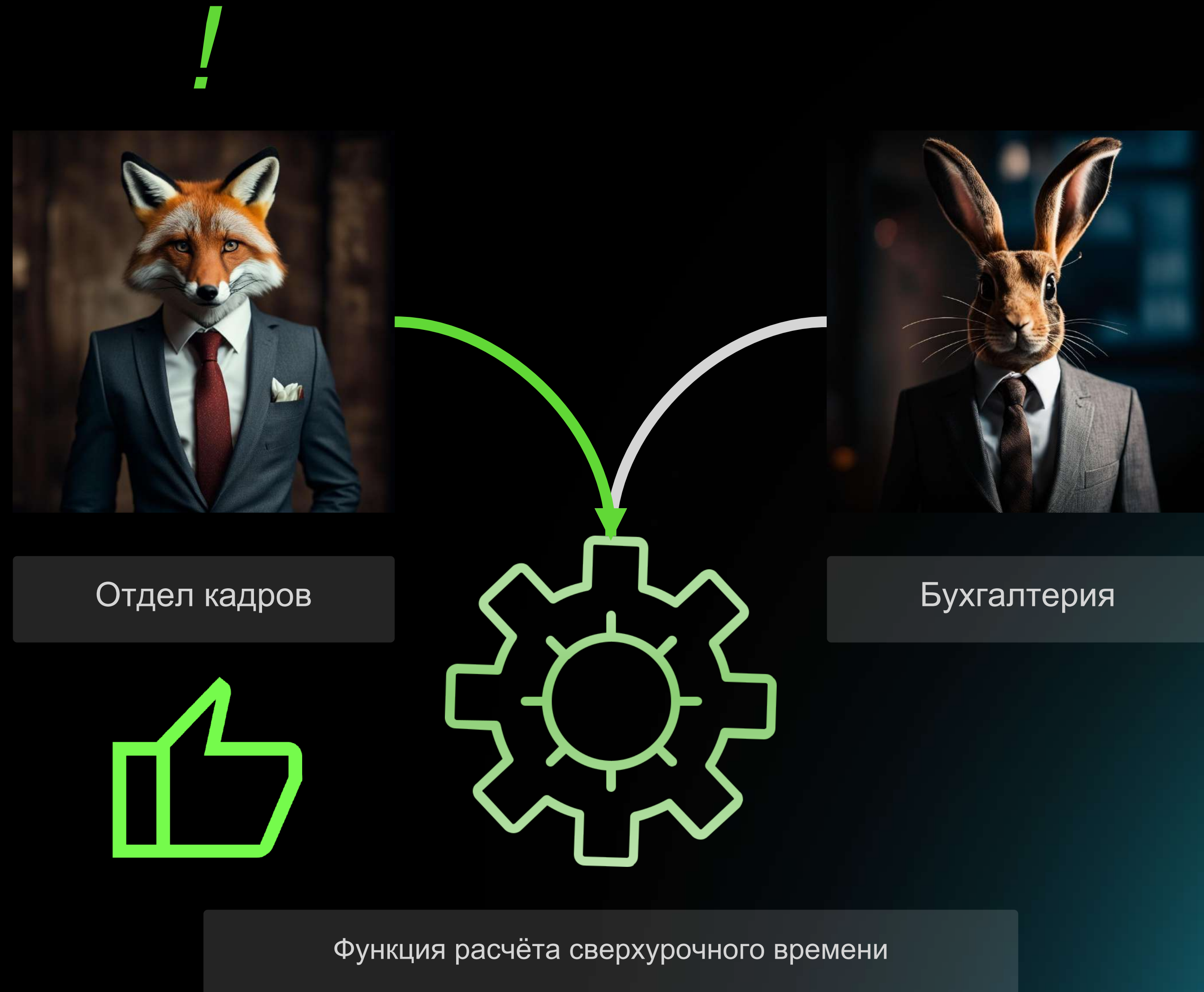
Хрупкость

Есть одна функция.

У функции есть два потребителя.

Функция отвечает перед двумя потребителями.

Первый потребитель меняет функцию.



Хрупкость

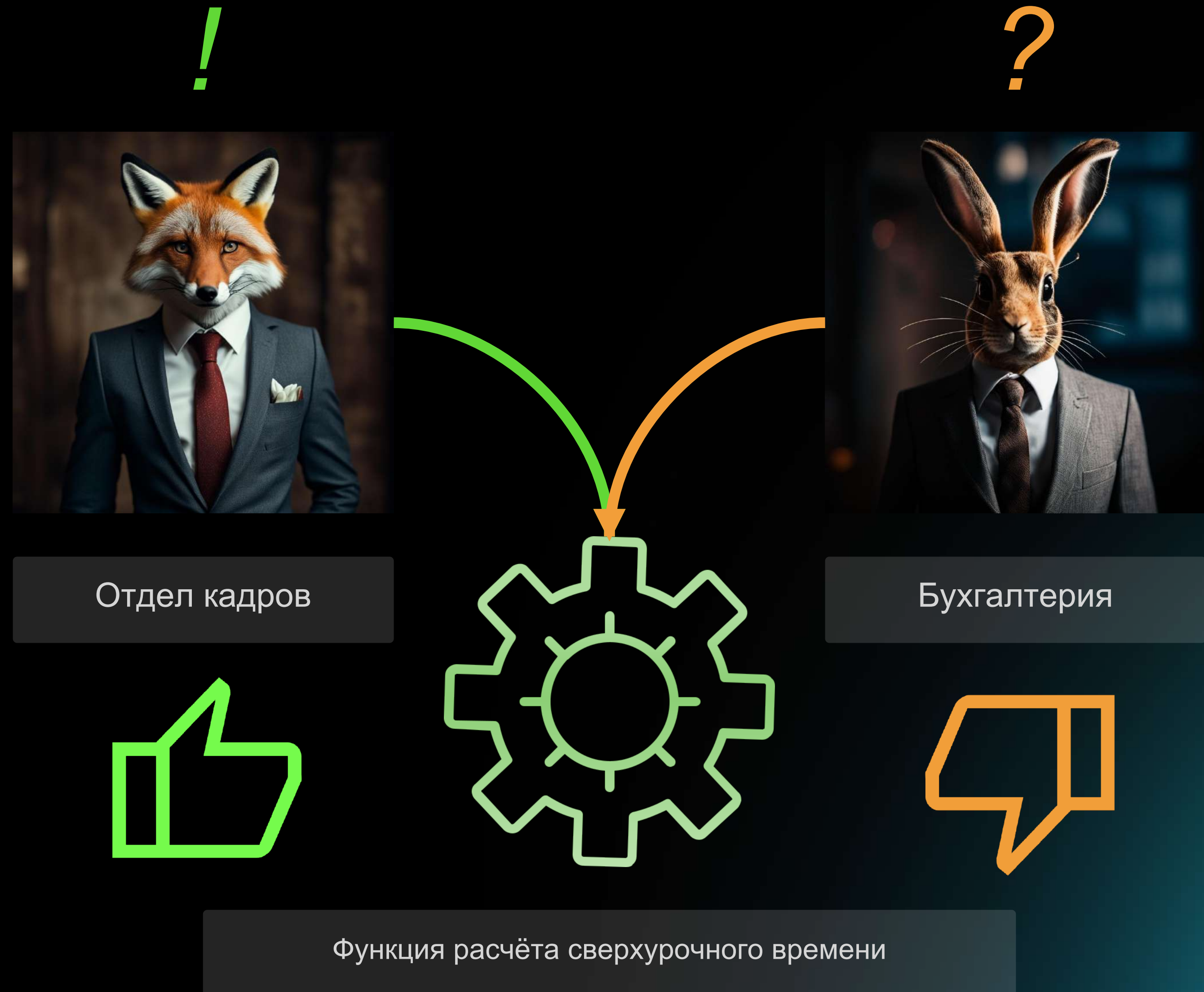
Есть одна функция.

У функции есть два потребителя.

Функция отвечает перед двумя потребителями.

Первый потребитель меняет функцию.

Работа функции меняется и для второго потребителя.



Хрупкость

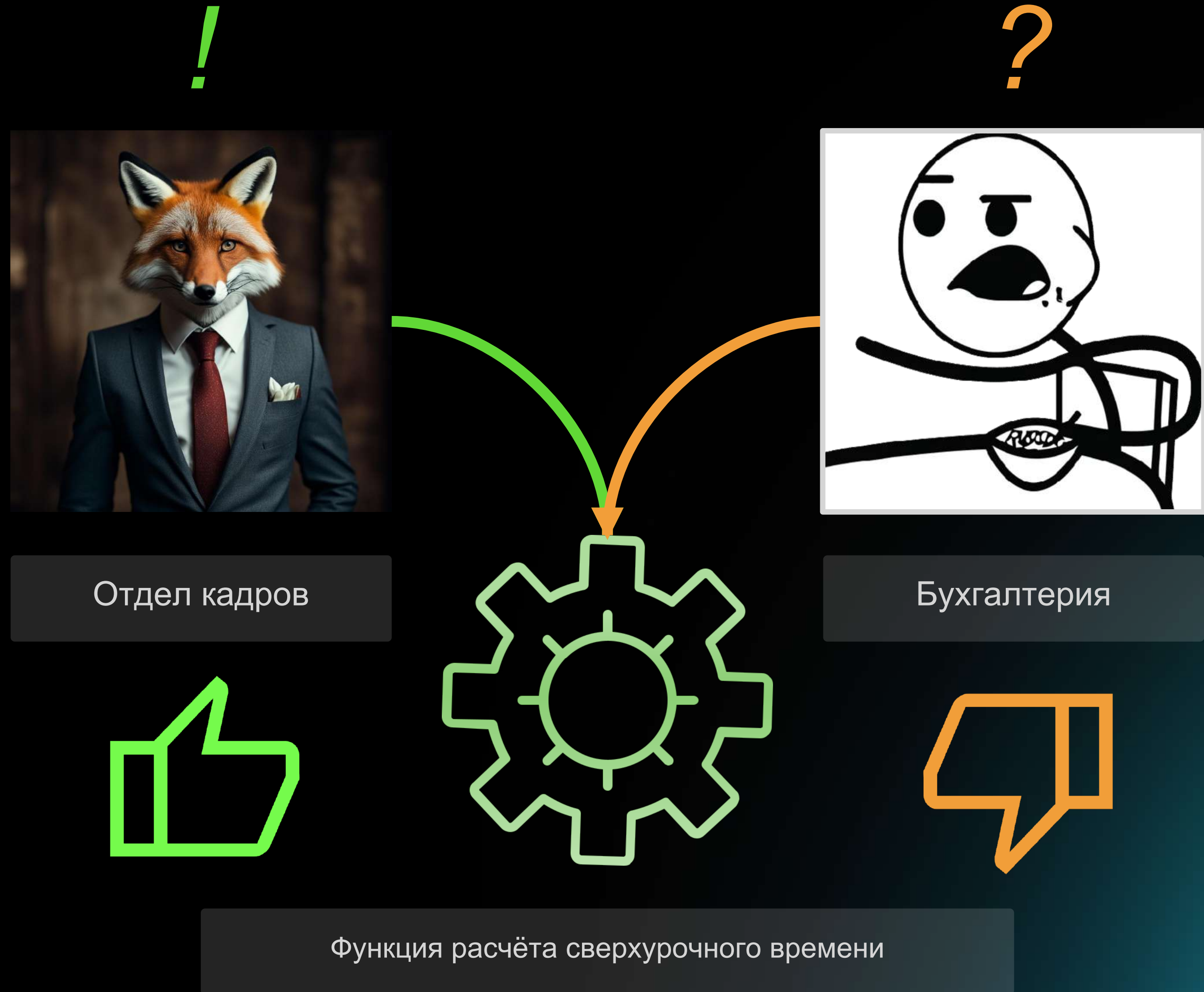
Есть одна функция.

У функции есть два потребителя.

Функция отвечает перед двумя потребителями.

Первый потребитель меняет функцию.

Работа функции меняется и для второго потребителя.



Хрупкость

Есть одна функция.

У функции есть два потребителя.

Функция отвечает перед двумя потребителями.

Первый потребитель меняет функцию.

Работа функции меняется и для второго потребителя.

Система становится хрупкой.





Single Responsibility Principle

Программный компонент
должен отвечать перед одним
потребителем.

Хрупкость

Есть одна функция.

У функции есть два потребителя.

Функция отвечает перед двумя потребителями.

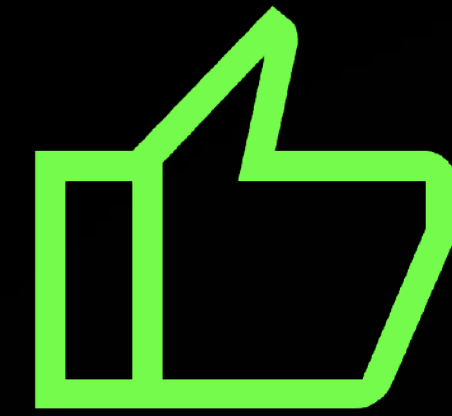
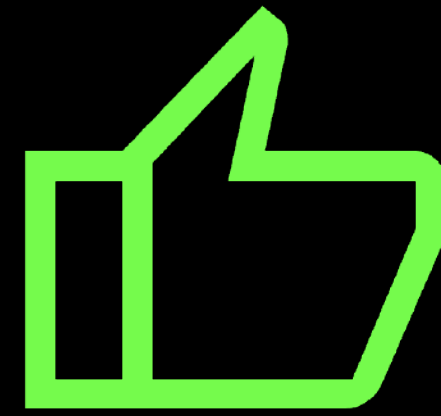
Первый потребитель меняет функцию.

Работа функции меняется и для второго потребителя.

Система становится хрупкой.

Решение:

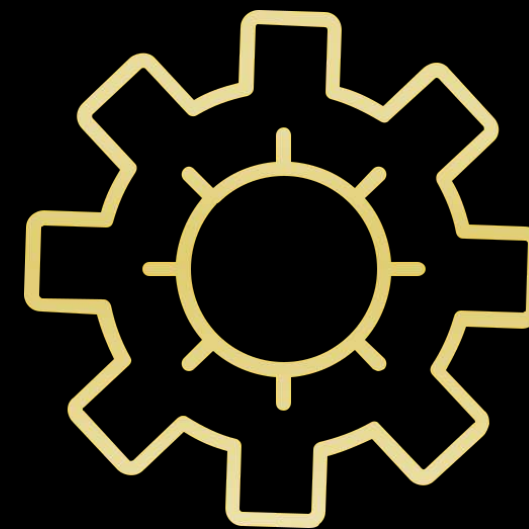
Перед каждым потребителем должна отвечать отдельная функция.



Отдел кадров



Бухгалтерия

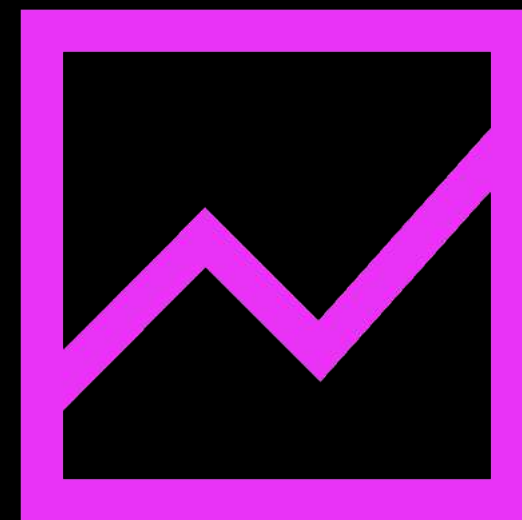


Функция для отдела кадров



Функция для бухгалтерии

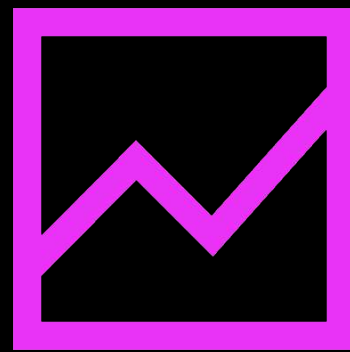
Что такое **плохая архитектура**?



Хрупкость

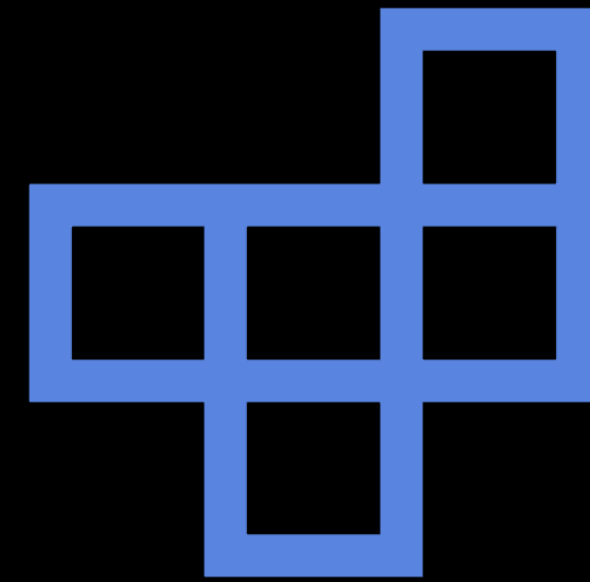
Состояние системы, в которой
один компонент отвечает за
множество реализаций.

Что такое плохая архитектура?



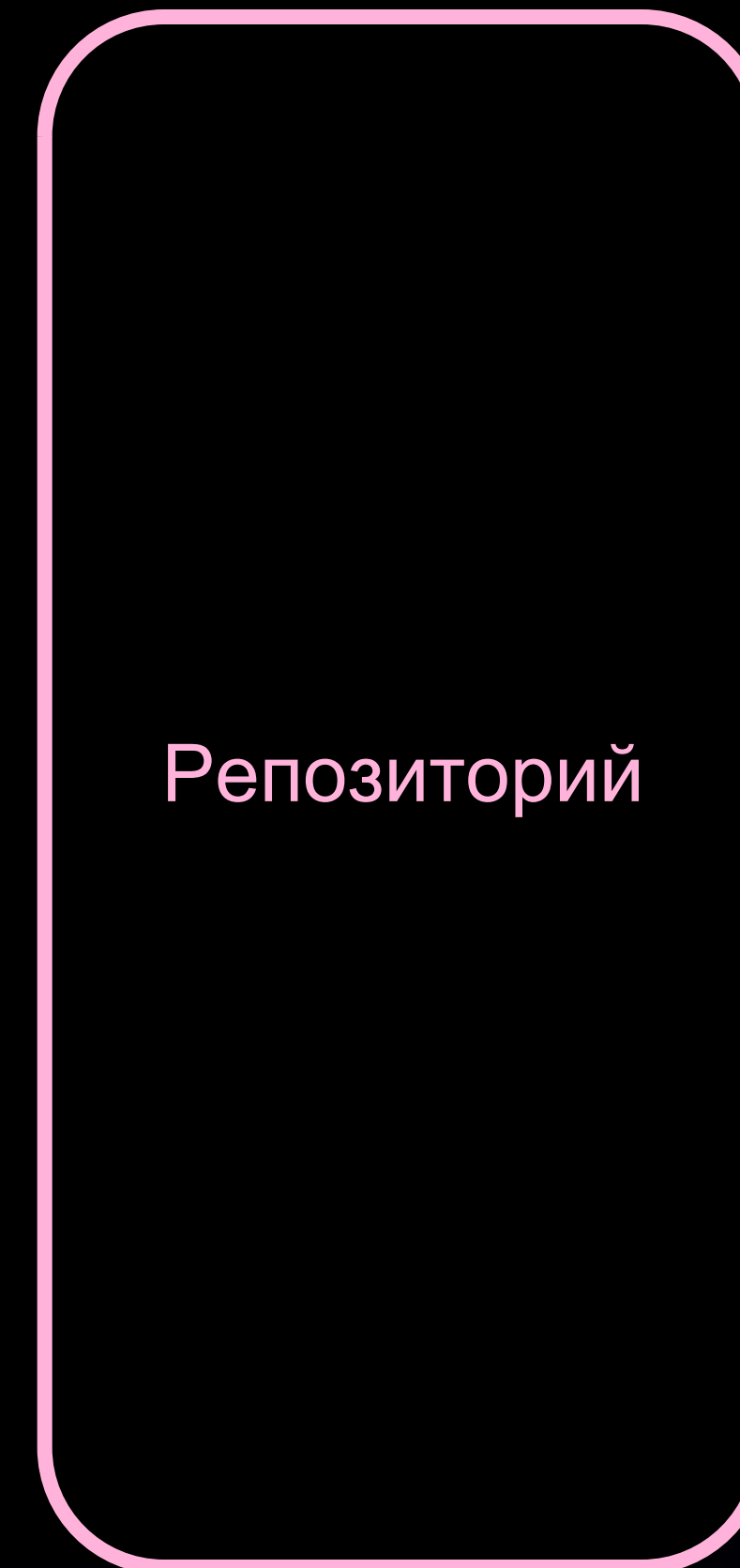
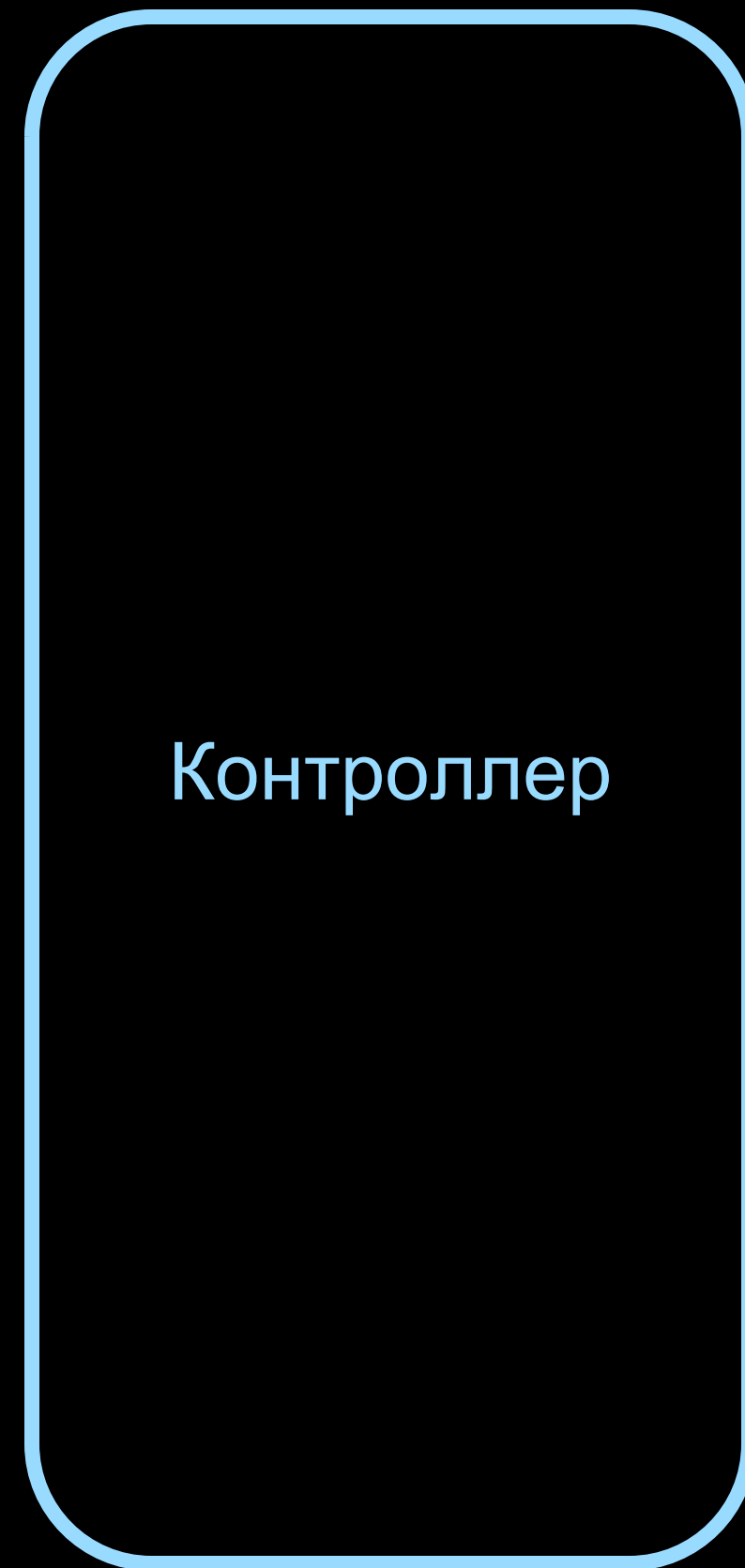
Хрупкость

Состояние системы, в которой один компонент отвечает за множество реализаций.

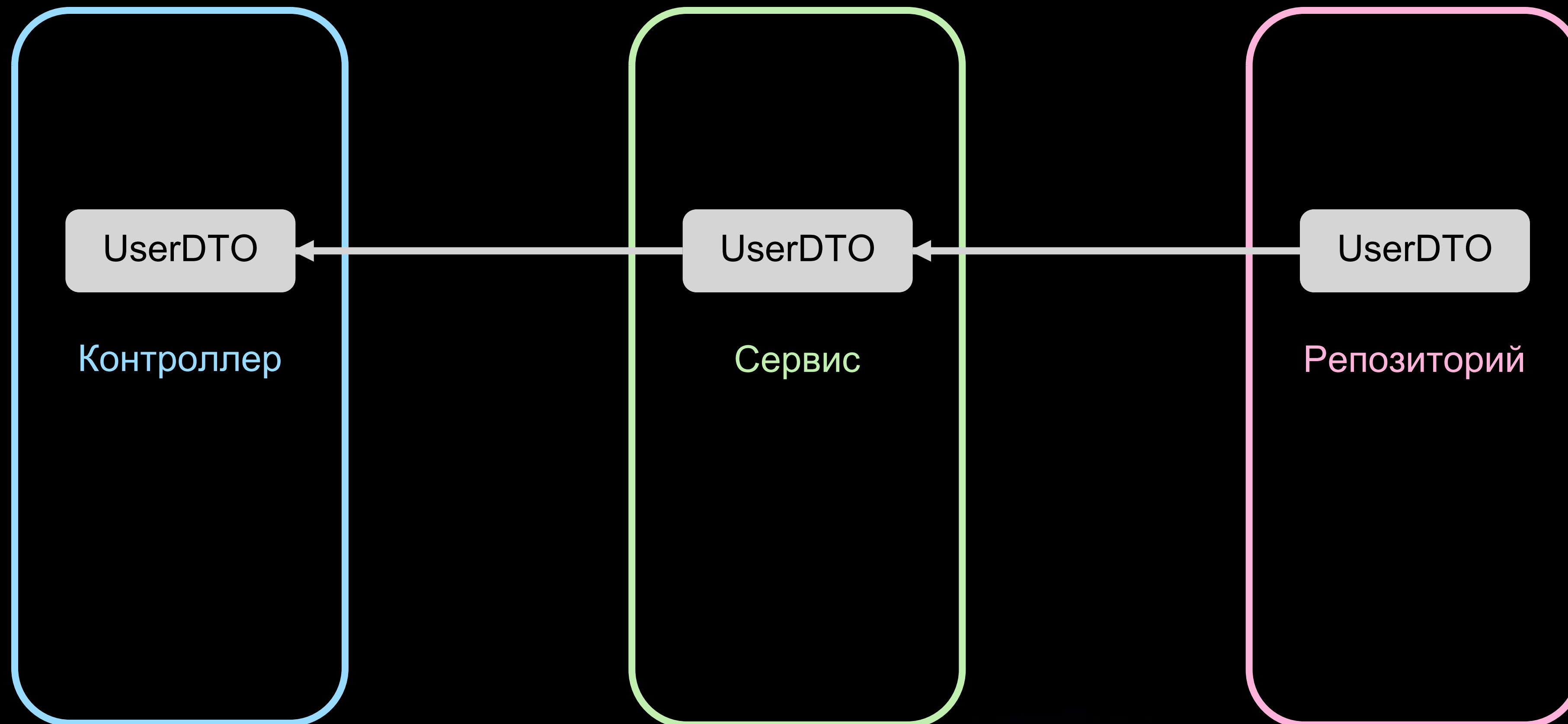


Жёсткость

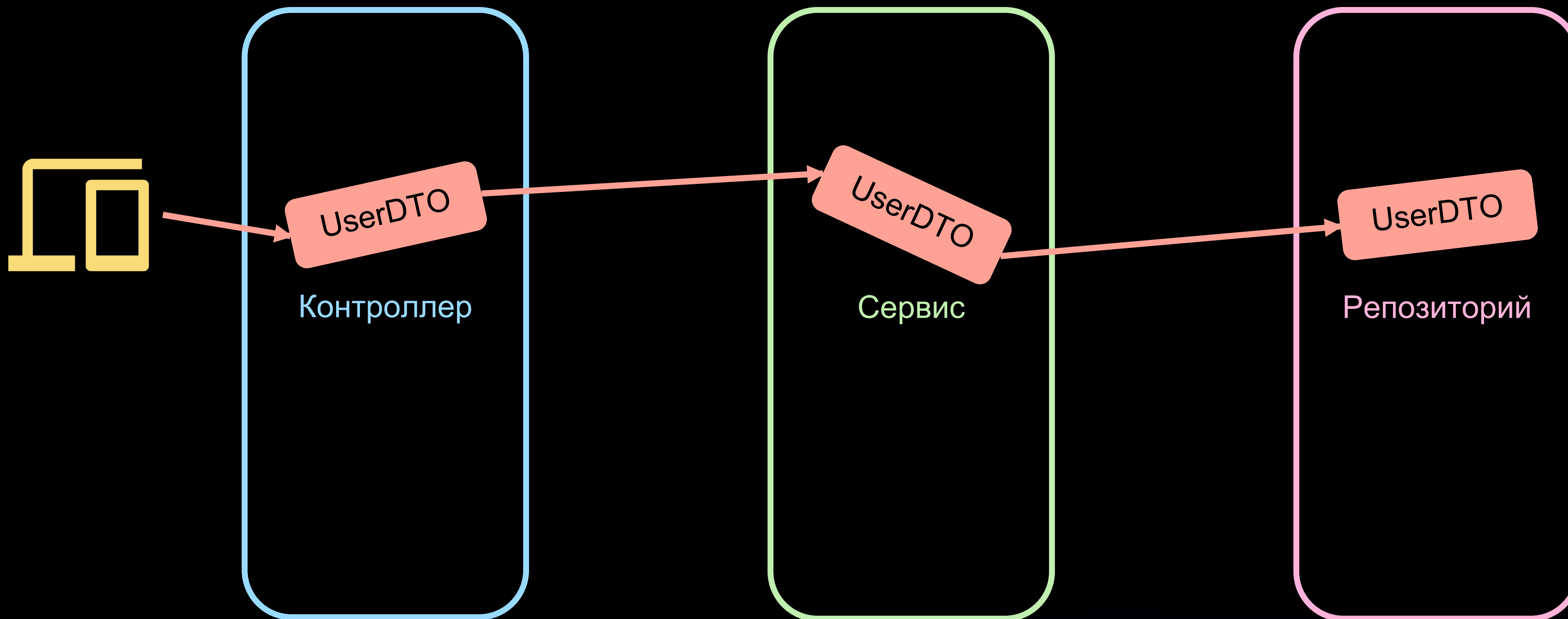
Свойство системы, при котором любое изменение одного компонента неизбежно затрагивает другие.



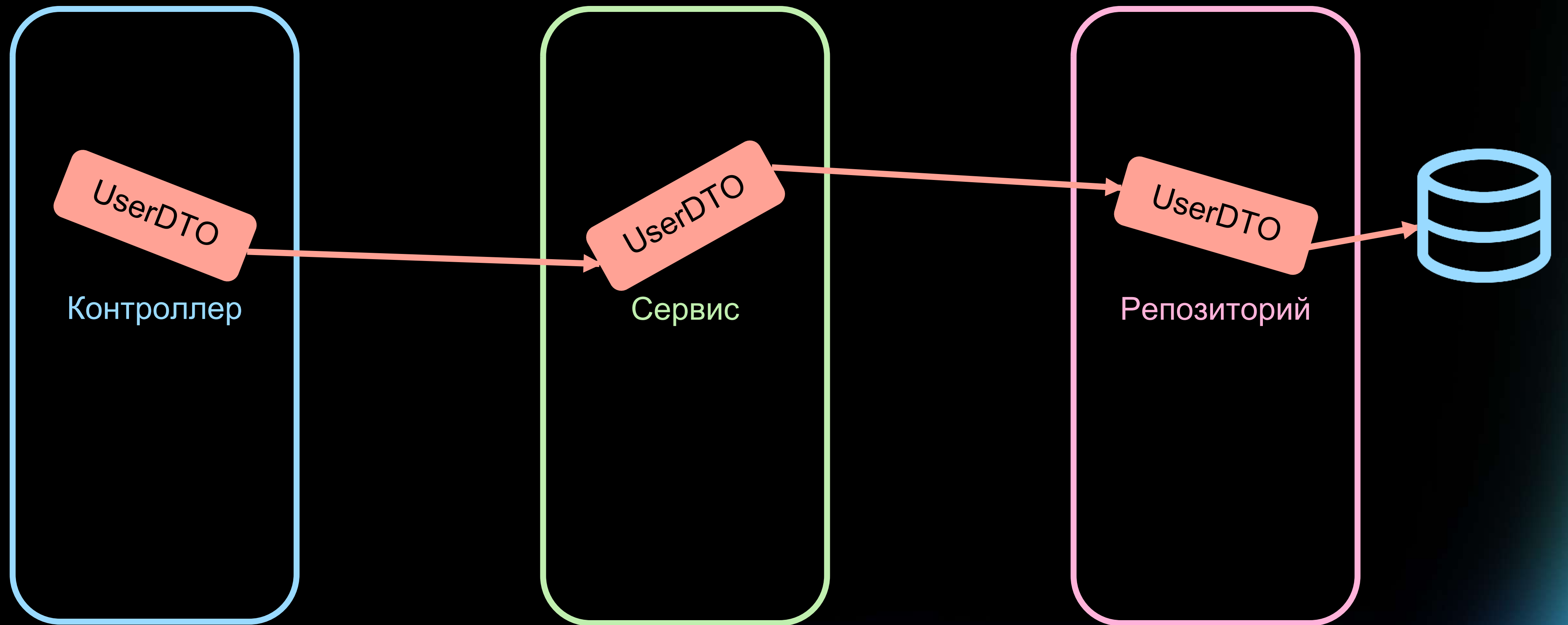
Самое обычное приложение



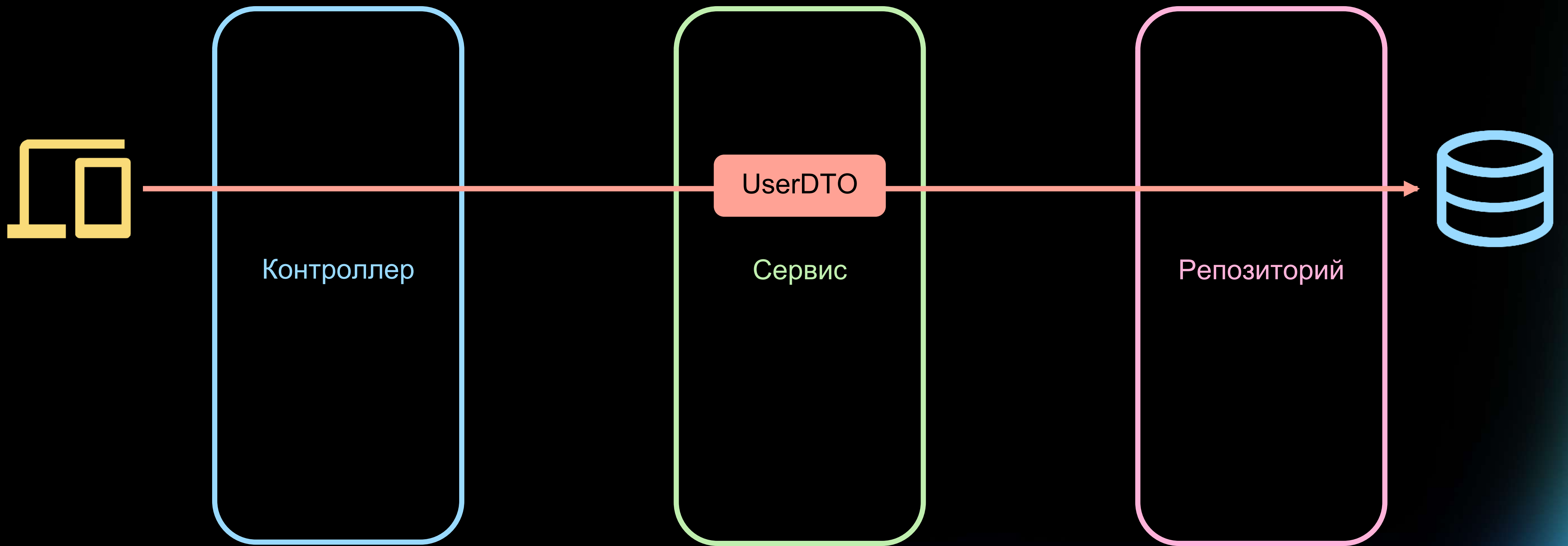
DTO протащили аж до репозитория



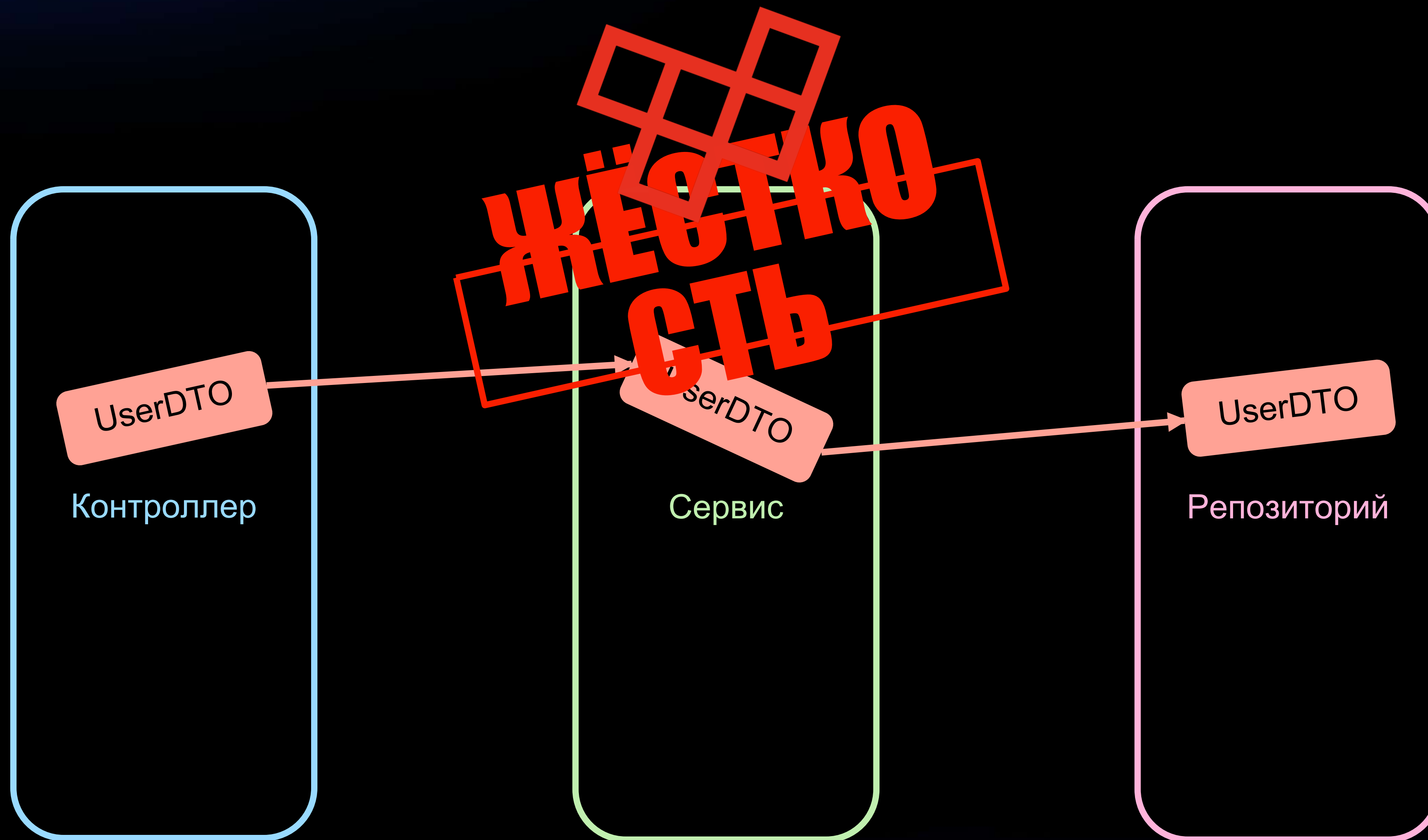
Любое изменение контракта затрагивает все компоненты



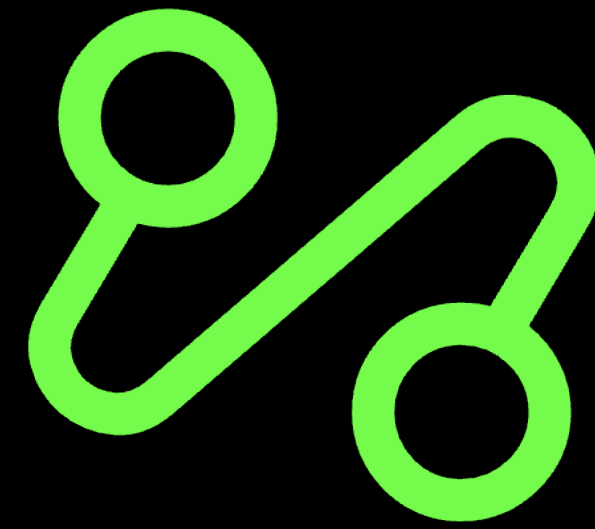
Любое изменение контракта затрагивает все компоненты



Все контракты жёстко зафиксированы между собой

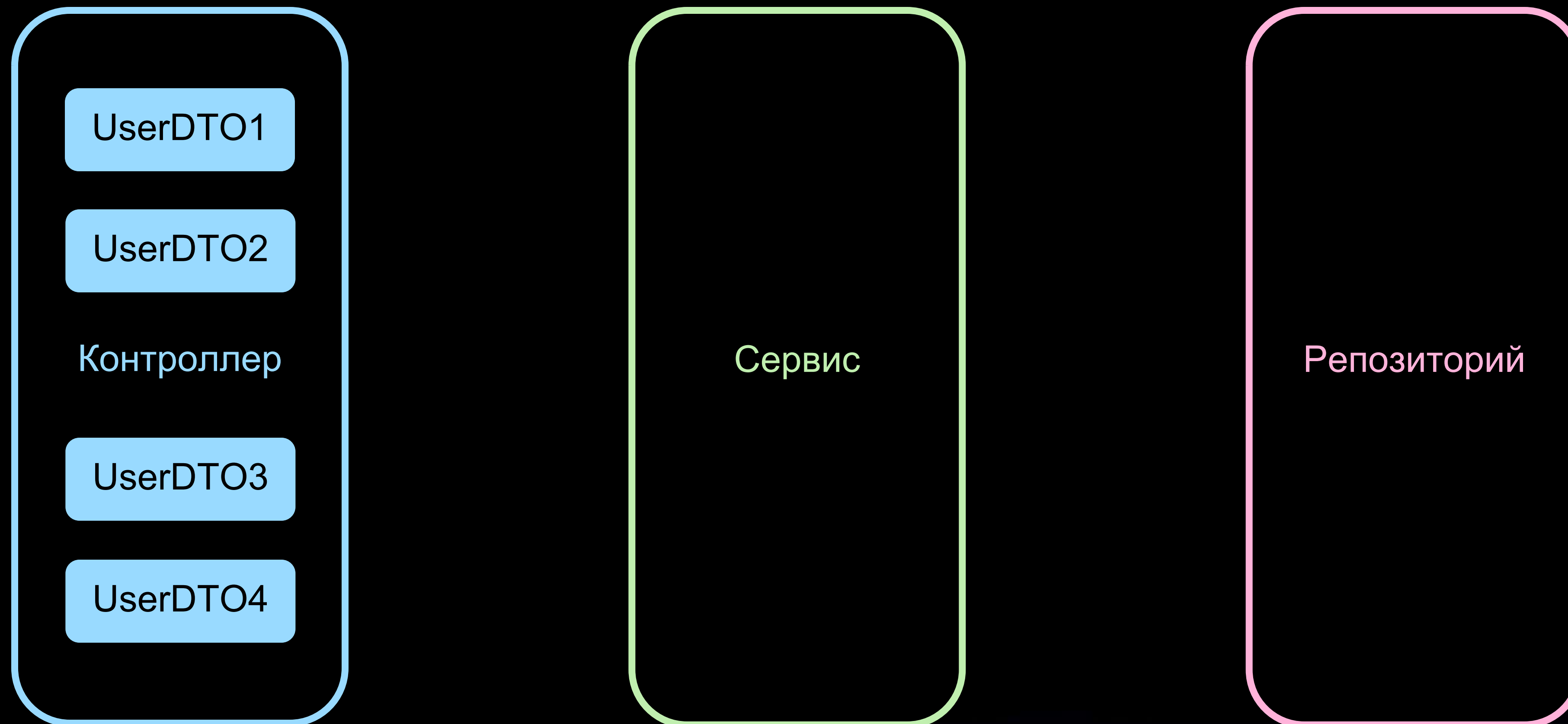


Приложение становится жёстким

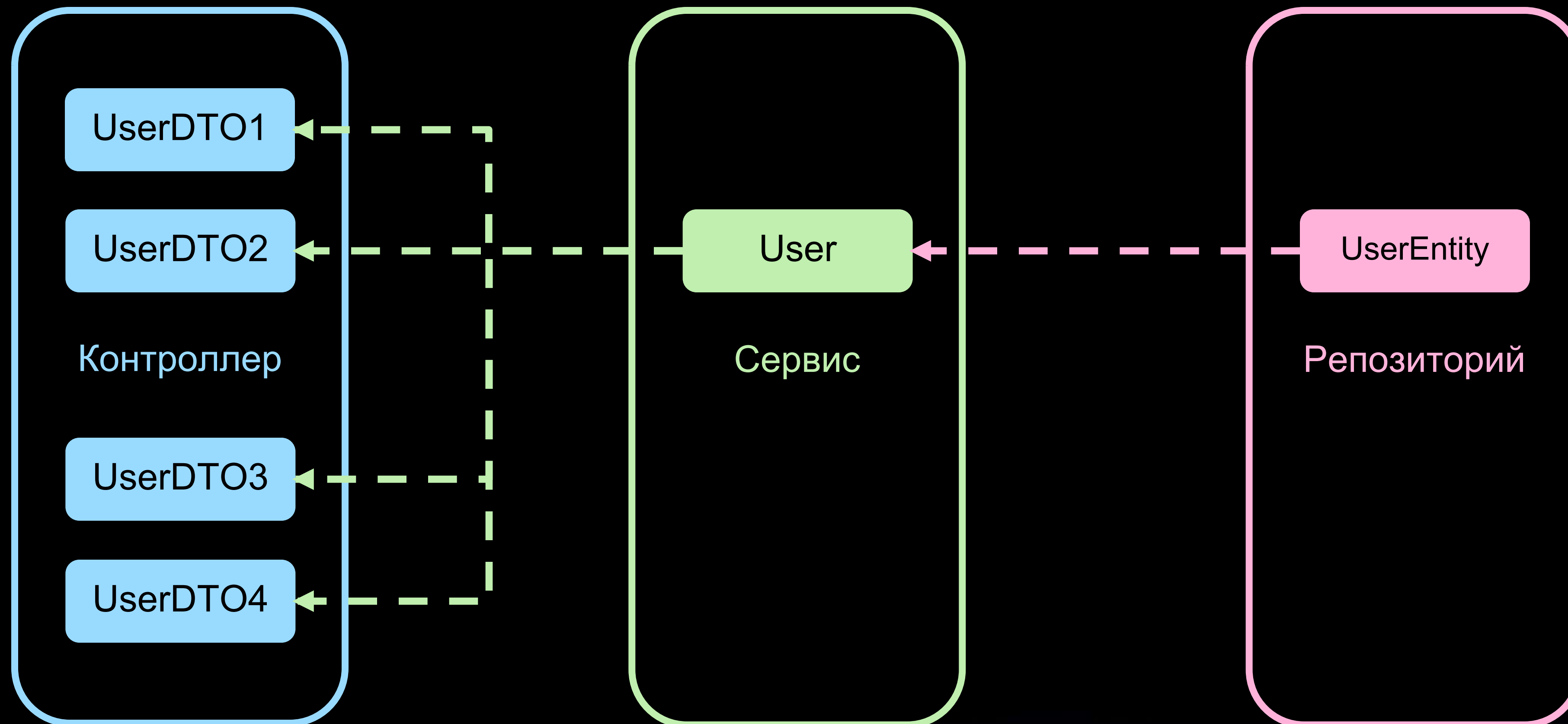


Гибкость

Свойство системы,
позволяющее производить
изменения в ней по
минимальной цене.

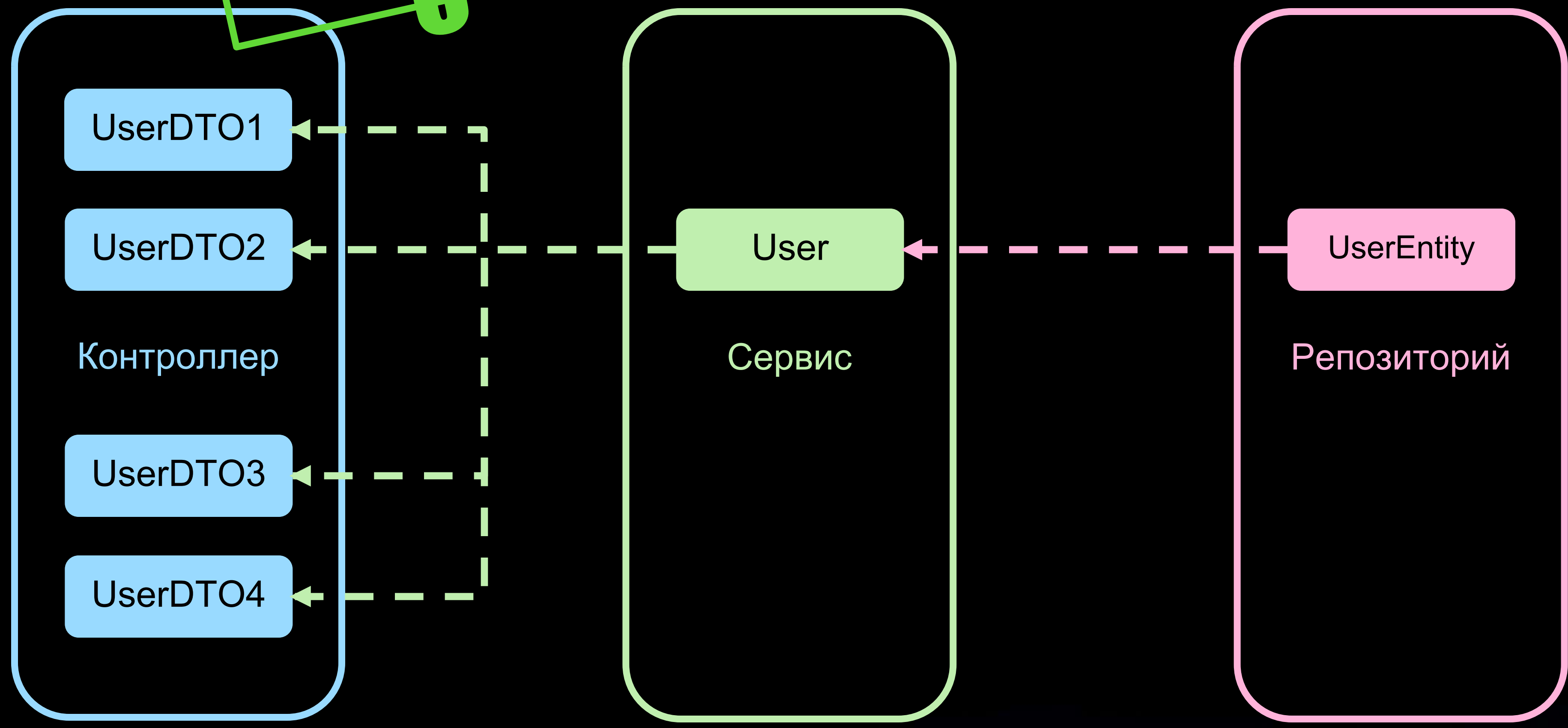


Инкапсулируем **DTO** в пределах своего клиента



Создаём необходимые сущности для каждого слоя

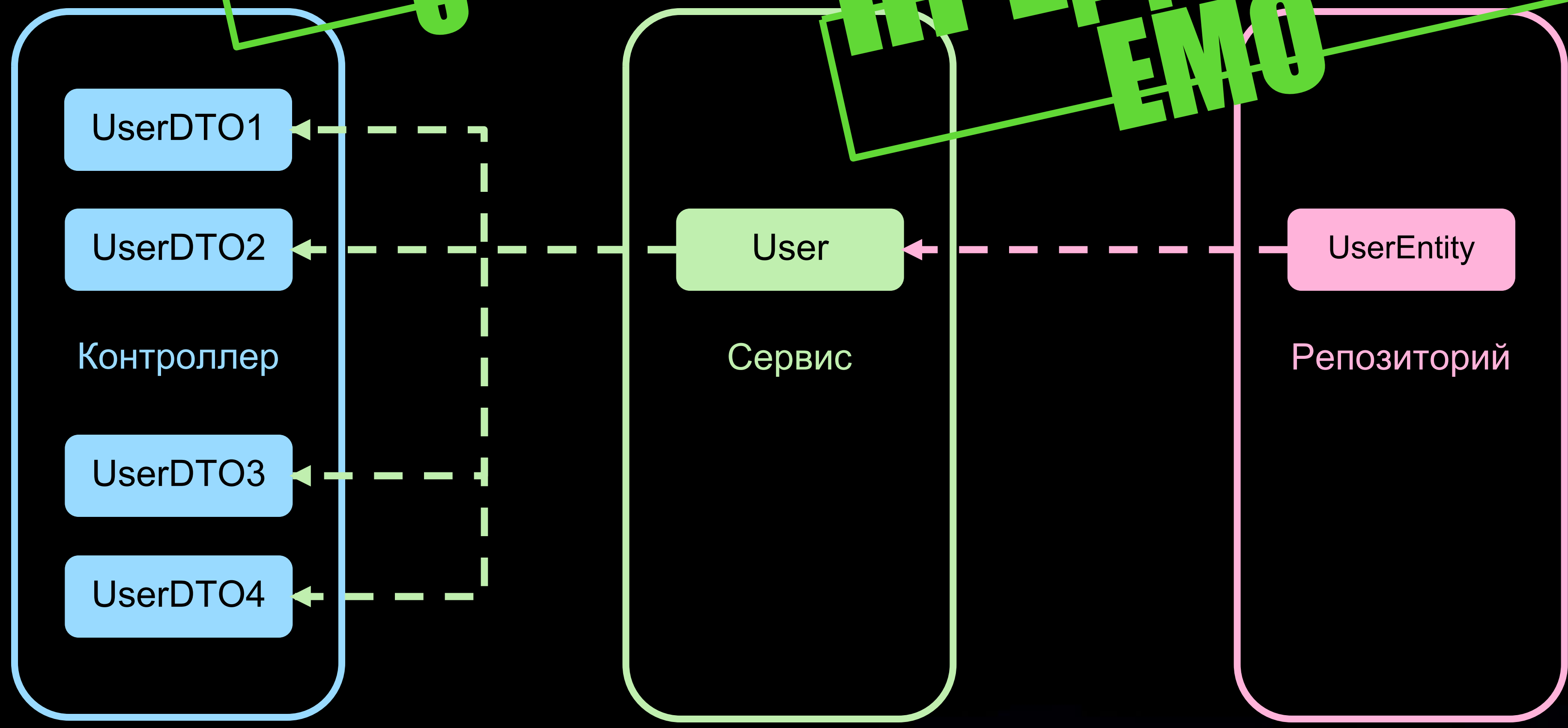
ДЁШЕВ



Это дешево

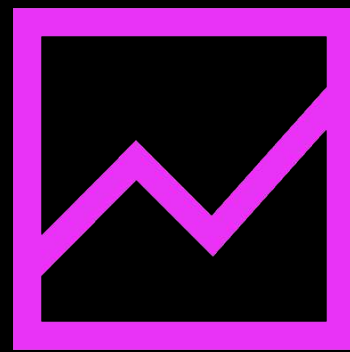
ДЁШЕВ
0

ПРЕДСКАЗУЕМО



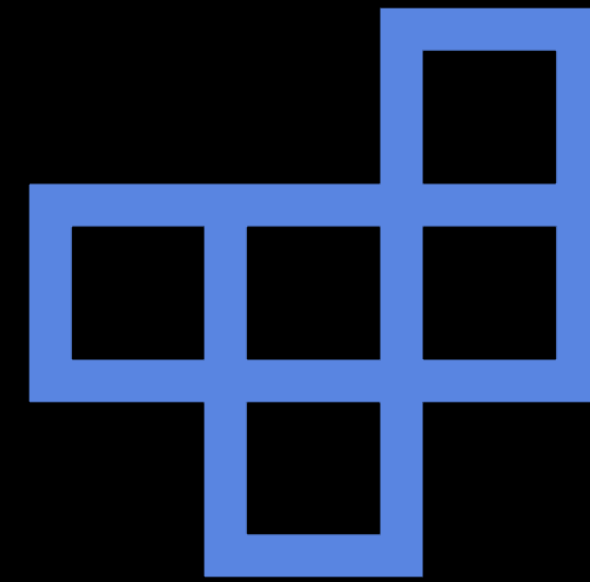
И предсказуемо

Что такое плохая архитектура?



Хрупкость

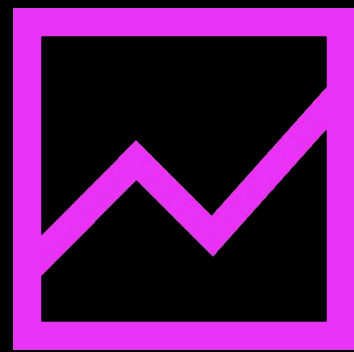
Состояние системы, в которой один компонент отвечает за множество реализаций.



Жёсткость

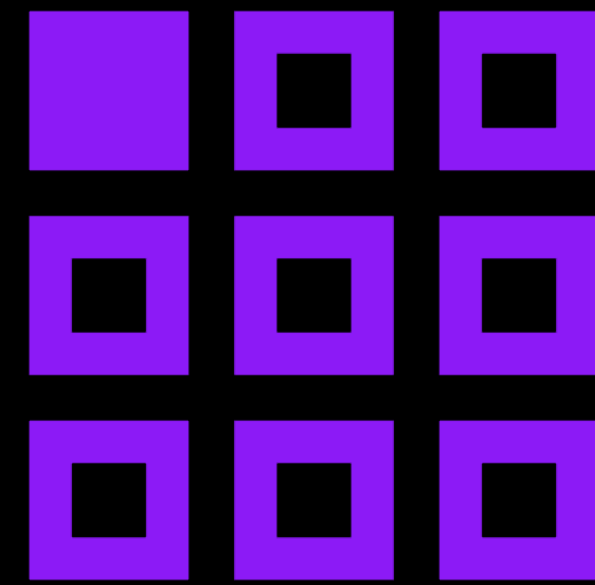
Свойство системы, при котором любое изменение одного компонента неизбежно затрагивает другие.

Что такое **плохая архитектура**?



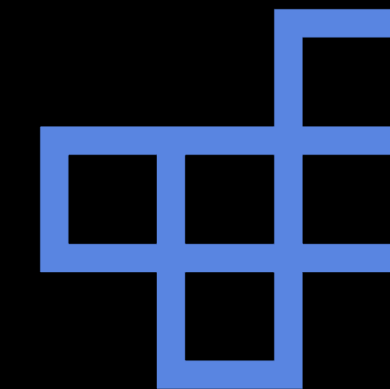
Хрупкость

Состояние системы, в которой один компонент отвечает за множество реализаций.



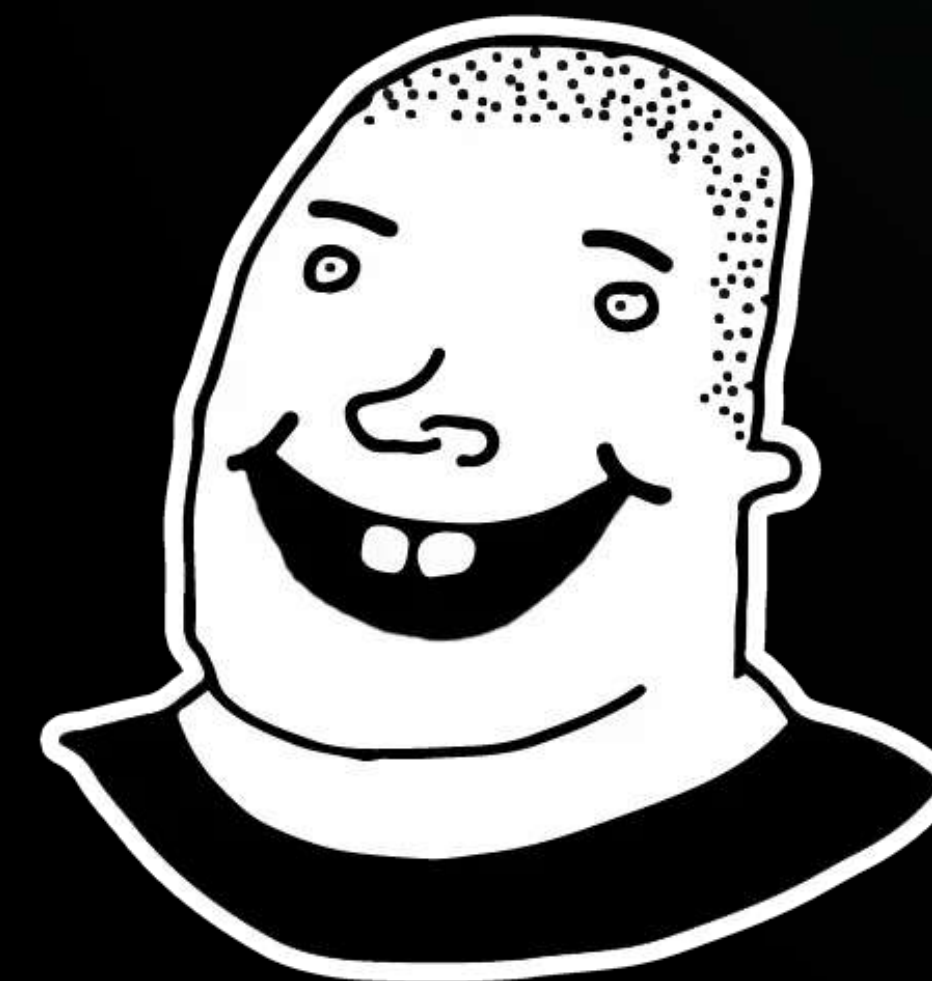
Неподвижность

Система написана с таким количеством специфичных особенностей, что её невозможно переиспользовать.

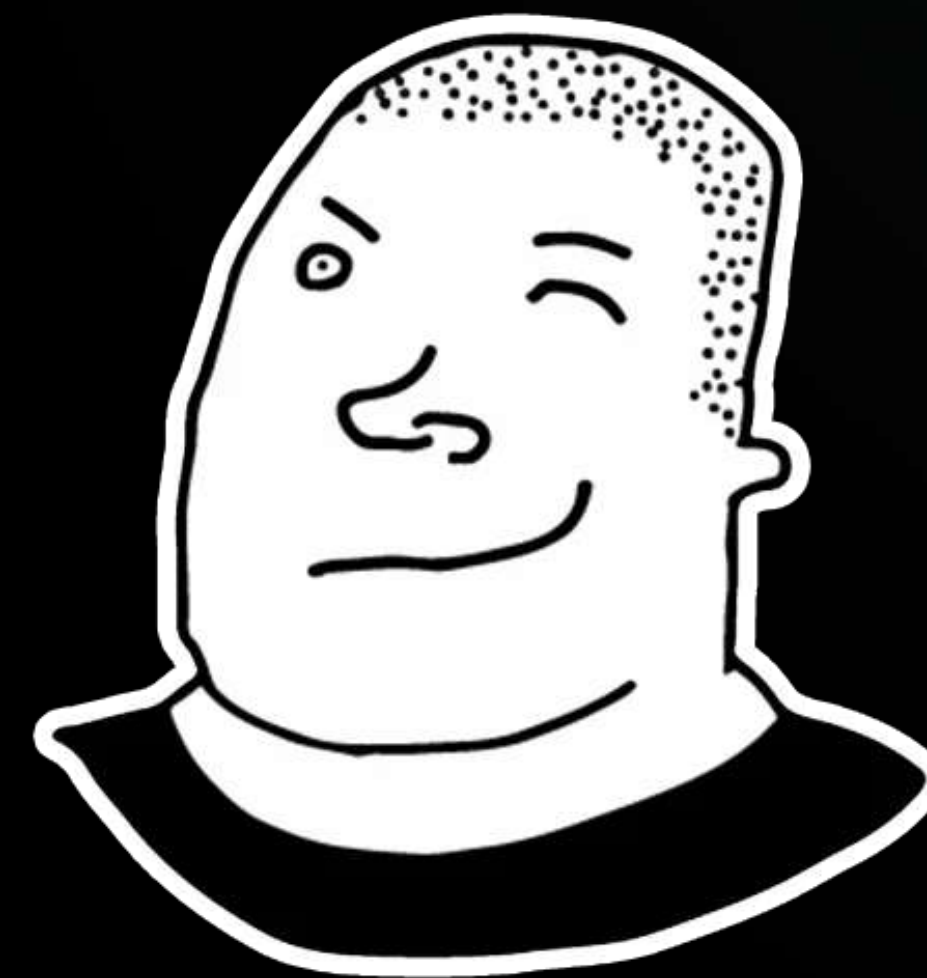
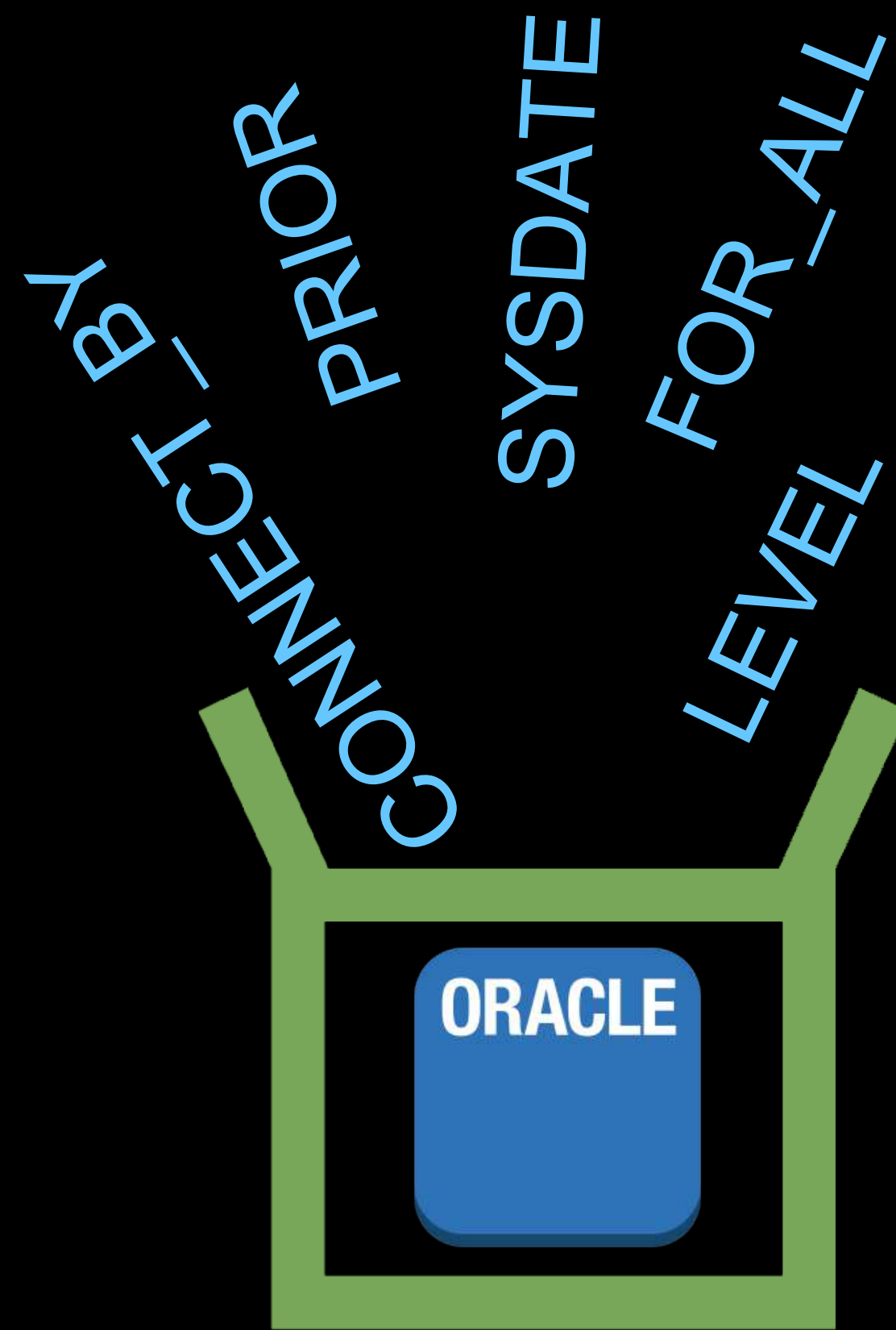


Жёсткость

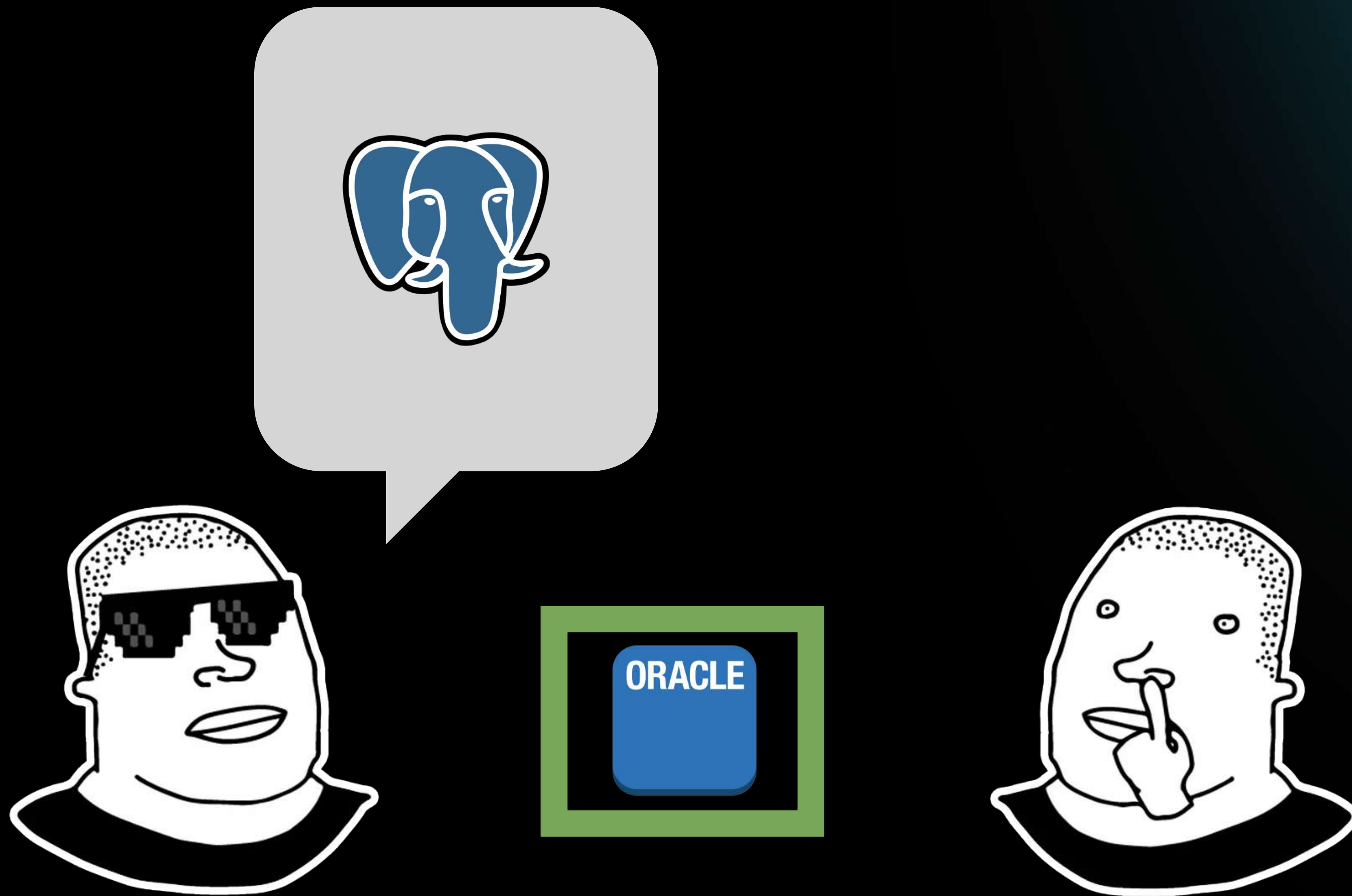
Свойство системы, при котором любое изменение одного компонента неизбежно затрагивает другие.



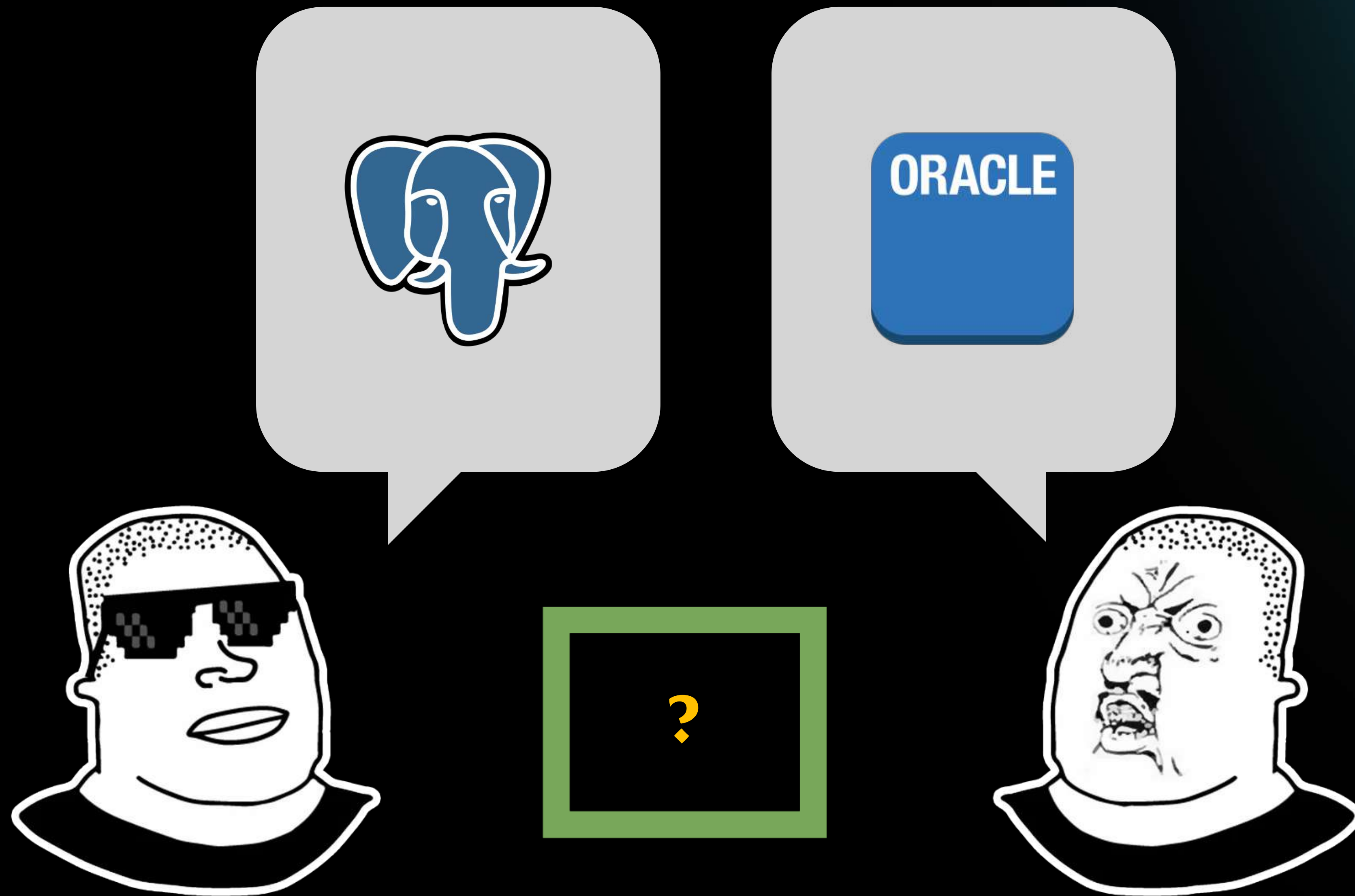
Итак, у вас Spring JDBC и база данных на Oracle DB.



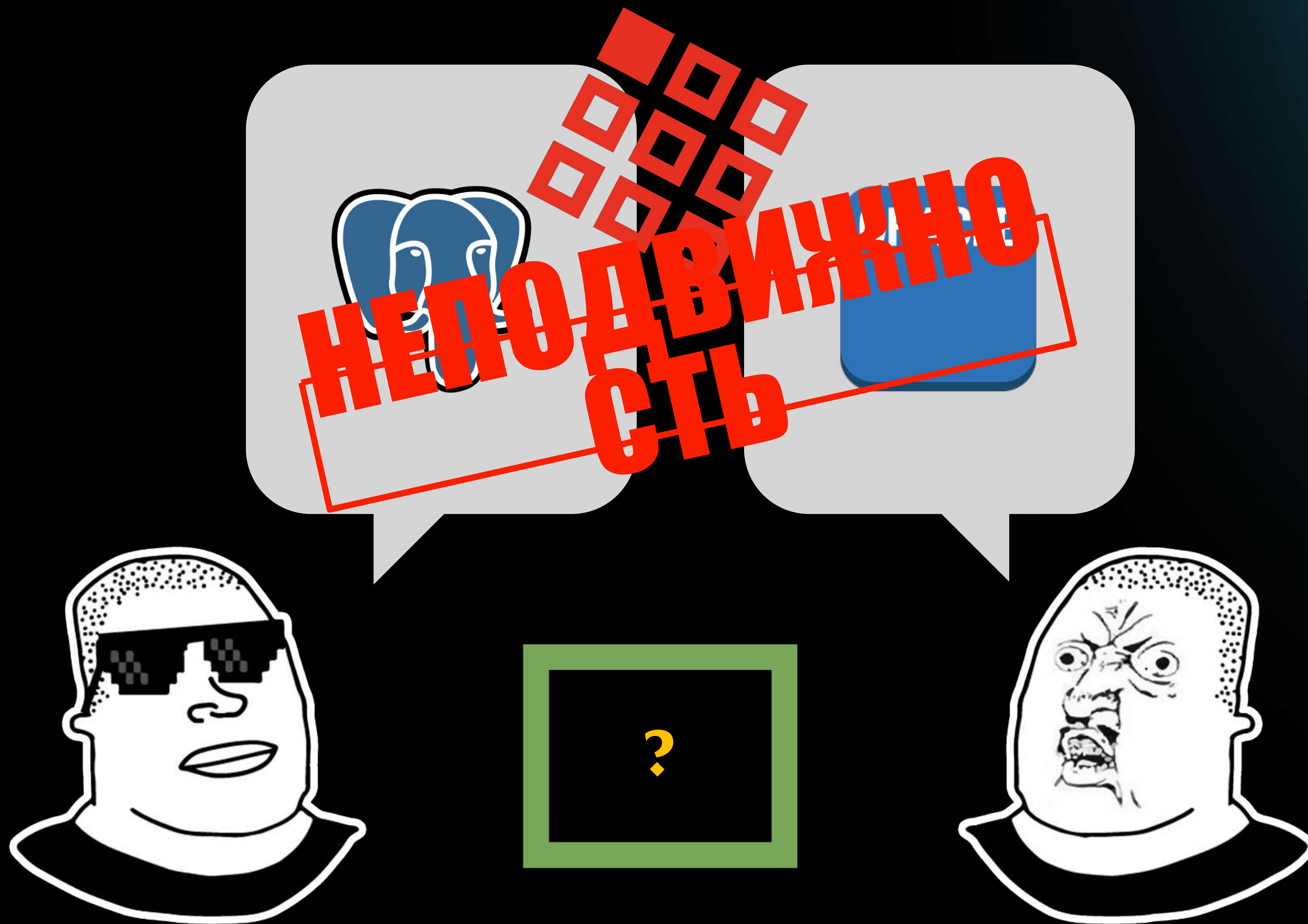
Все ваши запросы написаны вручную на диалекте Oracle DB.



Настало время переехать на [Postgres](#).

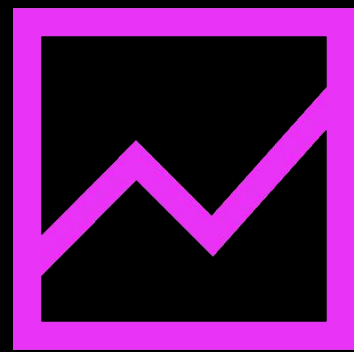


Какой Postgres? У нас OracleDB приколочен гвоздями через Spring JDBC!



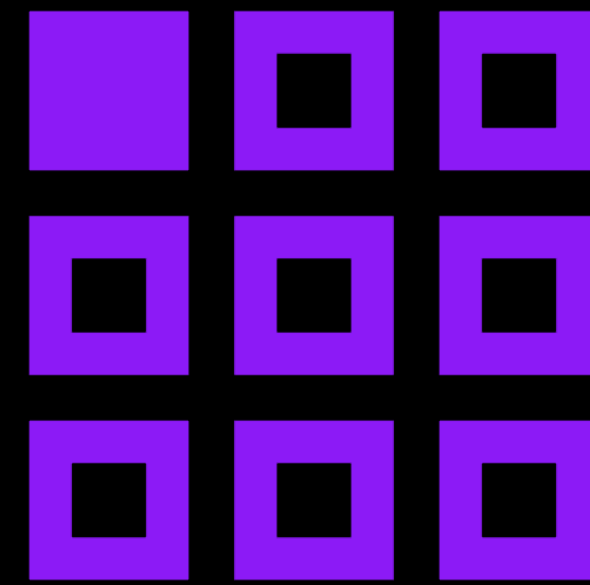
Наше приложение не может работать на другом ландшафте.

Что такое **плохая архитектура**?



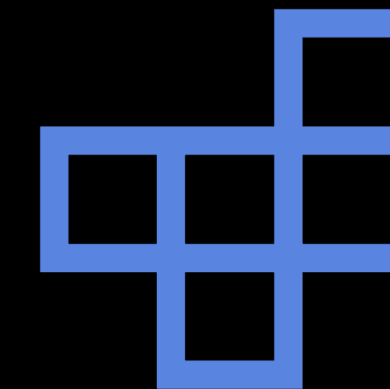
Хрупкость

Состояние системы, в которой один компонент отвечает за множество реализаций.



Неподвижность

Система написана с таким количеством специфичных особенностей, что её невозможно переиспользовать.



Жёсткость

Свойство системы, при котором любое изменение одного компонента неизбежно затрагивает другие.

Какие теории помогли бы нам избежать плохой архитектуры?

Фокус на **домене**

Глубоко понимать систему ещё на старте разработки.



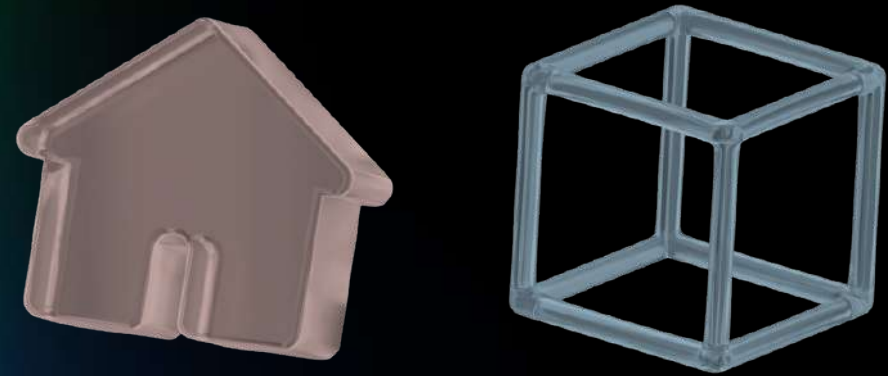
Какие теории помогли бы нам избежать плохой архитектуры?

Фокус на **домене**

Глубоко понимать систему ещё на старте разработки.

Чёткие **границы контекстов**

Не смешивать логику нескольких компонентов.



Какие теории помогли бы нам избежать плохой архитектуры?

Фокус на **домене**

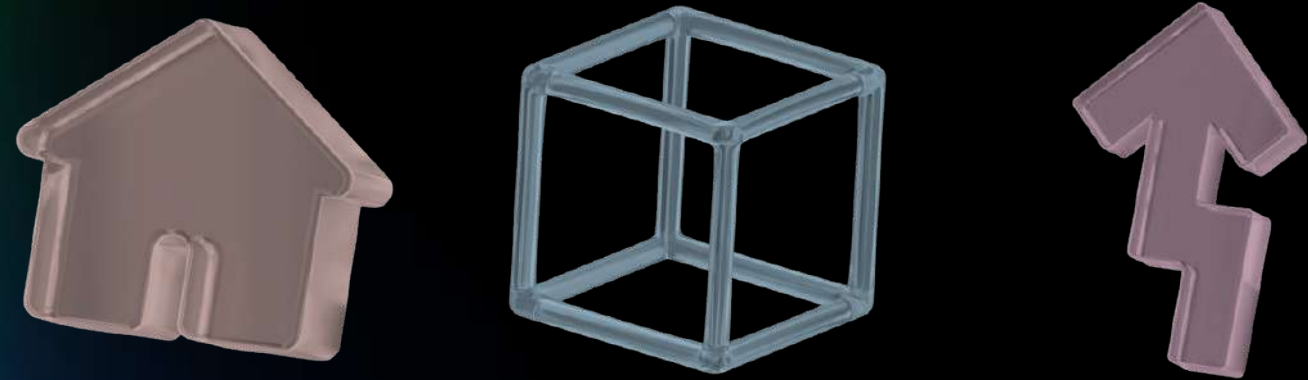
Глубоко понимать систему ещё на старте разработки.

Чёткие **границы контекстов**

Не смешивать логику нескольких компонентов.

Акцент на **гибкости**

Сделать приложение независимым от инфраструктуры.



Можно ли подвести под хорошие практики
общую теоретическую базу?

Можно ли подвести под хорошие практики
общую теоретическую базу?

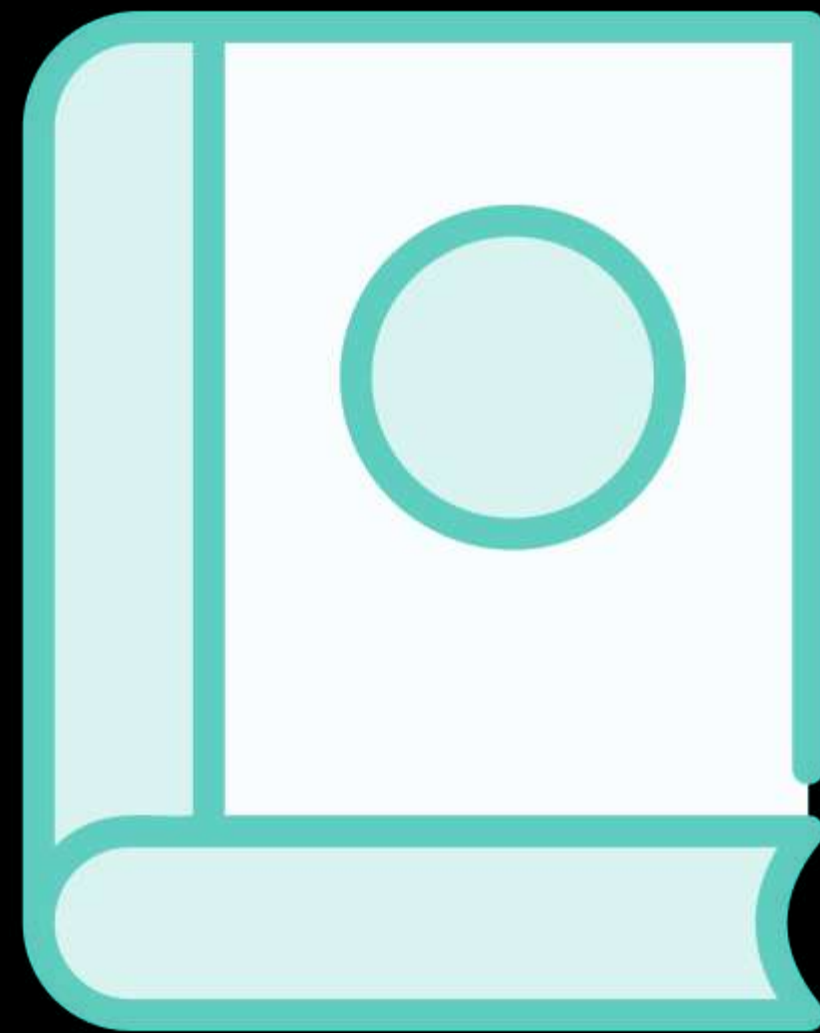
Можно.

Domain Driven Design

Предметно-ориентированное проектирование

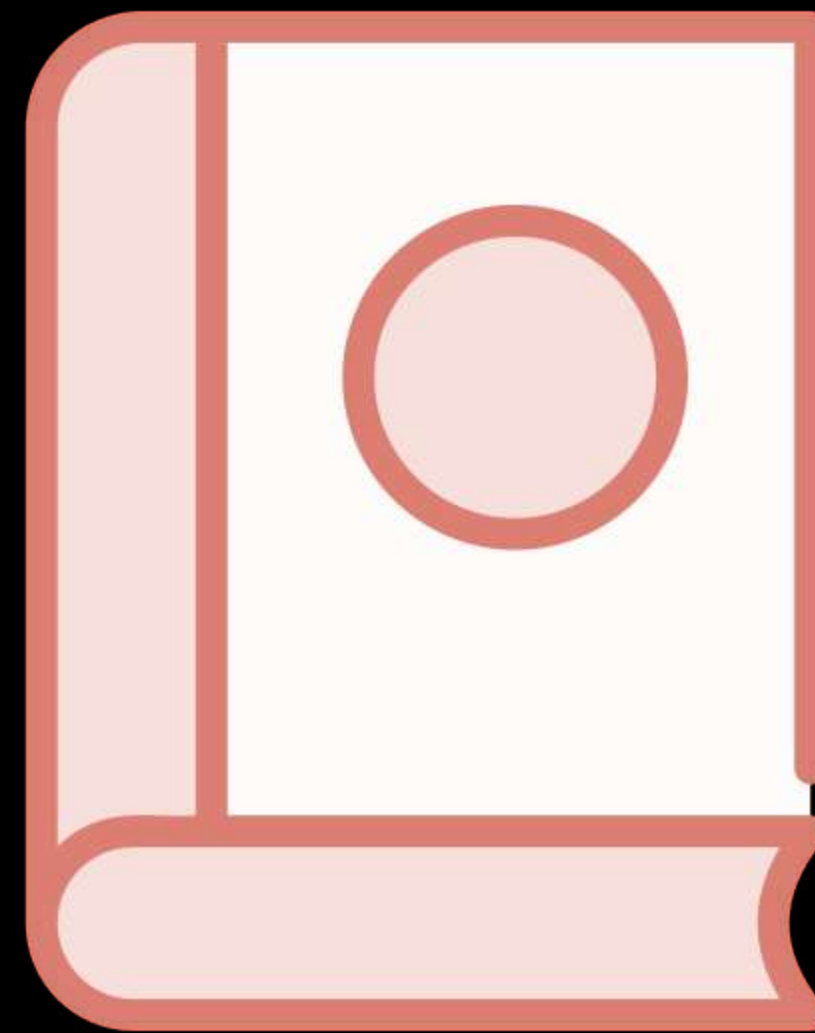
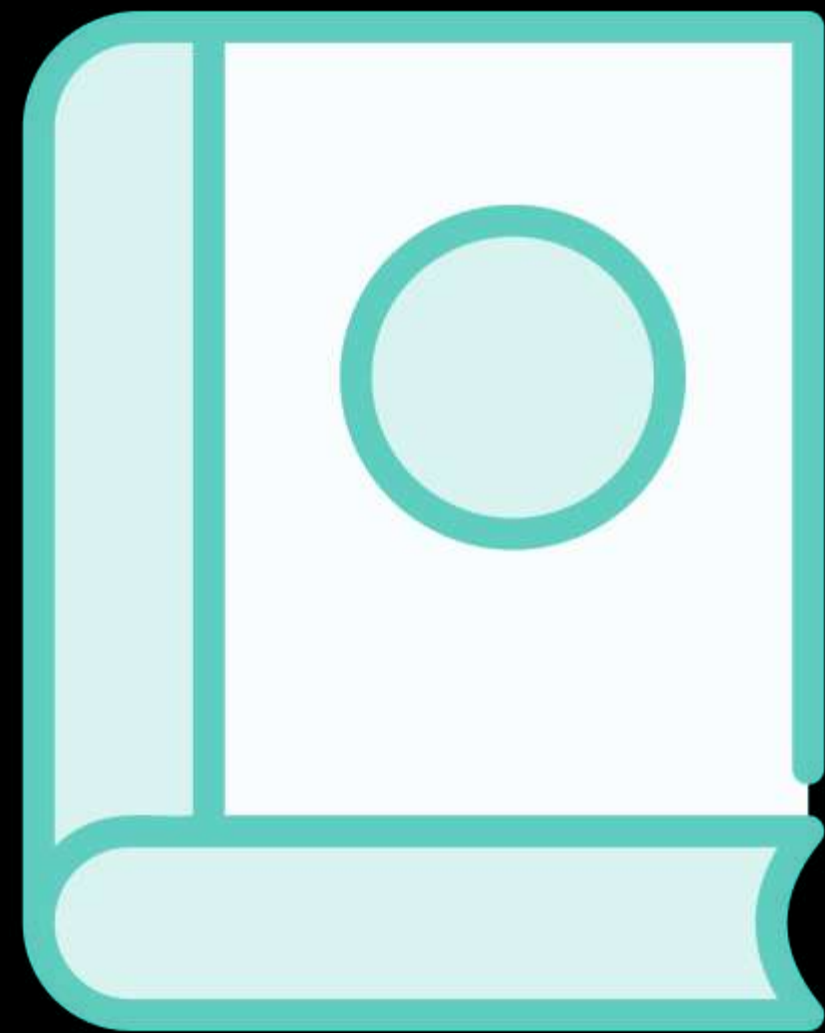
Domain Driven Design

Предметно-ориентированное проектирование



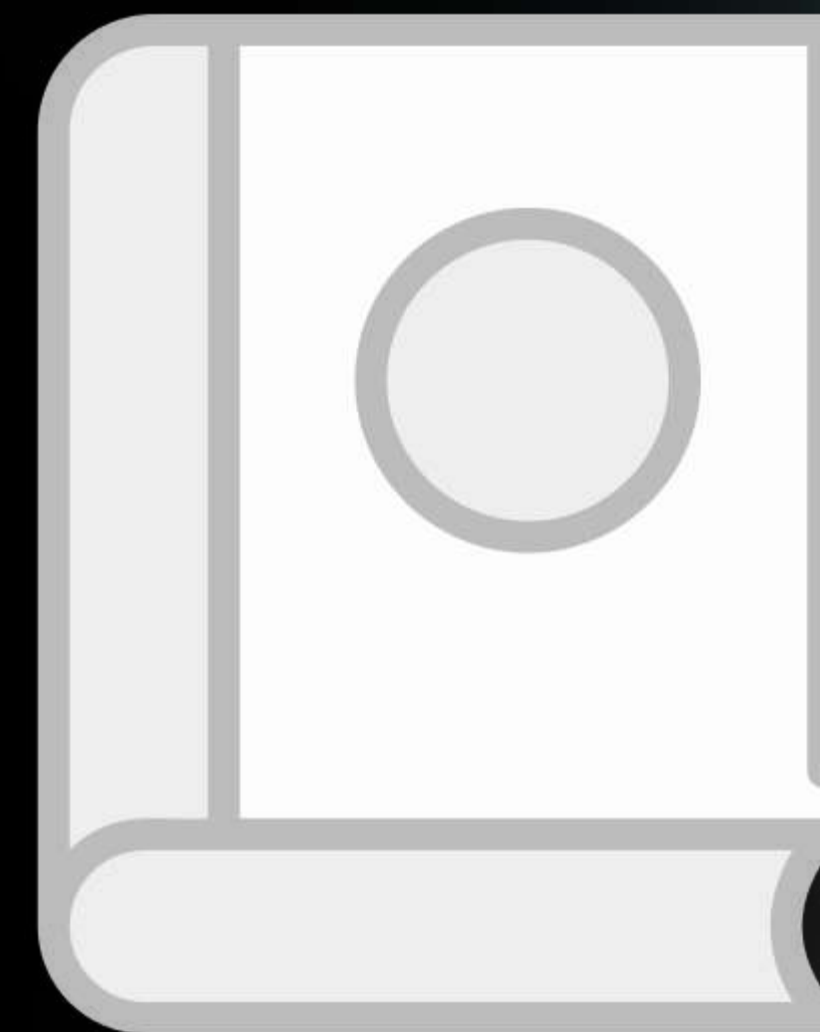
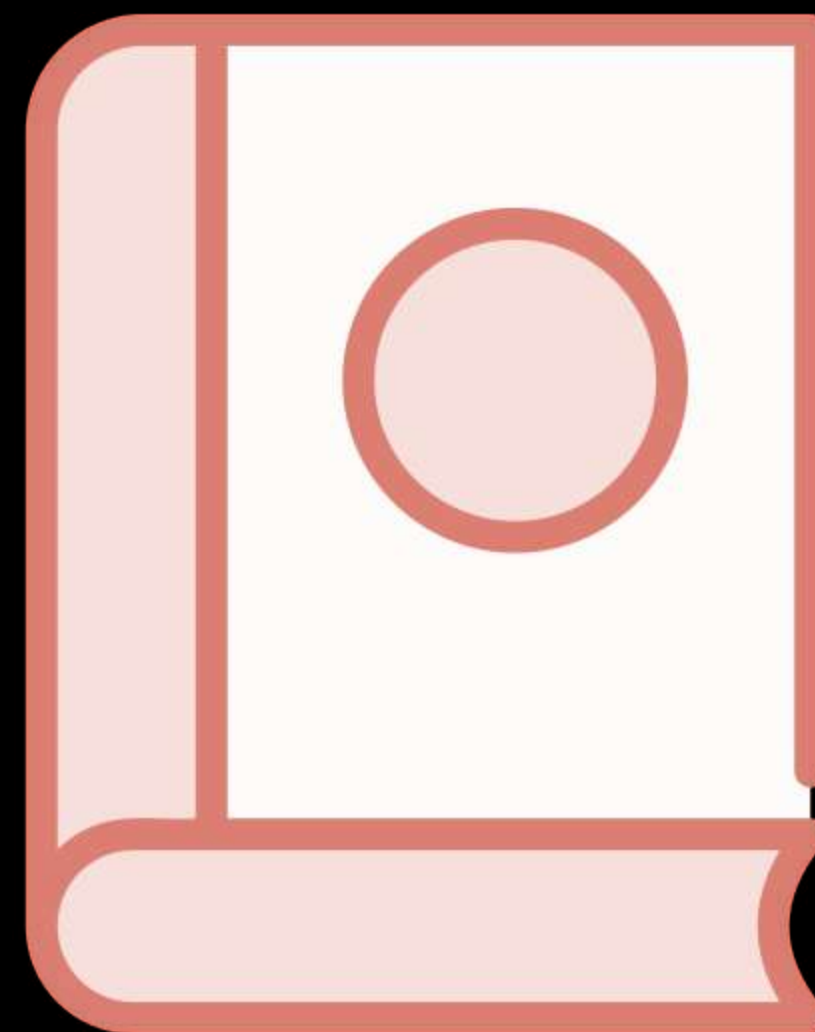
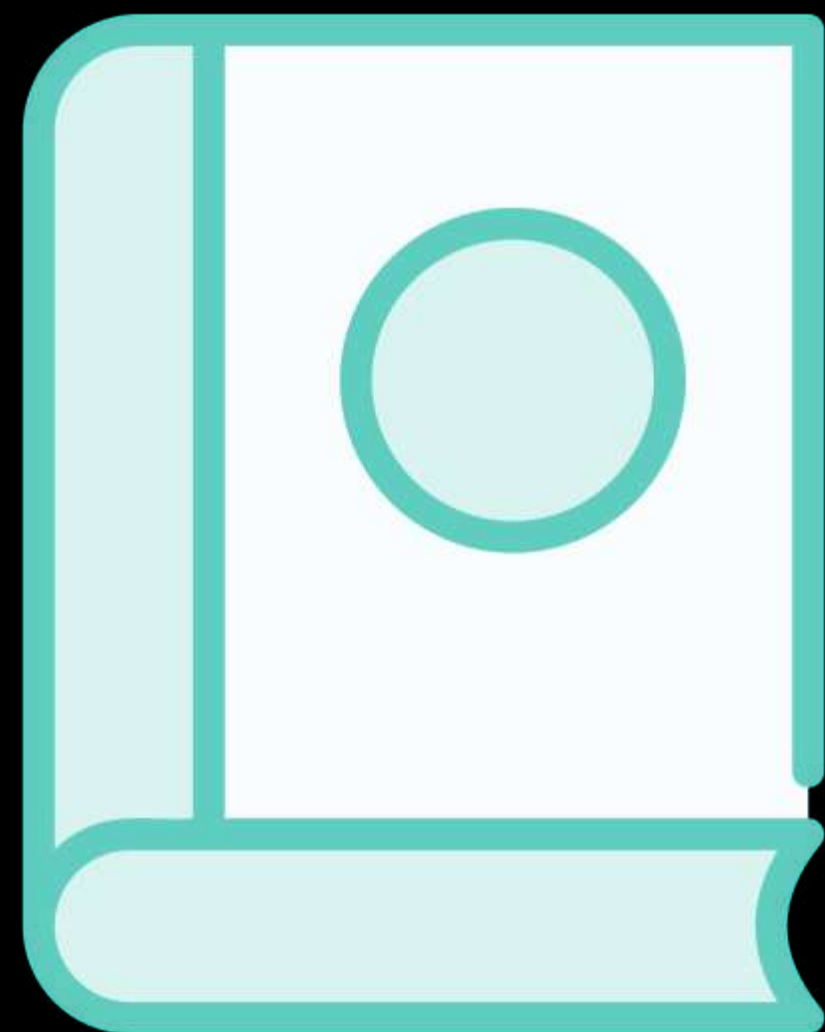
Domain Driven Design

Предметно-ориентированное проектирование



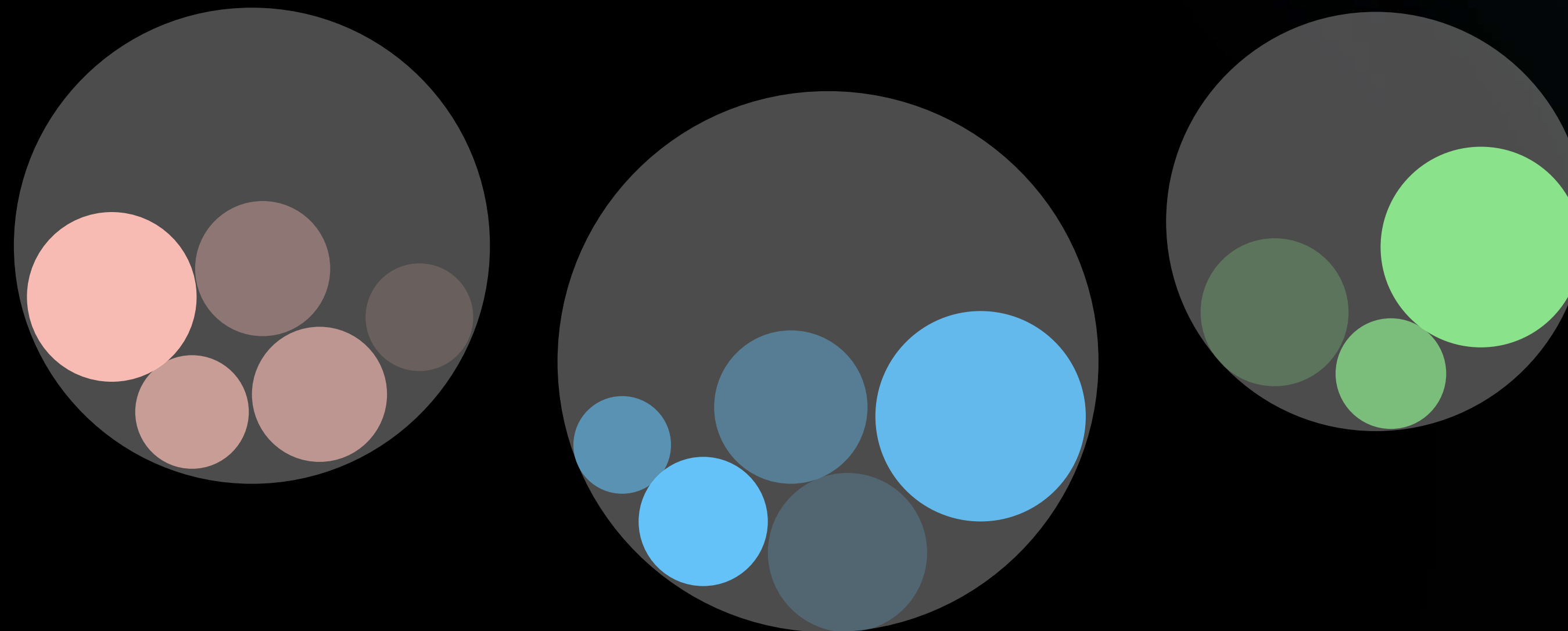
Domain Driven Design

Предметно-ориентированное проектирование



Domain Driven Design

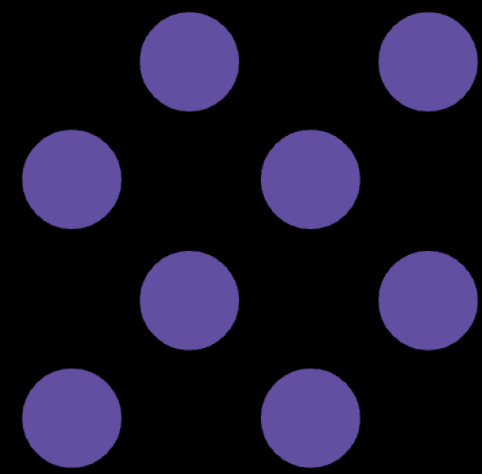
Предметно-ориентированное проектирование



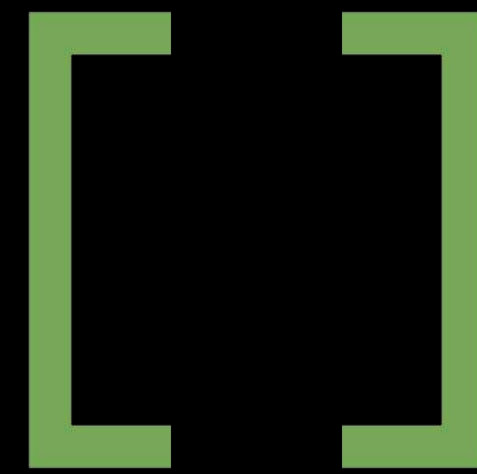
Более абстрактное воплощение **объектно-ориентированного** подхода,.

Domain Driven Design

Предметно-ориентированное проектирование



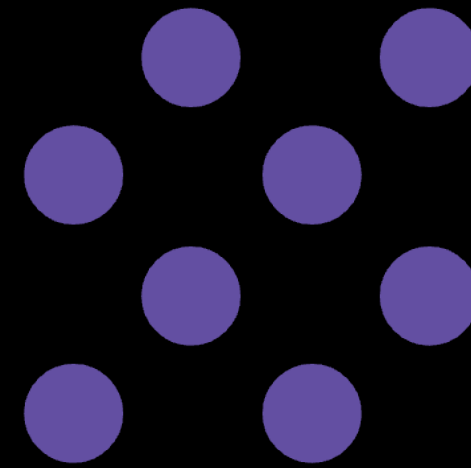
Предметная
область



Ограниченный
контекст



Единый
язык



Предметная область

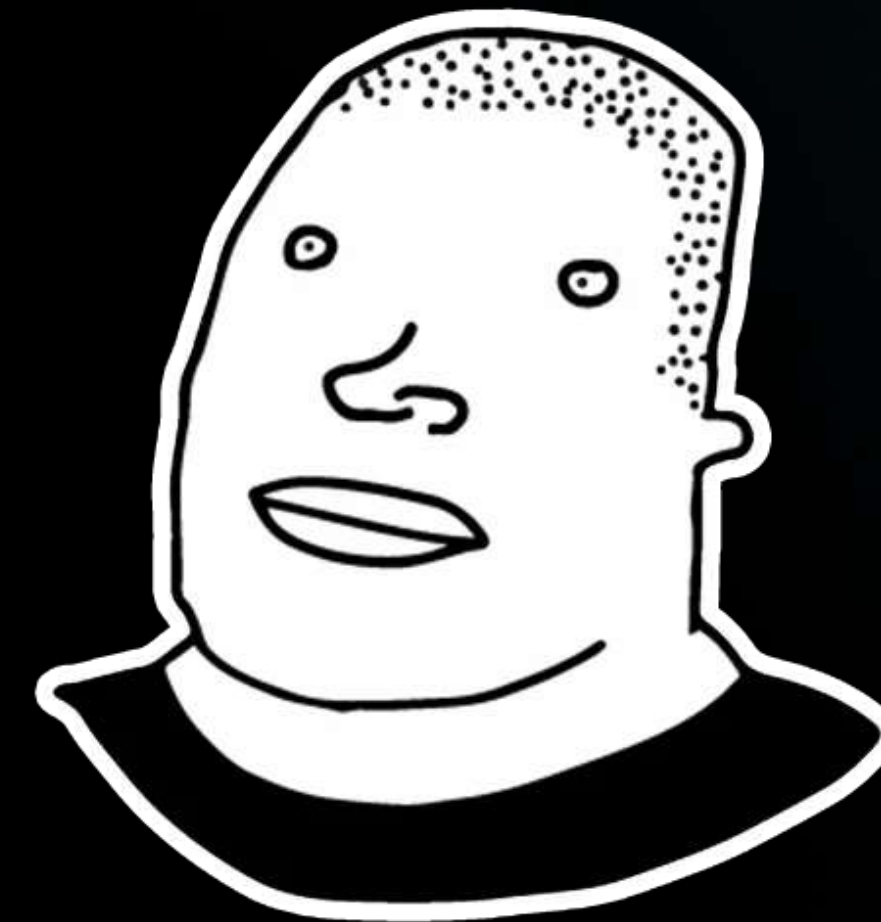
Совокупность тесно связанных понятий, описывающих свойства и поведение в пределах бизнеса-модели.

Предметная область:
что это такое, и откуда её взять?

Предметная область:
что это такое, и откуда её взять?



Бизнес



Команда разработки

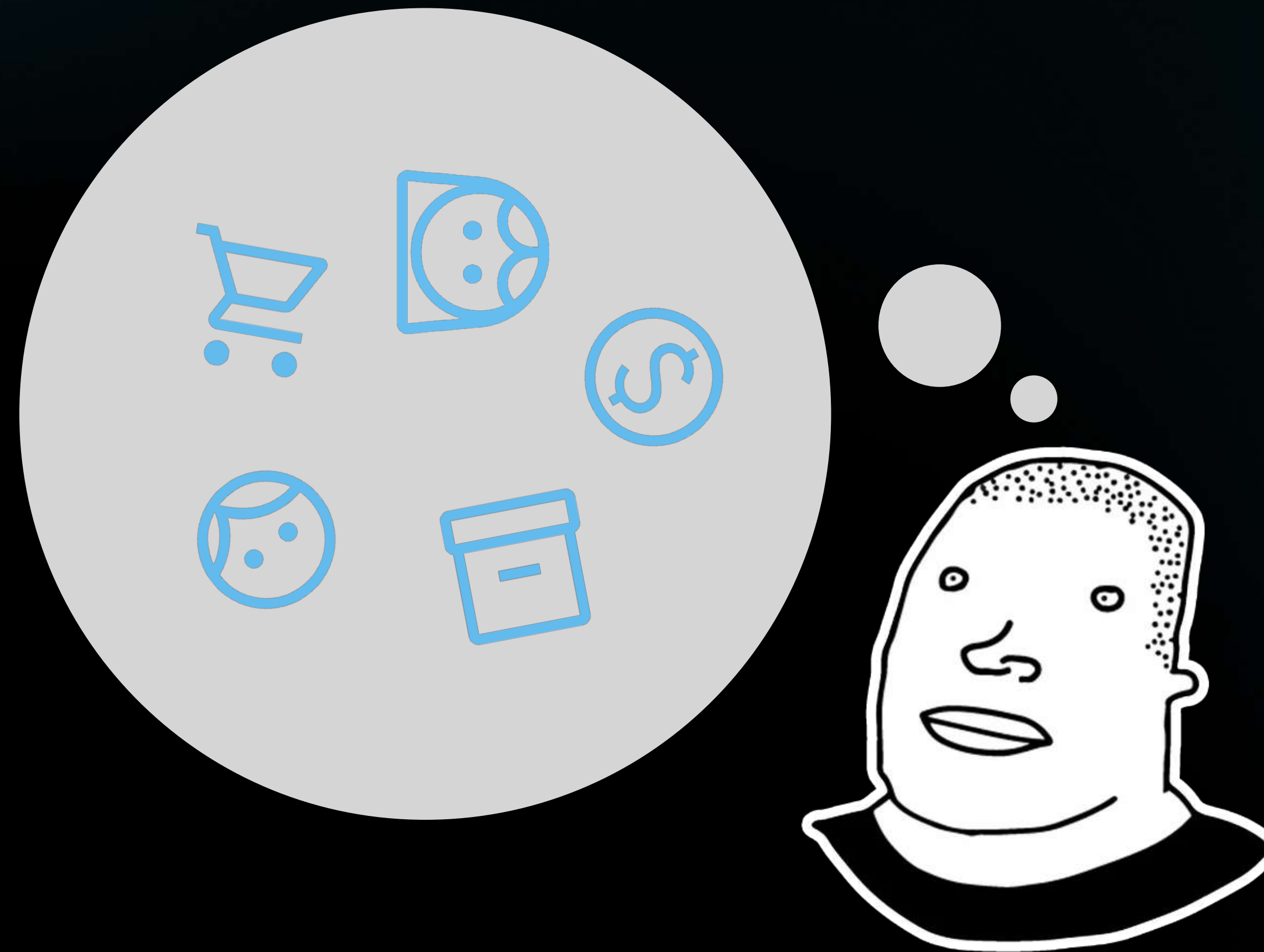
Предметная область:
что это такое, и откуда её взять?



Бизнес

Команда разработки

Предметная область:
что это такое, и откуда её взять?



Команда разработки

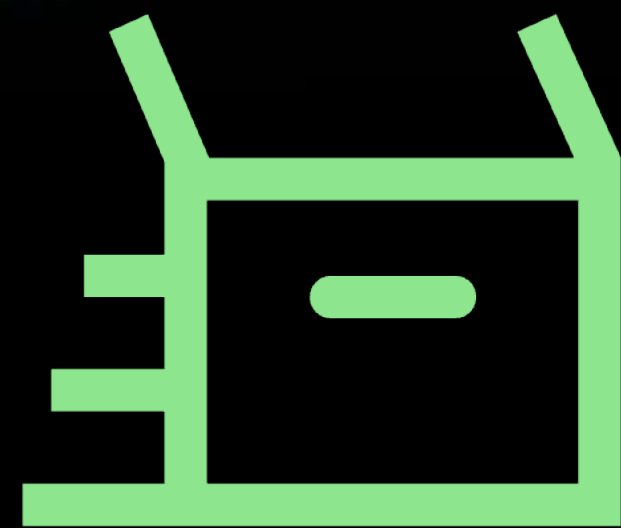
Маркетплейс



Маркетплейс



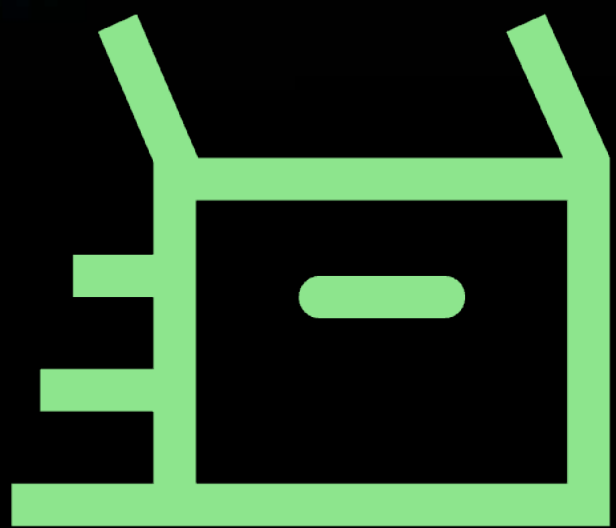
Служба доставки



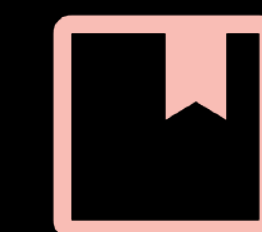
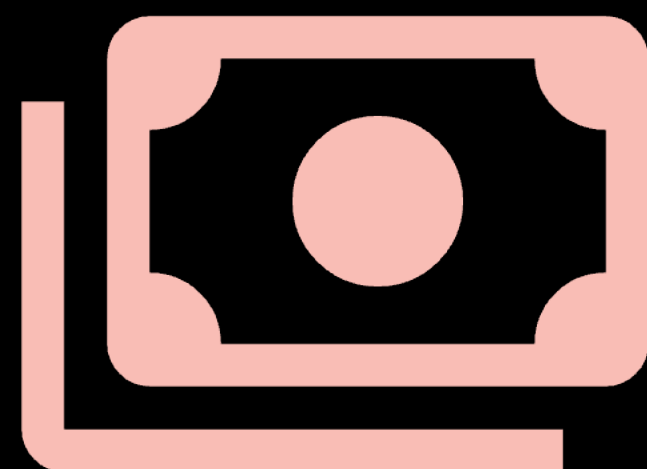
Маркетплейс



Служба доставки



Система оплаты





Интернет-магазин



Интернет-магазин



Ограниченный контекст

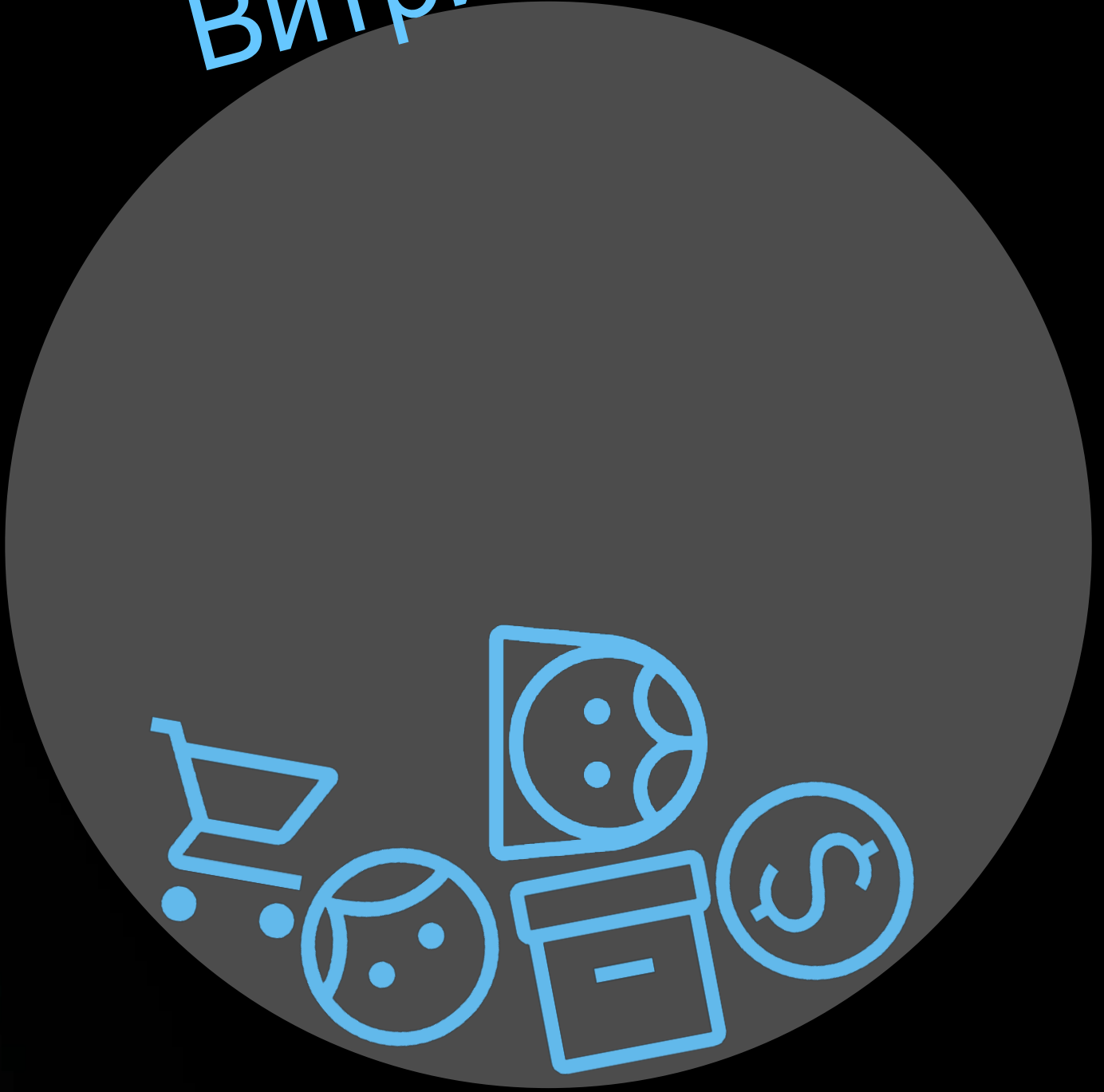
Понятийные границы бизнеса, в которые заключена предметная область.

Каждая из предметных областей ограничена своим **контекстом**, который жёстко инкапсулирует предметные области в своих границах.

Система оплаты



Витрина



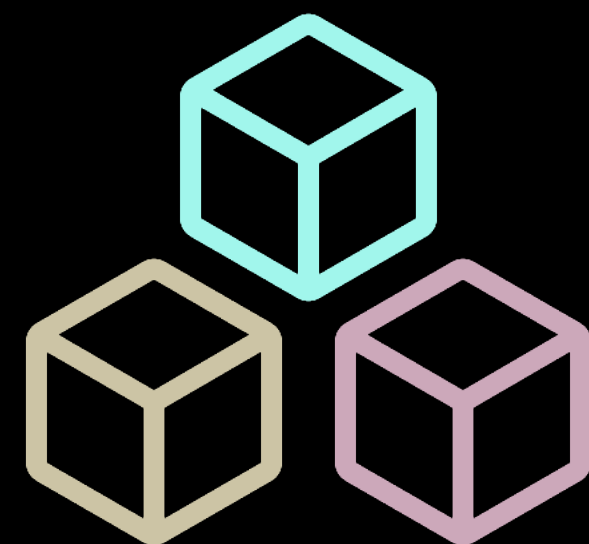
Доставка



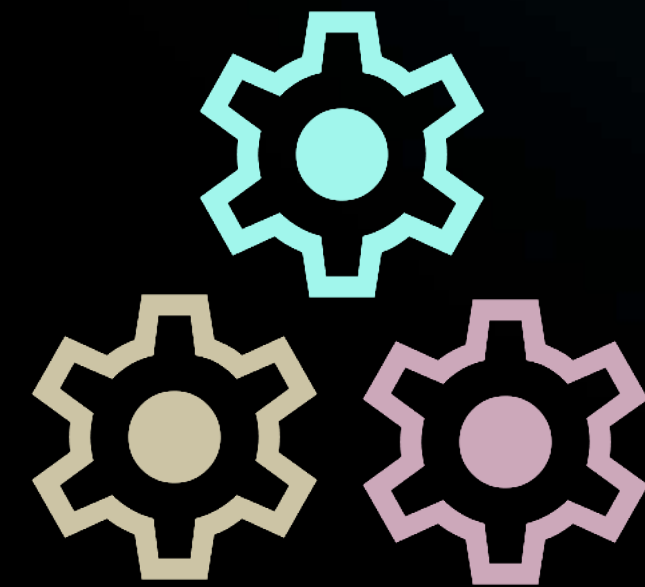
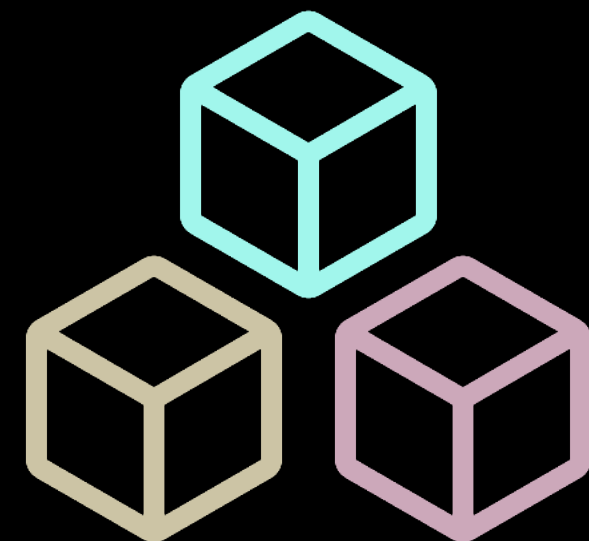
Разделение по пакетам,



Разделение по пакетам, модулям,

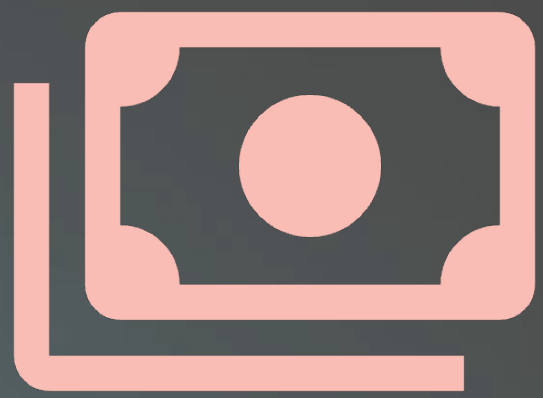


Разделение по пакетам, модулям, отдельным сервисам.



Назначение ограниченного контекста – свести к **нулю** количество связей между предметными областями.

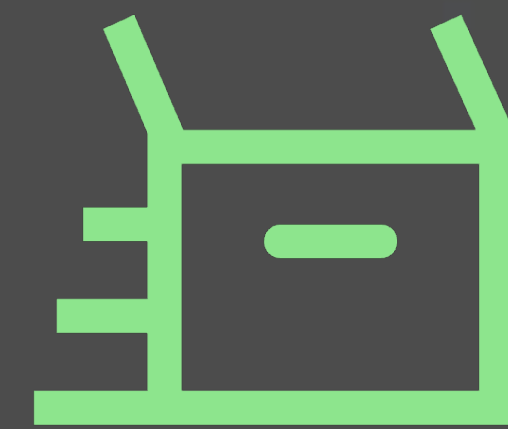
Система оплаты



Витрина



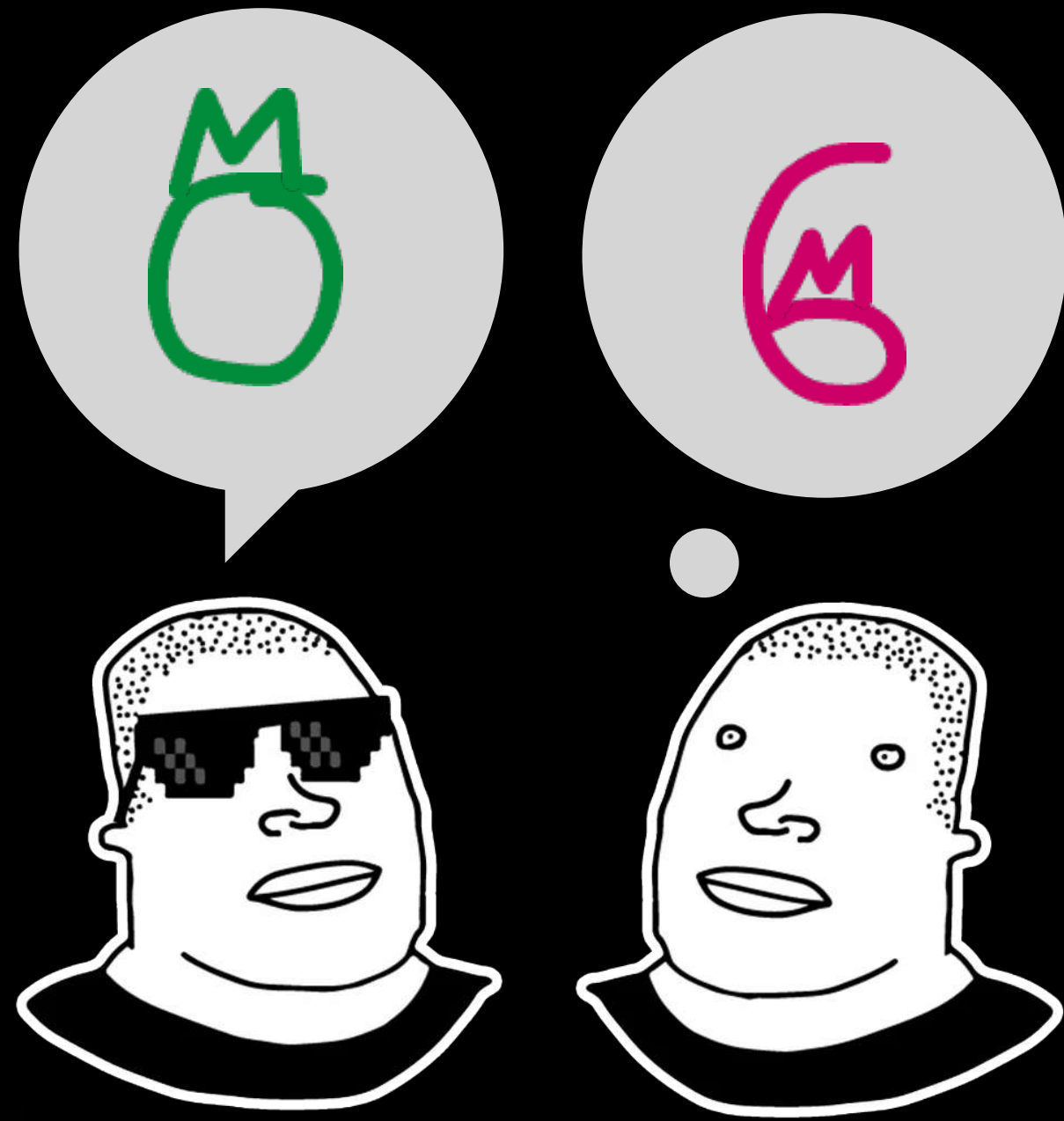
Доставка

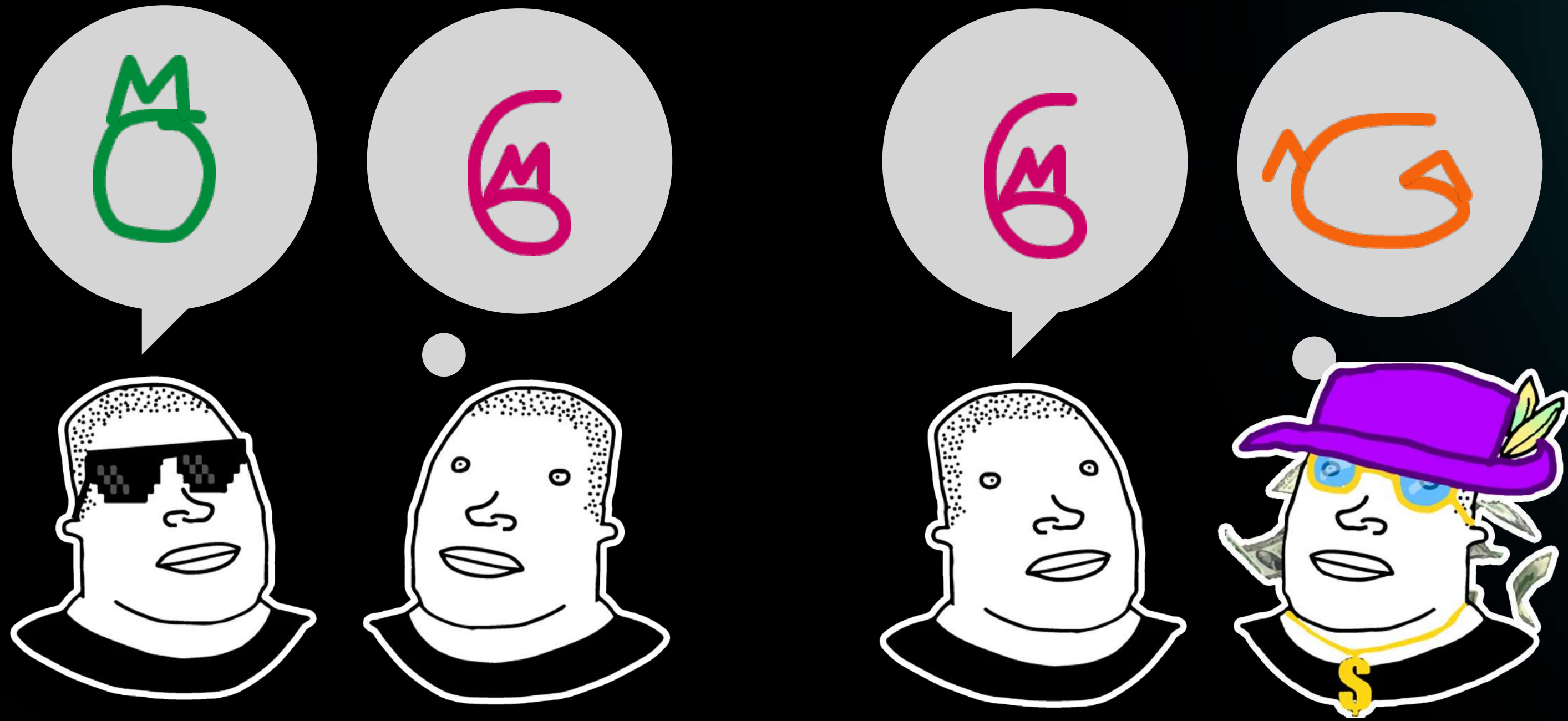


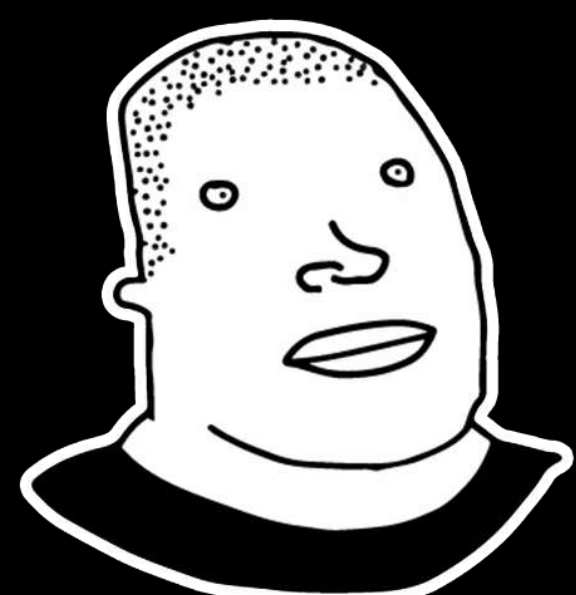
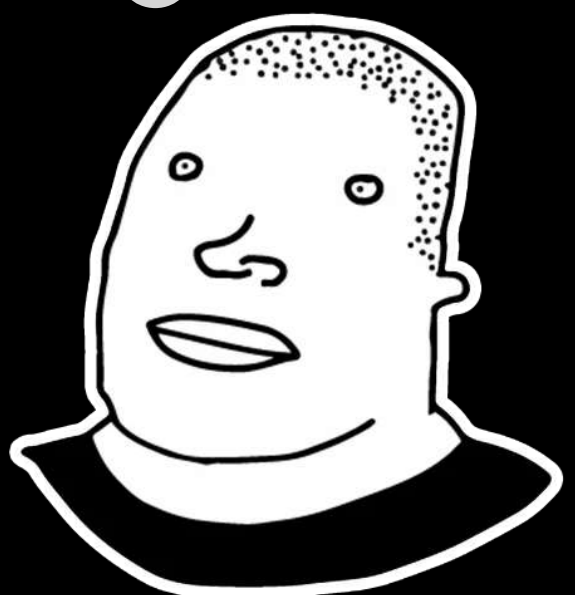
Мы определили предметные области и
ограничили их контекстом.

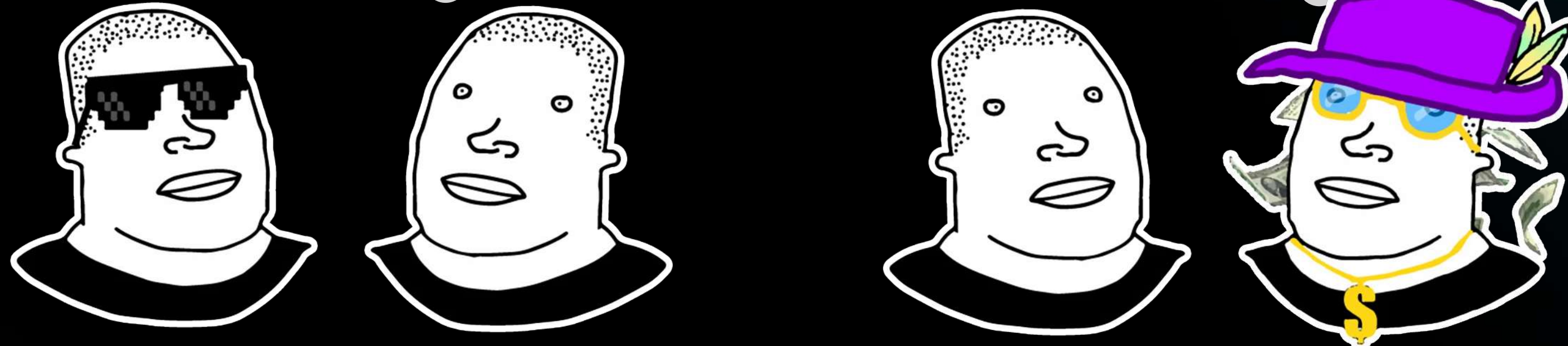
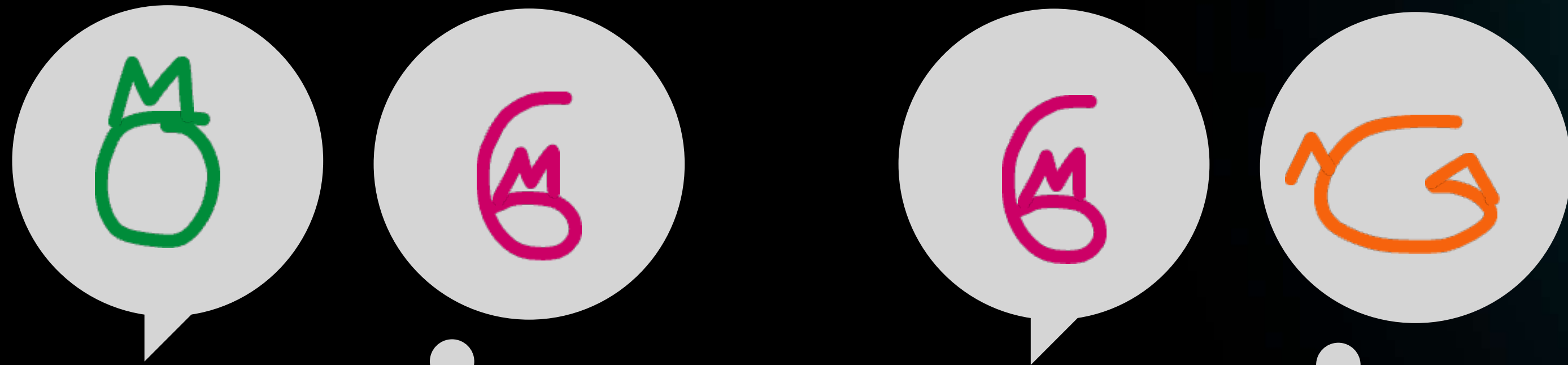
Мы определили предметные области и
ограничили их контекстом.

Но как добиться понимания **общих терминов,**
процессов и правил среди команды?







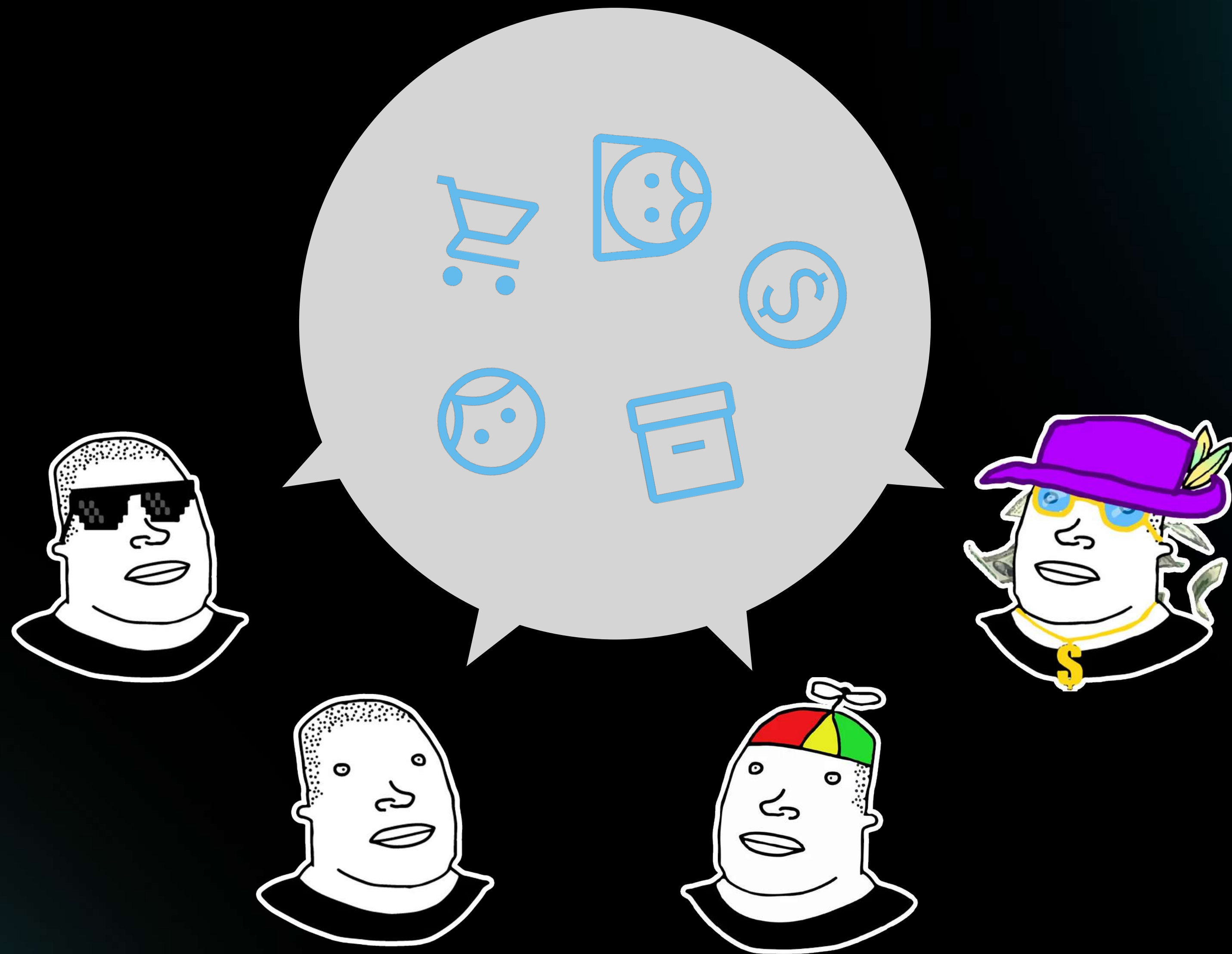


????!?



Единый язык

Общий язык терминов и понятий, описывающий предметную область.



Единый словарь ключевых терминов

Товар

Товар

Система оплаты

Маркетплейс

Доставка

Товар



Товар



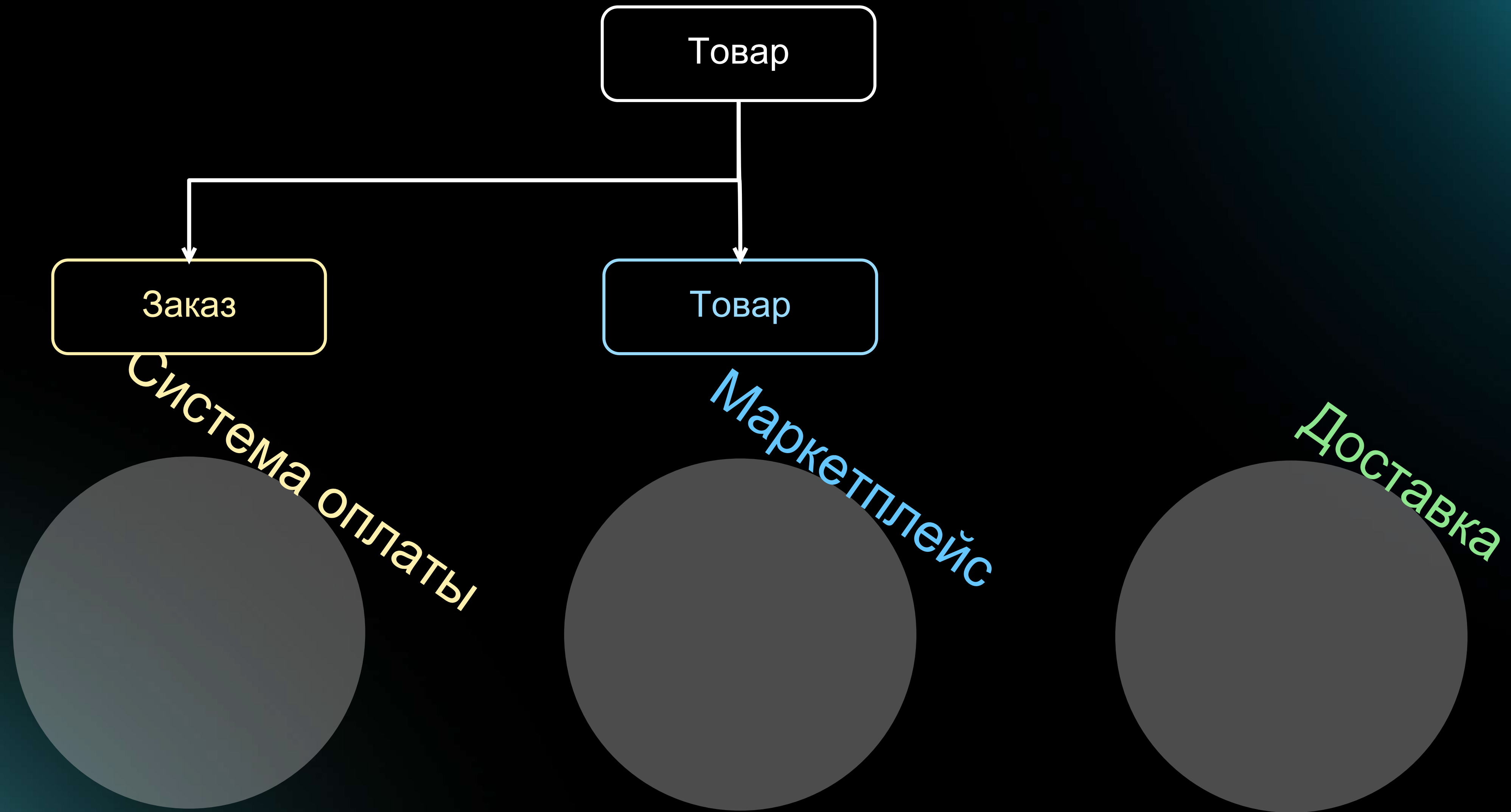
Система оплаты

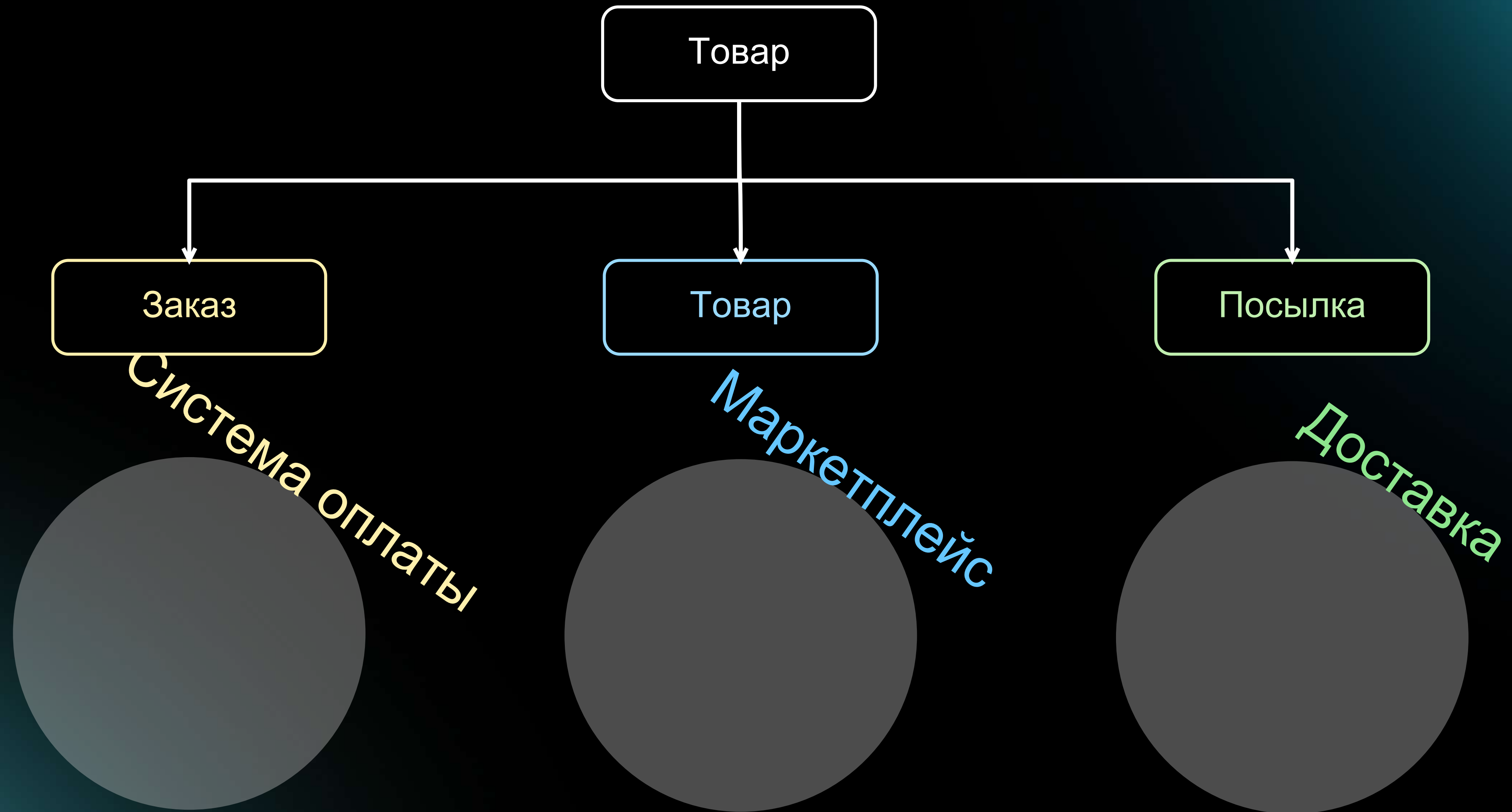


Маркетплейс



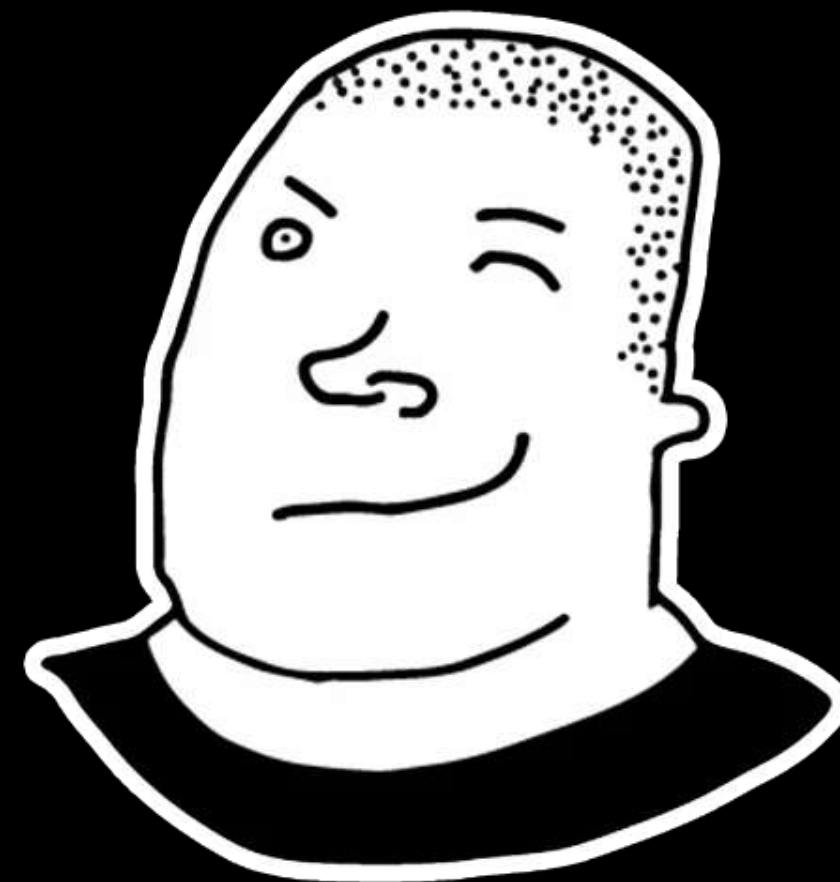
Доставка



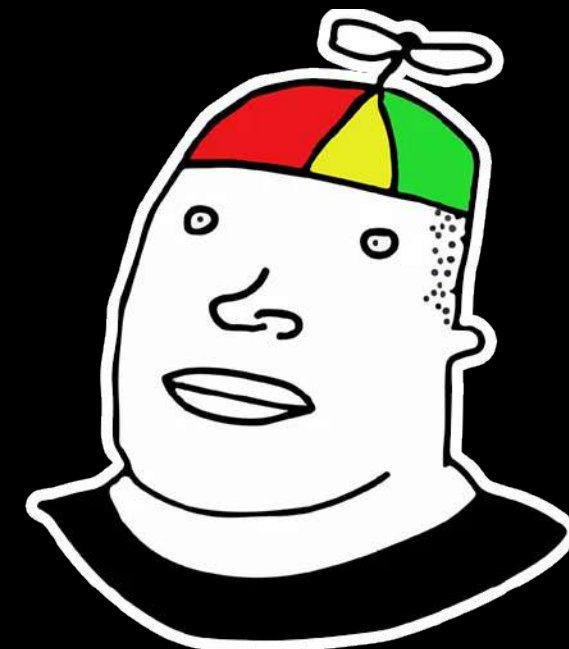
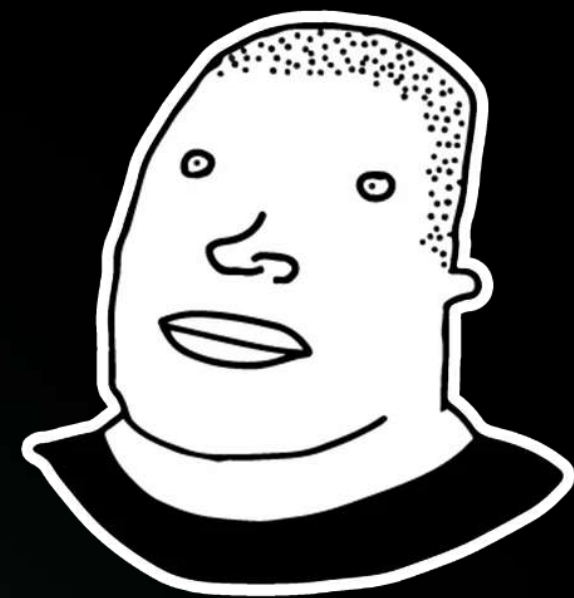
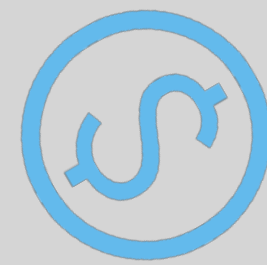
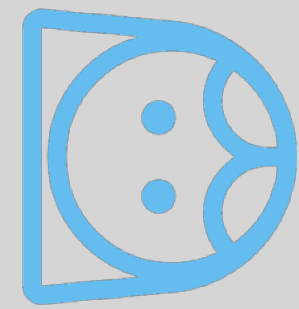


~~OrderManager().processOrder()~~

Order().cancel()



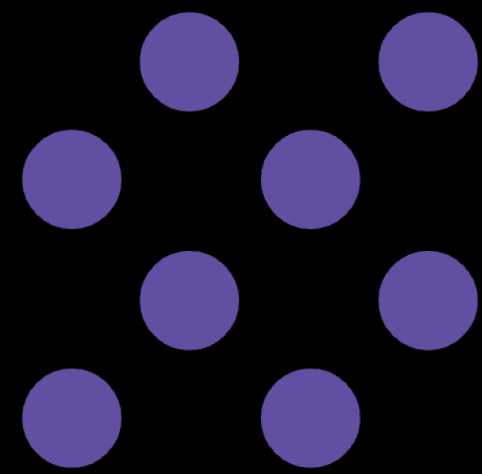
Буквальное отражение в коде



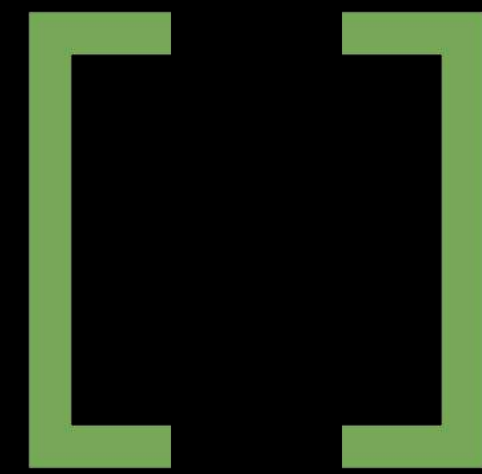
Благодаря **единому языку**, значительно упрощается обсуждение требований и разработка новых решений.

Domain Driven Design

Предметно-ориентированное проектирование



Предметная
область



Ограниченный
контекст



Единый
язык

Но как Domain-Driven Design помогает избежать плохой архитектуры?

Но как Domain-Driven Design помогает избежать плохой архитектуры?

Разберём иерархию мультисервисного приложения с точки зрения предметно-ориентированного проектирования.

Иерархия мультисервисного приложения

На самом верхнем уровне приложение имеет свои предметную область, ограниченный контекст и единый язык.



Иерархия мультисервисного приложения

На самом верхнем уровне приложение имеет свои предметную область, ограниченный контекст и единый язык.

Приложение имеет консолидированное API и пользовательский интерфейс.



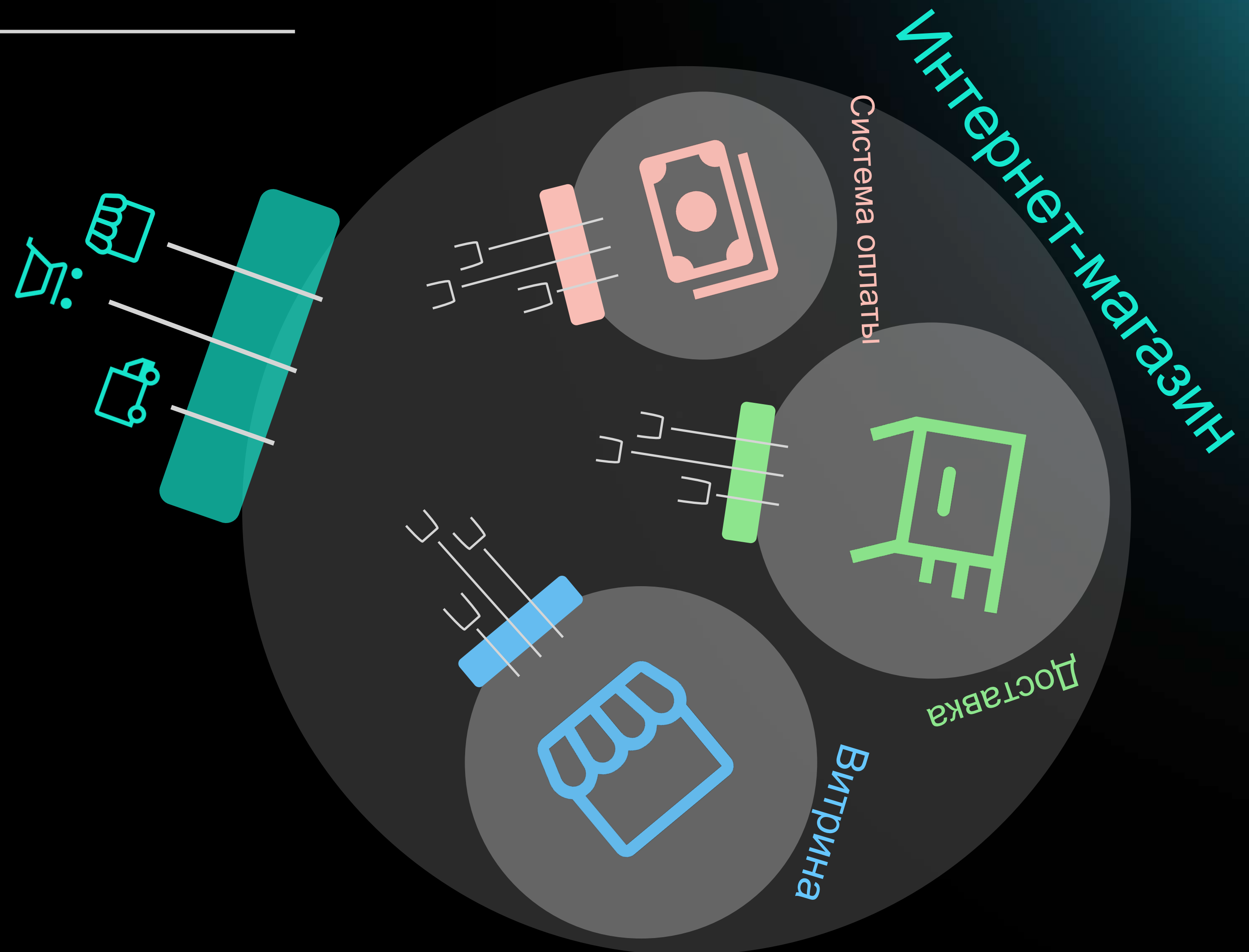
Иерархия мультисервисного приложения

На самом верхнем уровне приложение имеет свои предметную область, ограниченный контекст и единый язык.

Приложение имеет консолидированное API и пользовательский интерфейс.

Уровнем ниже приложение дробится на более мелкие компоненты, каждый из которых повторяет приложение в миниатюре — имеет свои предметную область, ограниченный контекст и единый язык.

И своё API.



Иерархия мультисервисного приложения

На самом верхнем уровне приложение имеет свои предметную область, ограниченный контекст и единый язык.

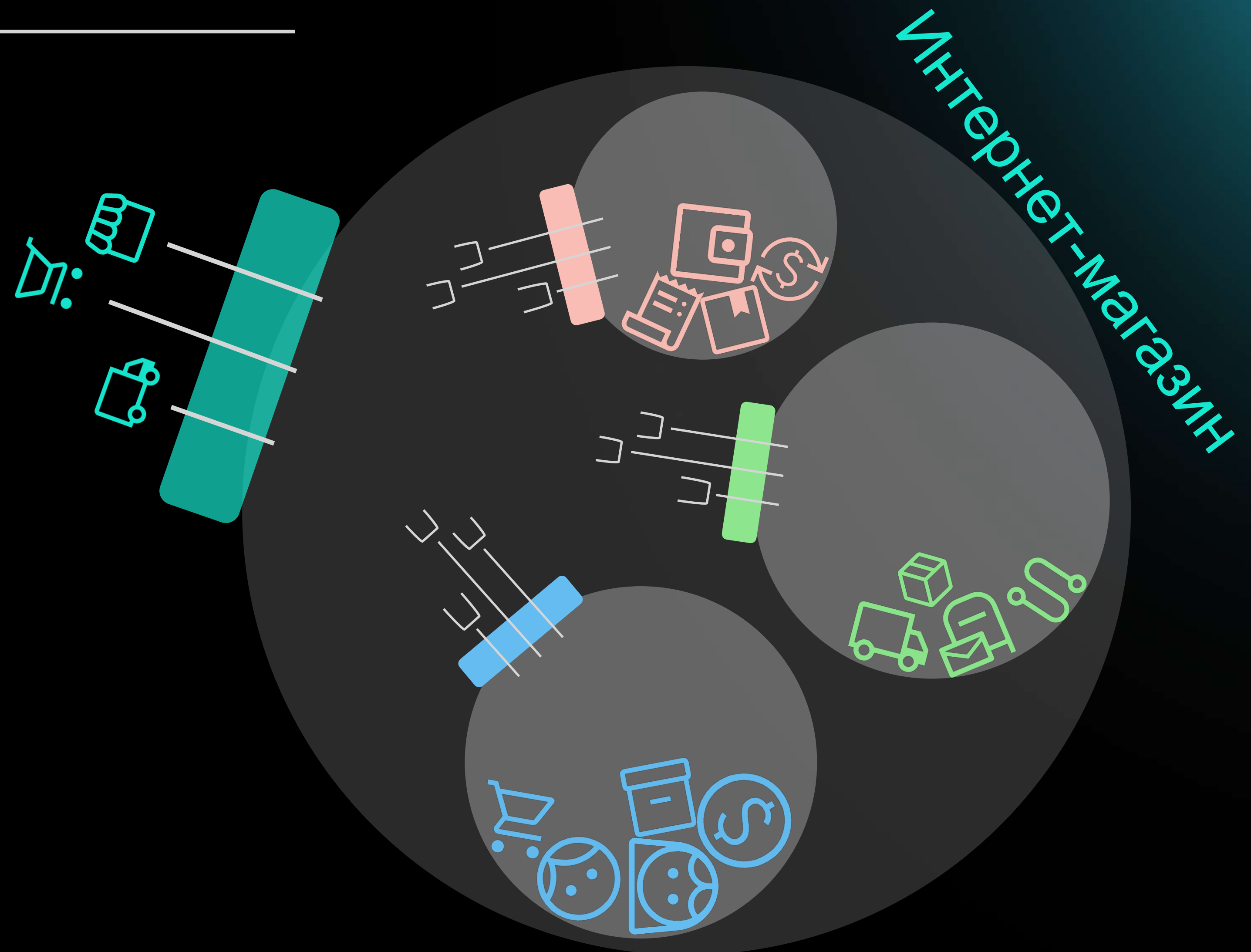
Приложение имеет консолидированное API и пользовательский интерфейс.

Уровнем ниже приложение дробится на более мелкие компоненты, каждый из которых повторяет приложение в миниатюре — имеет свои предметную область, ограниченный контекст и единый язык.

И своё API.

Ещё уровнем ниже более мелкие компоненты имеют всё то же самое.

Предметную область, ограниченный контекст и единый язык.



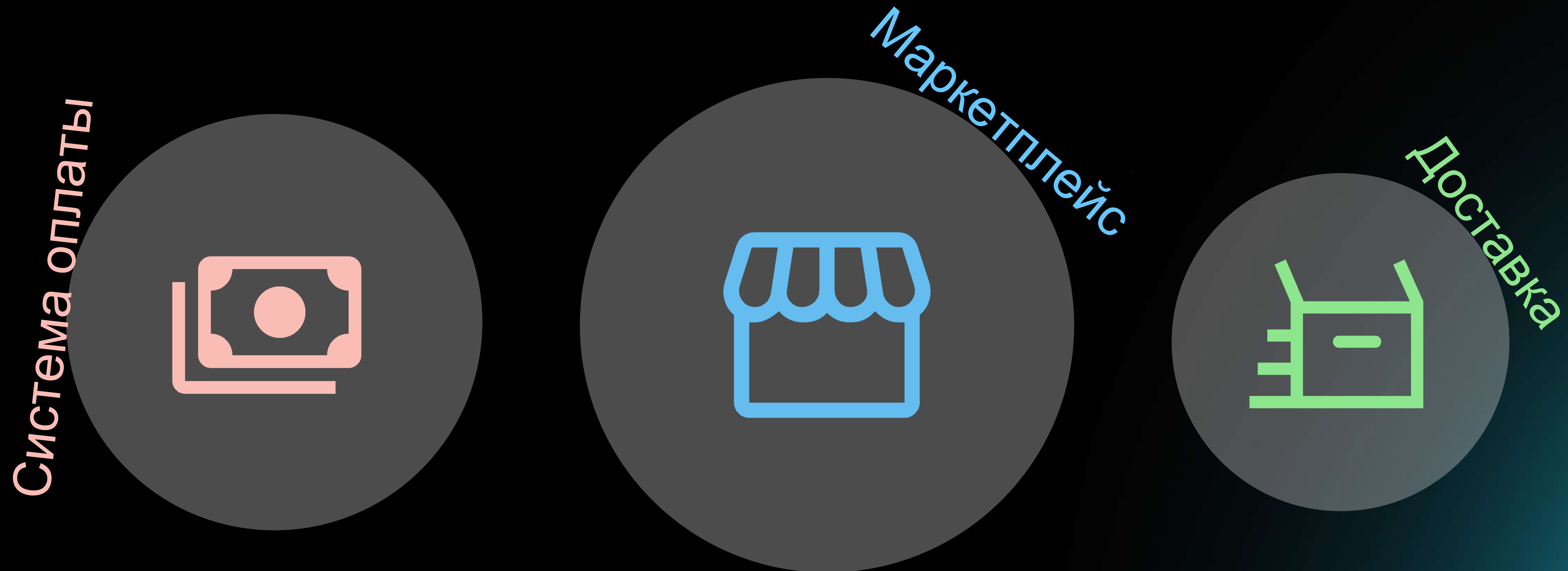
D.D.D...

D.D.D. everywhere!

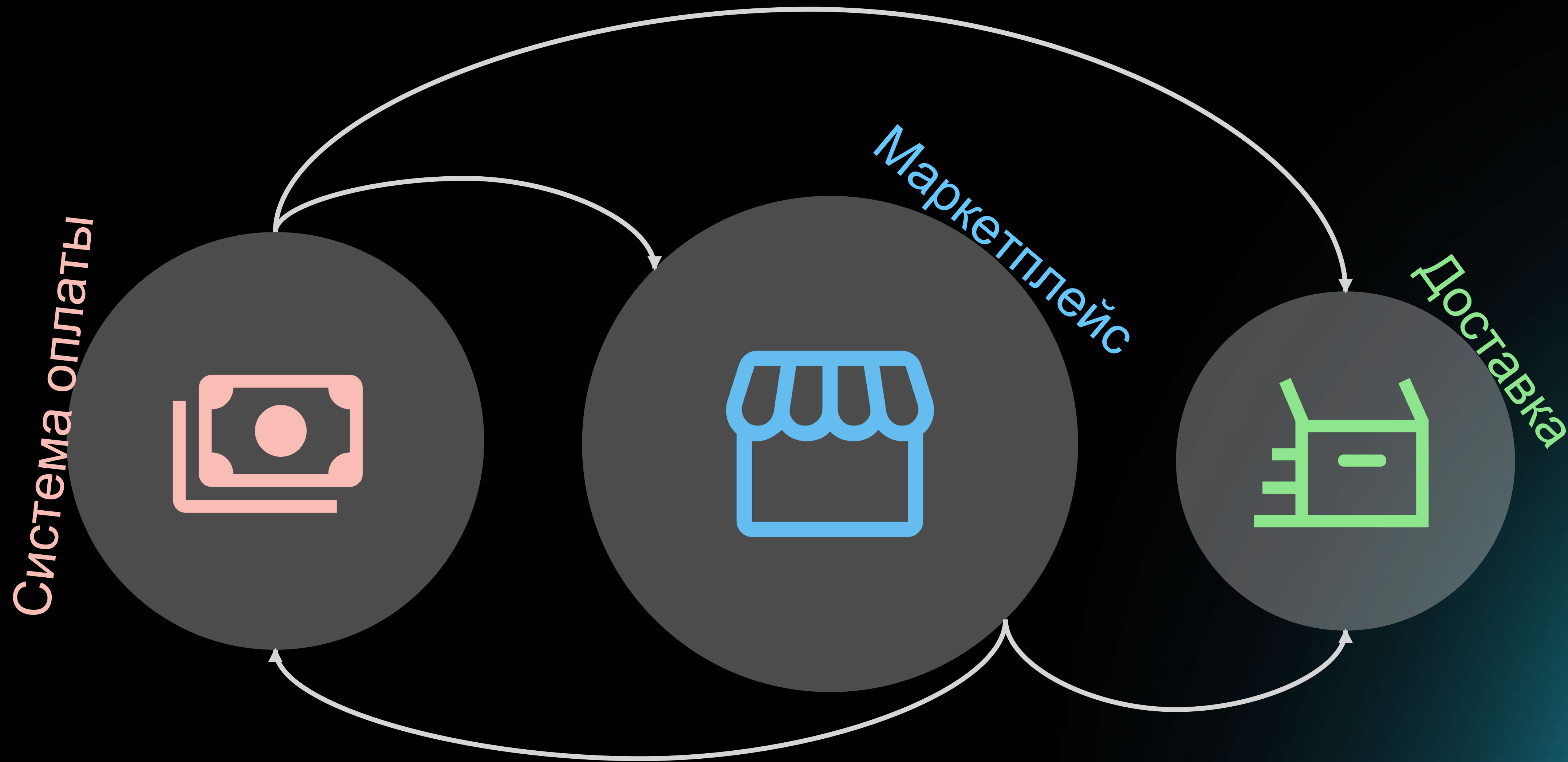


У нас есть несколько **микросервисов**.

Давайте спроектируем их архитектуру так, чтобы она нас устроила с точки зрения **предметно-ориентированного проектирования**.

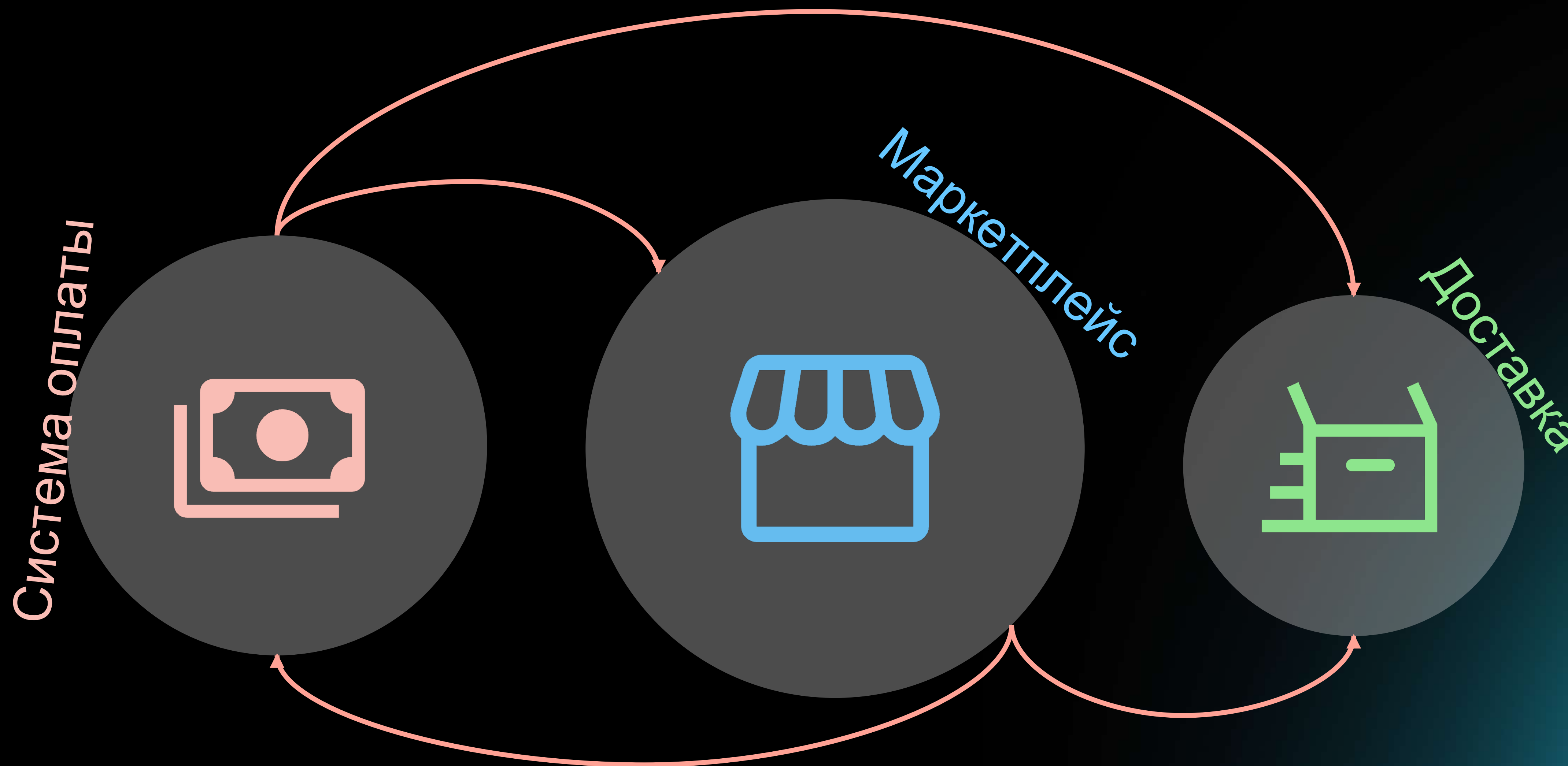


Проблема 1.
Перекрёстные связи.

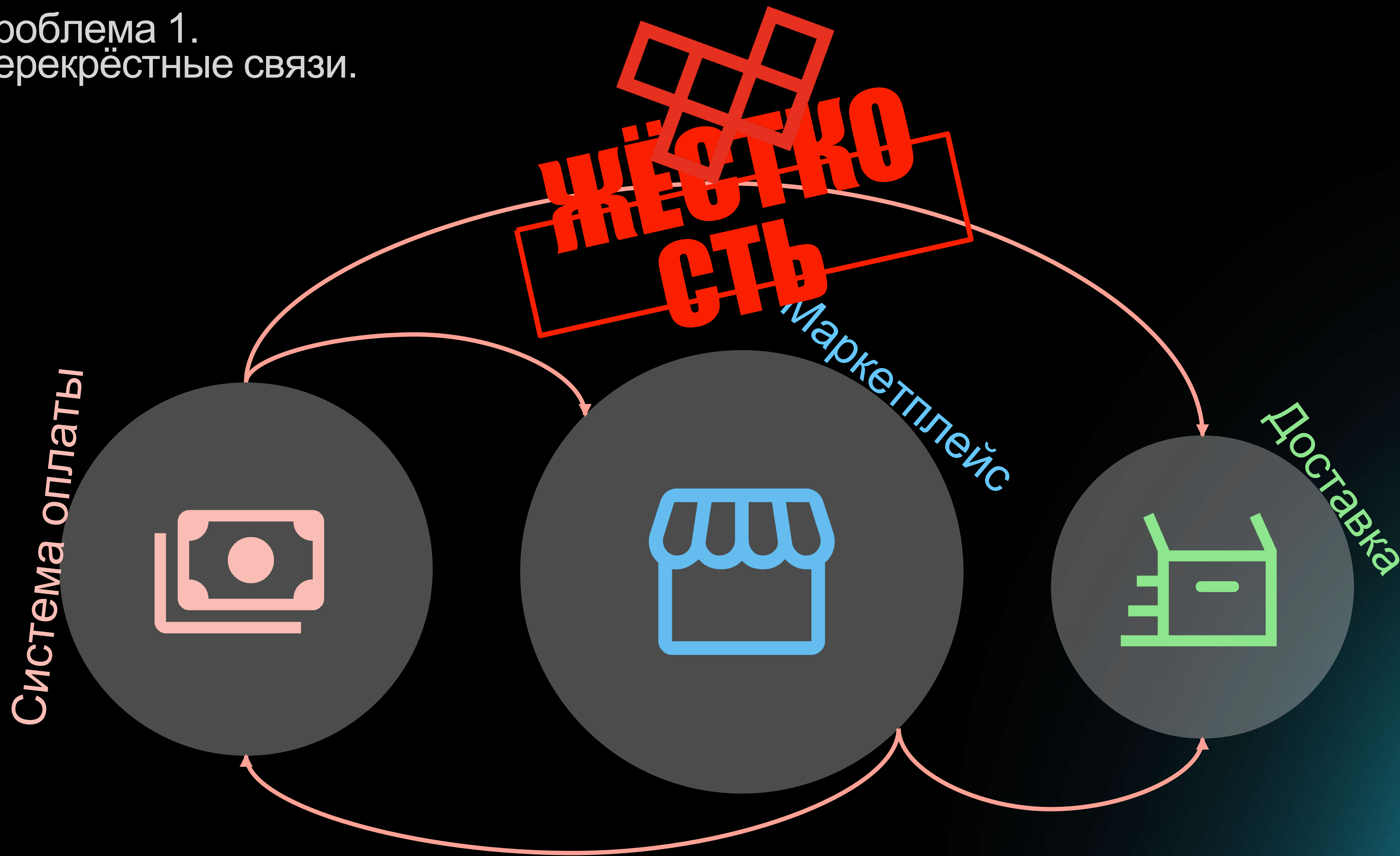


Проблема 1.
Перекрёстные связи.

Компоненты становятся **ЛОГИЧЕСКИ ЗАВИСИМЫМИ** друг от друга.



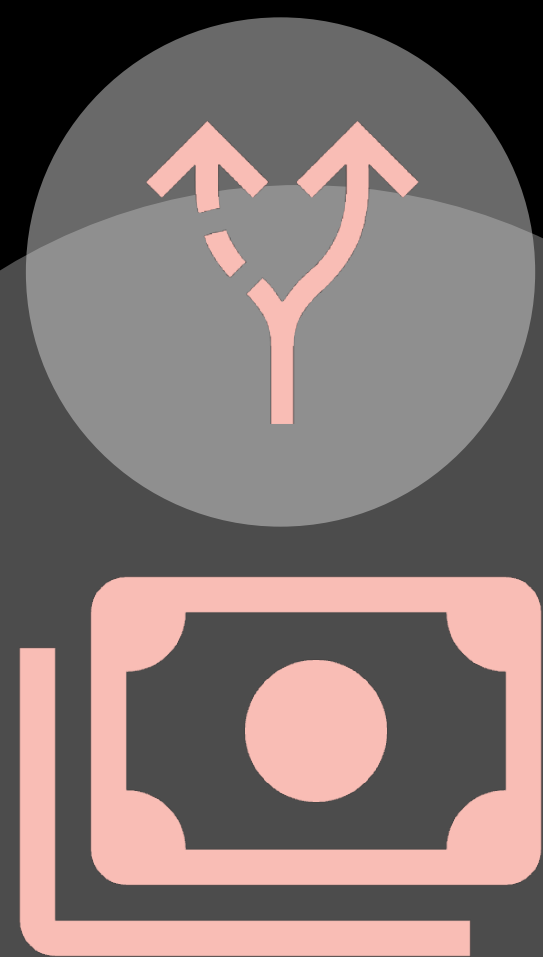
Проблема 1.
Перекрёстные связи.



Проблема 2. Сценарии.

Где их **хранить**?

Система оплаты



Маркетплейс



Доставка



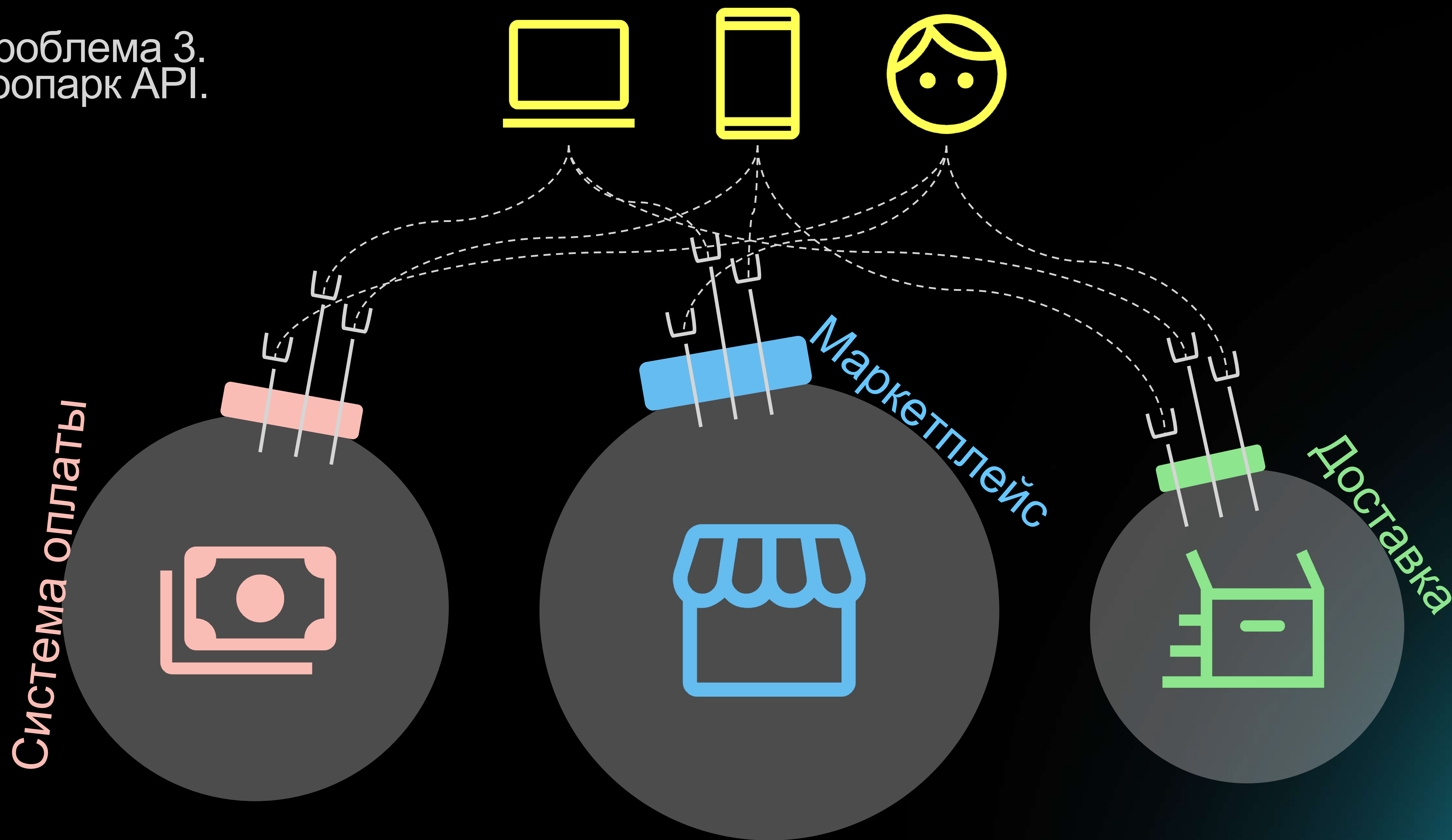
Проблема 2. Сценарии.



Проблема 2.
Сценарии.

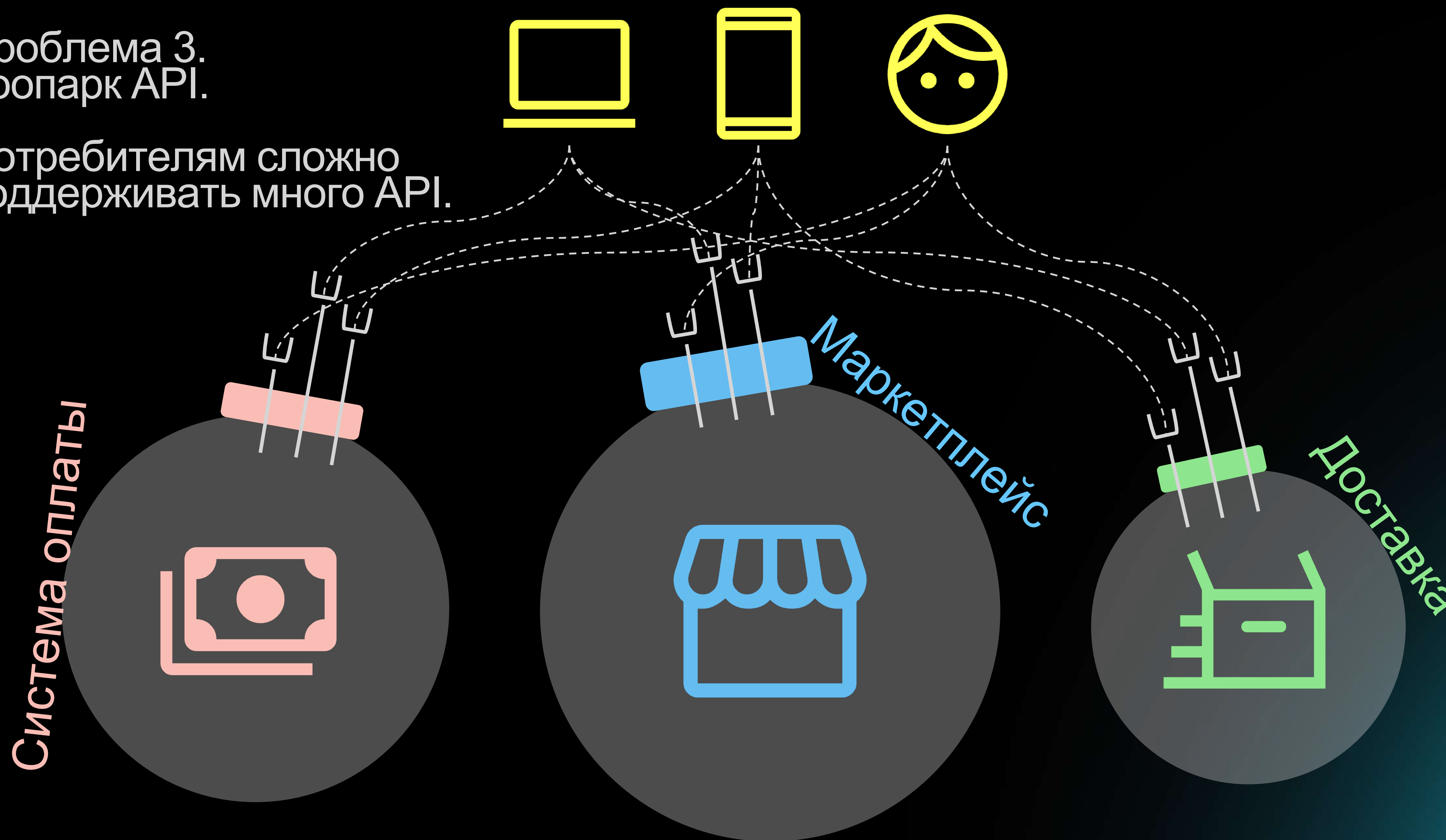


Проблема 3.
Зоопарк API.



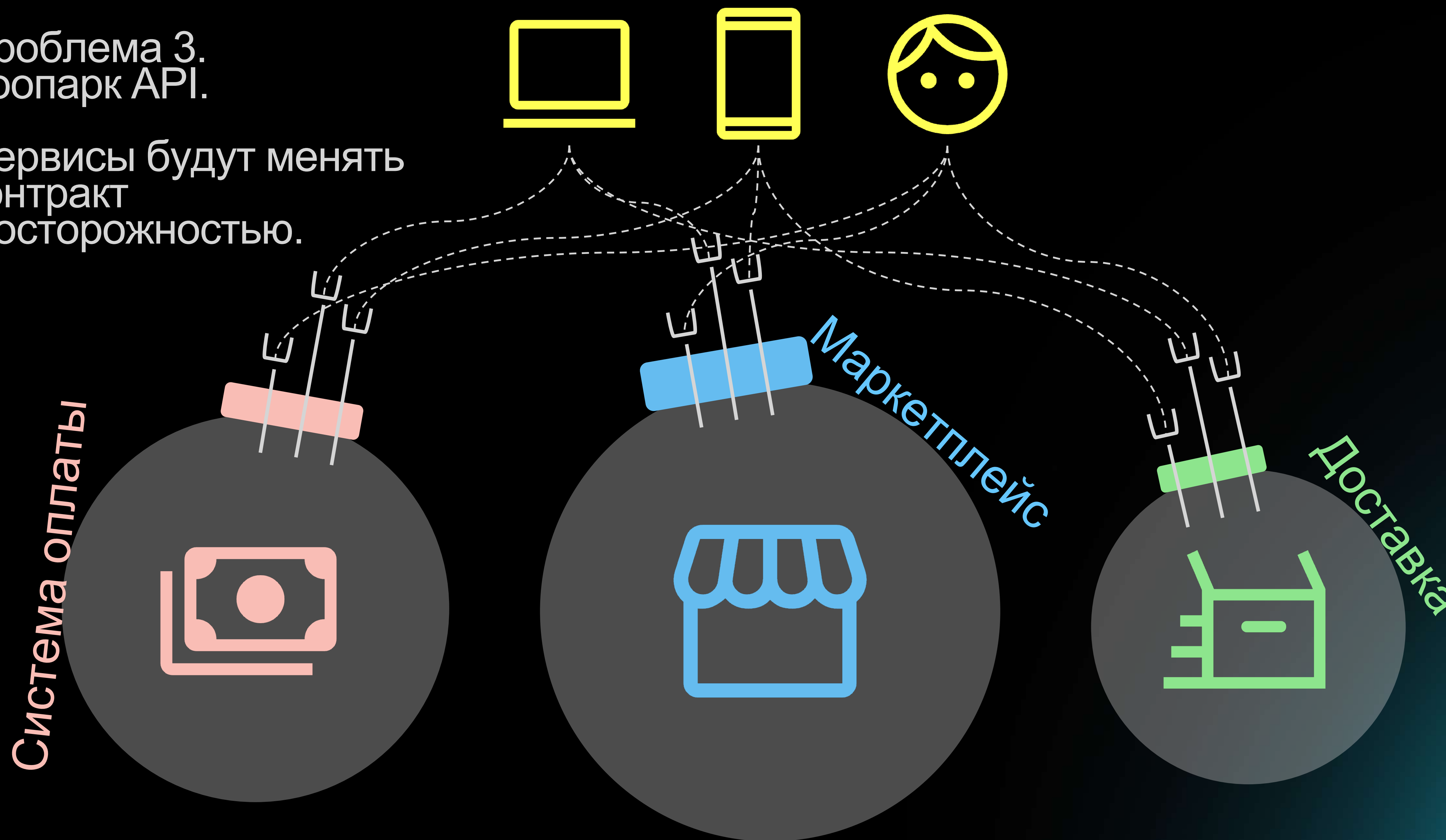
Проблема 3. Зоопарк API.

Потребителям сложно
поддерживать много API.



Проблема 3. Зоопарк API.

Сервисы будут менять
контракт
с осторожностью.



Проблема 3.
Зоопарк API.



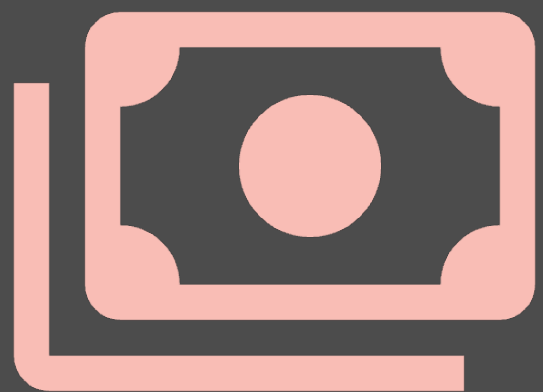
Мы получили весь букет **плохой архитектуры**.

Мы получили весь букет **плохой архитектуры**.

Какие **решения**?



Система оплаты



Маркетплейс

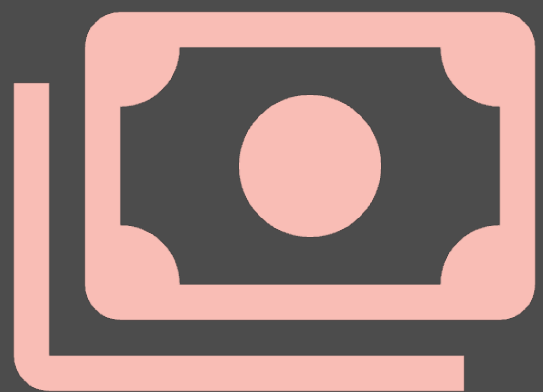


Доставка





Система оплаты



Маркетплейс

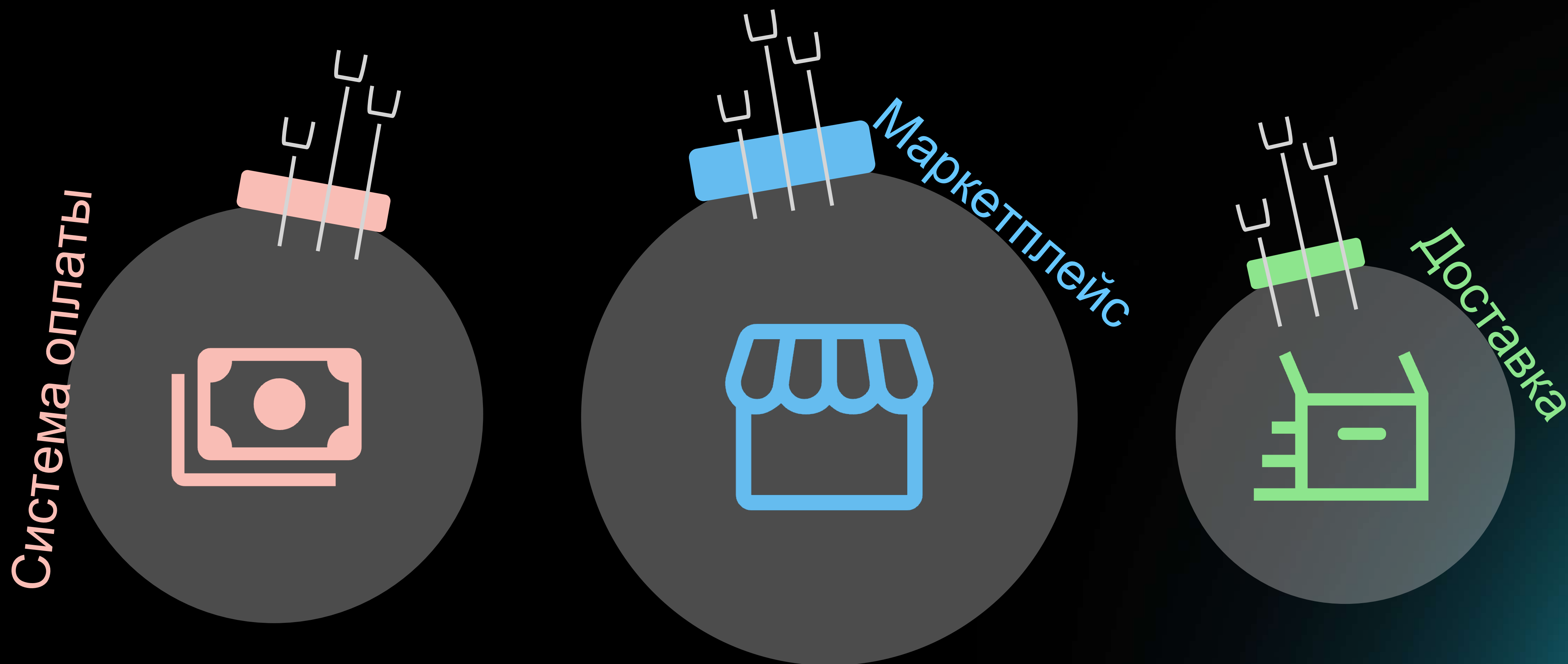


Доставка

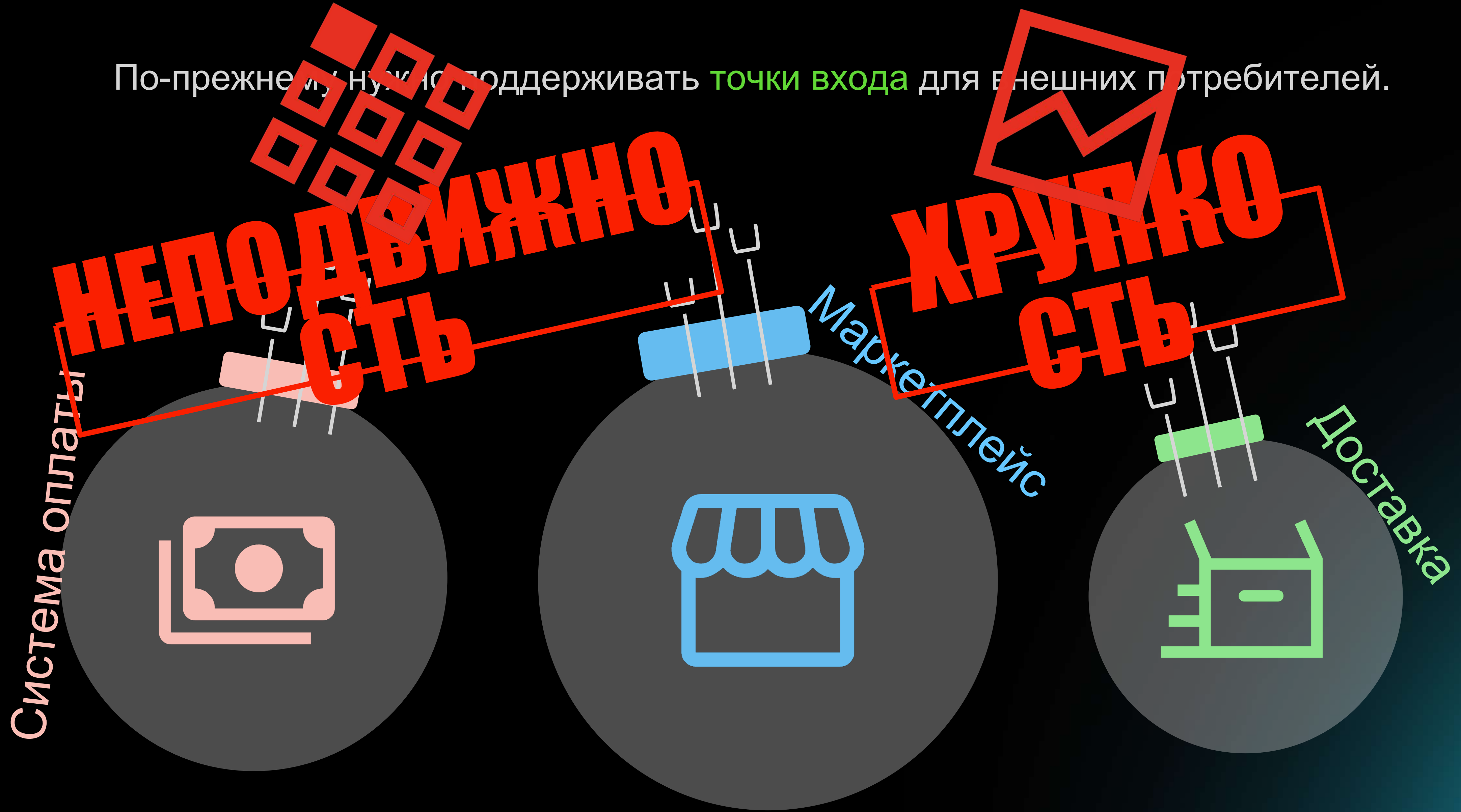
По-прежнему нет места для **общих сценариев**.



По-прежнему нужно поддерживать **точки входа** для внешних потребителей.



По-прежнему нужно поддерживать **точки входа** для внешних потребителей.



На текущем уровне абстракции задачу решить
НЕВОЗМОЖНО.

На текущем уровне абстракции задачу решить
НЕВОЗМОЖНО.

Значит, нужно поднимать уровень абстракции.



Подходящие архитектурные паттерны

Подходящие архитектурные паттерны



Фасад

Подходящие архитектурные паттерны



Фасад



Оркестратор

Подходящие архитектурные паттерны



Фасад



Оркестратор



Посредник

Подходящие архитектурные паттерны



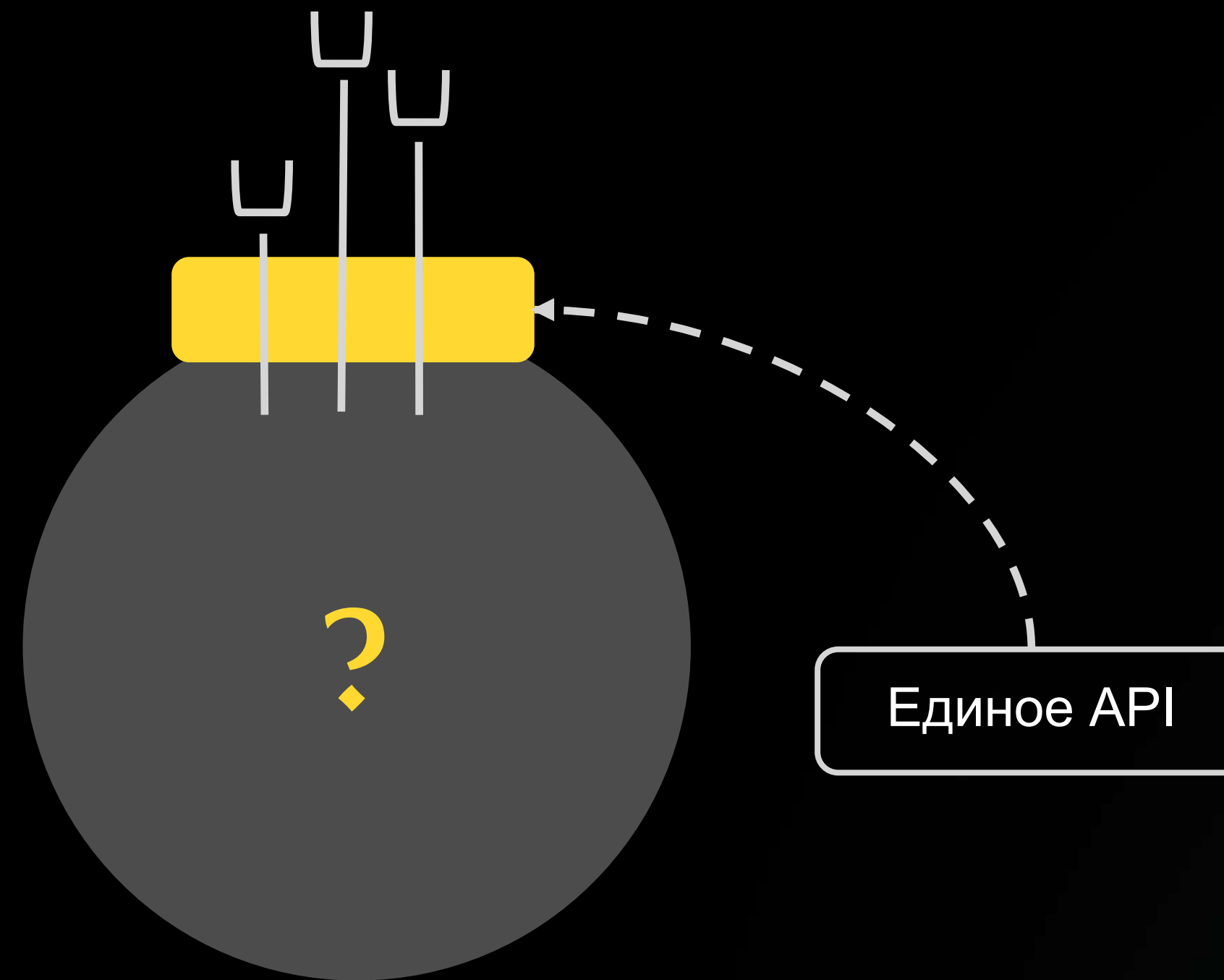
Фасад

Организация единого API

Фасад

Скрывает множество API разных микросервисов, предоставляя единое API.

Фасад

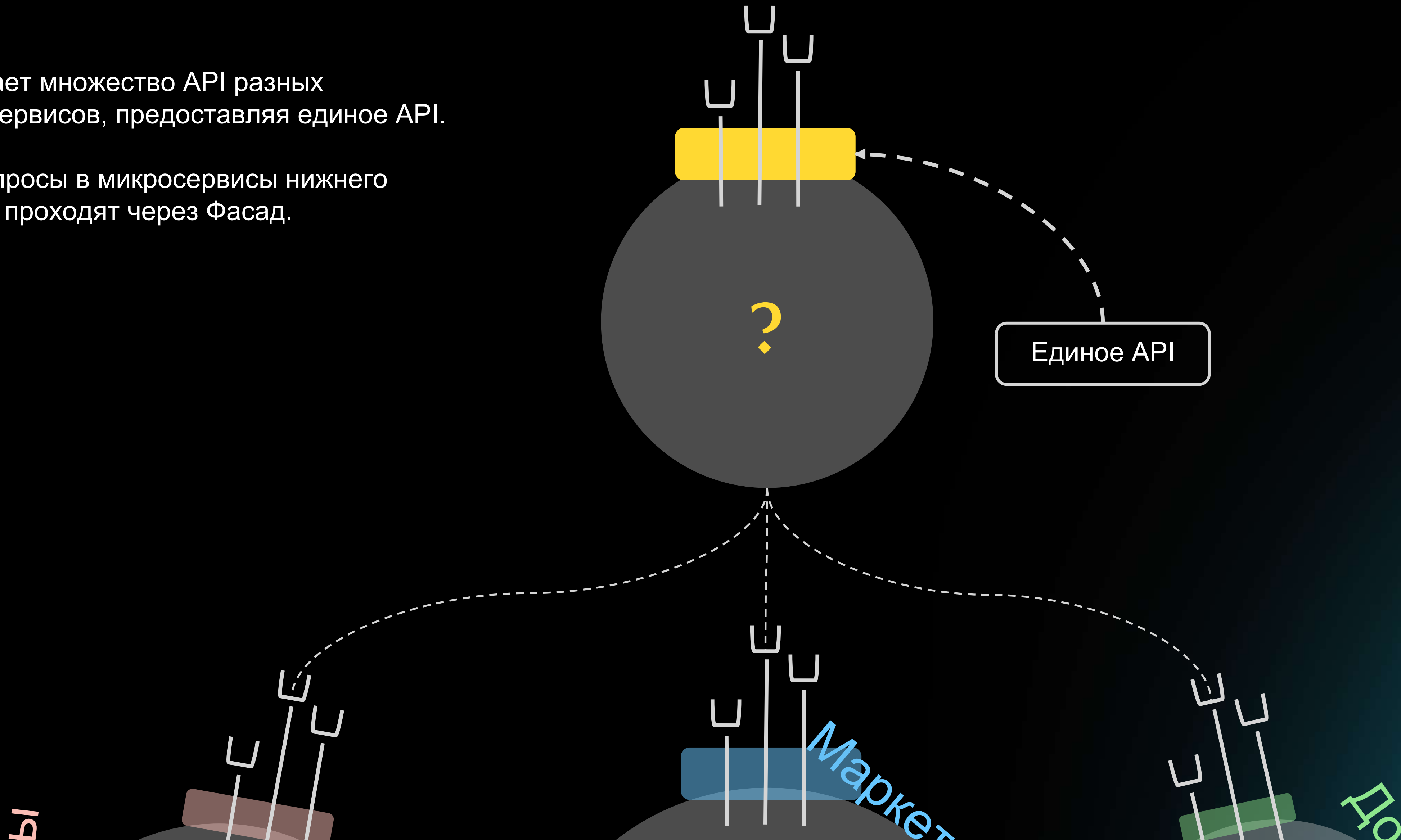


Фасад

Скрывает множество API разных микросервисов, предоставляя единое API.

Все запросы в микросервисы нижнего уровня проходят через Фасад.

Фасад



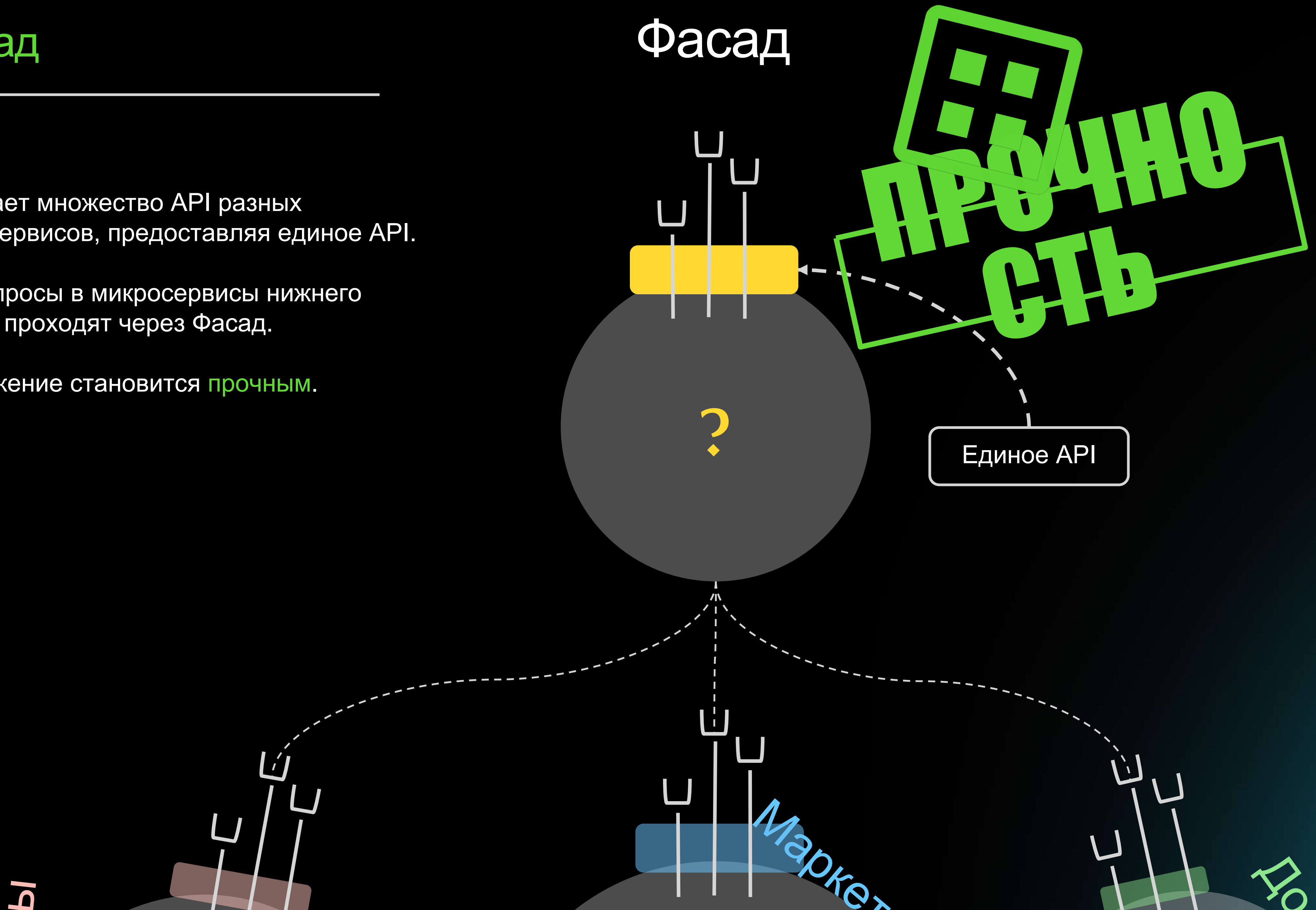
Фасад

Скрывает множество API разных микросервисов, предоставляя единое API.

Все запросы в микросервисы нижнего уровня проходят через Фасад.

Приложение становится **прочным**.

Фасад



Фасад

Скрывает множество API разных микросервисов, предоставляя единое API.

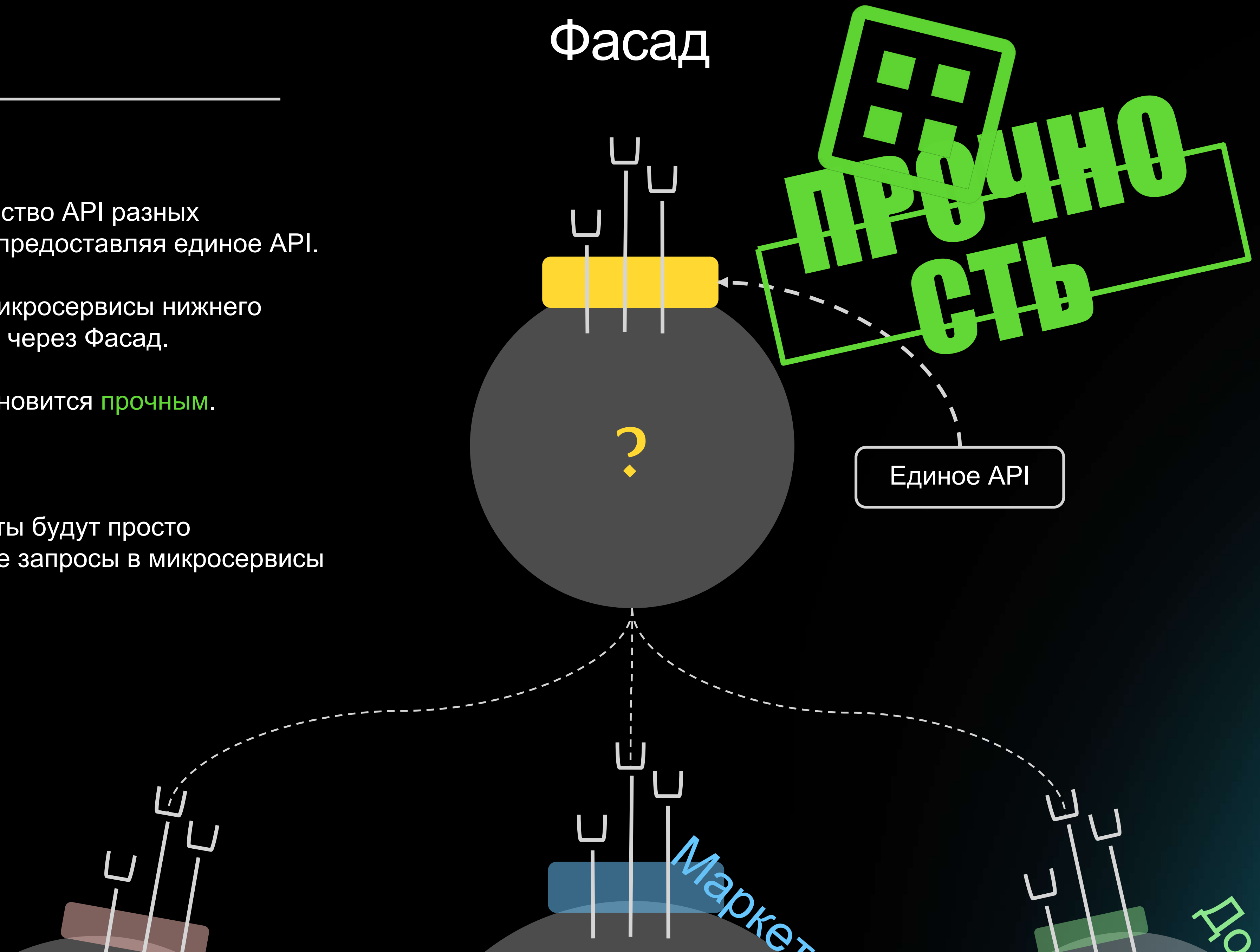
Все запросы в микросервисы нижнего уровня проходят через Фасад.

Приложение становится **прочным**.

Минусы:

Многие эндпоинты будут просто проксировать все запросы в микросервисы нижнего уровня.

Фасад



Подходящие архитектурные паттерны



Фасад

Организация единого API



Оркестратор

Хранение сценариев и
оркестрирование работы
сервисов

Оркестратор

Позволяет поднять сценарии на уровень выше.

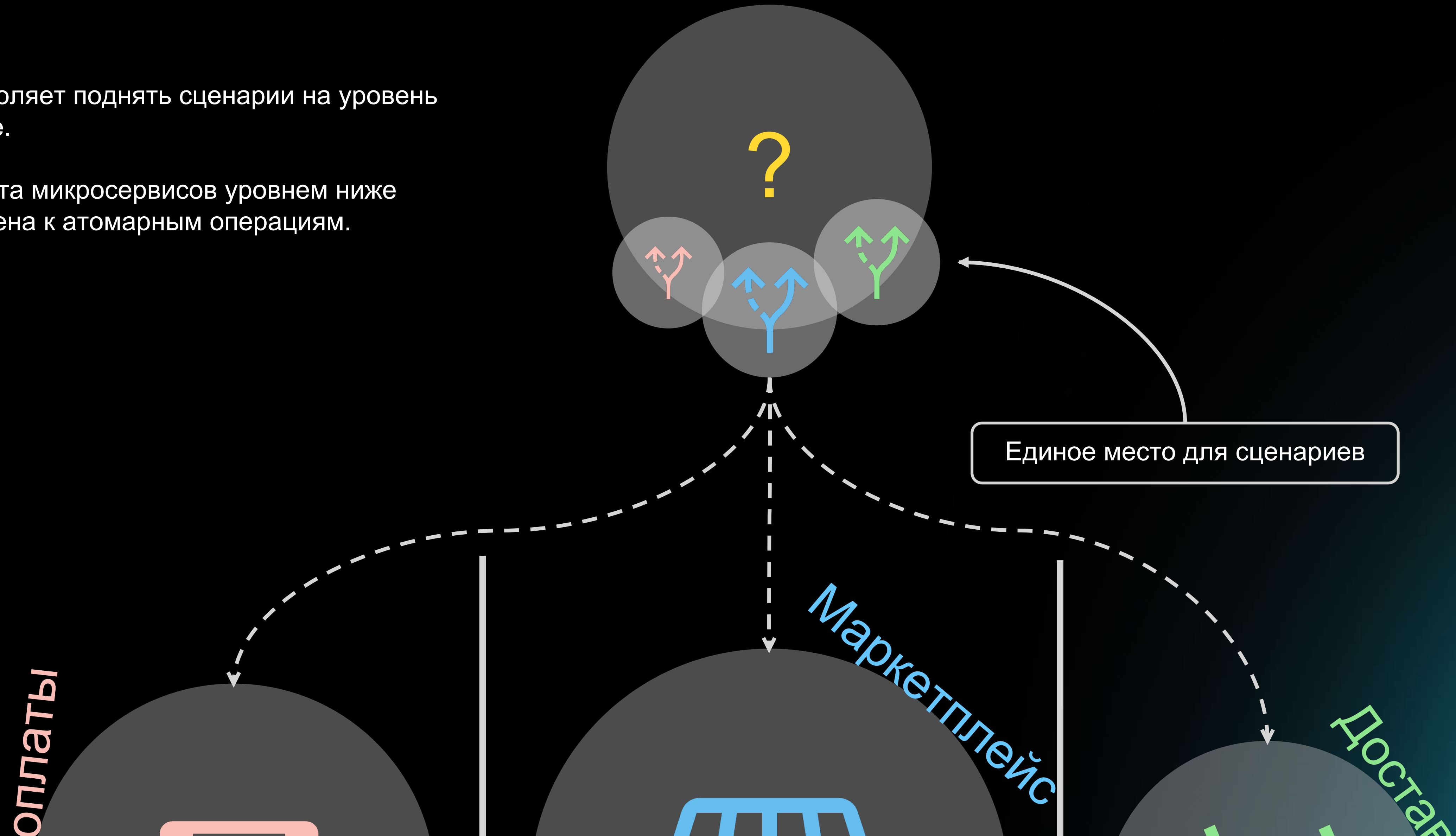


Оркестратор

Позволяет поднять сценарии на уровень выше.

Работа микросервисов уровнем ниже сведена к атомарным операциям.

Оркестратор



Оркестратор

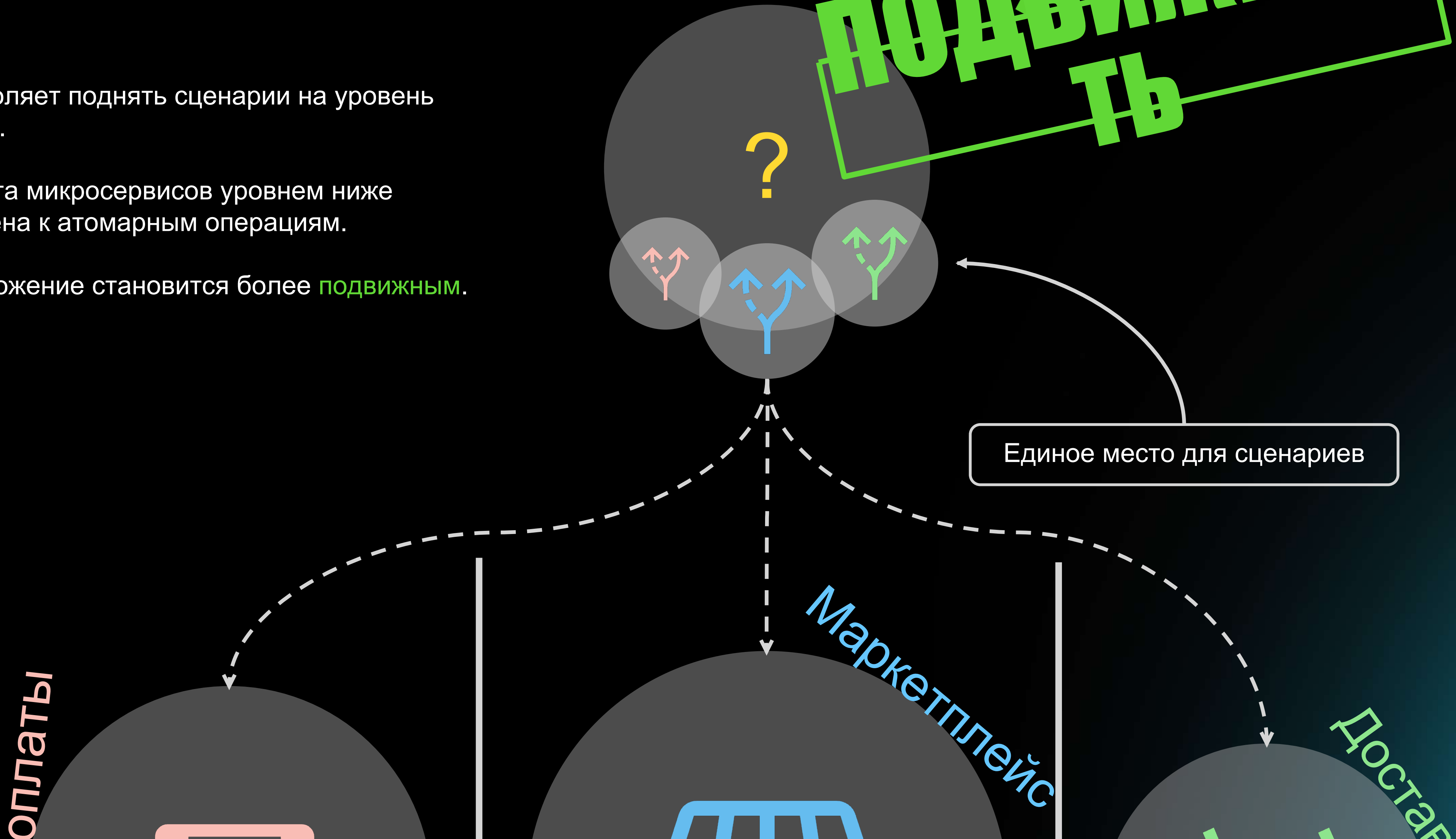
Позволяет поднять сценарии на уровень выше.

Работа микросервисов уровнем ниже сведена к атомарным операциям.

Приложение становится более **подвижным**.

Оркестратор

ПОДВИЖНОСТЬ



Подходящие архитектурные паттерны



Фасад

Организация единого API



Оркестратор

Хранение сценариев и оркестрирование работы сервисов

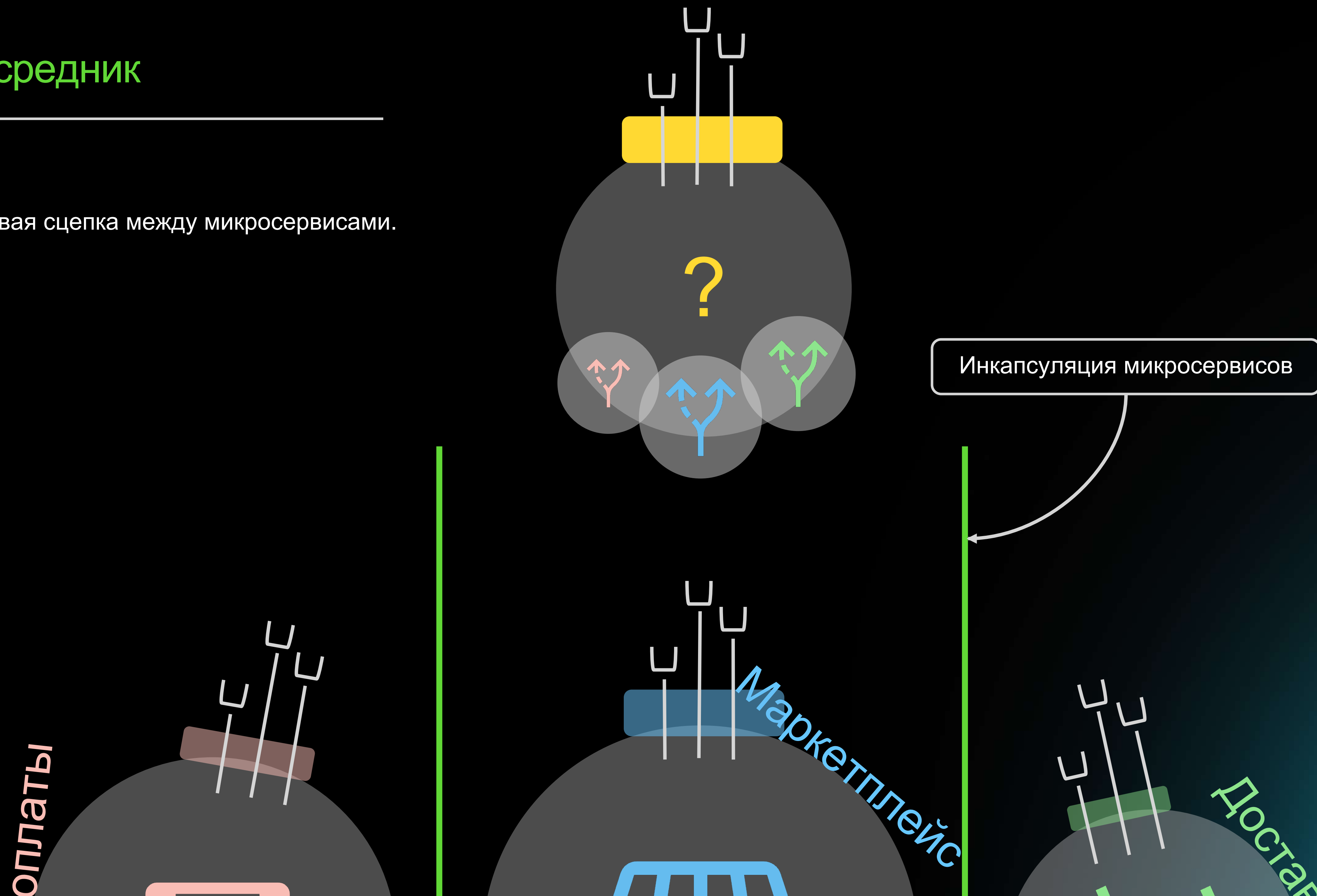


Посредник

Взаимосвязь между сервисами при сохранении их инкапсуляции

Посредник

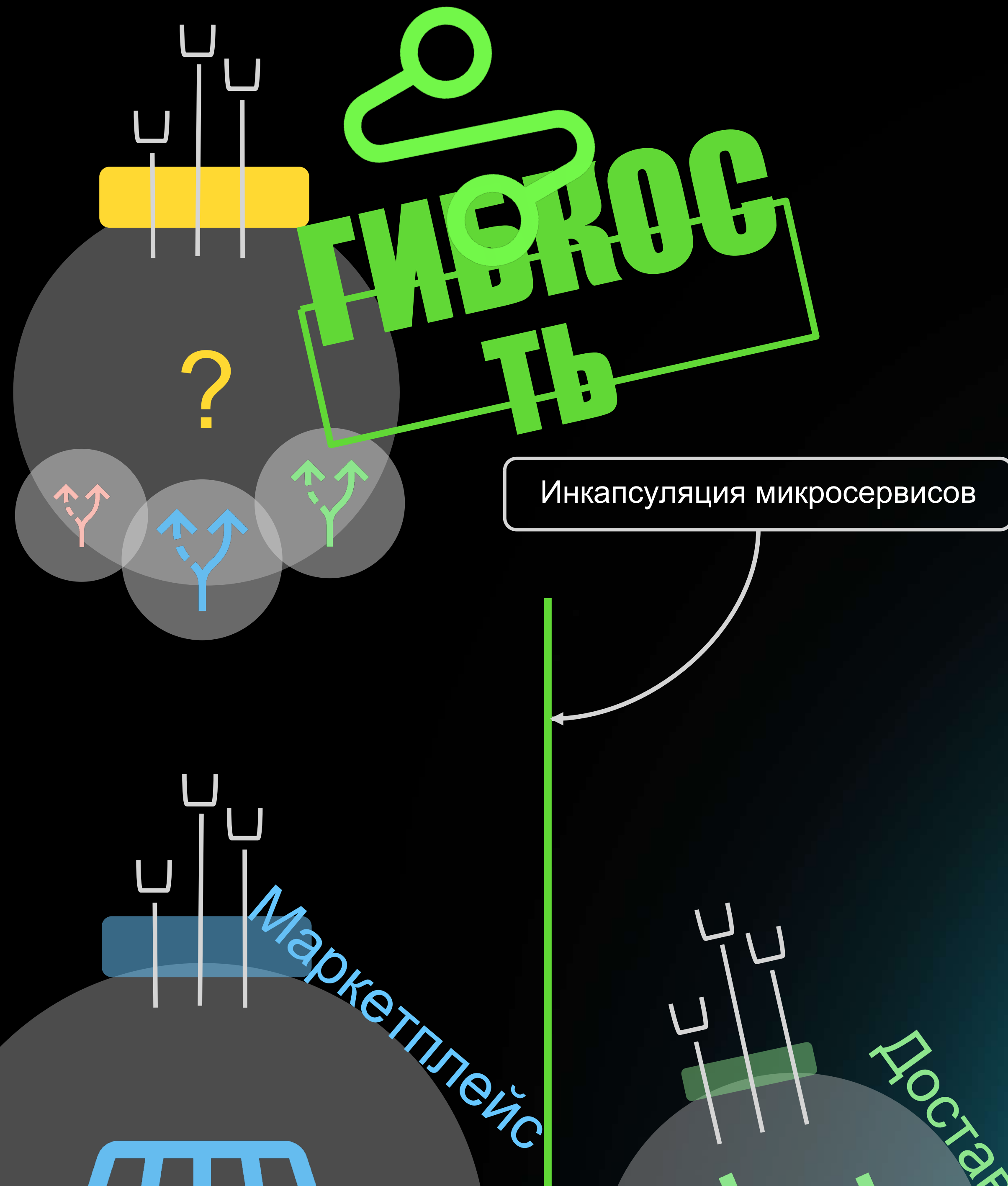
Нулевая сцепка между микросервисами.



Посредник

Нулевая сцепка между микросервисами.

Приложение становится гибким.



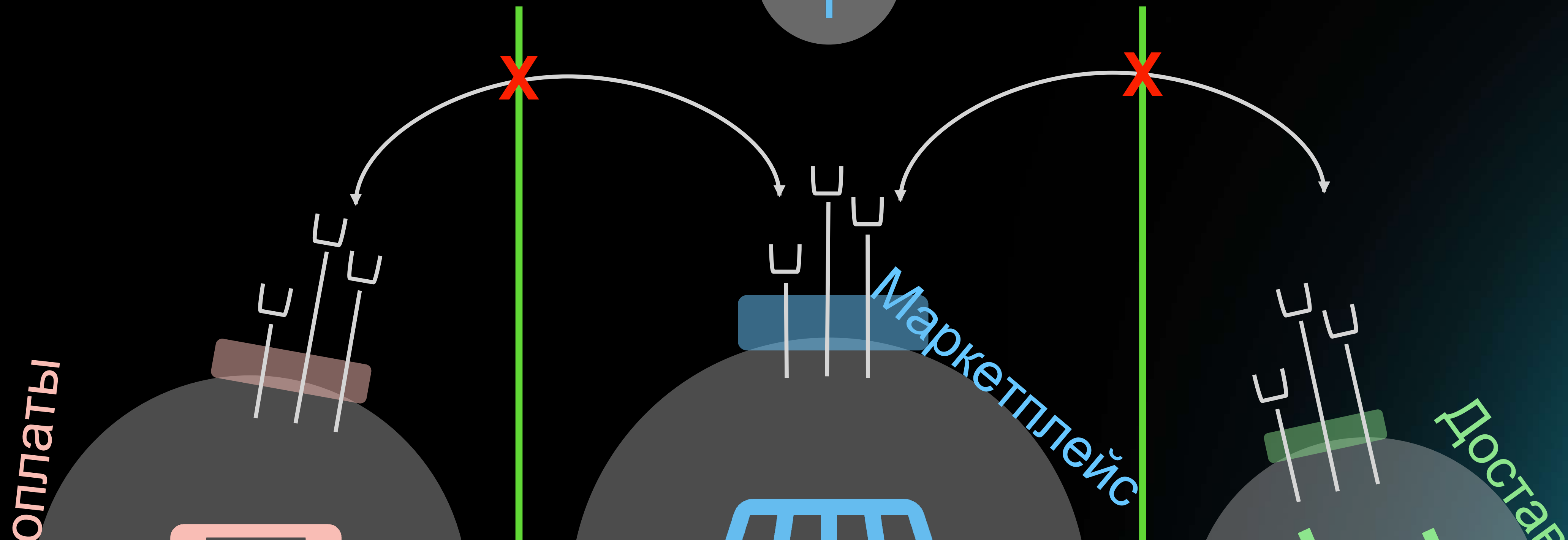
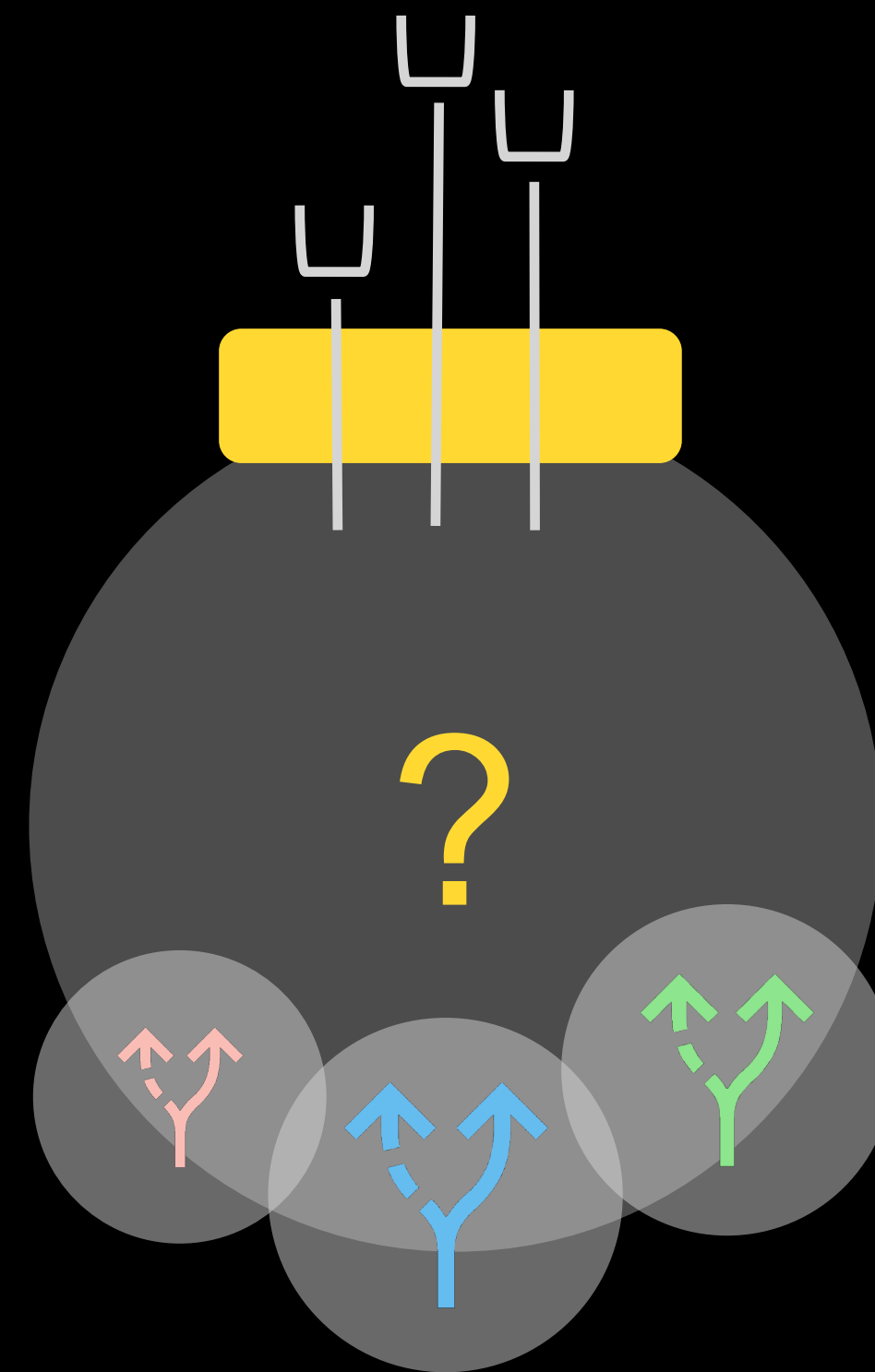
Посредник

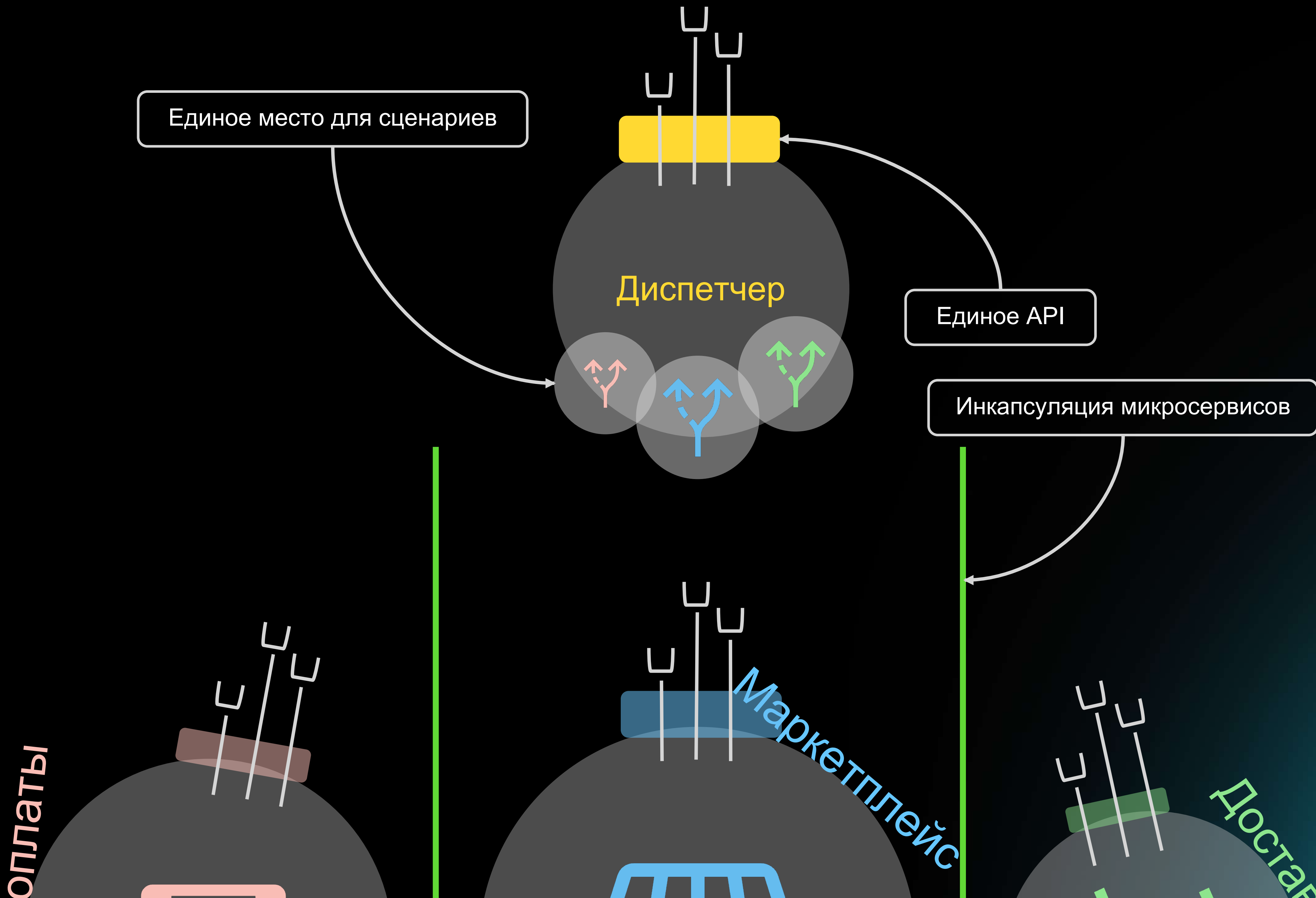
Нулевая сцепка между микросервисами.

Приложение становится **гибким**.

Замечание:

Между нижестоящими сервисами отныне не может быть никаких коммуникаций.





Domain-Driven Design: в чём профит в нашем случае?

Domain-Driven Design: в чём профит в нашем случае?

Фокус на **домене**, а не на
ТЕХНОЛОГИЯХ

Разработчики вынуждены глубоко понимать систему, а не просто писать код.

Это снижает риск *over-engineering* и нерелевантных абстракций и помогает обнаружить архитектурные проблемы в самом начале.



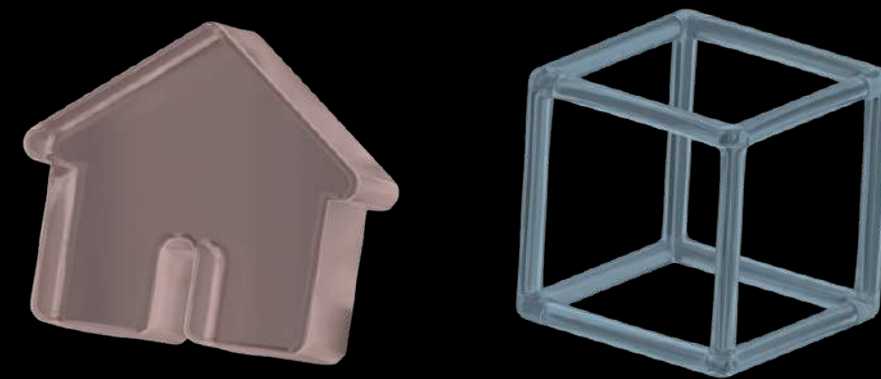
Domain-Driven Design: в чём профит в нашем случае?

Фокус на **домене**, а не на
ТЕХНОЛОГИЯХ

Чёткие границы контекстов

Система разделяется на ограниченные контексты, каждый со своей моделью и логикой.

Это предотвращает риск возникновения спагетти-кода и неконтролируемых зависимостей.



Domain-Driven Design: в чём профит в нашем случае?

Фокус на **домене**, а не на
ТЕХНОЛОГИЯХ

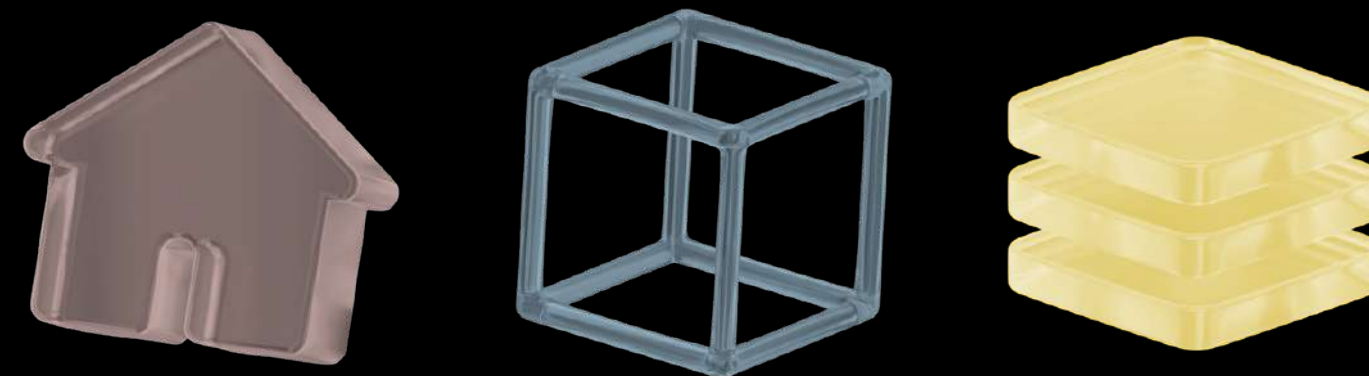
Чёткие **границы контекстов**

Слоистая архитектура

DDD явно разделяет код на:

- Домен (ядро бизнес-логики)
- Приложение (оркестрация)
- Инфраструктуру (технические детали)

Это предотвращает смешивание логики (как, в нашем случае, DTO в репозитории).



Domain-Driven Design: в чём профит в нашем случае?

Фокус на **домене**, а не на
ТЕХНОЛОГИЯХ

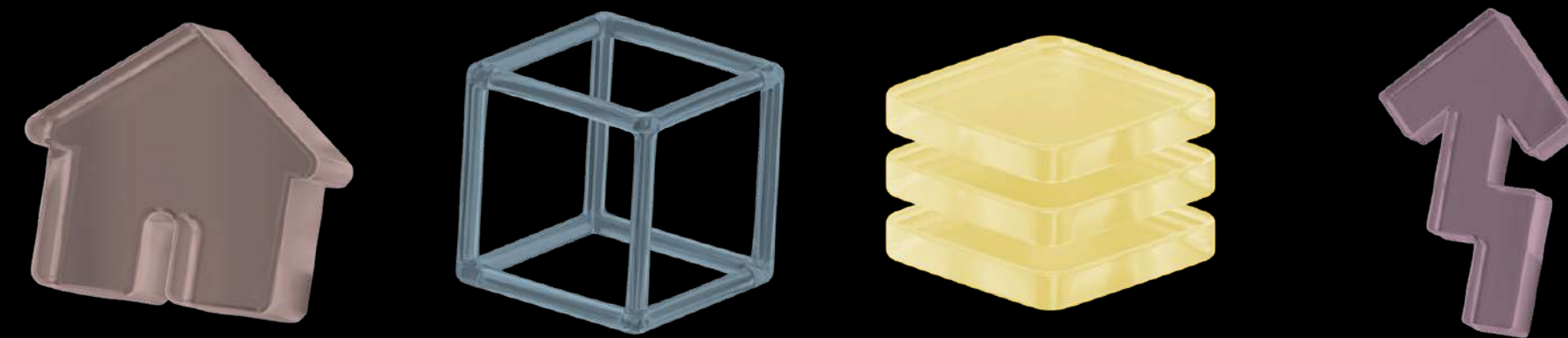
Чёткие **границы контекстов**

Слоистая архитектура

Акцент на **гибкости**

Чистые доменные модели легко тестировать.

Инфраструктурные детали вынесены за пределы домена, что упрощает замену технологий (пример с базой данных).



Domain-Driven Design: в чём профит в нашем случае?

Фокус на **домене**, а не на
ТЕХНОЛОГИЯХ

Чёткие **границы контекстов**

Слоистая архитектура

Акцент на **гибкости**

Чистые
Инфраструктуру
замену



Как полюбить модульное тестирование:
обратная сторона TDD



Domain-Driven Design: в чём профит в нашем случае?

Фокус на **домене**, а не на технологиях

Чёткие границы контекстов

Слоистая архитектура

Акцент на **гибкости**

Избегание «Большого кома грязи»

DDD поощряет модульность, низкую связанность (low coupling) и высокую связность (high cohesion) компонентов.

Это позволяет избежать бизнес-логики в клиентах, общей «божественной» модели и жёсткой привязки к базе данных.



Domain-Driven Design: в чём профит в нашем случае?

Фокус на **домене**, а не на технологиях

Чёткие границы контекстов

Слоистая архитектура

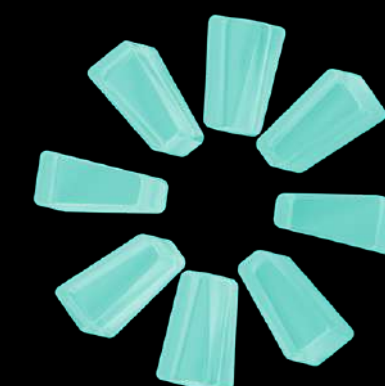
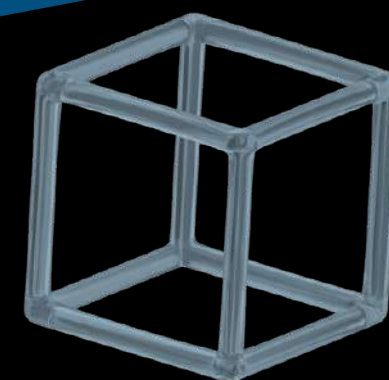
Акцент на **гибкости**

Избегание «**Больших
грязи**»

DDD поощряет модульность, низкую связанность (low coupling) и высокую связность (high cohesion) компонентов.

Это позволяет избежать бизнес-логики в клиентах, общей «божественной» модели и жёсткой привязки к базе данных.

Кто начинает разработку с проектирования базы данных?



Domain-Driven Design: в чём профит в нашем случае?

Фокус на **домене**, а не на технологиях

Чёткие **границы контекстов**

Слоистая архитектура

Акцент на **гибкости**

Избегание «**Большого кома грязи**»

DDD поощряет модульность, низкую связанность (low coupling) и высокую связность (high cohesion) компонентов.

Это позволяет избежать бизнес-логики в клиентах, общей «божественной» модели и жёсткой привязки к базе данных.



Domain-Driven Design системно борется с главными причинами плохой архитектуры:

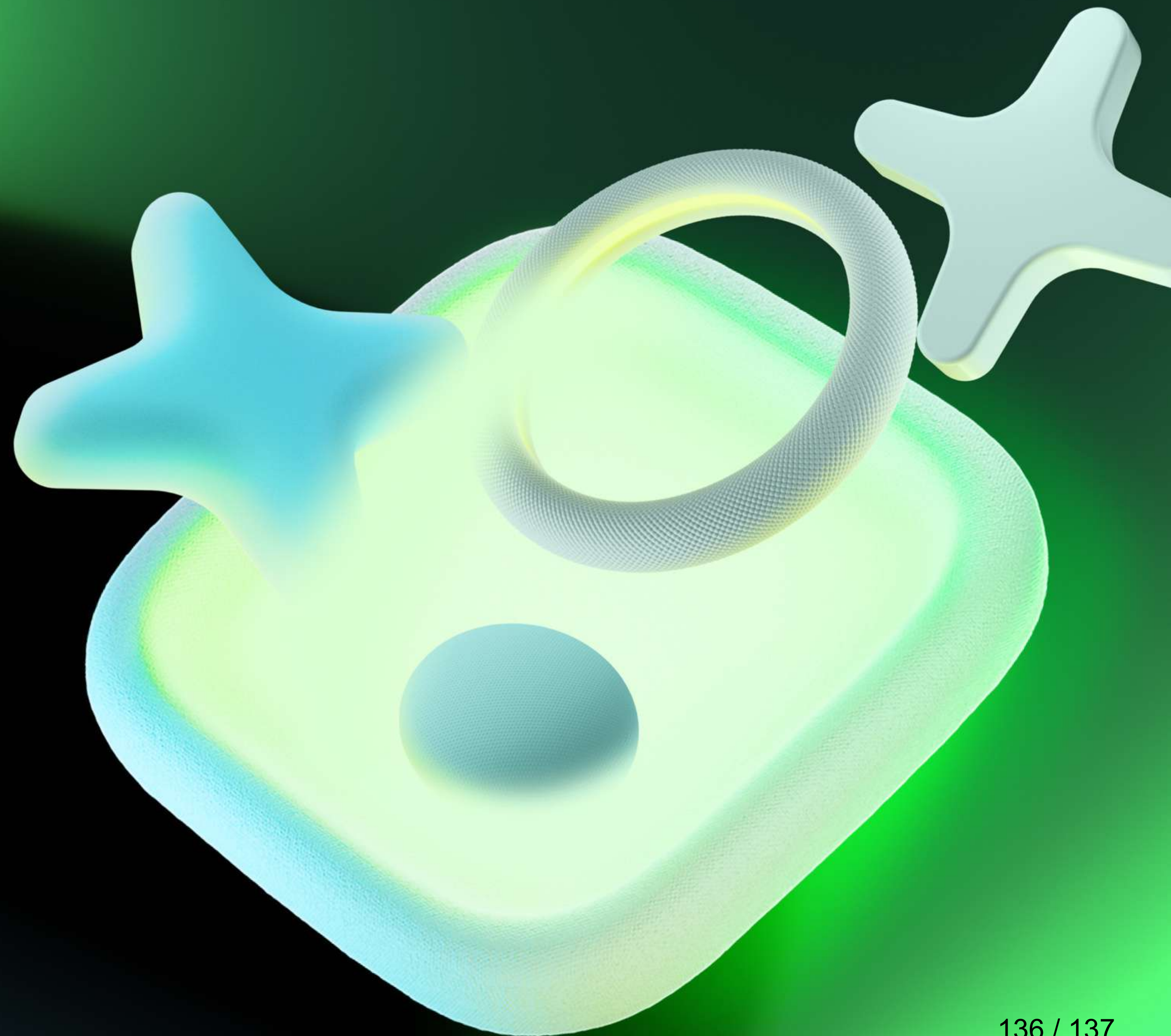
непонимание бизнес-логики, смешивание слоёв, отсутствие чётких границ.



Команда
Сбера

**Познай домен —
и ты познаешь систему!**

/ Сократ /





Команда
Сбера



Статья на Хабре



Канал в Telegram