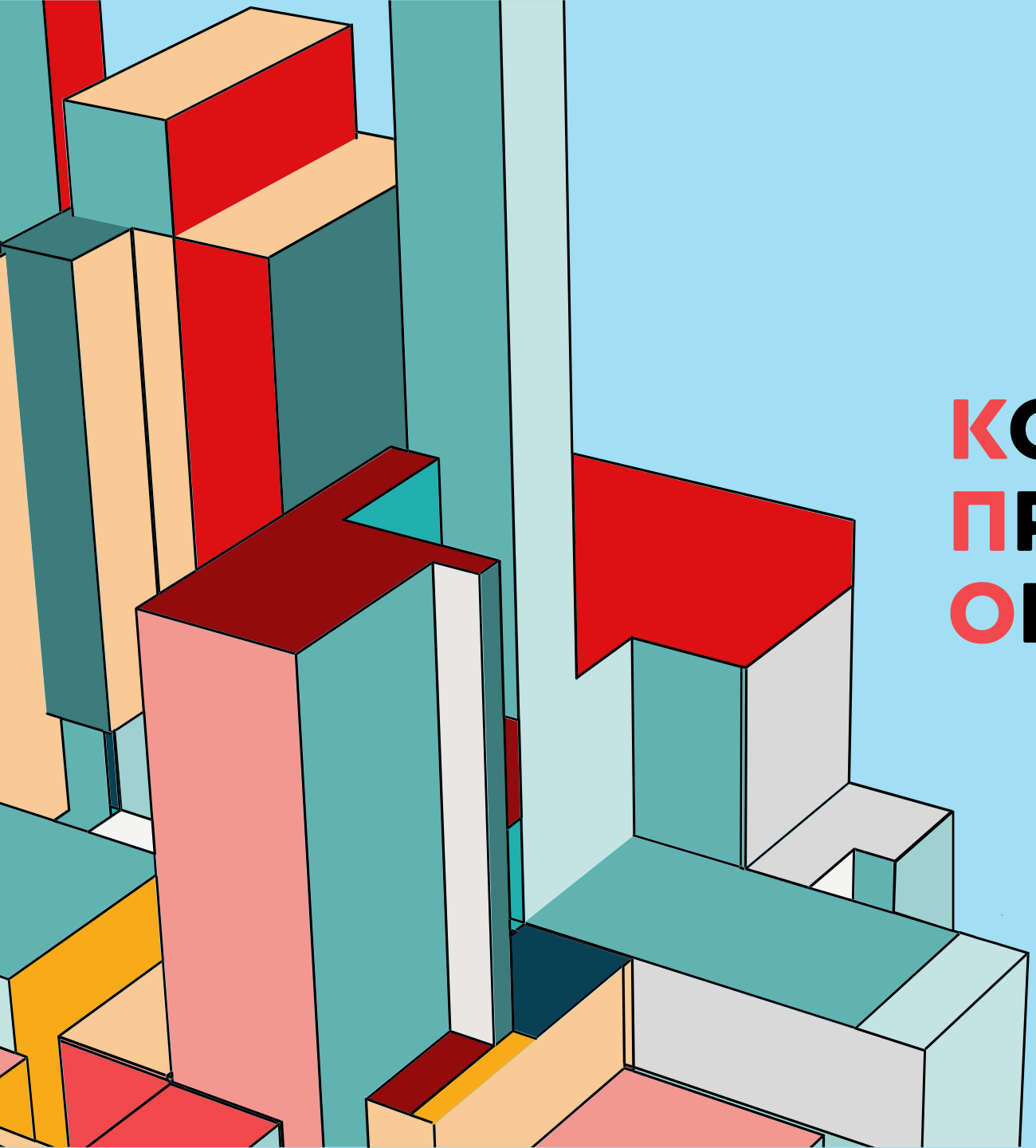


КОНСТРУИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ПЛАН ЛЕКЦИИ № 8

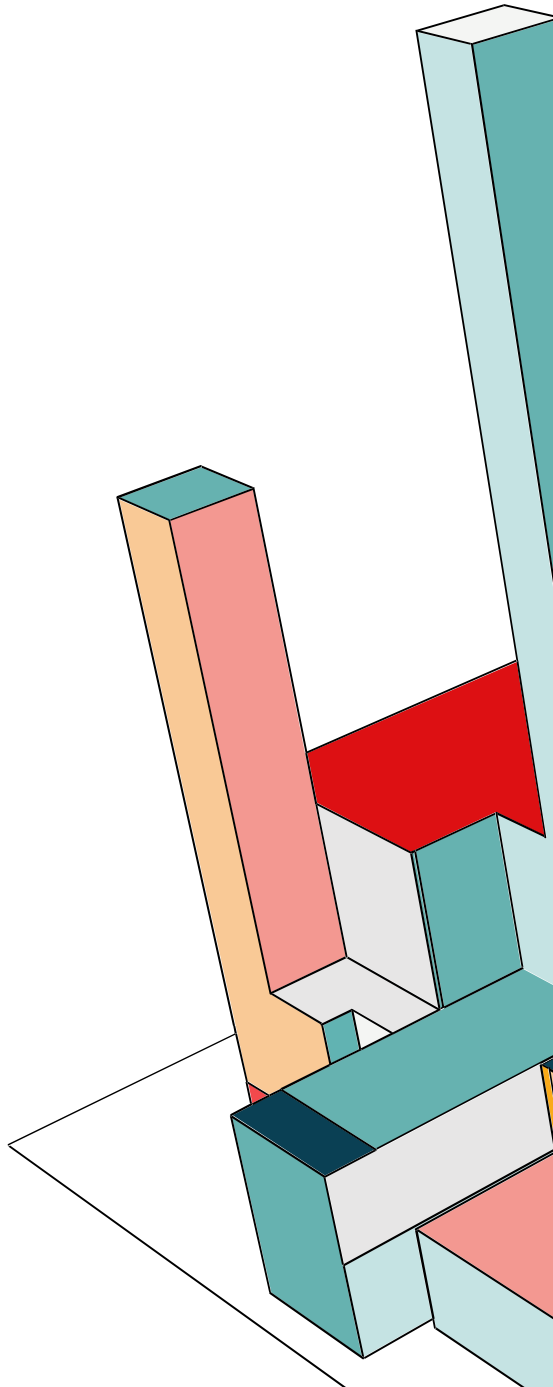
От MVC до Clean Architecture

Эволюция Enterprise-архитектур

1. Трехслойная архитектура
2. Ports And Adapters
3. Onion Architecture
4. Clean Architecture

MV*-паттерны презентационного слоя

Model-View-Controller, Model-View-Presenter, Presentation Model, Model-View-ViewModel



ЗАЧЕМ ЗНАТЬ ИСТОРИЮ?

Современные архитектуры сложные.
Ретроспективно их понять легче

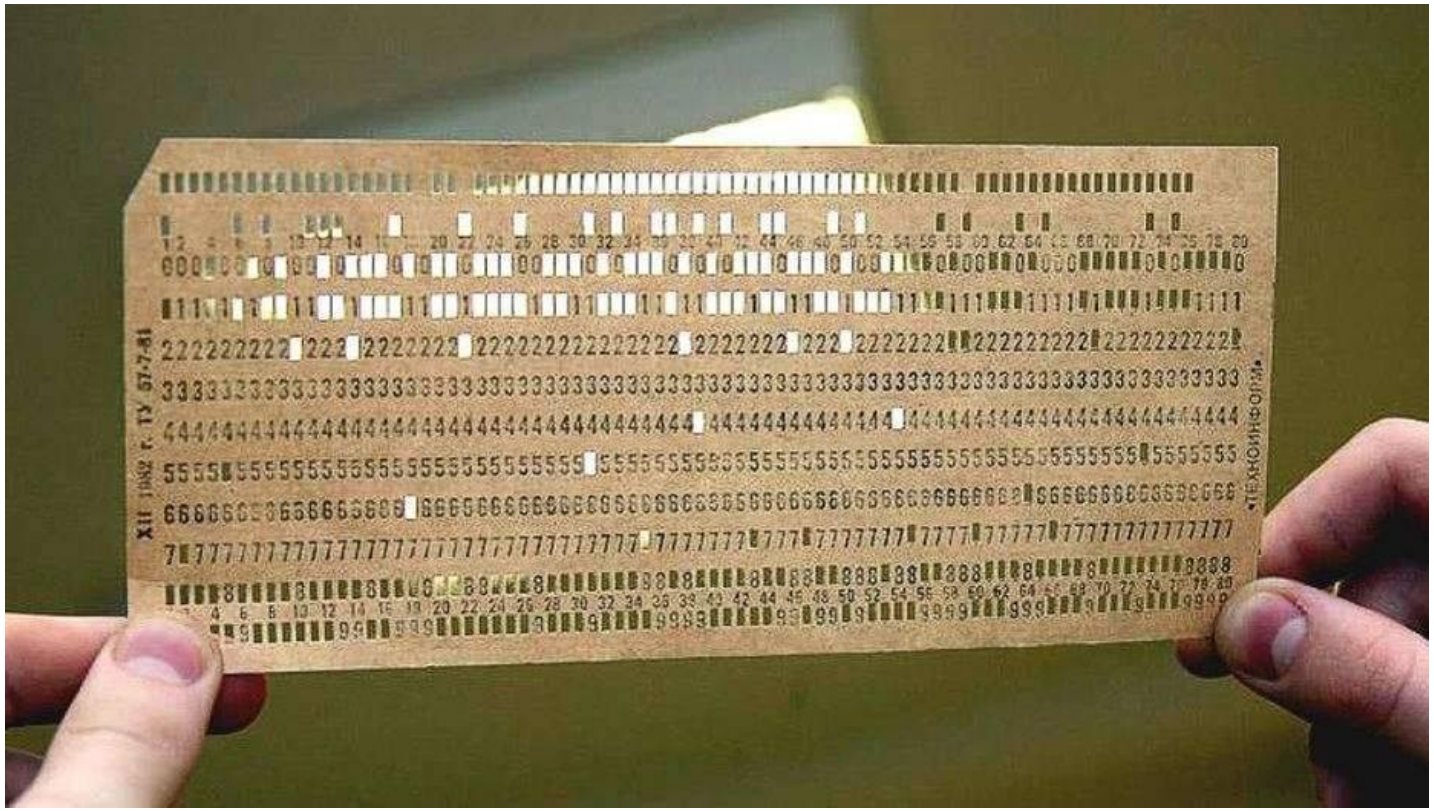
Отбрасывать лишнее, добавлять собственное.

Предсказывать будущее.

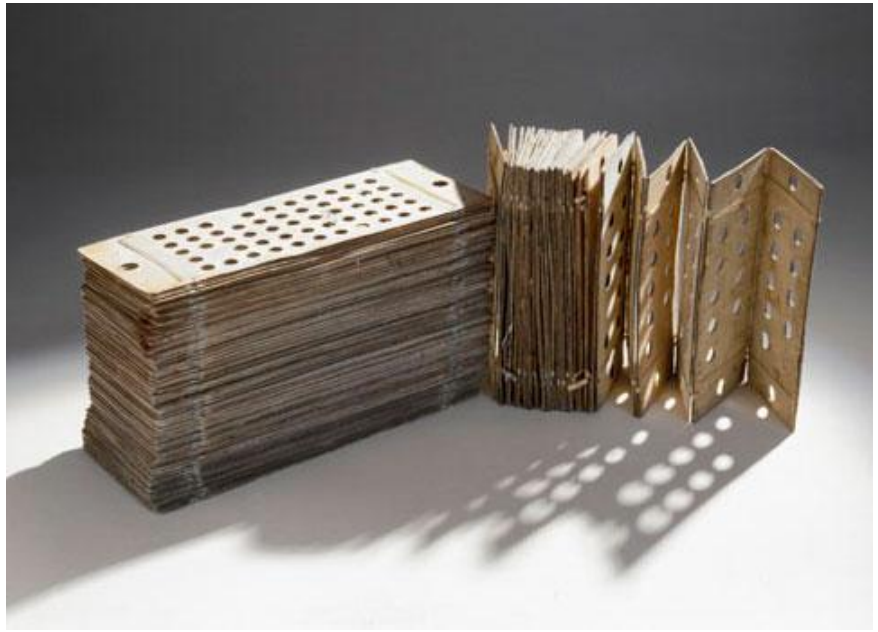
ЧЕГО НЕ БУДЕТ

1. Всех архитектур
2. Топологии
3. Хирургической точности

ЭТО СЛОЖНО?



СЛОЖНО...



КАК ПОБЕДИТЬ СЛОЖНОСТЬ?

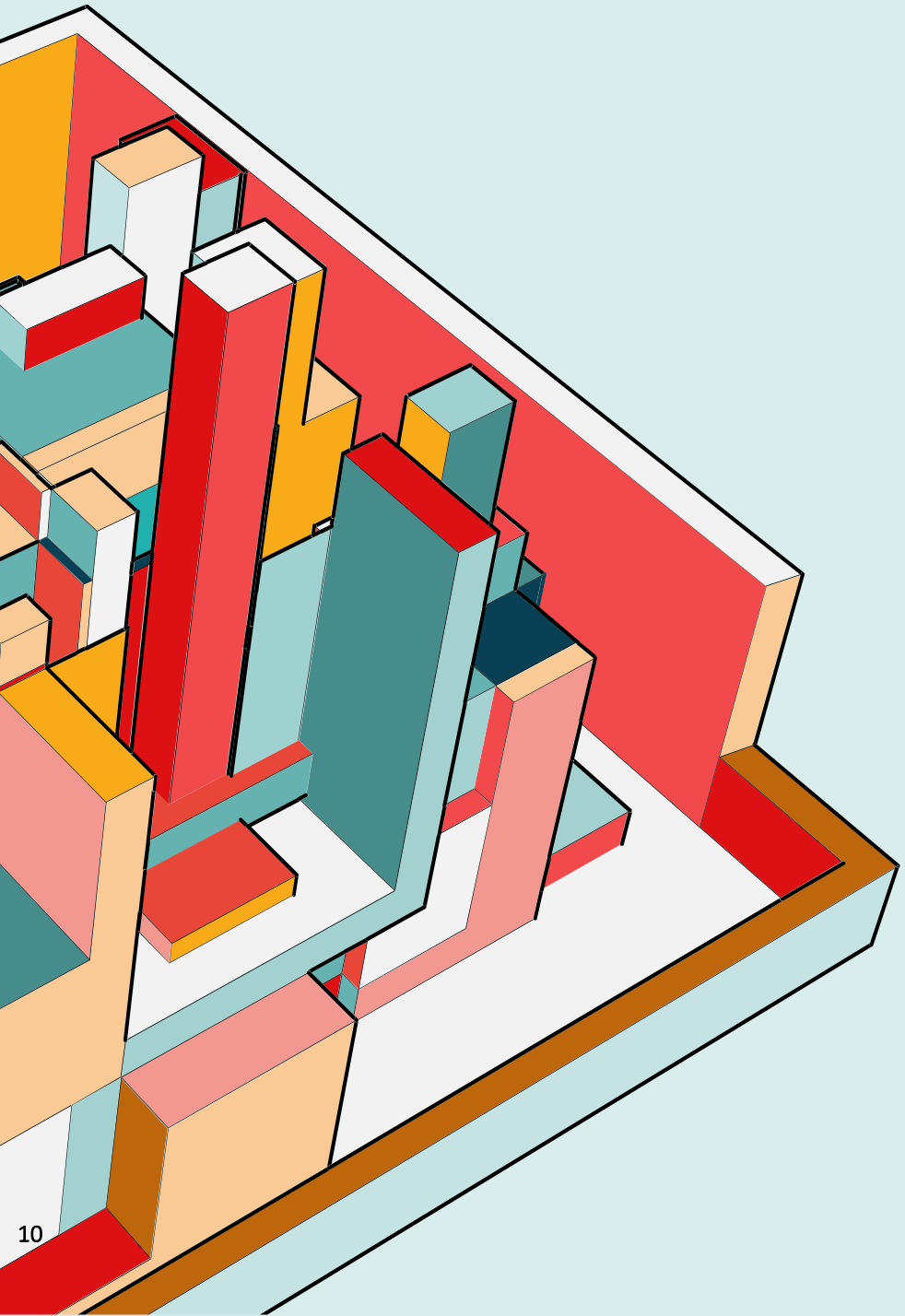
1. Упорядочить
2. Поднять уровень абстракции

ЗАДАЧА АРХИТЕКТУРЫ

1. Победить сложность
2. Обеспечить адаптивность
3. Дать общий язык

ХОРОШАЯ АРХИТЕКТУРА

1. Low coupling
2. Don’t Repeat Yourself
3. Баланс и здравый смысл



MODEL VIEW CONTROLLER

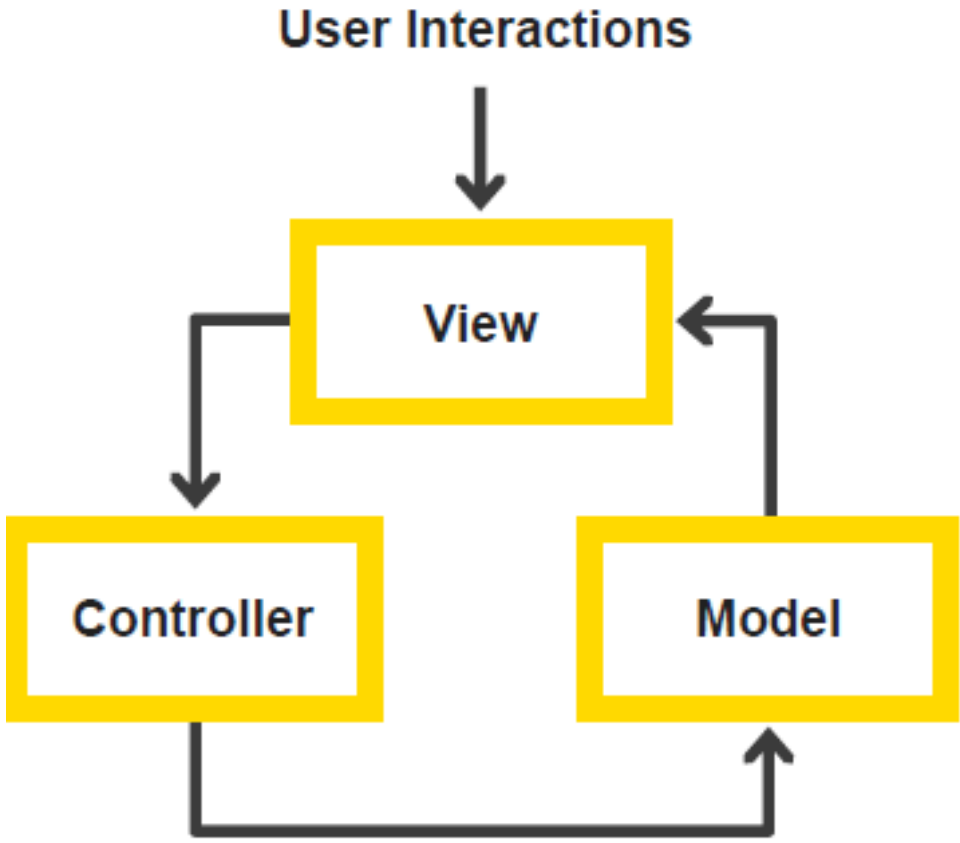
1979 год

MVC

Отделяет бизнес-логику
от представления

Множество вариаций
по логике взаимодействия

Множество вариаций
под конкретные среды



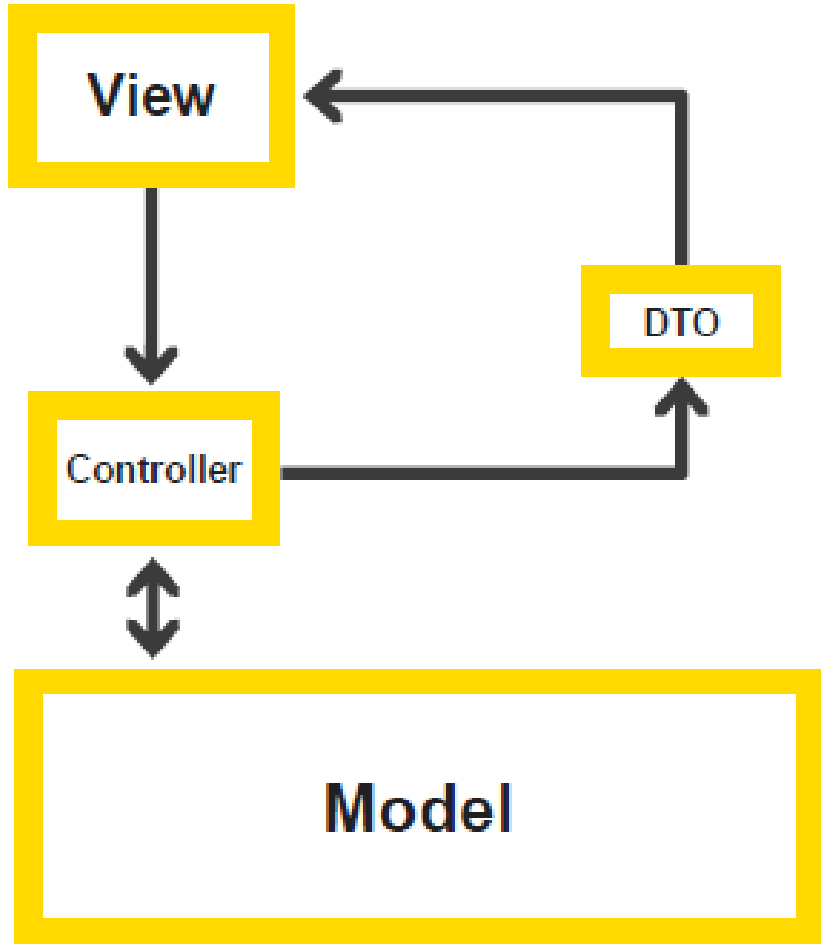
MV-...
НАШИ ДНИ

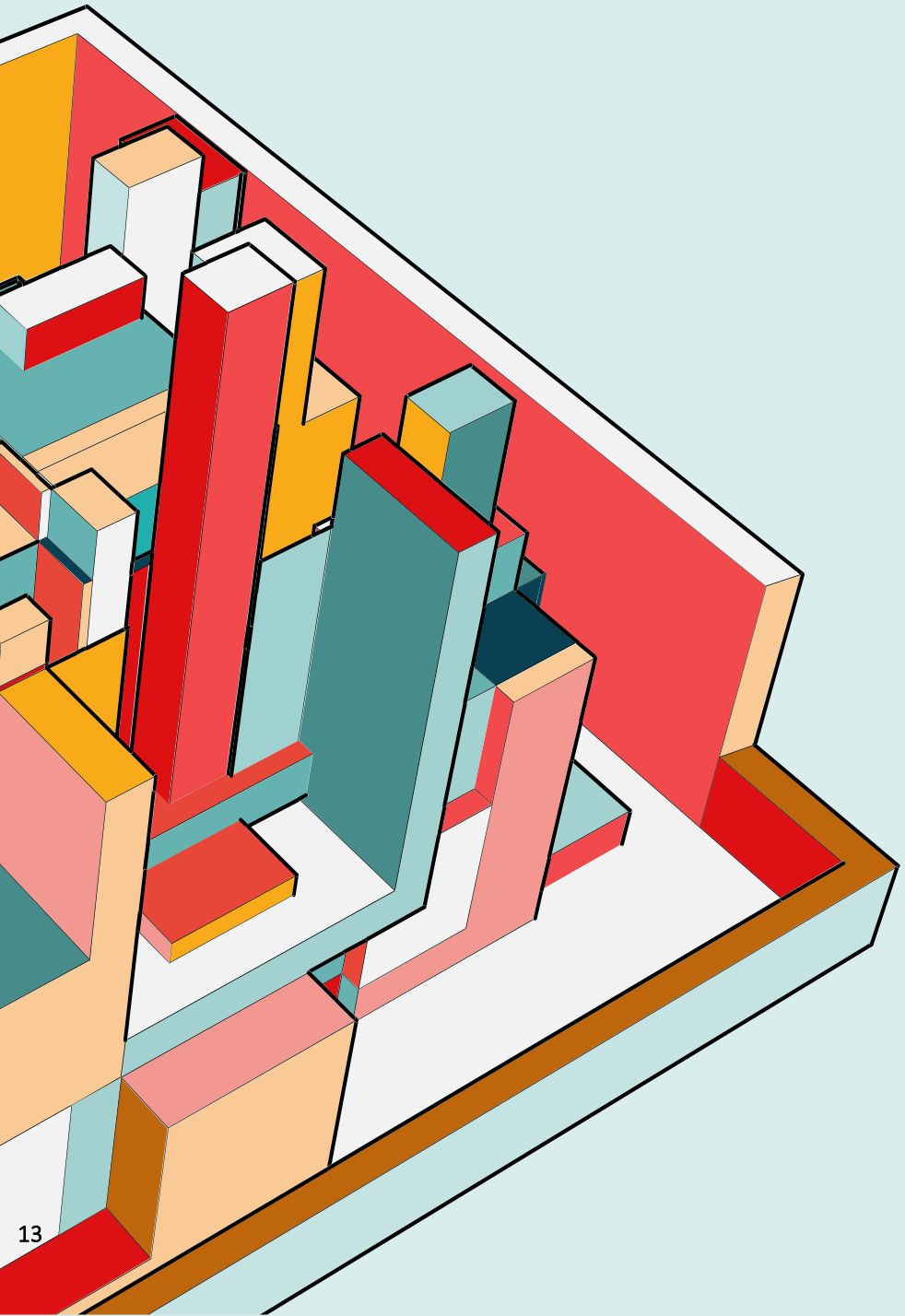
View стал проще благодаря технологиям

Controller - цикломатическая сложность 1

Прямая связь **Model < - > View -** нехорошо

Сложность **Model** выросла, получил свои архитектуры

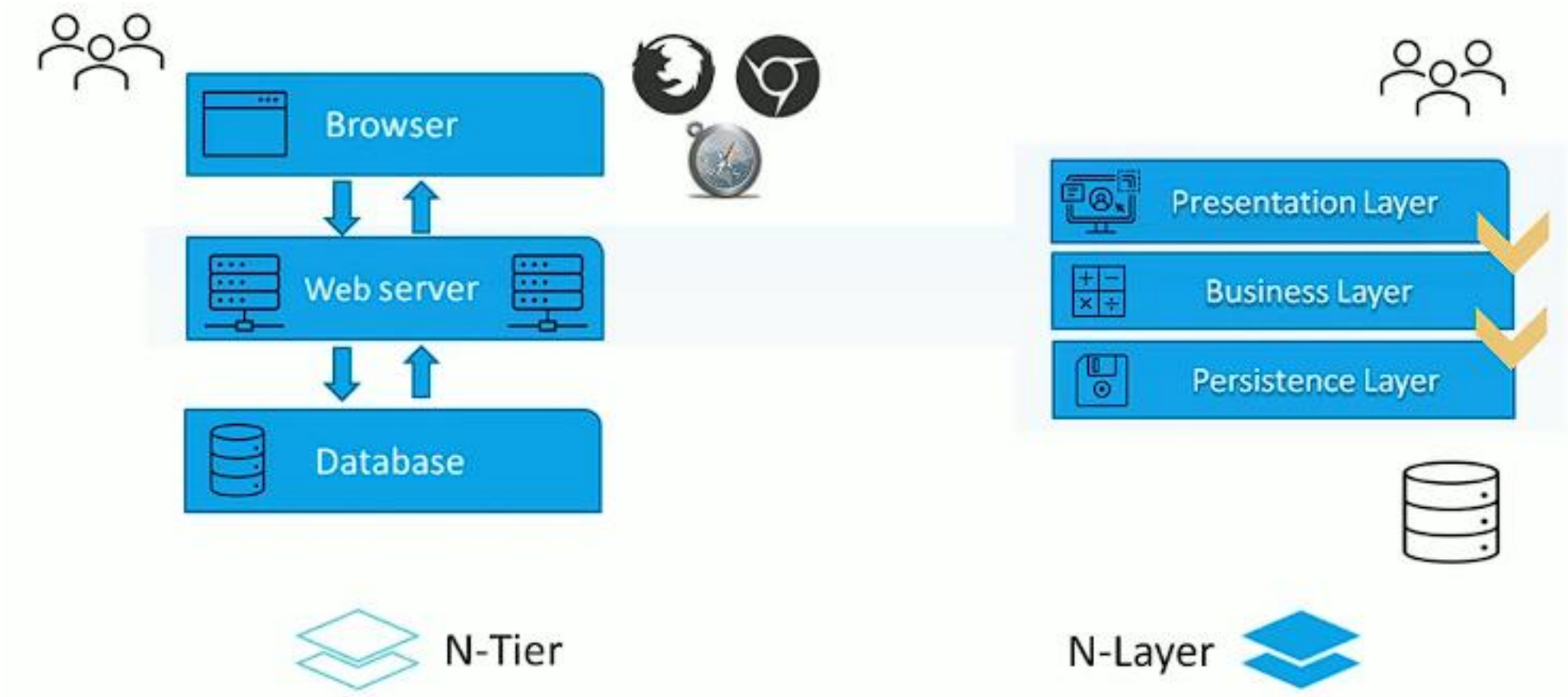


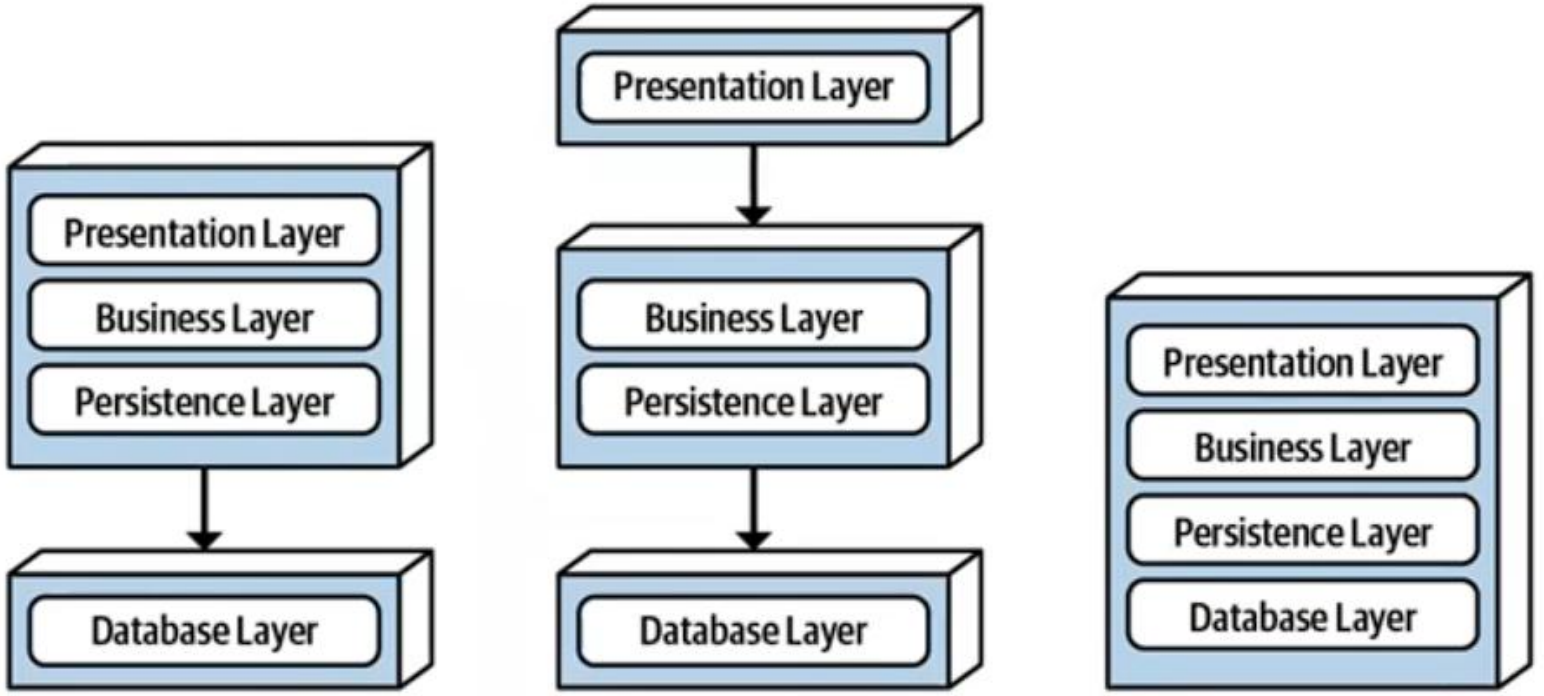


ТРЕХСЛОЙНАЯ АРХИТЕКТУРА

2002 год

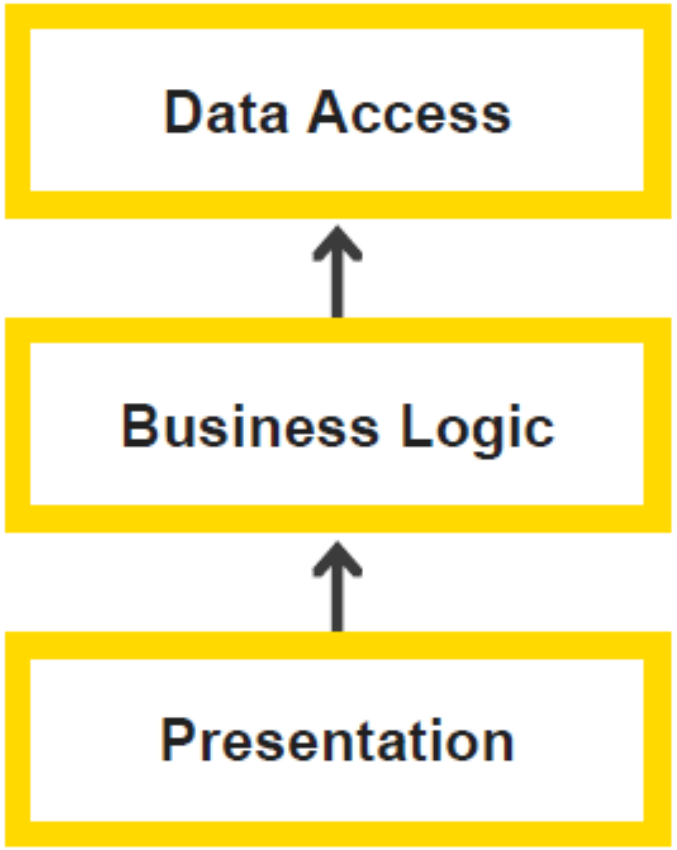
N – TIER VS N – LAYER




Presentation Layer
Business Layer
Persistence Layer
Database Layer
Presentation Layer
Business Layer
Persistence Layer
Database Layer
Presentation Layer
Business Layer
Persistence Layer
Database Layer

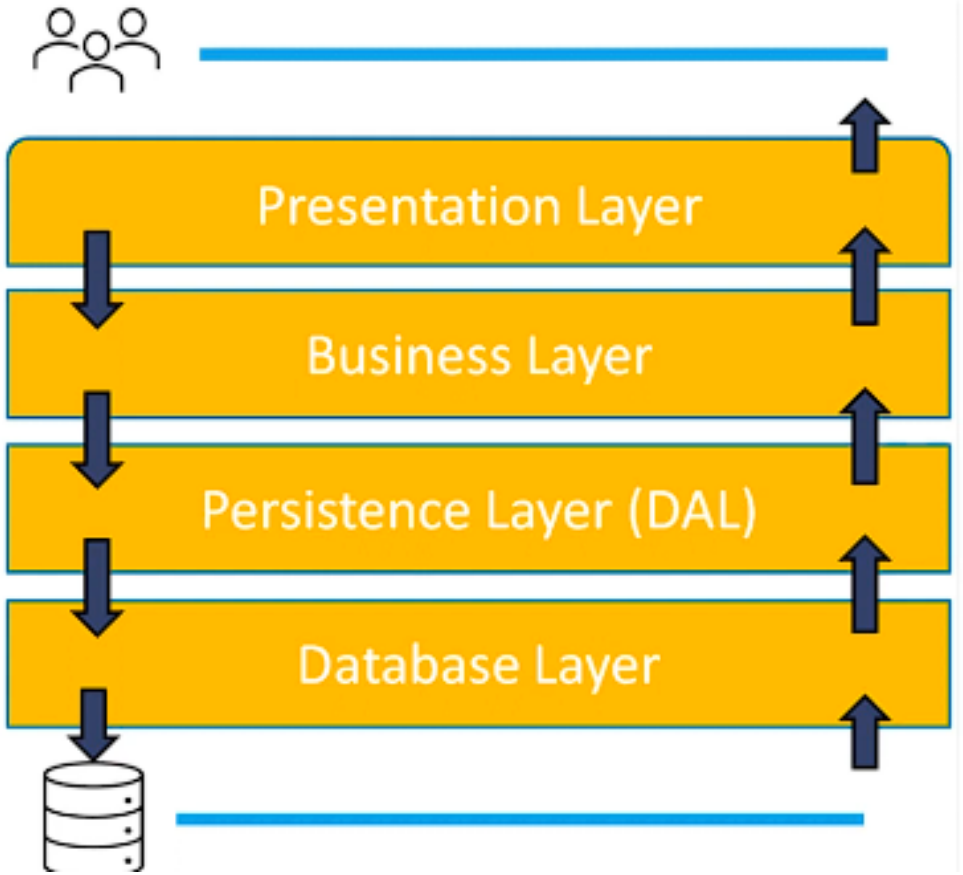
ТРЕХСЛОЙНАЯ АРХИТЕКТУРА

- Отдельные слои с собственными тактическими шаблонами
- Четкая иерархия связей
- Полное отделение бизнес логики от БД

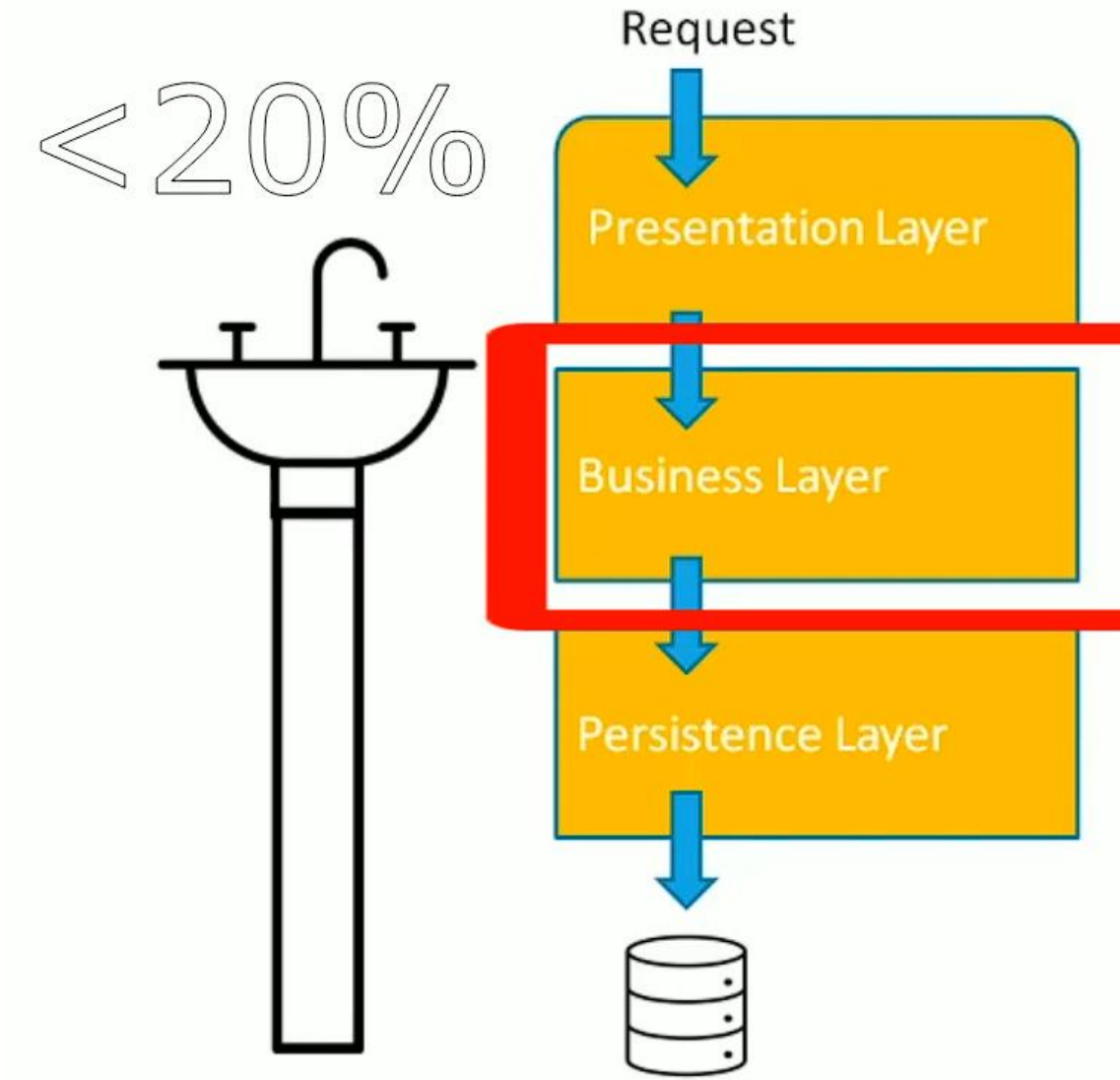


CLOSED LAYER

Layer isolation (изоляция слоев) – подход, при котором изменение кода в одном слое не требует значительных изменений других слоев



ARCHITECTURE SINKHOLE – АНТИ-ПАТТЕРН

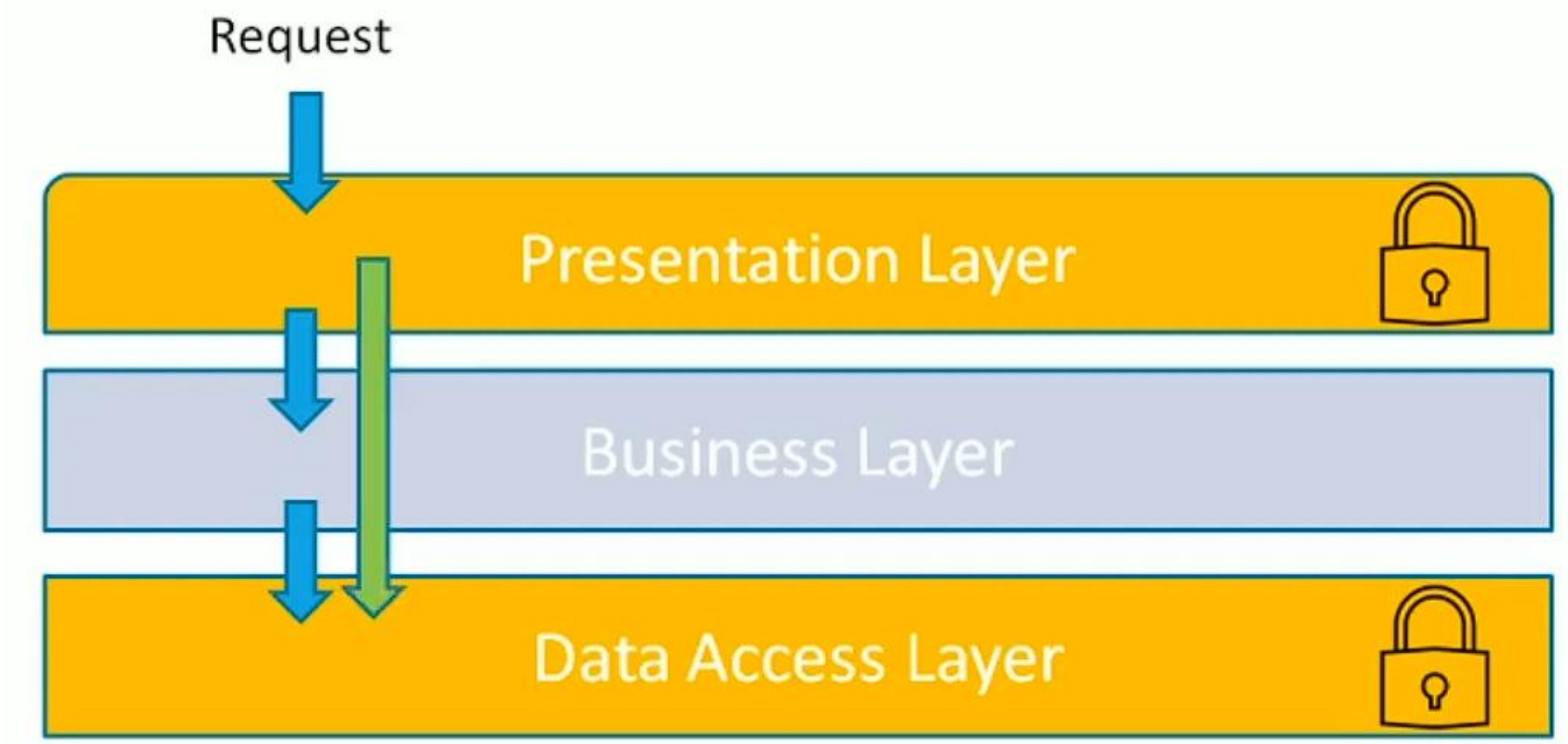


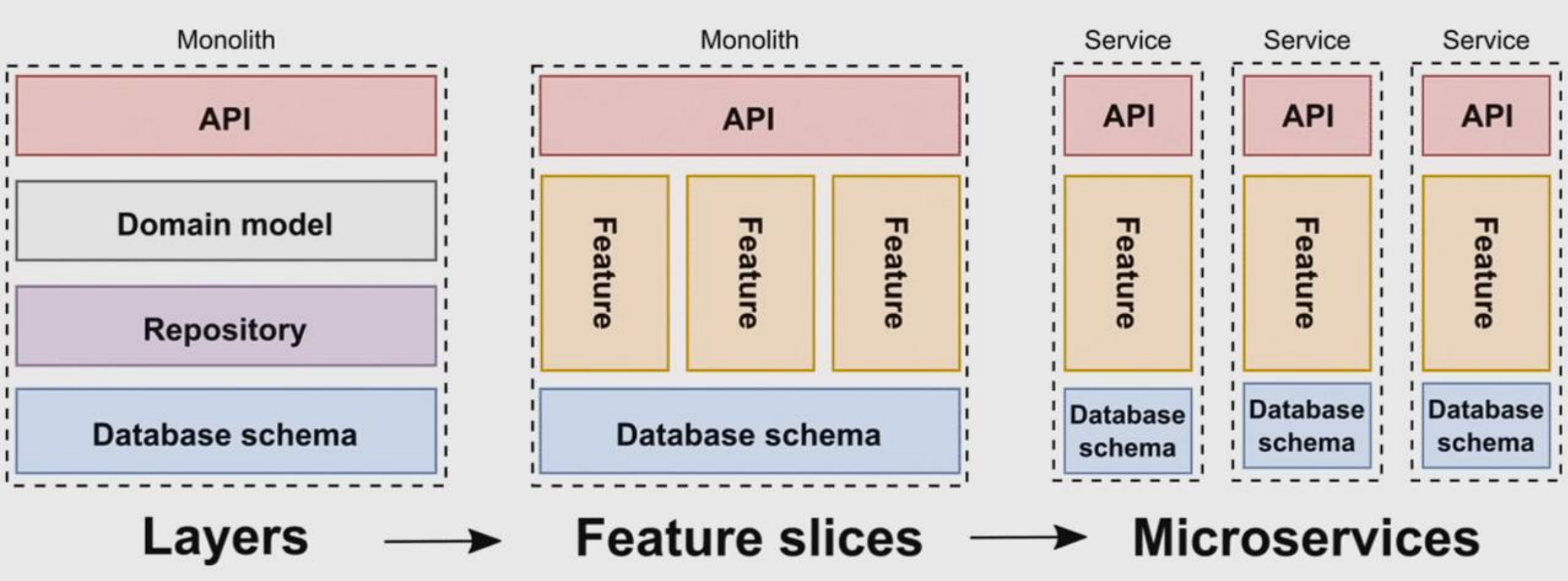

```
public HttpResponseMessage GetAboutPage()
{
    var aboutPage = _pageService.GetPage("about");
    return CreateResponse(HttpStatusCode.Ok, aboutPage);
}
```

```
// inside pageService
public string GetPage(string pageName)
{
    _pagesRepository.GetPage(pageName);
}
```

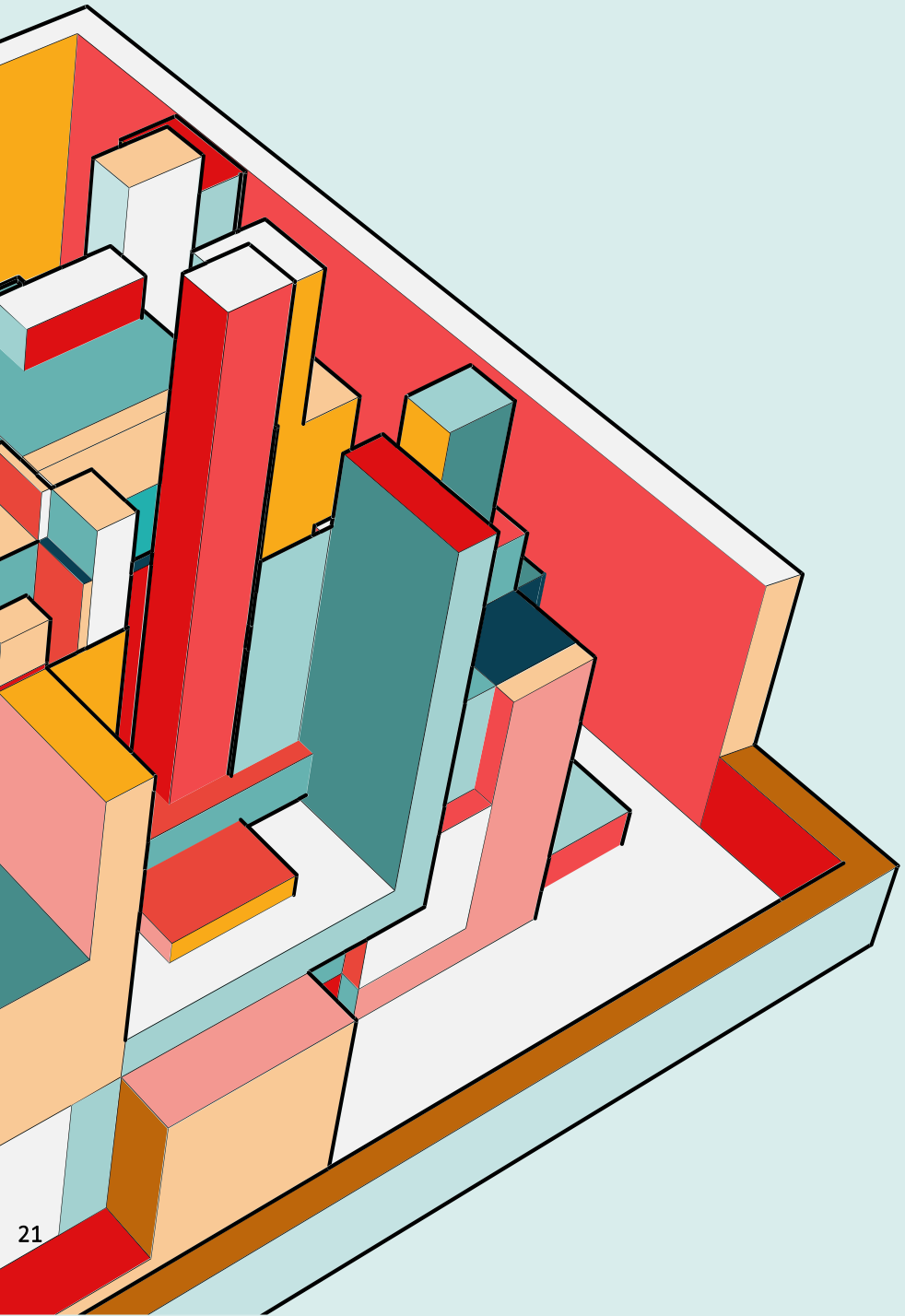
```
// inside pagesRepository
public string GetPage(string pageName)
{
    return _db.Pages.SingleOrDefault(pageName);
}
```

OPENED LAYER




Monolith
API
Domain model
Repository
Database schema
Monolith
API
Feature
Feature
Feature
Database schema
Service
Service
Service
API
API
API
Feature
Feature
Feature
Database schema
Database schema
Database schema
Layers
Feature slices
Microservices

2000-**e**



DOMAIN DRIVEN DESIGN

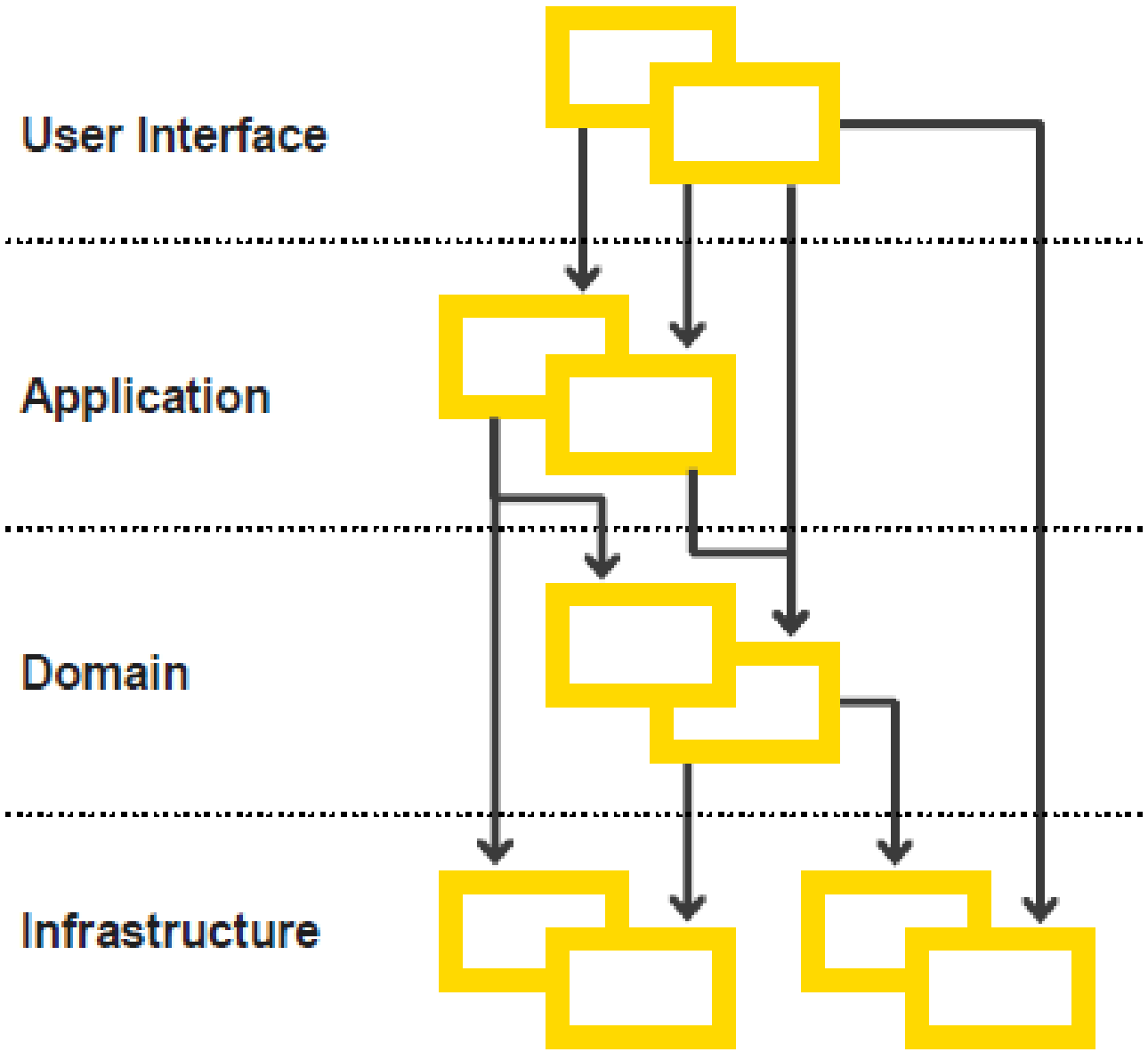
2003 год

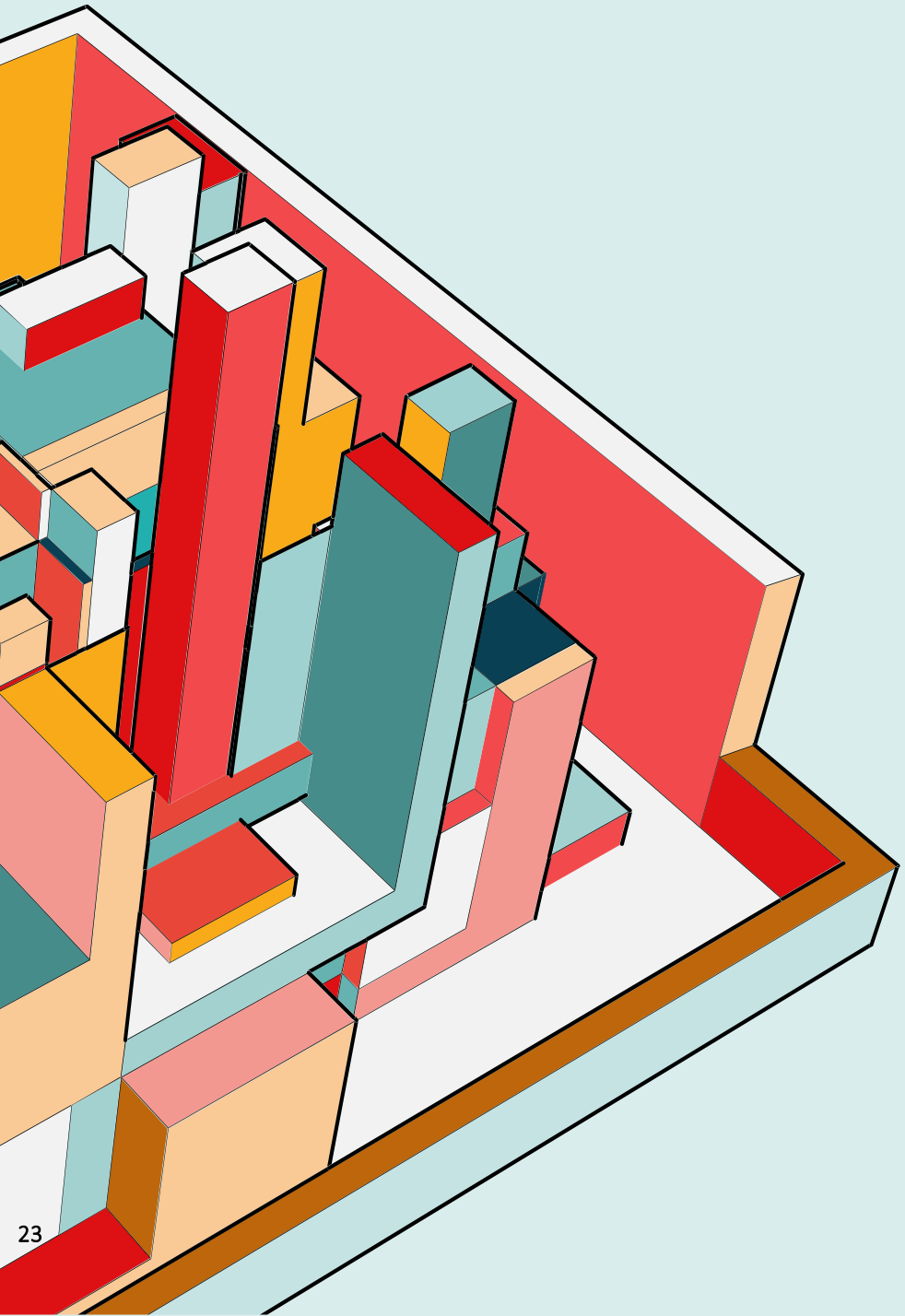
DDD

База Данных – не главное,
а лишь инфраструктура

Больше слоев и разделение
на use cases

Bounded Contexts, Shared
Kernel, Anti Corruption Layer

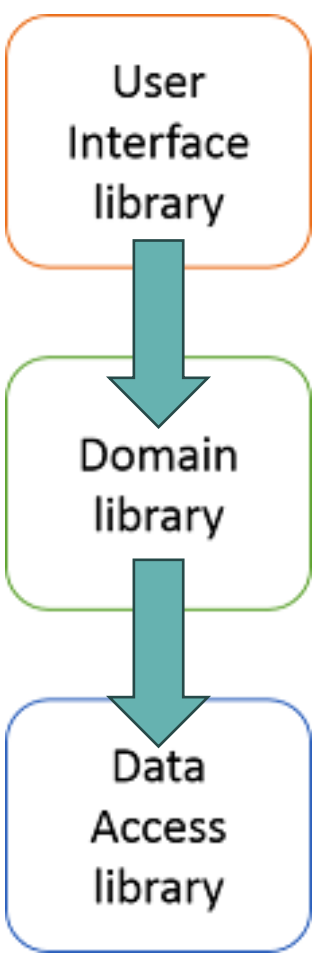


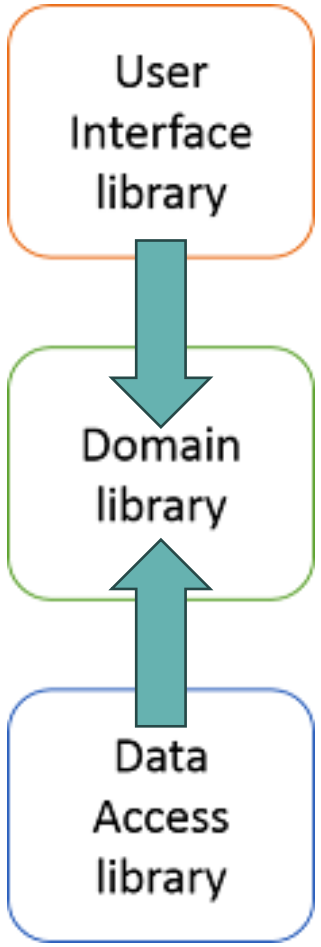


PORTS AND ADAPTERS

2005 год

ПРИНЦИП ИНВЕРСИИ ЗАВИСИМОСТЕЙ





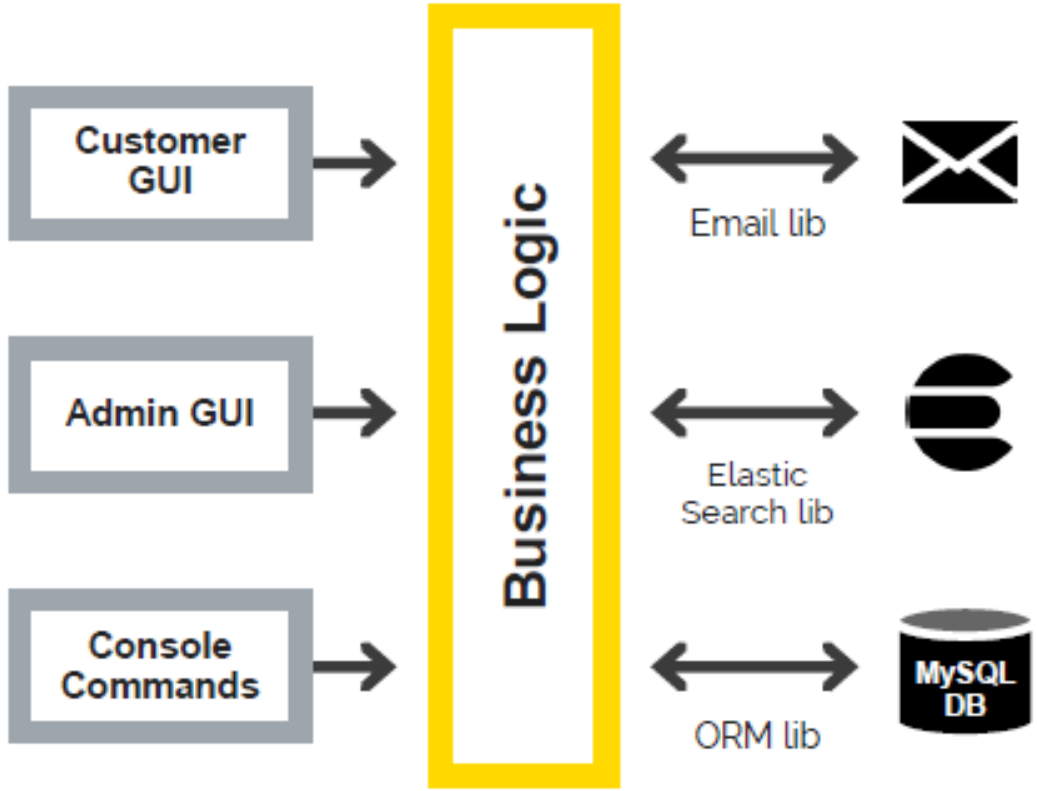
PORTS AND ADAPTERS

Domain это ключевое ядро системы в полной изоляции от технических деталей

Последовательность вызова вместо иерархии

Разворот зависимости
Data Access --> Domain

Работа с Presentation (Infrastructure) через порты



ОСНОВНАЯ ИДЕЯ

Основная логика (Core):

Здесь сосредоточены все "мозги" приложения — бизнес-логика, правила и процессы.

Порты (Ports):

Это интерфейсы, которые описывают, что нужно основной логике.

Например, "сохранить данные книги" или "получить данные о пользователе".

Но порты не говорят, как именно это делается.

Адаптеры (Adapters):

Это "переходники", которые реализуют порты. Они связывают основную логику с конкретными технологиями, например:

- Работа с базой данных.
- Отправка HTTP-запросов к внешнему API.
- Обработка запросов от веб-клиента.

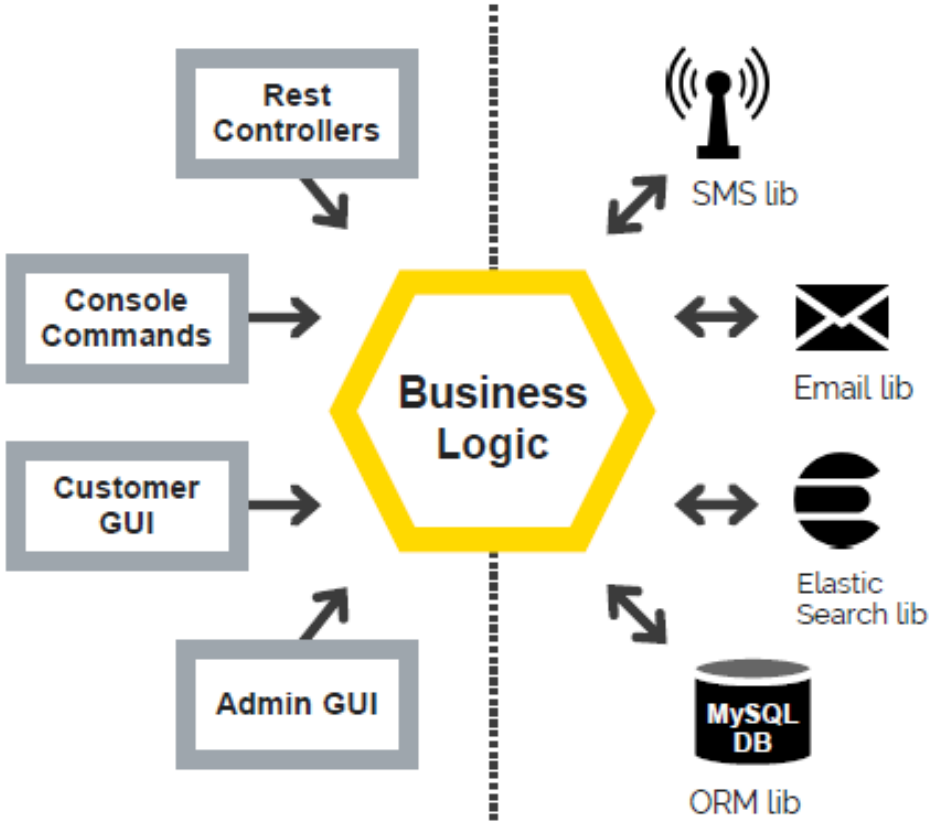
PORTS AND ADAPTERS

Domain это ключевое ядро системы в полной изоляции от технических деталей

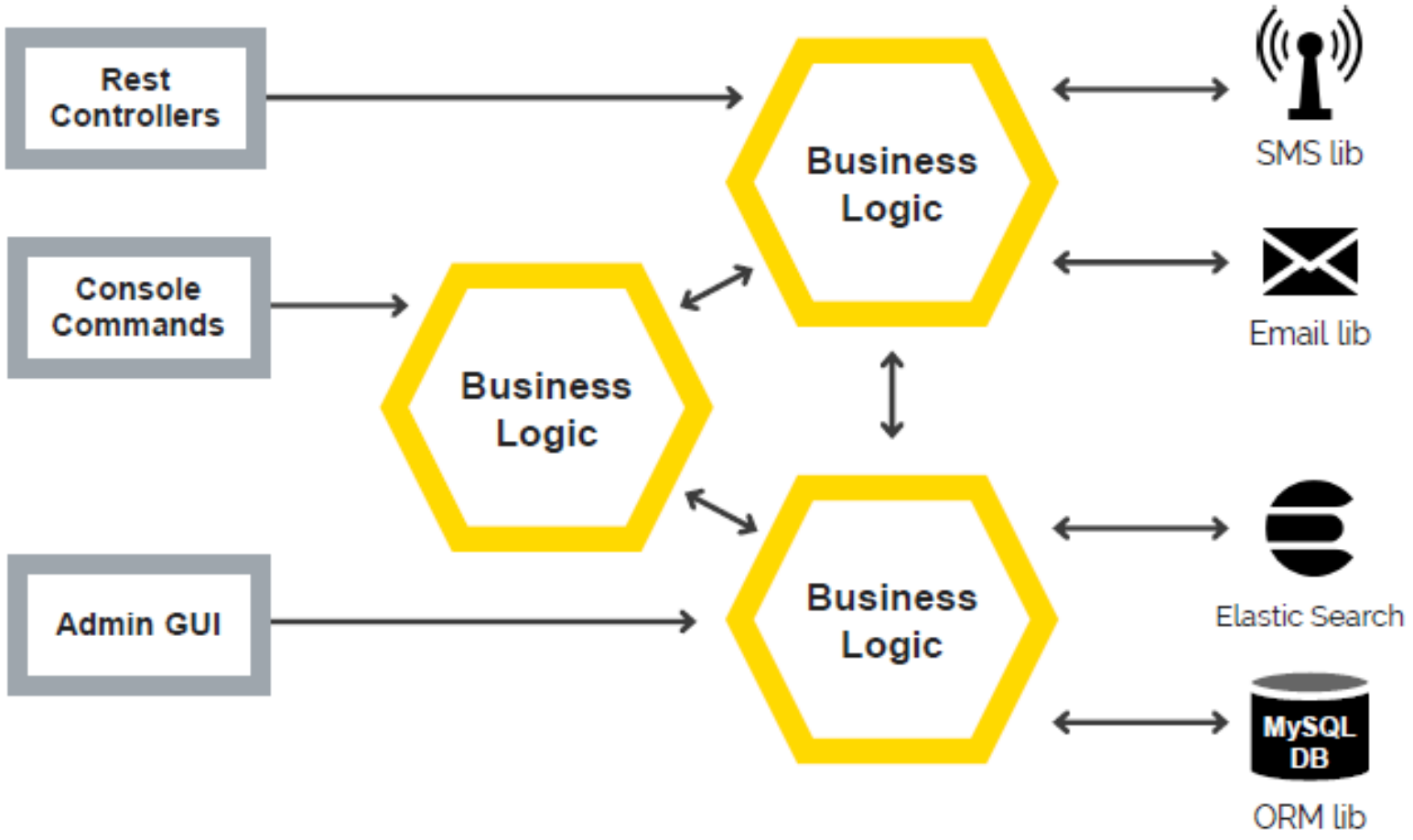
Последовательность вызова вместо иерархии

Разворот зависимости
Data Access --> Domain

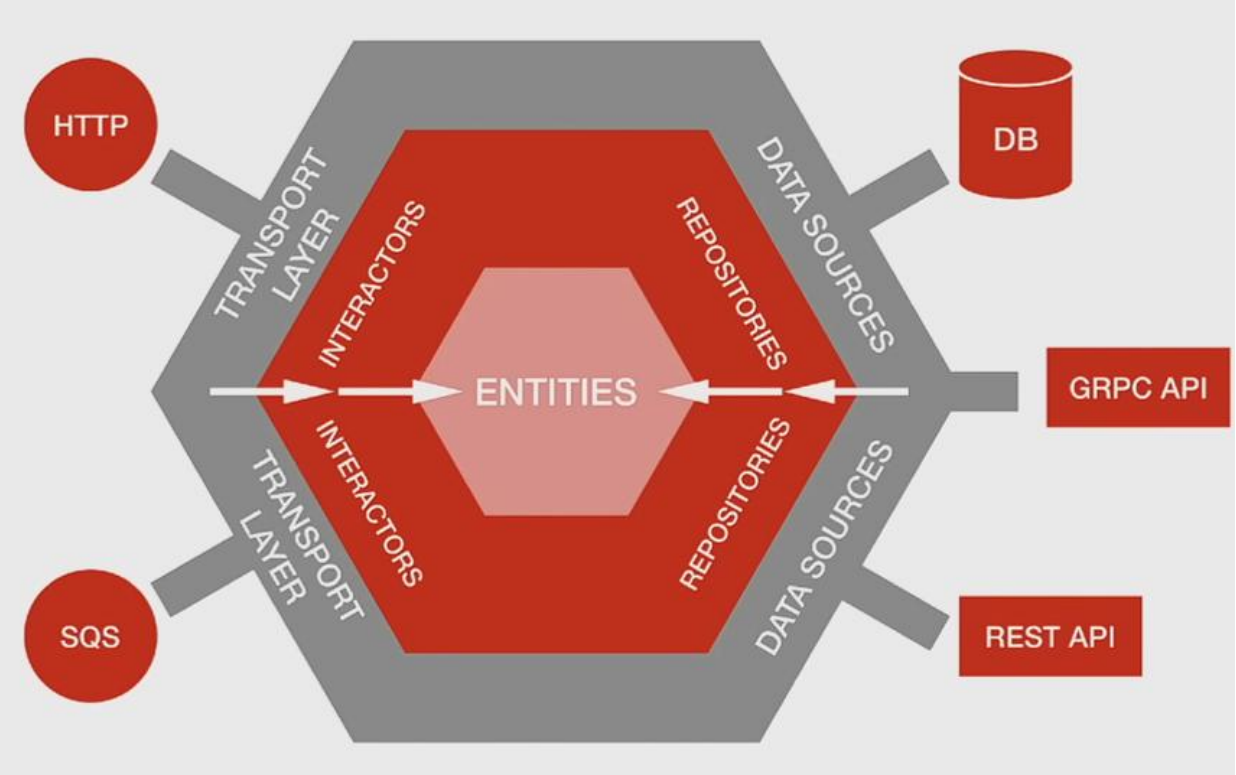
Работа с Presentation (Infrastructure) через порты

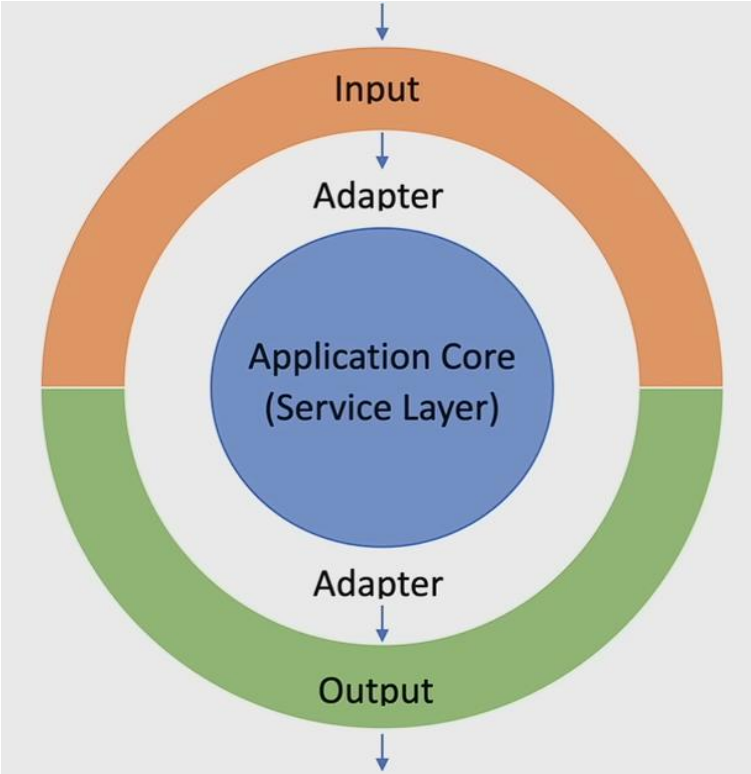


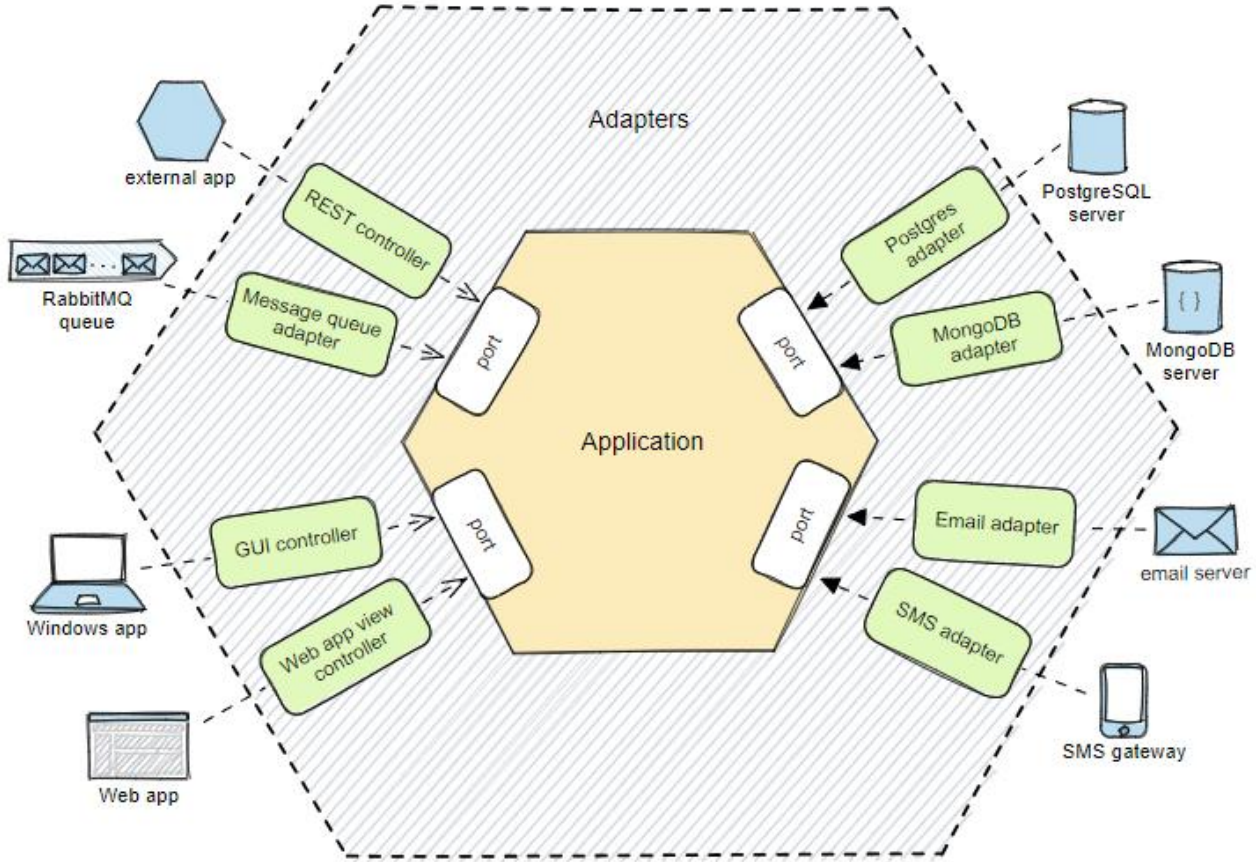
ЭТО ТОЖЕ
PORTS AND ADAPTERS

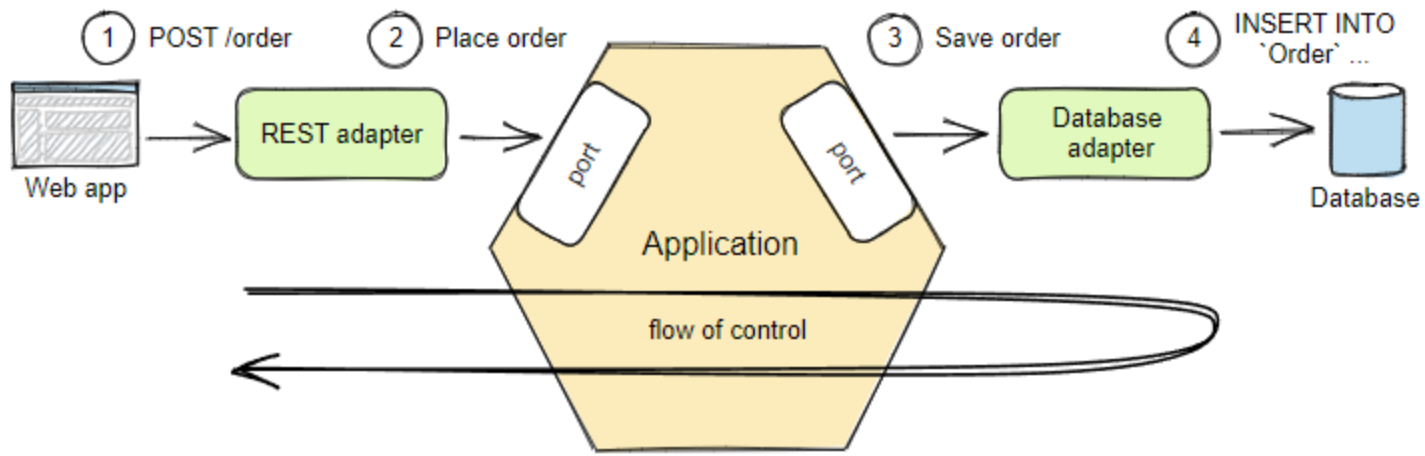


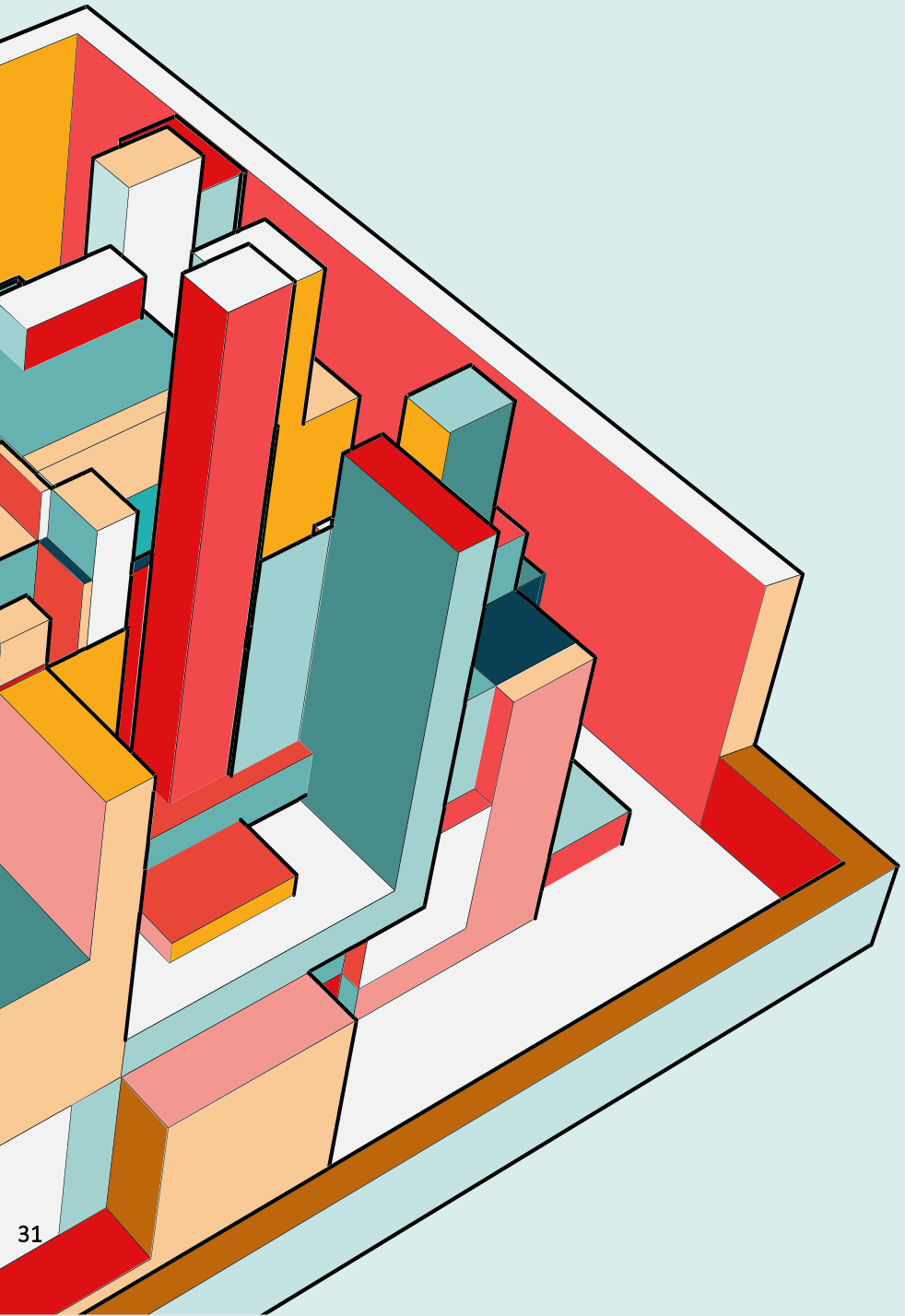
СОВРЕМЕННАЯ ИНТЕРПРЕТАЦИЯ ОТ NETFLIX






Adapters
external app
REST controller
RabbitMQ queue
Message queue adapter
port
Application
port
Postgres adapter
PostgreSQL server
MongoDB adapter
MongoDB server
port
GUI controller
Windows app
port
Email adapter
email server
Web app view controller
SMS adapter
Web app
SMS gateway


1 POST /order
2 Place order
3 Save order
4 INSERT INTO `Order` ...
Web app
REST adapter
port
port
Application
Database adapter
Database
flow of control



ONION ARCHITECTURE

2008 год

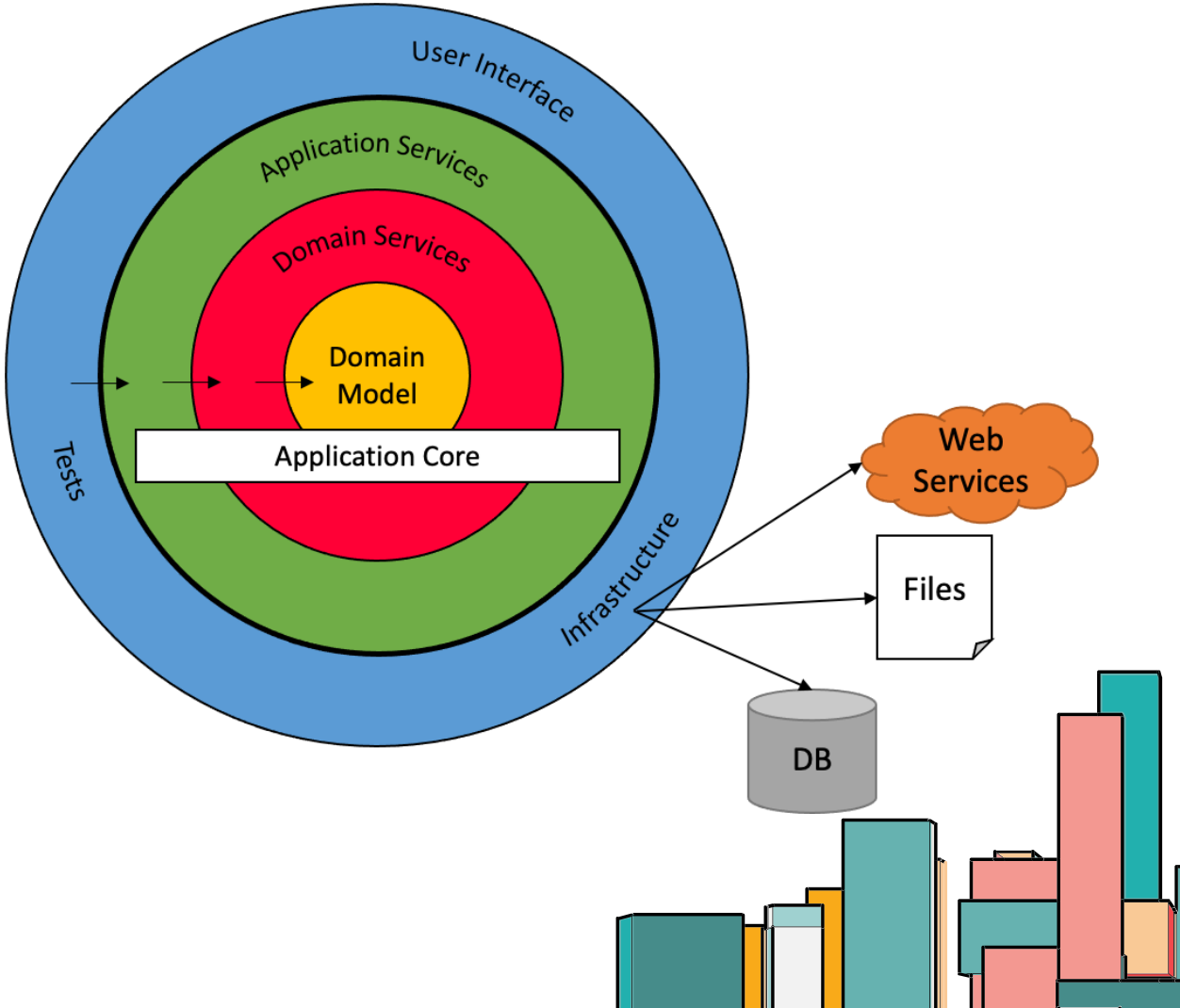
ONION ARCHITECTURE

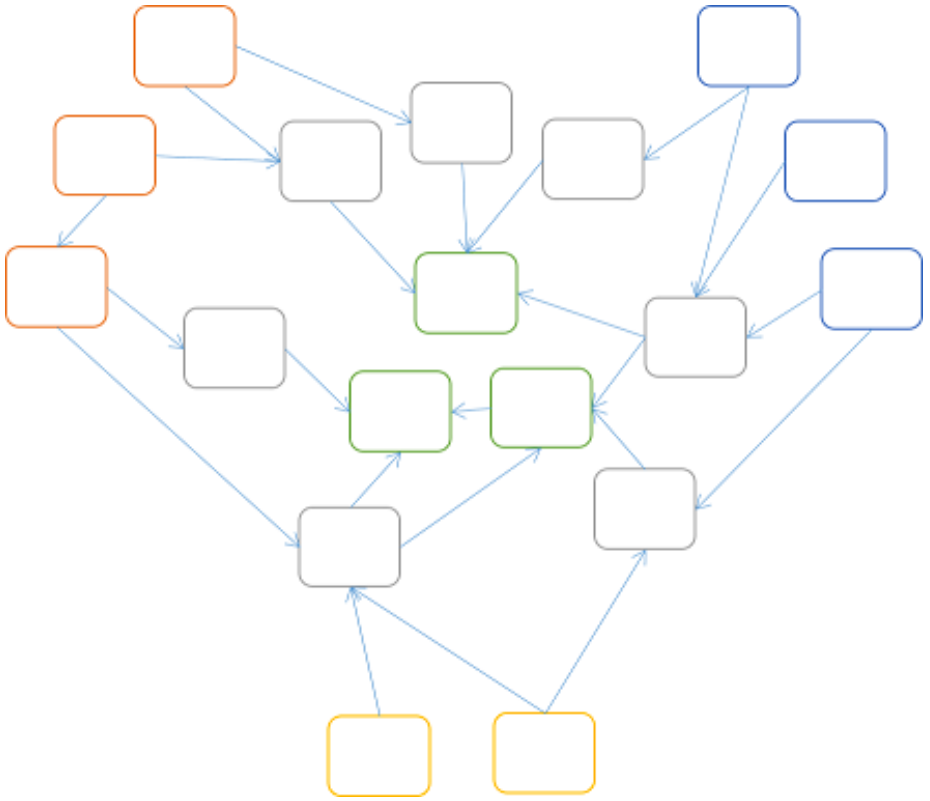
Деление ядра системы на три отдельных слоя

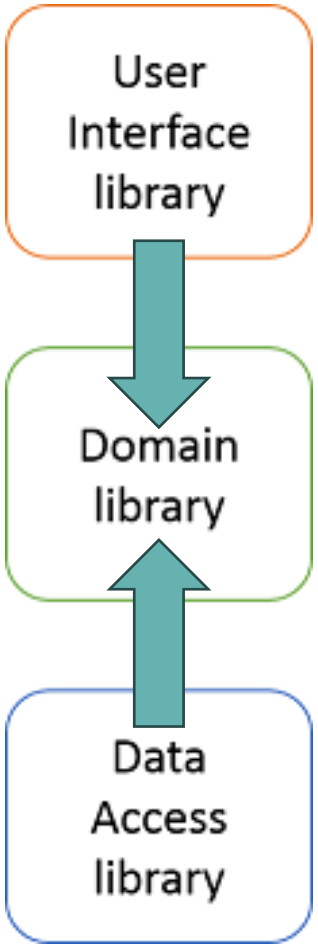
Зависимости только внутрь

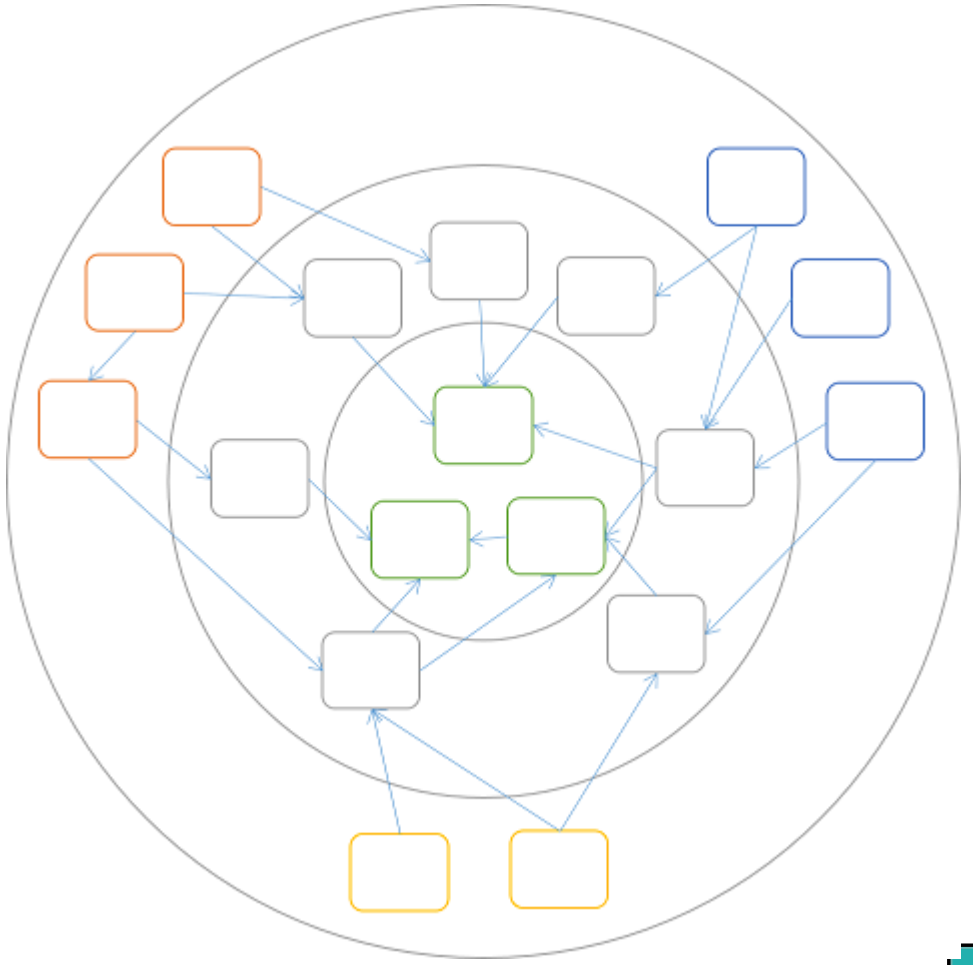
Dependency Inversion Principle – архитектурный принцип

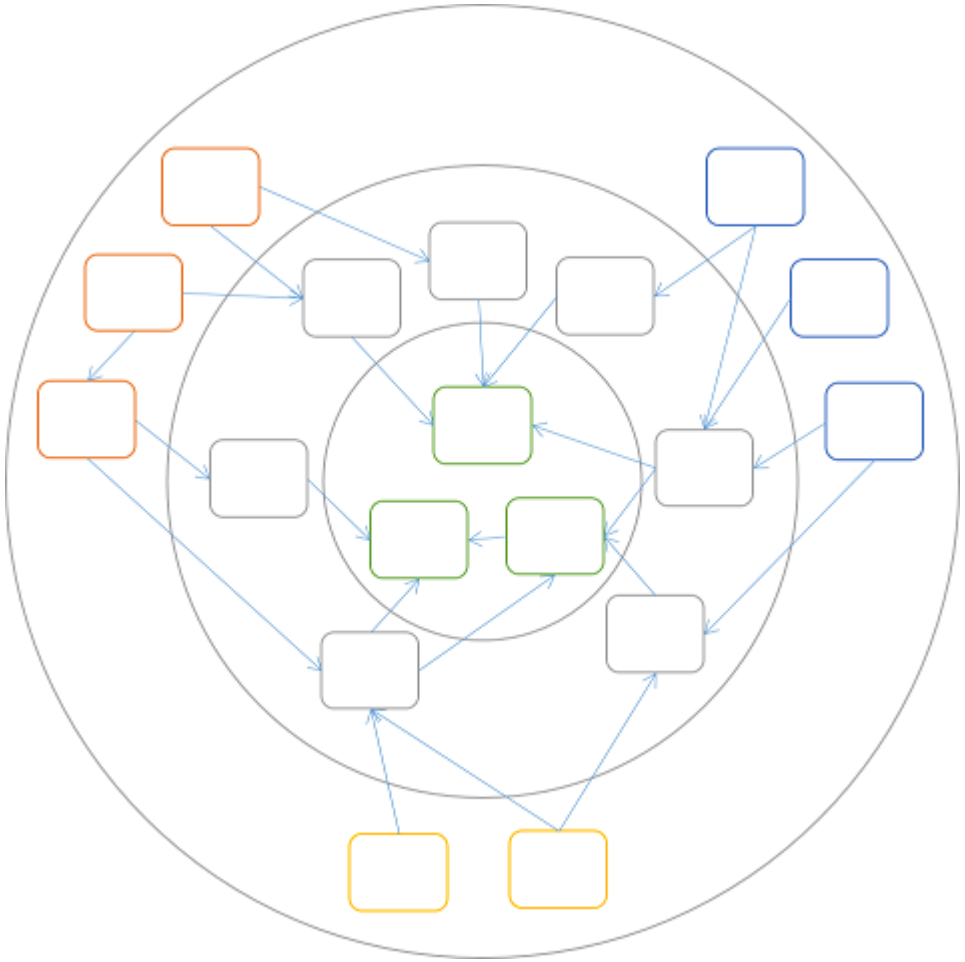
Бизнес логика работоспособна даже без внешних слоев

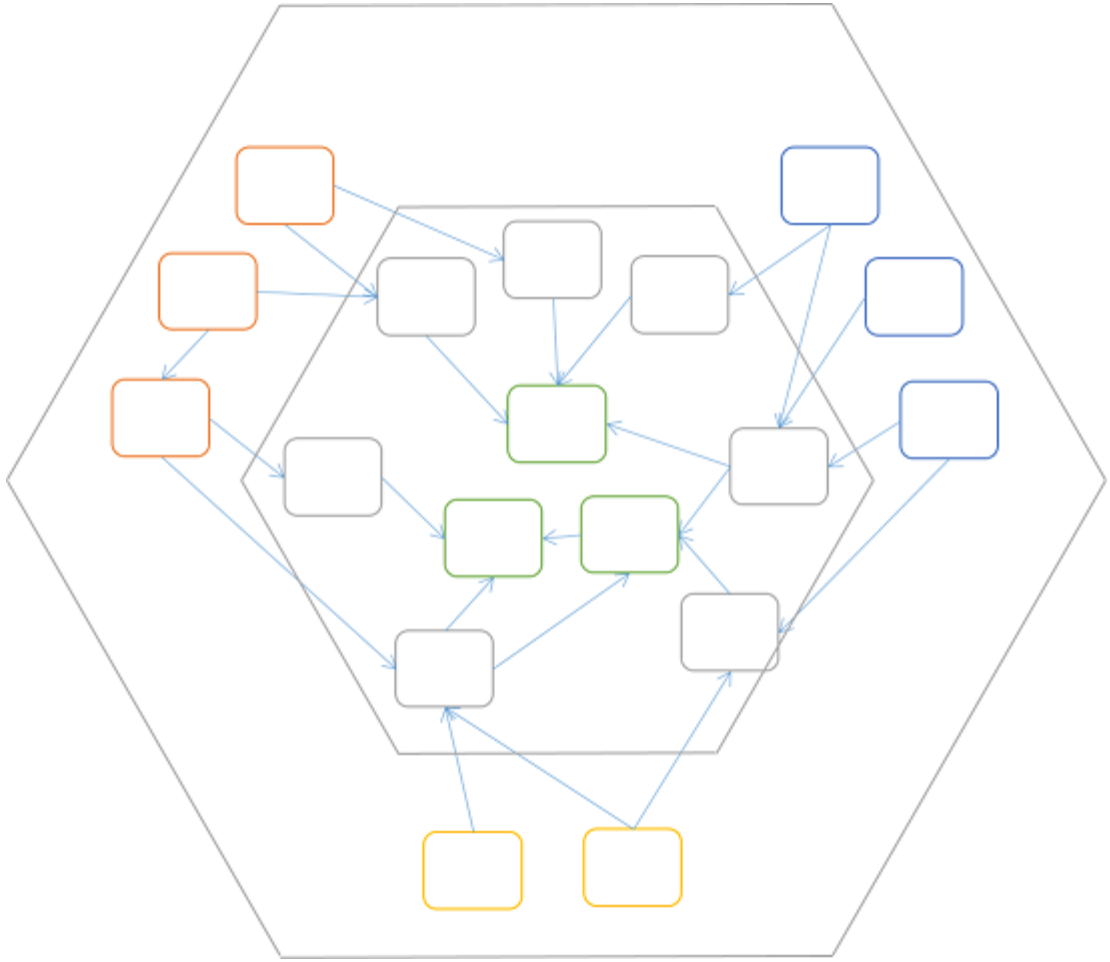


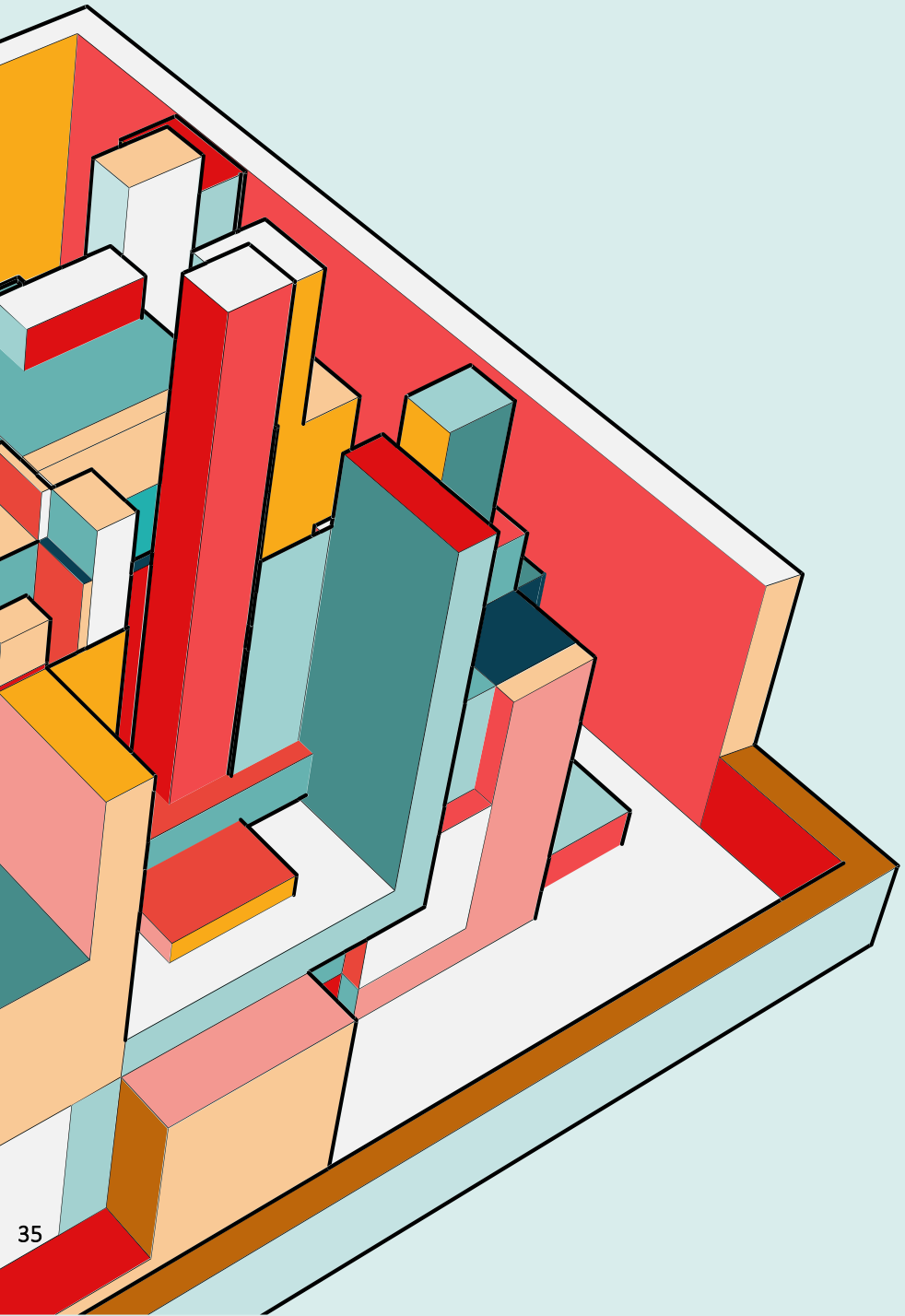



User Interface library
Domain library
Data Access library









CLEAN ARCHITECTURE

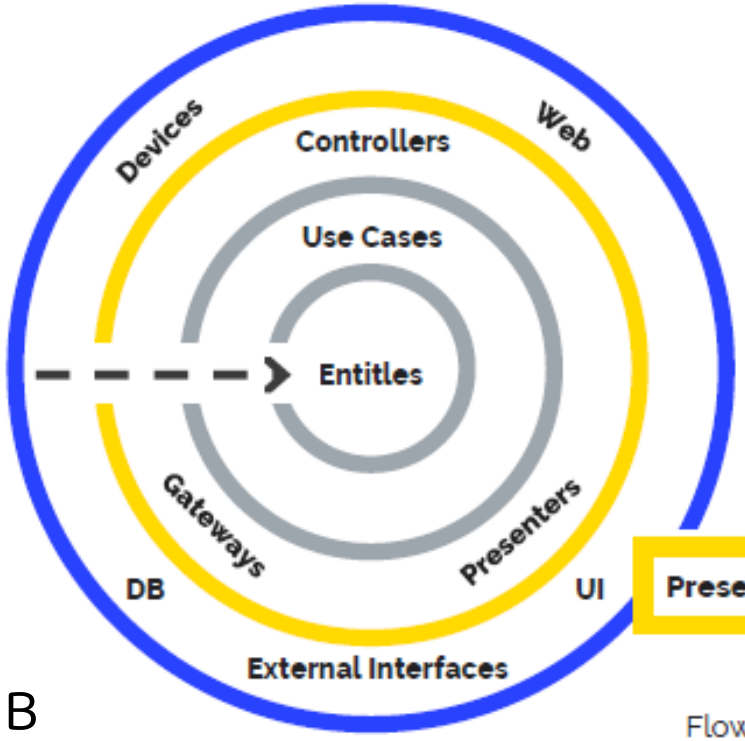
2012 год

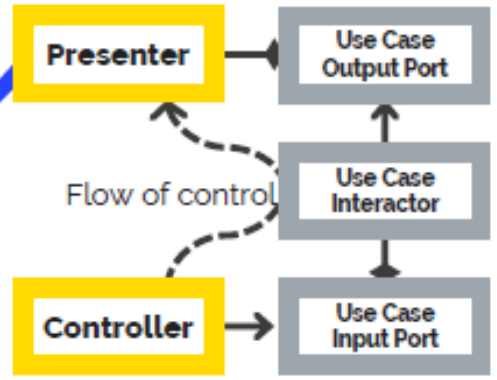
CLEAN ARCHITECTURE

Деление и «горизонтально» и «вертикально» (SRP)

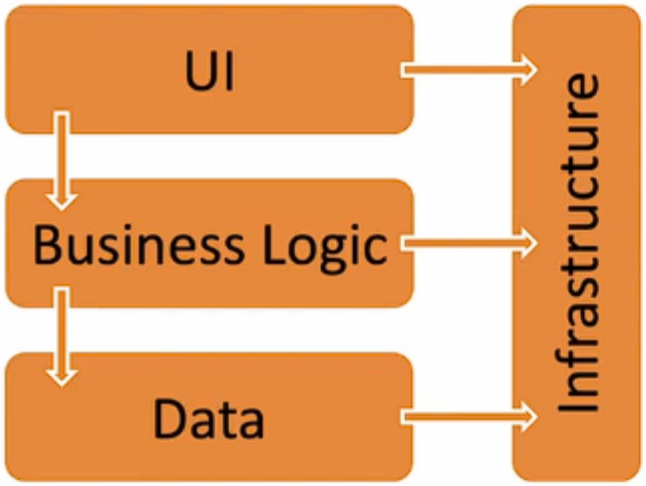
Направление зависимостей по уровню компонента

Закрепление базовых принципов (DI, SRP, OCP)

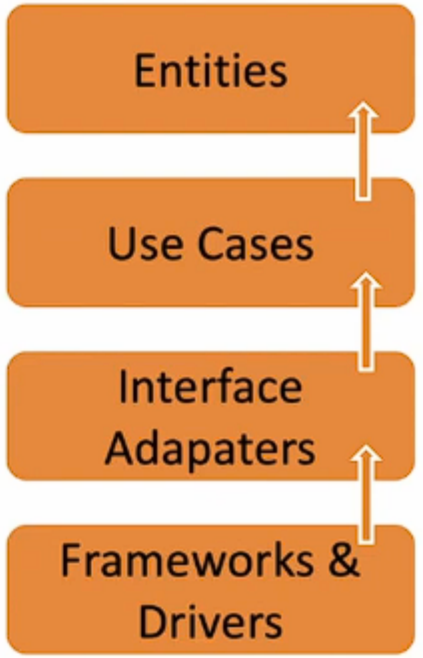




МНОГОСЛОЙНАЯ ИЛИ ЧИСТАЯ



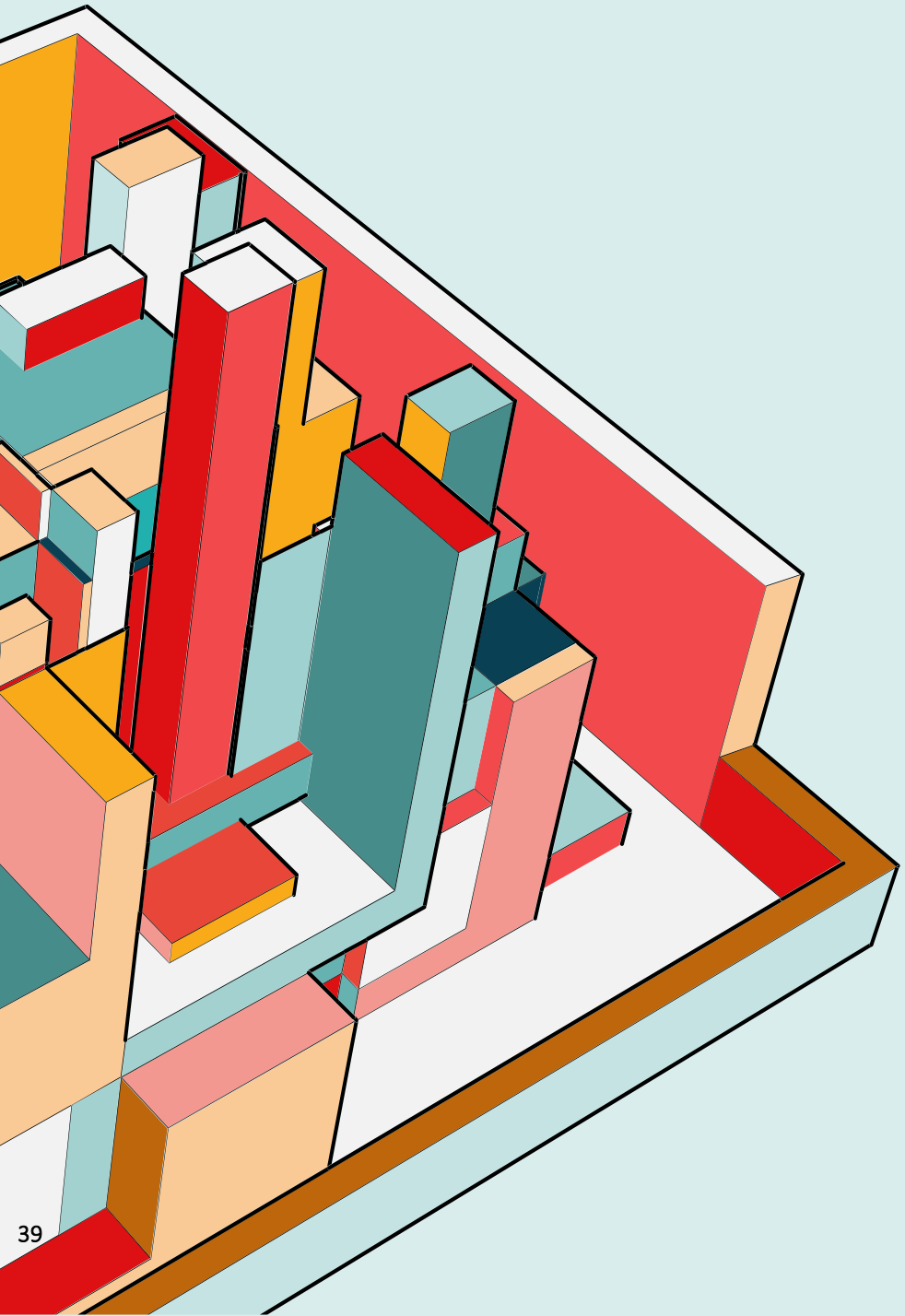
N-tier Architecture



Clean Architecture

КАК ЖИТЬ ДАЛЬШЕ?

1. Понять сложность своего проекта
2. Взять соразмерную архитектуру
3. Выбросить все лишнее = YAGNI



MV ?

паттерны презентационного слоя

UI БЕЗ ПАТТЕРНОВ

Усложнение поддержки

Изменения в UI, логике или данных, скорее всего, повлекут за собой изменения в остальных частях. Поэтому вносить правки гораздо сложнее, что затрудняет поддержку.

Ухудшение тестируемости

Логика и данные приложения могут быть написаны таким образом, чтобы каждый компонент мог быть протестирован отдельно.

Уменьшение возможности переиспользования

Если ваш UI-код смешан с кодом логики и данных, то его становится гораздо сложнее переиспользовать.

ЦЕЛИ MV*-ПАТТЕРНОВ

Отделить UI-код (View)

от кода логики (Presenter, Controller, ViewModel и т. д.)

и кода обработки данных (Model).

Это позволяет каждому из них развиваться самостоятельно.

Например, вы сможете изменить внешний вид и стиль приложения, не затрагивая логику и данные.

Одним из самых первых паттернов для отделения представления от логики и модели стал Model-View-Controller (MVC).

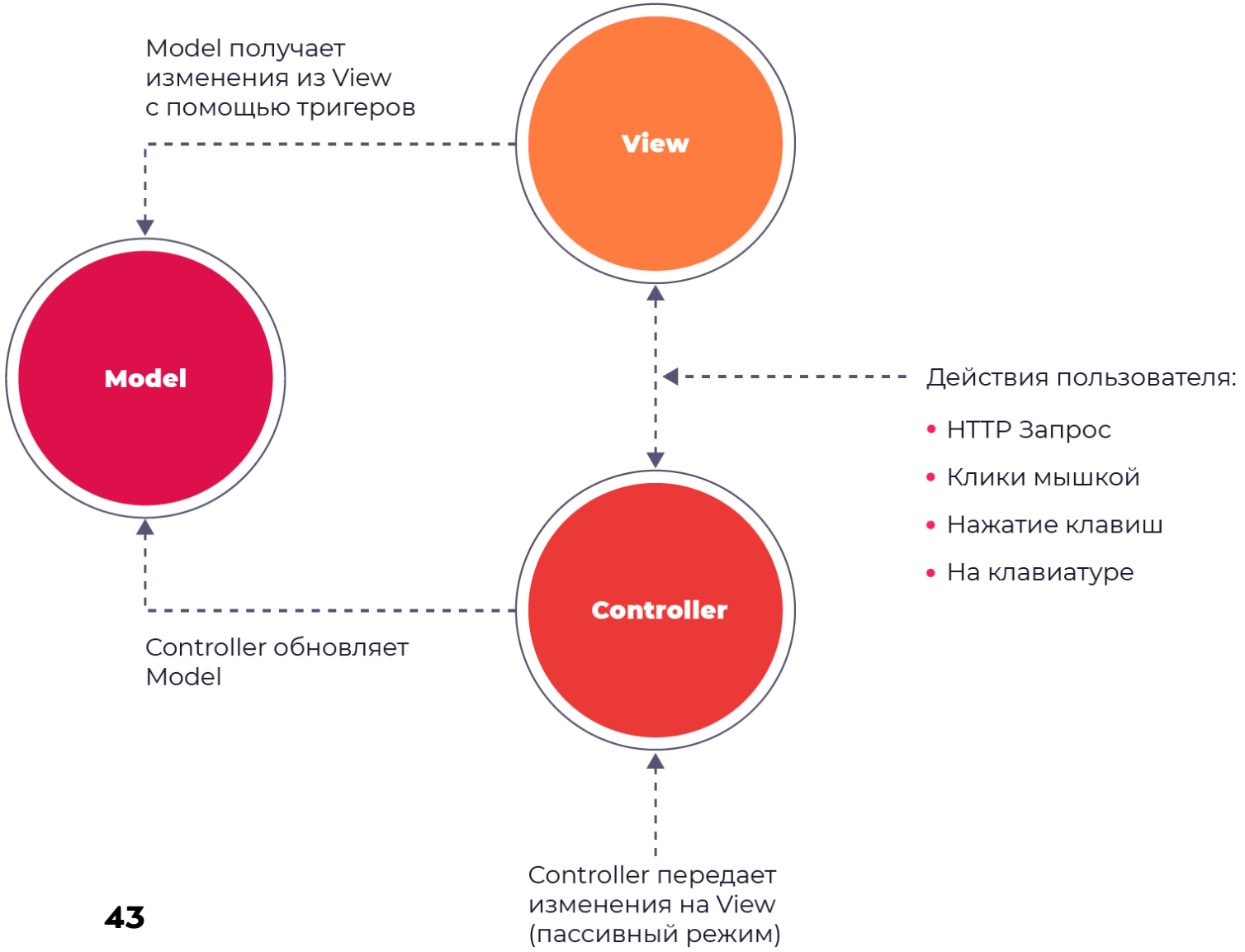
Эта концепция была описана Трюгве Реенскаугом в 1979 году!

MODEL И VIEW

- **View** — это визуальный интерфейс (UI). Он может состоять как из отдельных элементов, так и из виджетов.

- **Model** — это данные приложения, которые отображаются с помощью интерфейса, а также процесс их получения и сохранения.

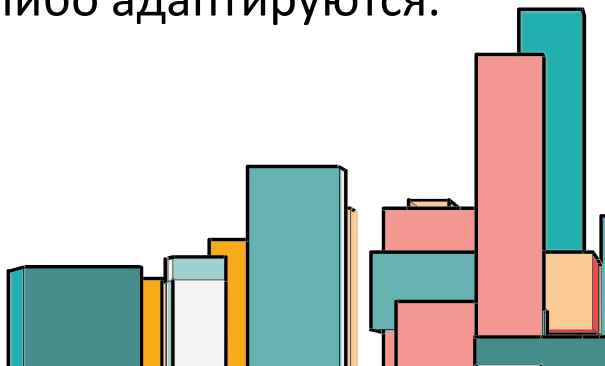
MODEL-VIEW-CONTROLLER



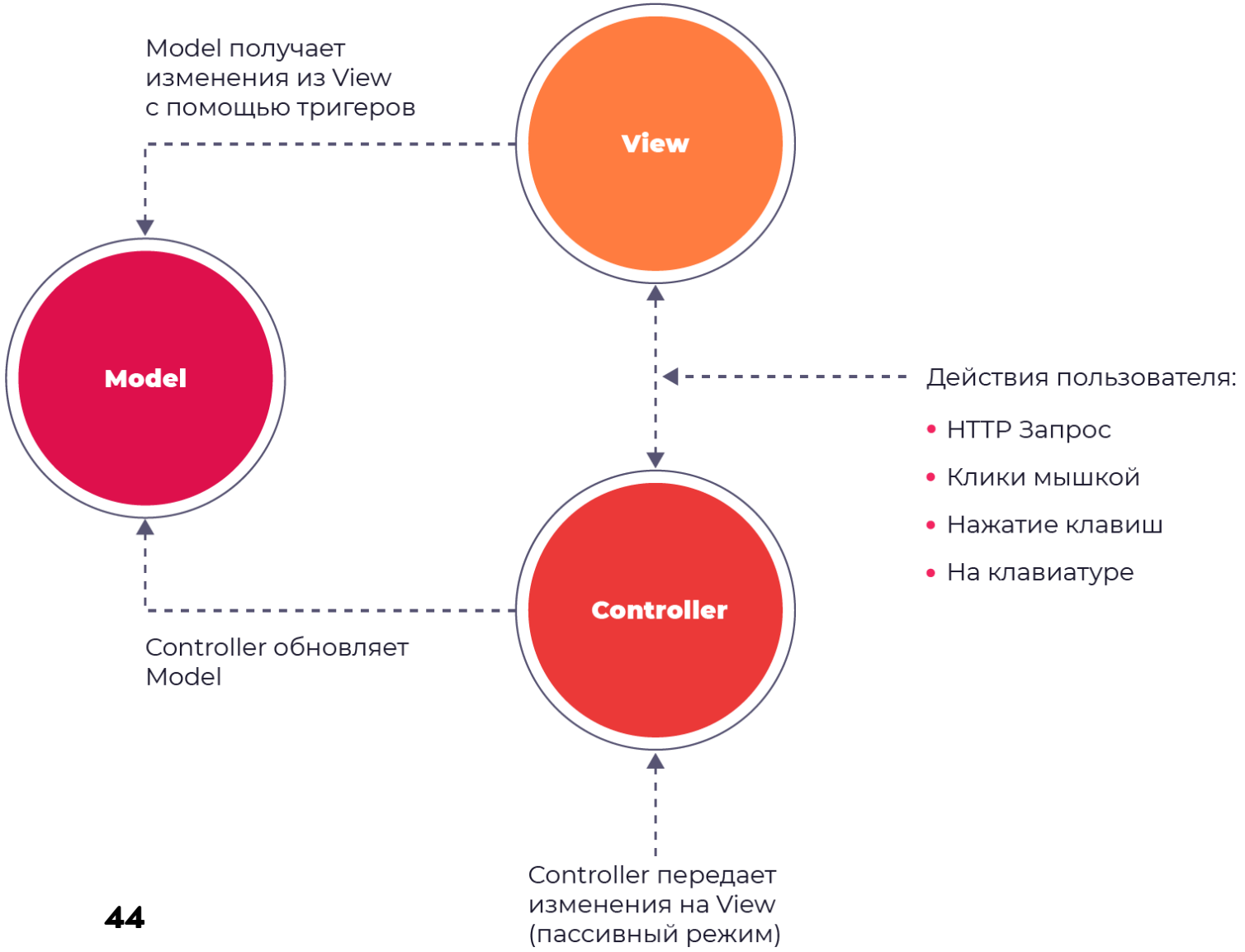
Model — данные приложения с логикой их получения и сохранения.

Чаще всего модель оперирует данными из базы данных или результатами работы веб-сервисов.

Данные либо сразу выводятся на экран, либо адаптируются.



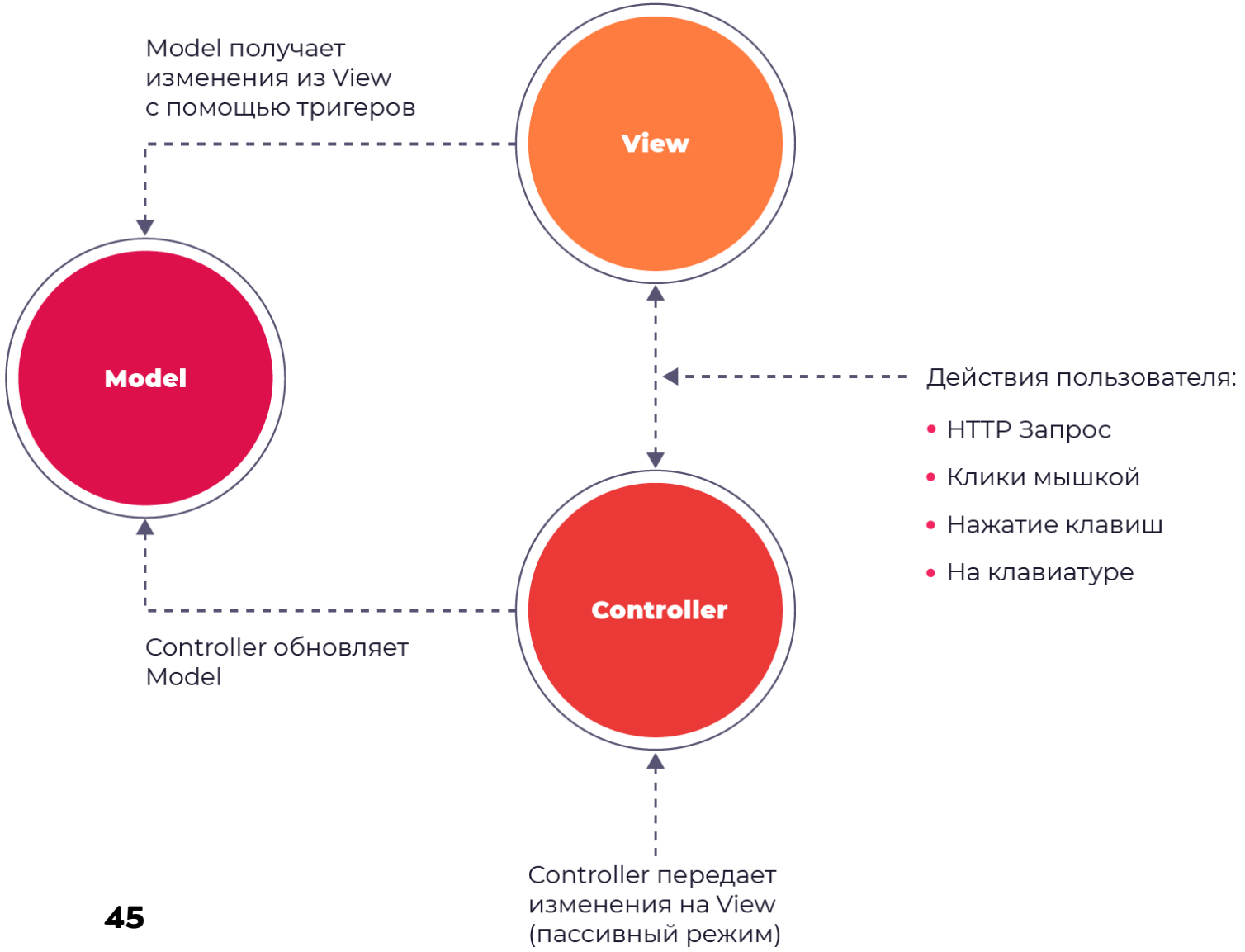
MODEL-VIEW-CONTROLLER



View — визуальный интерфейс, отрисовка кнопок, надписей, полей ввода и других элементов форм.

Может следить за Model и отображать данные из неё.

MODEL-VIEW-CONTROLLER



Controller — следит за действиями пользователя и решает, что с ними делать.

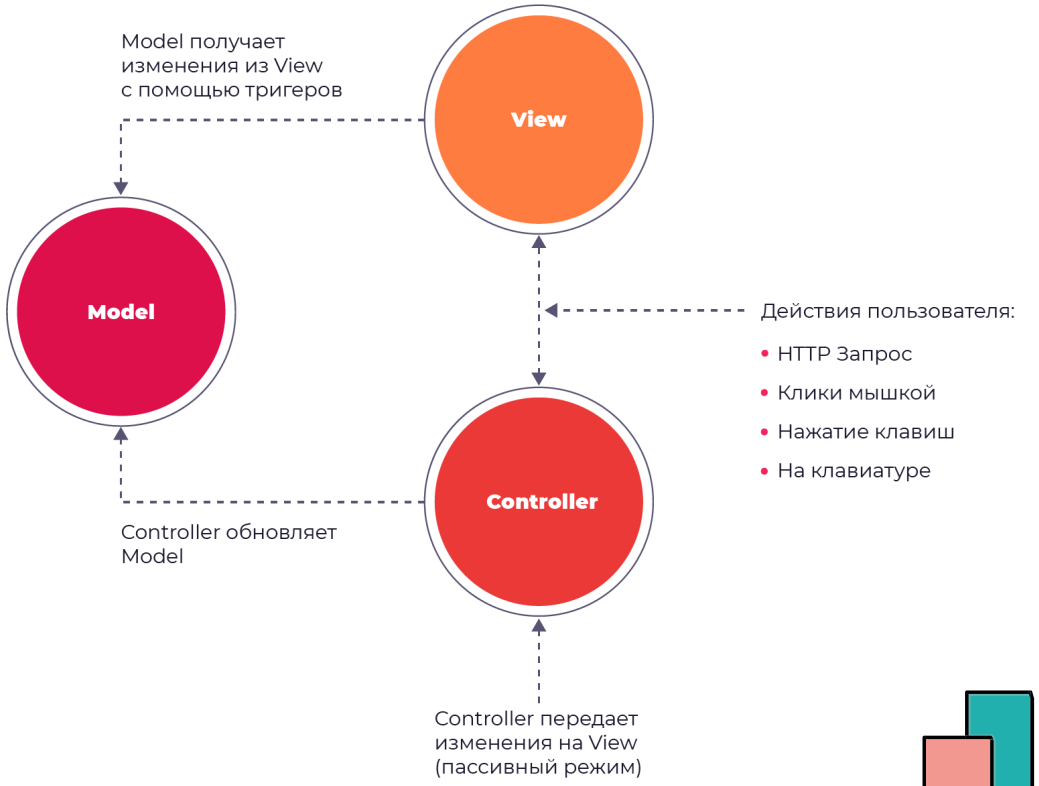
Обновляет Model и View.

ПРИНЦИП РАБОТЫ ПАТТЕРНА MVC

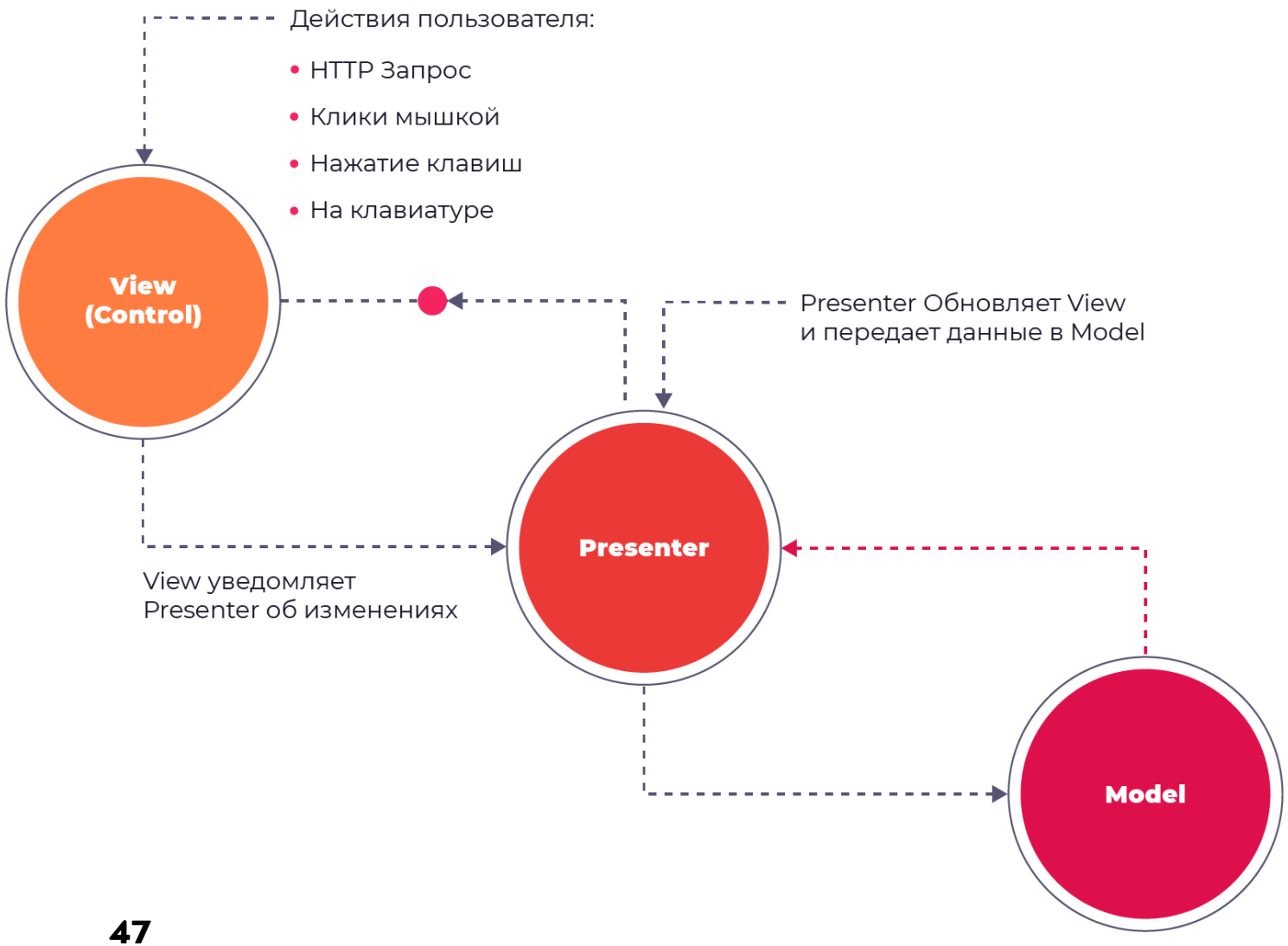
Controller обрабатывает действия пользователя — клики мышкой, нажатие клавиш клавиатуры или входящие http-запросы.

Обработанные изменения Controller передаёт в Model и отрисовывает на View (пассивный режим), или в модель попадают изменения напрямую из View (активный режим).

Главная задача View — отобразить данные из Model с помощью Controller.



MODEL-VIEW-PRESENTER



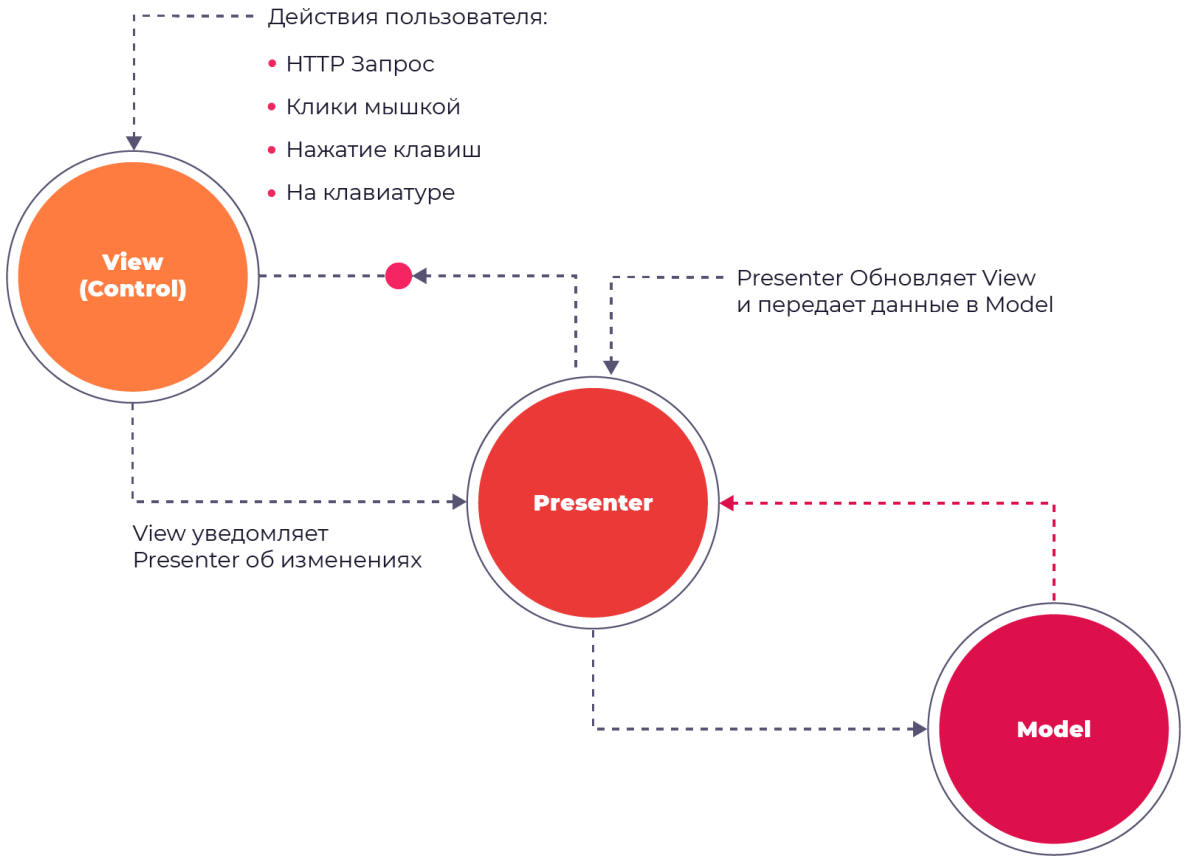
Развитие виджетов упразднило отрисовку отдельных элементов View, таким образом, и отдельный класс Controller стал не нужен.

Но отделить логику приложения от данных всё равно необходимо.

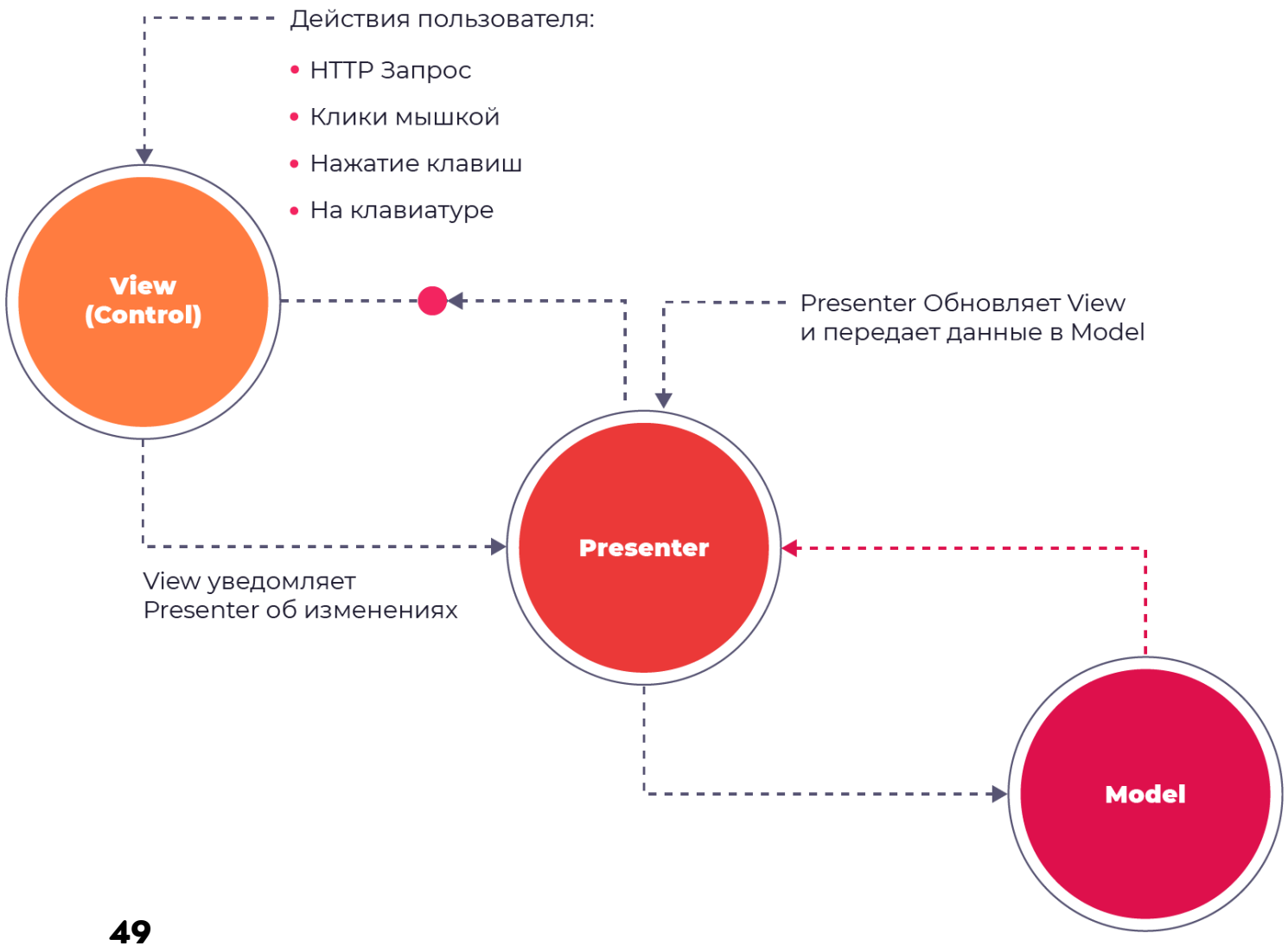
ВМЕСТО CONTROLLER ПОЯВИЛСЯ PRESENTER И ВСЁ?

Если сравнивать с MVC,
то **Model не изменилась**.

View теперь сам обрабатывает действия пользователей (с помощью виджетов, например), а **если это действие что-то меняет** в логике интерфейса, то оно **передаётся в Presenter**.



MODEL-VIEW-PRESENTER

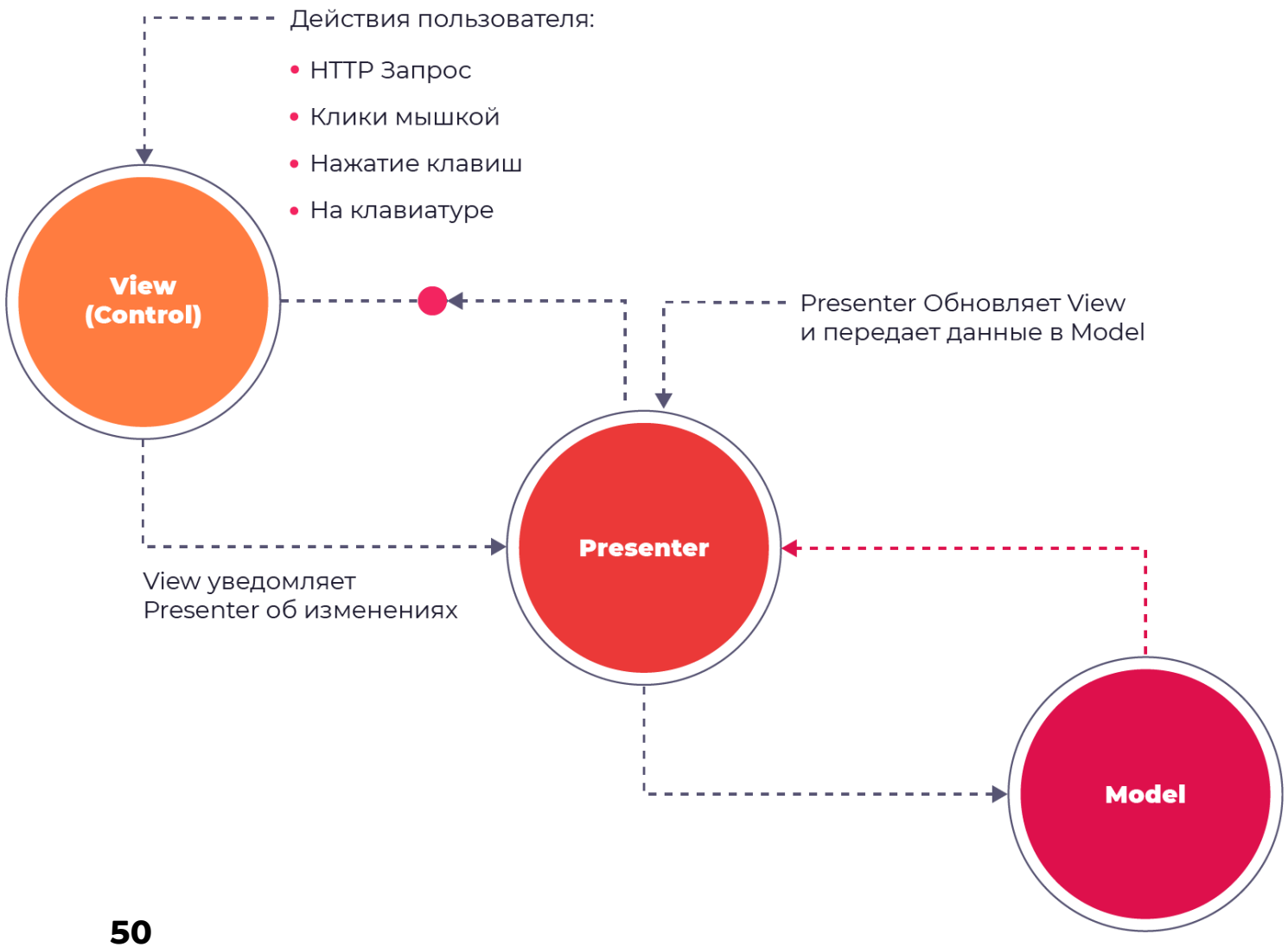


Presenter как дирижёр — отвечает за синхронную работу Model и View.

Если он получает уведомление от View о совершённом пользователем действии, то обновляет модель и синхронизирует изменения с View.

Всё общение происходит через интерфейс, что и даёт их разделение.

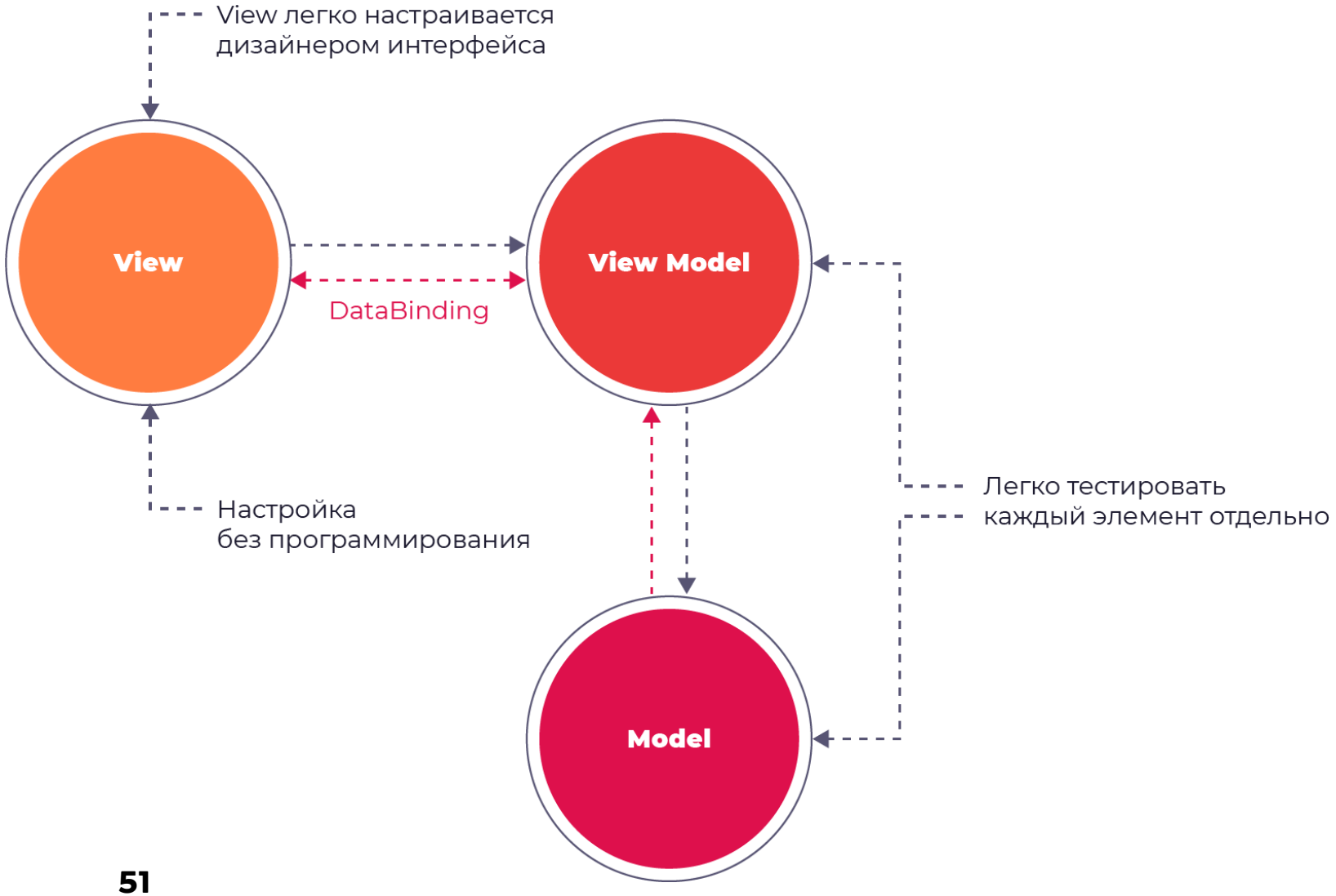
MODEL-VIEW-PRESENTER



Есть две реализации:

- **Passive View**, где View ничего не знает о Model, а за получение информации из Model и обновление View отвечает Presenter
- **Supervising Controller**, где View знает о Model и сам связывает данные с отображением

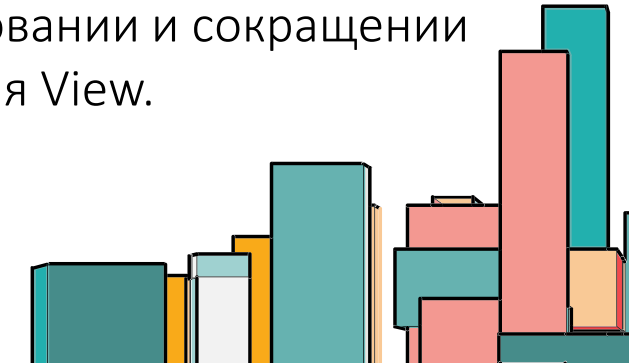
MODEL-VIEW-VIEWMODEL



Здесь нет прямого общения между ViewModel и View, оно происходит посредством команд.

ViewModel — совмещение Model и Controller

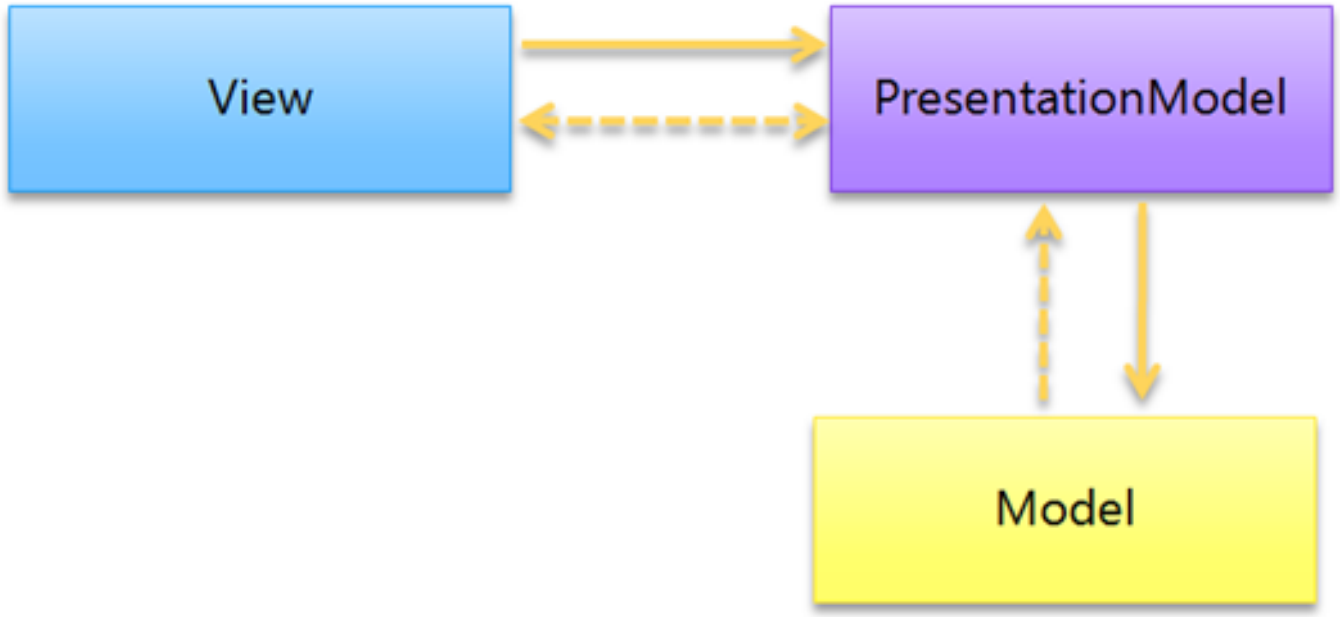
Главные преимущества MVVM в лёгком проектировании интерфейсов, независимом тестировании и сокращении кода для View.



ОСНОВНЫЕ ВЫВОДЫ

1. Модель во всех паттернах выглядит одинаково и имеет одну и ту же цель – получение, обработка, а также сохранение данных.
2. В классическом MVC пользовательский ввод обрабатывает Controller, а не View.
3. Современные ОС и библиотеки виджетов берут на себя обработку пользовательского ввода, поэтому больше нет нужды в контроллере из MVC.
4. Цель MV*-паттернов: отделить друг от друга отображение UI, логику интерфейса и данные (их получение и обработку).
5. Используя MV*-паттерн в своем приложении, вы упрощаете его поддержку и тестирование, отделяете данные от способа их визуализации.
6. MVP достаточно универсальный паттерн и подойдет во многих случаях (это мое личное мнение). Какой вариант использовать: Passive View или Supervising Controller – решать вам. Руководствуйтесь тем, что вам нужно: больше контроля и тестируемости либо лаконичности и краткости кода. Лавируйте между задачами и применяйте тот или другой подход.
7. Если в системе присутствует хорошая реализация автоматического связывания данных (databinding), то MVVM – это ваш выбор.

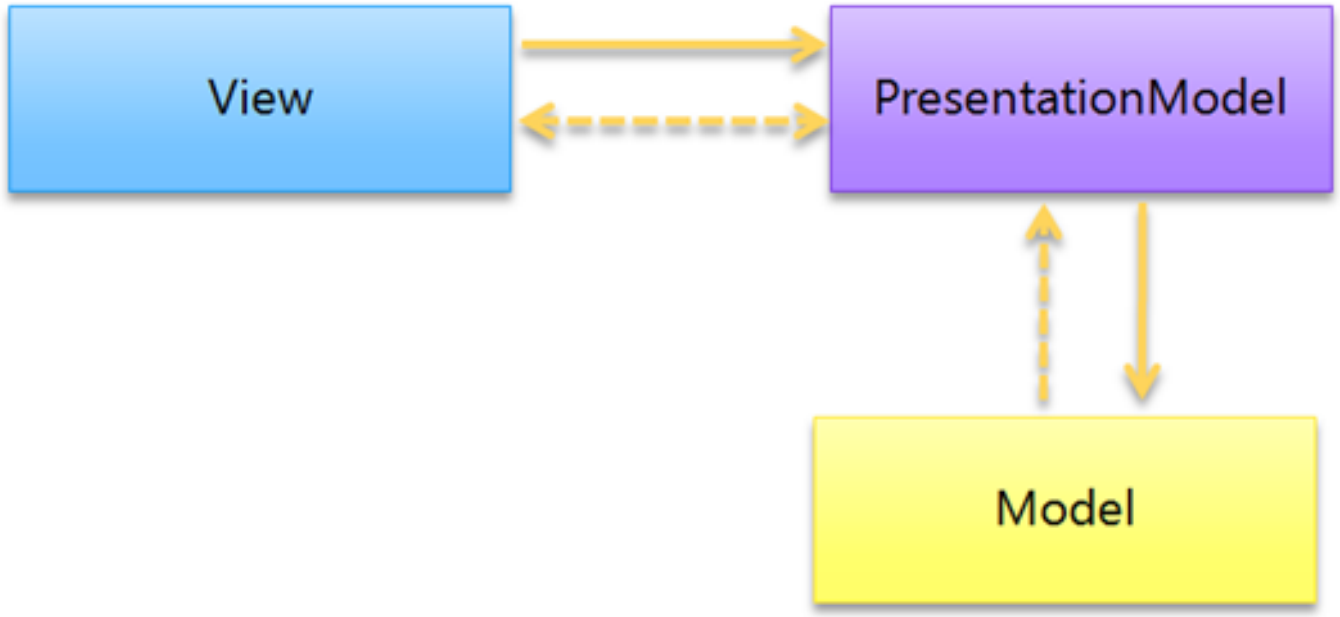
PRESENTATION MODEL
ХОРОШАЯ АЛЬТЕРНАТИВА MVVM



Полезна там, где нет автоматического связывания.

Но придется писать код связывания самостоятельно

PRESENTATION MODEL
ХОРОШАЯ АЛЬТЕРНАТИВА MVVM



PresentationModel:

1. Содержит логику пользовательского интерфейса:

2. Предоставляет данные из модели для отображения на экране

3. Хранит состояние пользовательского интерфейса

ОПТИМИЗАЦИЯ И ПРОИЗВОДИТЕЛЬНОСТЬ

1. **View выносим на устройство клиента** (ноутбук, ПК, смартфон). За ускорение отвечает технология одностраничного приложения SPA. Сложные расчёты выполняются на конечном оборудовании пользователя, тем самым снижая нагрузку на backend-сервера.
2. **Controller** (Presenter, ViewModel) **выносятся на отдельный backend-сервер**, который обрабатывает логику.
3. **Model** — самый объёмный и производительный элемент, поэтому для него **необходим сервер с быстрой системой хранения данных**, а лучше с возможностью кэширования «горячей» информации в оперативной памяти.

Дополнительное преимущество можно получить, разделив эти элементы по разным вычислительным единицам.

ЛИТЕРАТУРА

📕 Шпаргалка по MV-паттернам: https://habr.com/ru/articles/151219/

📕 Перевод хорошей обзорной статьи по MV*-паттернам: https://habr.com/ru/companies/mobileup/articles/313538/

📕 Статья которая дает понимание исторического становления MV* паттернов: https://simpleone.ru/blog/mv-patterns-in-web-application-development

📕 Главу "Слоистая архитектура" из книги Марка Ричардса «Паттерны архитектуры программного обеспечения».

https://systems.education/layered-architecture

📕 Перевод статьи Mark Seemann о популярных архитектурах разработки ПО и о том, что между ними общего https://habr.com/ru/articles/344164/

СПАСИБО!

Виденин Сергей

@videninserg