Разработка мобильных приложений

Лекция 6

Интерфейс пользователя

Ключев А.О. к.т.н., доцент ФПИиКТ Университета ИТМО
Санкт-Петербург

2025

Литература

- Книги
  - Роберт Мартин. Чистая архитектура
  - Yair Carreno. Building Modern Apps for Android
- Примеры исходных текстов к лекции -
  - https://github.com/kluchev/activity-example
- Полезные практические ресурсы
  - https://developer.android.com/guide
  - https://github.com/android/compose-samples

В моей практике было довольно мало проектирования UI

и у вас есть шанс превзойти мои скромные познания в сотни раз =)

- Сделать сложный и навороченный UI не так просто. В моем случае, я десятки раз делал простые и топорные интерфейсы для встроенных систем (там как правило был маленький ЖК дисплей и основная проблема была в отсутствии ресурсов микроконтроллера), а также разные технологические интерфейсы для инструментальных систем, в основном для Windows, в последнее годы - кроссплатформенные.
- Простой UI можно делать как угодно, так как в трех соснах трудно заблудиться.
- Из вышесказанного проистекает и стиль данной лекции, в ней я не могу похвастаться своими озарениями в области строительства UI, не могу назвать свой любимый архитектурный паттерн UI, также как я не могу похвастаться и огромным опытом в создании UI. Единственное что я могу сказать, я делал UI за деньги, неправильно и мне очень стыдно ;)
- Этот курс заставил меня хоть как-то разобраться с Jetpack compose и я из чистой любви к искусству попытался разложить по полочкам интерфейс пользователя своей коммерческой программы, хотя там можно было обойтись и без этого.

Имеет ли смысл мудрить и делать все правильно?

- Если у вас простая задача – нет. Я и не мудрю обычно. На страшный код не страшно смотреть, если он маленький (или пока он маленький, паровозы нужно убивать, когда они еще чайники =).
  - Конечно вам решать, но когда я вижу гору кода на пустом месте мне становится смешно.
  - Вы должны быть как опытные бойцы, которые не красуются перед девушками, а уничтожают своего противника максимально быстро и эффективно (а не эффектно).
  - Короче, чаще вспоминайте про принцип KISS – будь проще, тупица!
  - Ну а для сложных и запущенных случаев есть редизайн и рефакторинг, но вдруг пронесет и это все не понадобится… Практика показывает, что рабочие программы ужасны внутри (открыл код, а он состоит из одних костылей), а **красивые программы часто не доживают до релиза, так как кончается время на редизайн**.
  - В этот самый момент я занимаюсь таким редизайном, но если я не успею, то вносить изменения в старый код не получится, так как я в нем окончательно запутался…
- Если ваш проект действительно сложный, вы работаете в большой фирме, над проектом работает большая команда (с типичной проблемой текучки кадров и легаси кодом), проект имеет перспективы развития и требует постоянных переделок, вам нужно организовать полноценное тестирование, то вам придется делать все «по науке». Иначе вы просто ничего не сделаете.
- В пограничных случаях можно делать и так и так. Если этот проект единственный в своем роде, то не мудрите, а если у вас будет вал работы в этой области, то идите и изучайте архитектурные паттерны…

Сколько нужно времени, чтобы изучить тонкости построения UI для Android?

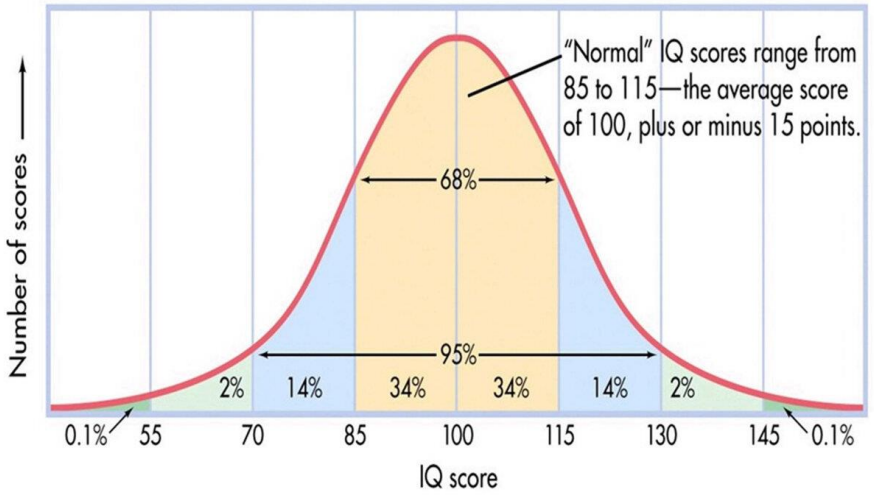
- Немного (единицы дней на базовый уровень, но лично мне пришлось поломать голову над состояниями и тем, как прикрутить корутины к Jetpack, там оказалось не все так просто и однозначно), при ряде условий:
  - У вас есть мотивация заниматься вот этим вот всем.
  - Вы хорошо знакомы с азами вычислительной техники и можете отличить компилятор от процессора и оперативную память от блока питания. То есть в книгах Таненбаума про компьютер, сети и ОС вам все все более-менее понятно и у вас есть уже даже есть какое-то свое мнение по поводу того, что написано в этих книгах.
  - Вы понимаете паттерны проектирования UI (и вообще знакомы с понятием паттернов проектирования, моделей вычислений и архитектуры);
  - Вы достаточно глубоко понимаете язык программирования на котором вы пишете UI (в нашем случае это Kotlin, с его грэдлом, классами, объектами, лямбдами, аннотациями, колбэками, интерфейсами, корутинами и другими штуками, непонятными простому смертному =);
  - Вы умеете изучать килотонны документации за короткое время.
  - У вас достаточно большая практика в программировании (без практики, сидя на диване освоить что-либо в нашей области не получится, слишком много деталей и неоднозначностей, а документация как правило ужасна).
- В случае если вы вообще ничего не понимаете в вычислительной технике, то у вас уйдут годы, а ваш путь будет напоминать путь мага из фэнтезийной книжки, который годами учит наизусть заклинания из единственной книги и вообще не понимает, что он учит.

Лирическое отступление о качестве технической документации…

- *Каждый интеллигентный человек должен знать, что такое сепульки:*
  - *СЕПУЛЬКИ – элемент цивилизации ардритов, связанный с сепулькариями (см. СЕПУЛЬКАРИИ).*
  - *СЕПУЛЬКАРИИ – объекты, предназначенные для сепуления (см. СЕПУЛЕНИЕ).*
  - *СЕПУЛЕНИЕ – регулярное действие, процесс производства сепулек, совершаемый ардритами, живущими на планете Энтеропия (см. СЕПУЛЬКИ). «Космическая энциклопедия»*
- (С) Станислав Лем, Звездные дневники Ийона Тихого.

Почему так происходит мы уже знаем…

- Реальная архитектура (та, которая придумана разработчиками и существует на самом деле) довольно сложна.
- Архитектурное описание (даже простое) как правило не по зубам большинству разработчиков.
- Получается, что архитекторам нет смысла возиться с описанием, им и так весело, а технические писатели, делающие документацию, в архитектуре ничего не понимают.
- Другими словами: чаще всего архитектура не покидает головы архитектора



Быстрая смена технологий: имеет ли смысл изучение всех технологий подряд? Пример из жизни.

- Я делал проекты с использованием Android в 2017-2019 годах
- С 2019 года произошли следующие изменения:
  - Java в Android была заменена на Kotlin
  - Стек технологий и архитектура приложений сильно поменялся
- Фактически, все знания, которые у меня были по проектированию мобильных приложений морально устарели всего за 3-4 года.
- В этом смысле изучение библиотек и фреймворков впрок – скорее всего плохая идея, так как через несколько лет вероятно ими уже никто не будет пользоваться. Большинство фреймворков, которые я видел несколько лет назад сейчас вообще не актуальны.

Что оказалось полезным

к вопросу о постоянных спорах со студентами, на тему «зачем нам все это нужно»

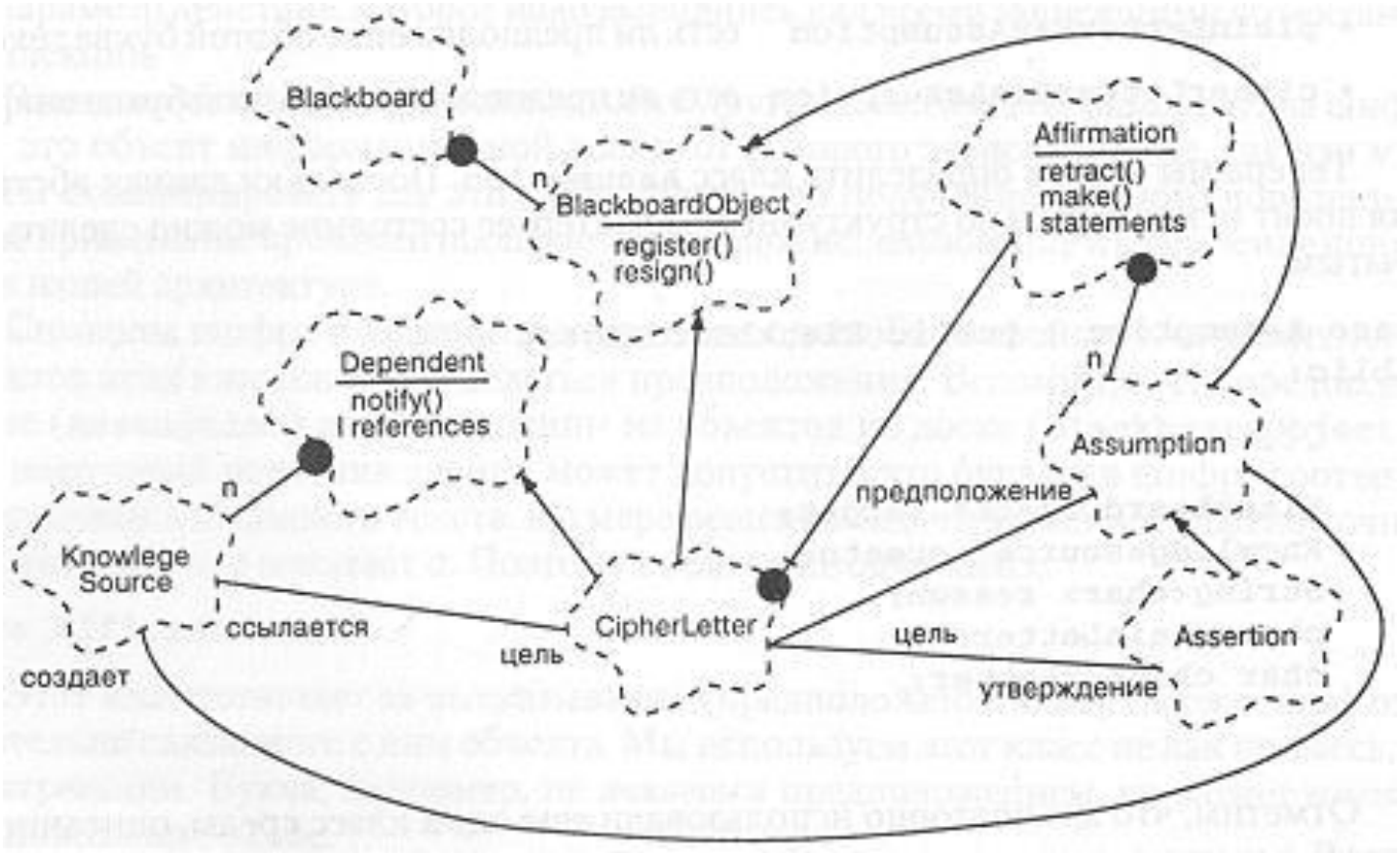
- Понимание моделей вычислений
  - Понимание цифровой и аналоговой схемотехники, понимание HDL
  - Понимание языков программирования и их основ (переход C-> C++ -> C# -> Java -> Kotlin)
  - Понимание устройства микропроцессоров, систем-на-кристалле, процессоров, гипервизоров и виртуальных машин
  - Понимание принципов работы операционных систем (ОС РВ, мобильных ОС и ОС общего назначения)
- Понимание работы сетей
  - Контроллерные сети
  - Локальные вычислительные сети
  - Стеки протоколов обмена
- Архитектурное проектирование
  - Опыт проектирования различных архитектур с последующей проверкой решений на практике оказался крайне полезен
- Глубокое понимание архитектурных шаблонов
  - Не бездумное использование готовых шаблонов из книжки Design Patterns, а умение изобретать свои при необходимости.

Когда это имеет смысл?

- Когда ваша работа сложная, то есть если вы сами создаете новые технологии, а не используете готовые, по шаблону делая простые проекты.
- На простых проектах скорее всего поставит крест бурно развивающийся ИИ и судьба программистов, закончивших курсы и с трудом освоивших изготовление простых сайтов и простых бекэндов не очень завидна.

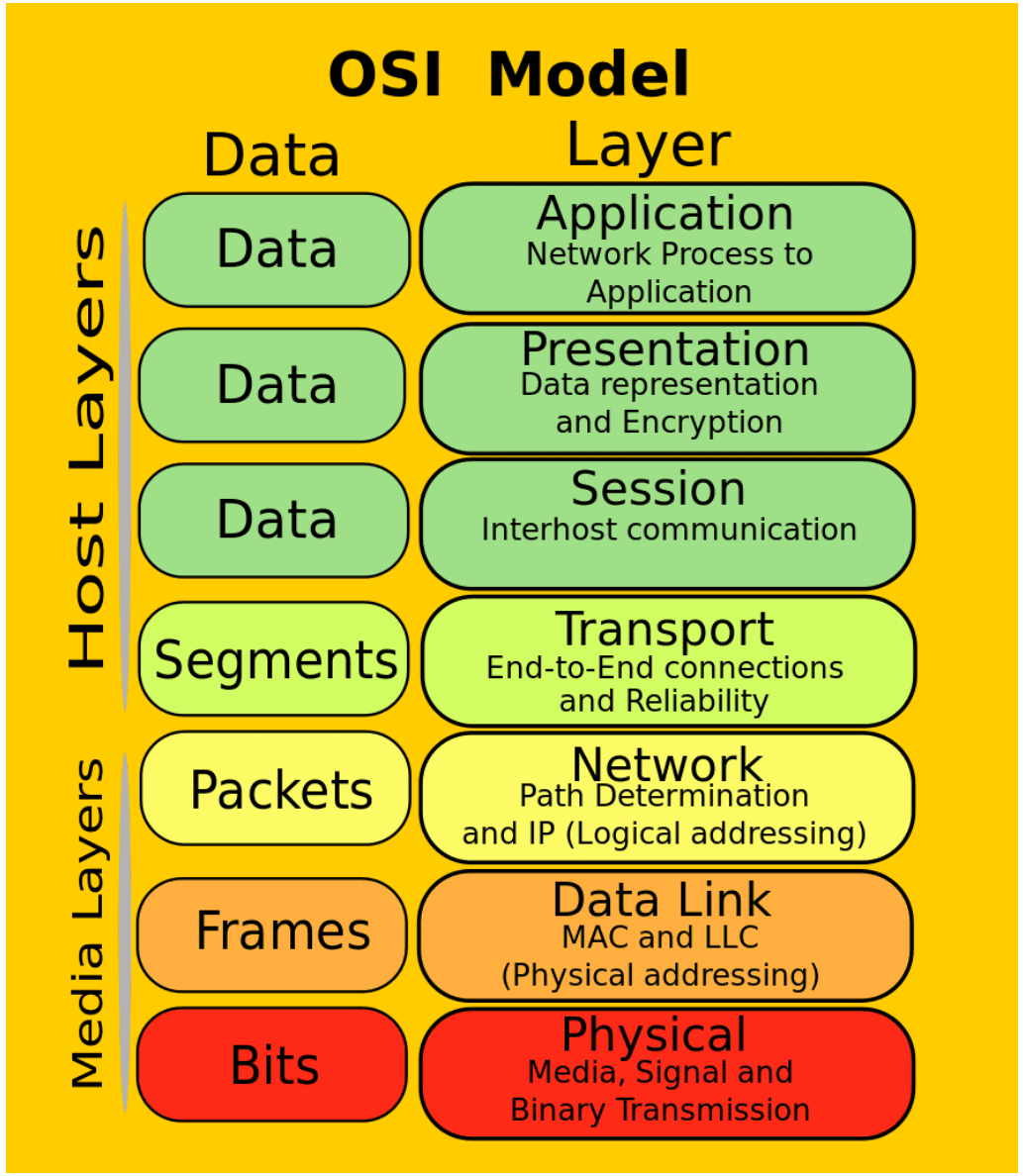
ООП и всякое такое…

- Классификация
- Отношения между объектами
- Категоризация
- Обобщение
- Конкретика и абстракции
- Анализ и синтез
- Декомпозиция

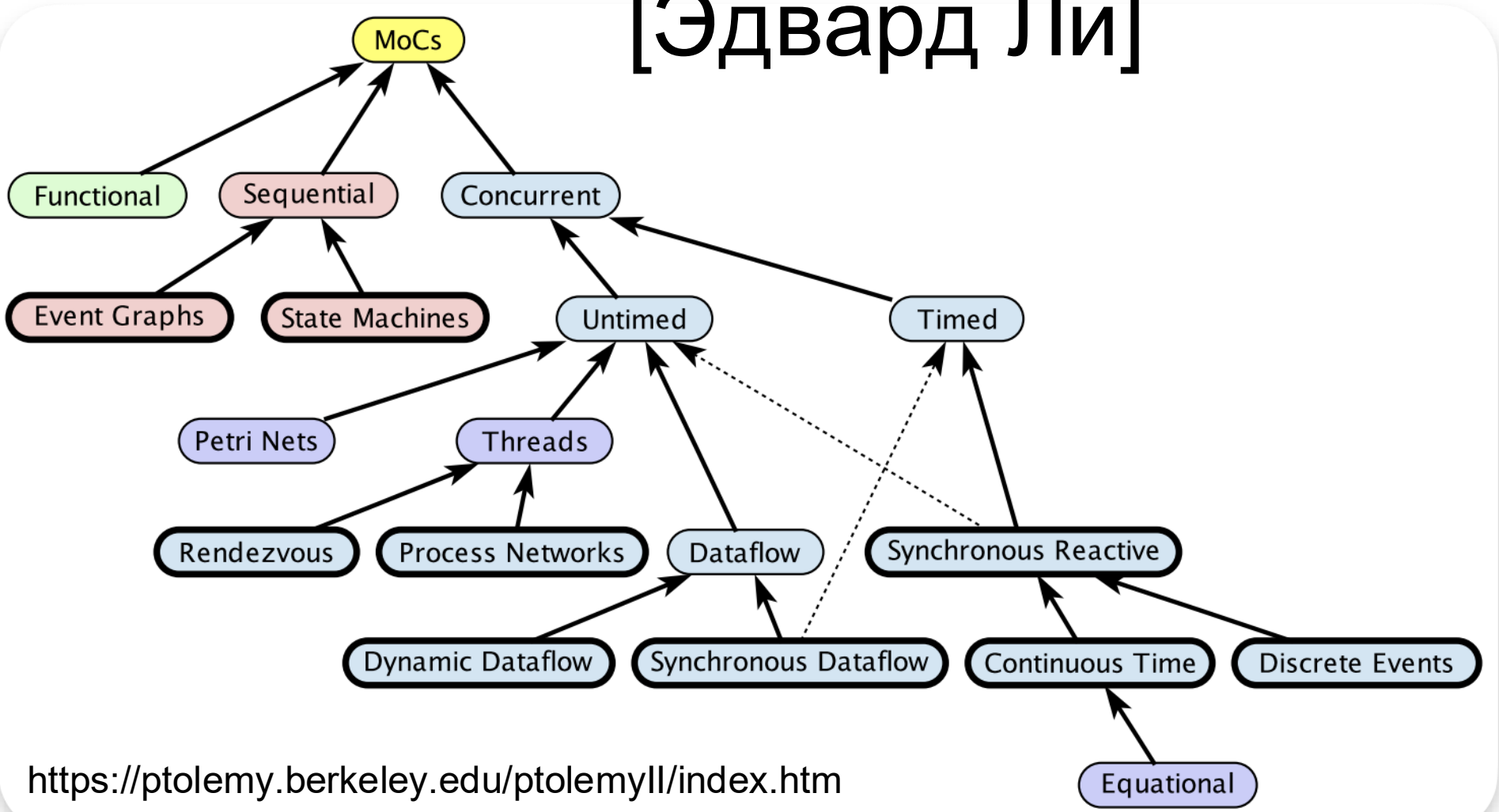


Модели вычислений и уровневая архитектура

На базе каких законов работает вычислительный процесс на каждом уровне абстракции?



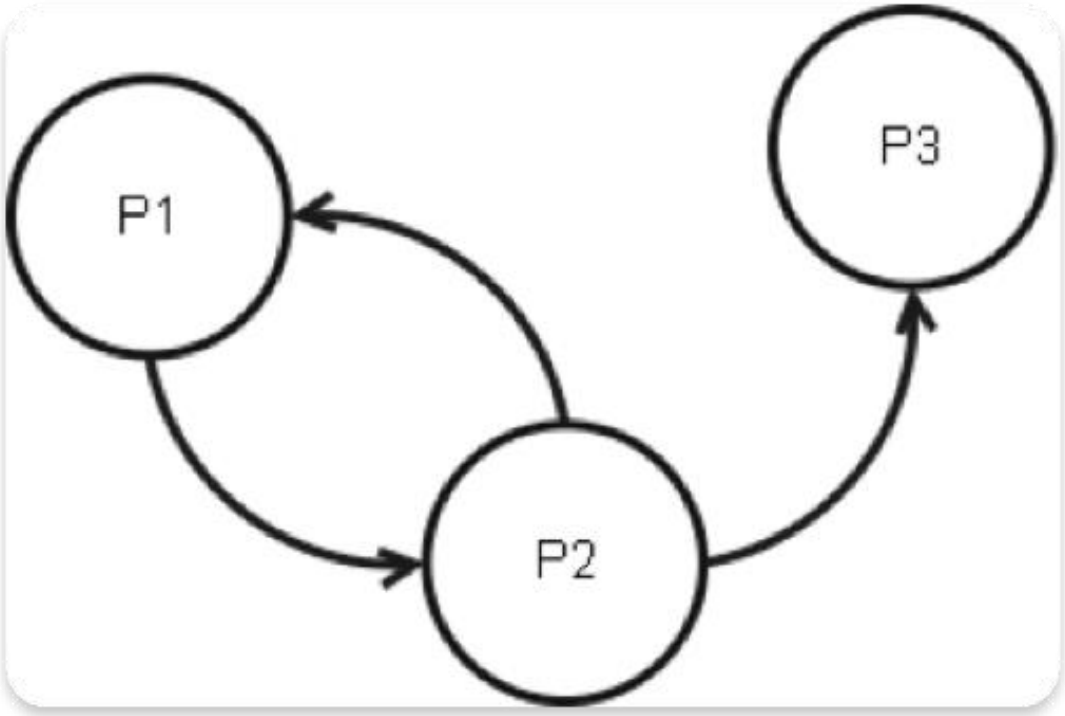
Классификация моделей вычислений [Эдвард Ли]



https://ptolemy.berkeley.edu/ptolemyII/index.htm

Process Network

- Process Network (PN), сеть процессов, сеть потоков данных - модель вычислений, в которой система представляется в виде ориентированного графа, вершины которого представляют собой процессы (вычисления), а дуги представляют собой упорядоченные последовательности элементов данных.



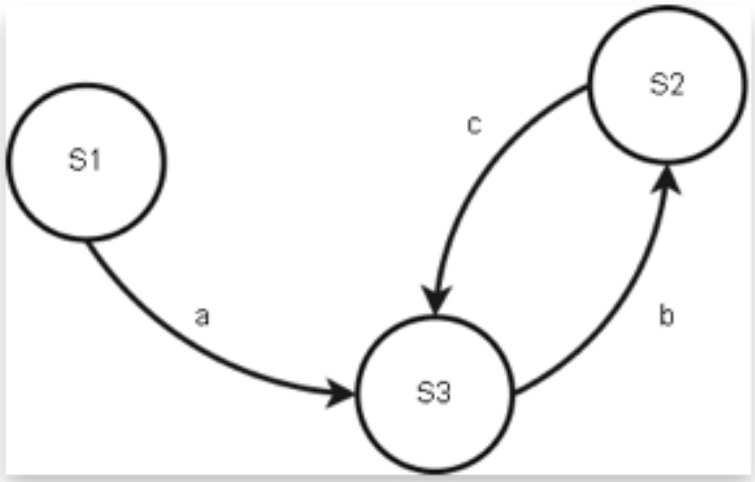
Конечный автомат

- FSM (finite state machine) , (finite automaton), конечный автомат, автомат — модель вычислений предполагающая наличие состояний объекта, переходов от состояния к состоянию и условий перехода.
- Конечный автомат - математическая модель устройства с конечной памятью. Конечный автомат перерабатывает множество входных дискретных сигналов в множество выходных сигналов. Различают синхронные и асинхронные конечные автоматы.
- Конечный автомат в теории алгоритмов - математическая абстракция, позволяющая описывать пути изменения состояния объекта в зависимости от его текущего состояния и входных данных, при условии что общее возможное количество состояний конечно. Конечный автомат является частным случаем абстрактного автомата.

Конечный автомат

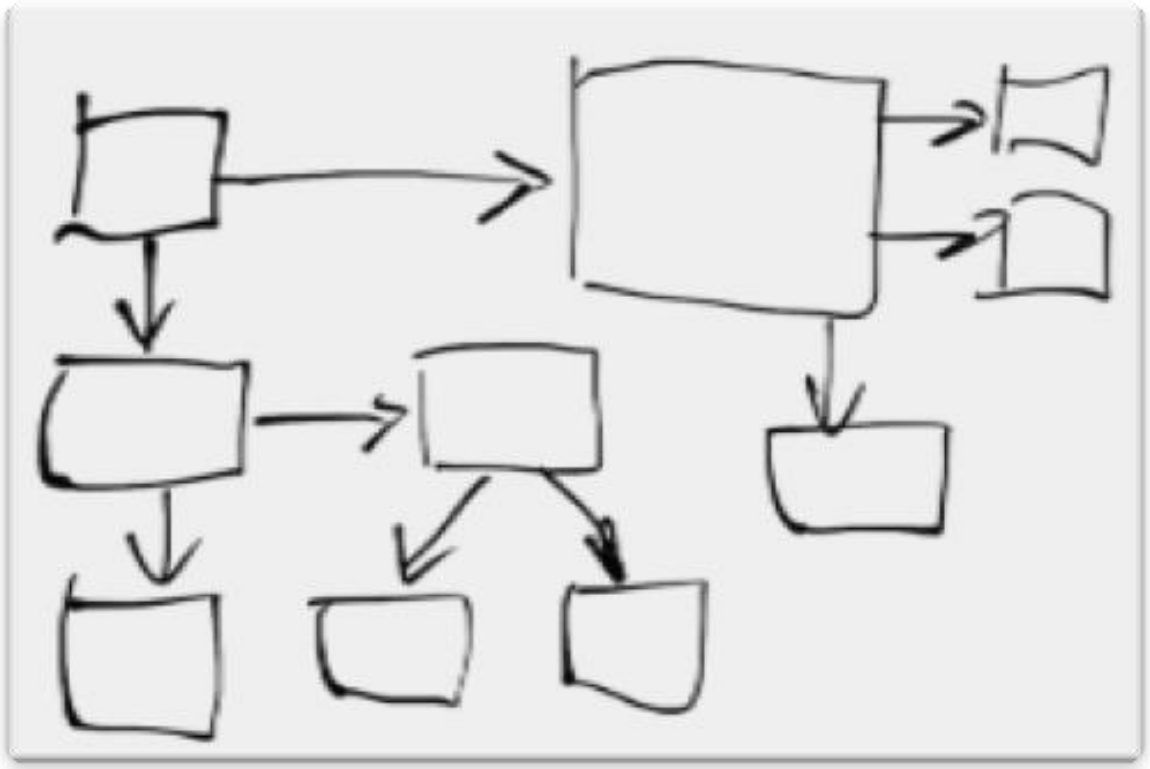
```
state = S1;

while( true )
{
   switch( state )
   {
      case S1: if( a ) state = S3;
      case S2: if( c ) state = S3;
      case S3: if( b ) state = S2;
   }
}
```

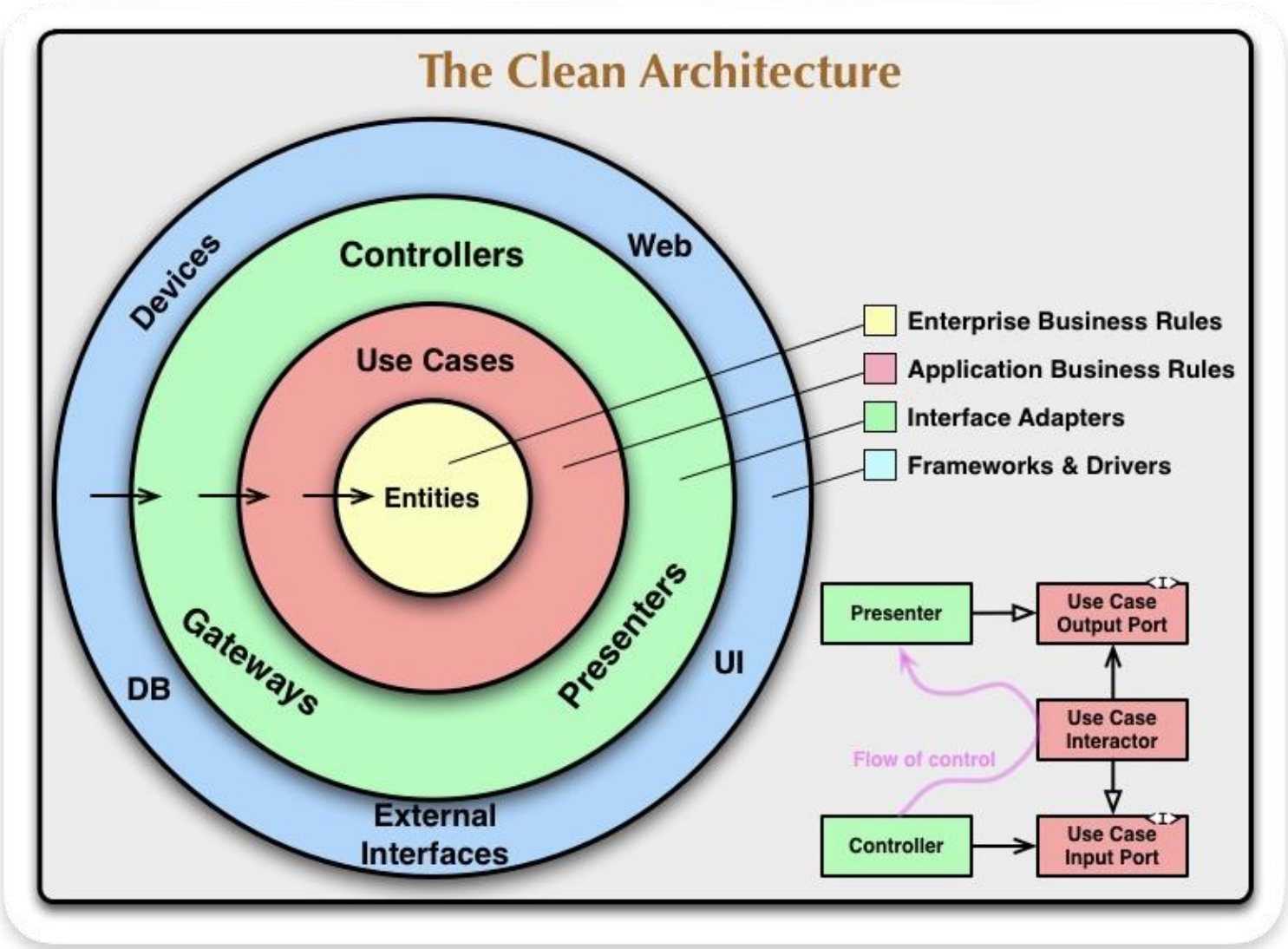


Архитектура

- Структура это совокупность связей между частями объекта.
- Архитектура – более ёмкое понятие, включающее в себя множество структур. MVVM – не архитектура приложения!!! Это маленький кусочек общей архитектуры!
- В рамках любой модели вычислений систему можно представить как структуру, в виде совокупности акторов и связей между ними.



Чистая архитектура Роберта Мартина

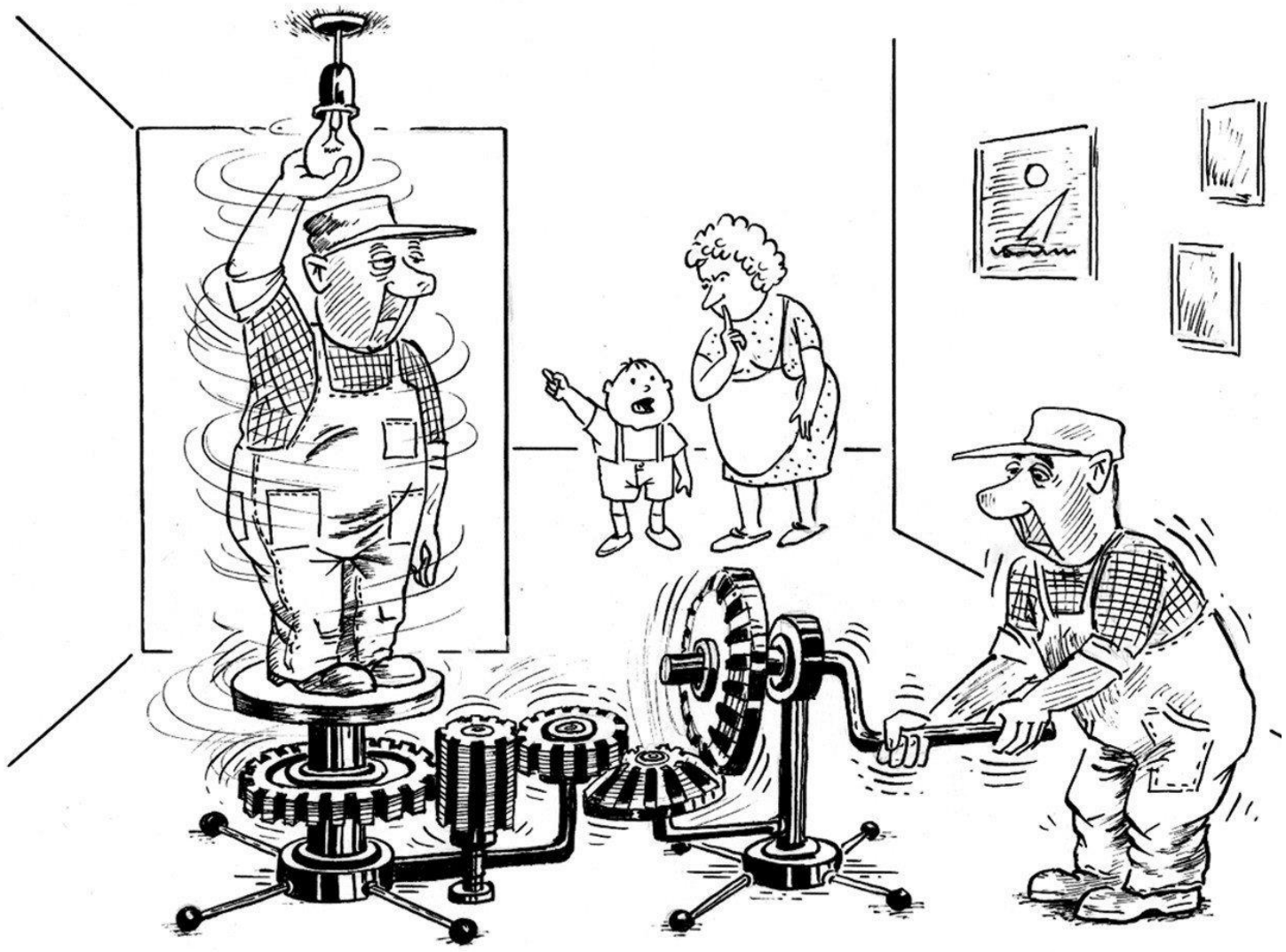


Чистая архитектура

1. Независимость от фреймворков
2. Тестируемость
3. Независимость от интерфейса пользователя
4. Независимость от баз данных
5. Независимость от каких либо внешних сервисов

Платформа

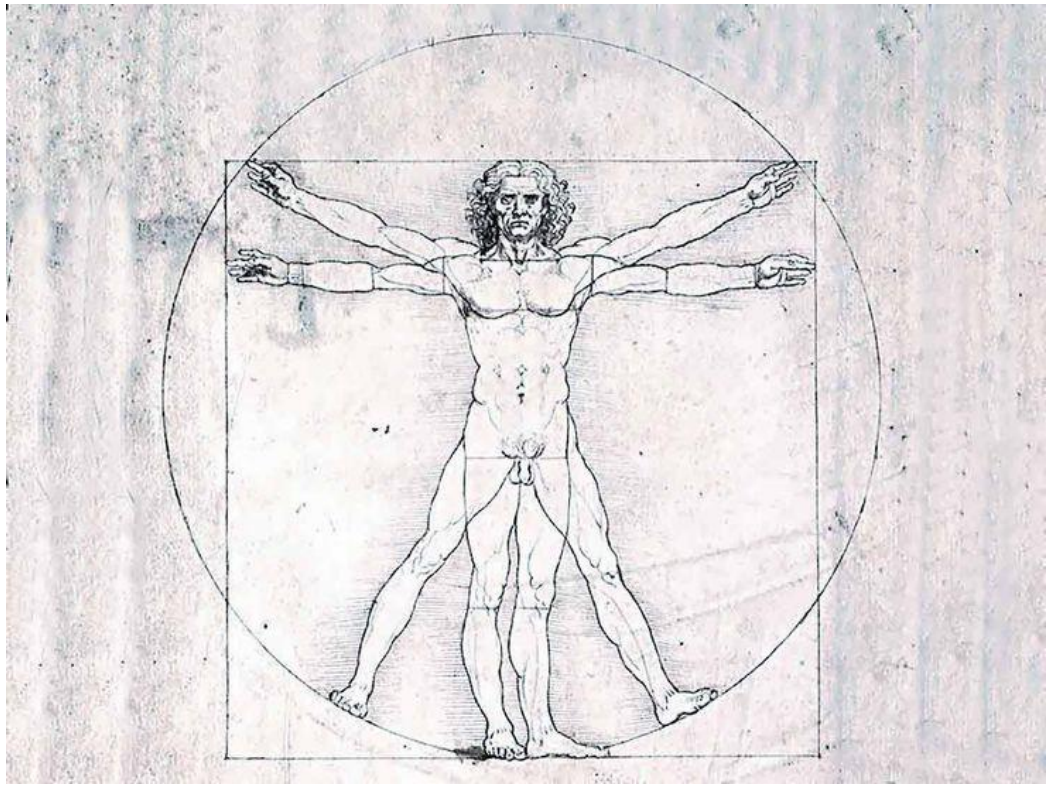
Платформа это то, на чем удобно стоять и делать какую-либо полезную работу



Для кого нужен UX/UI?

Для ЧЕЛОВЕКА!

Из знаний о том как видят, чувствуют, думают и поступают люди проистекают основные принципы построения UX/UI

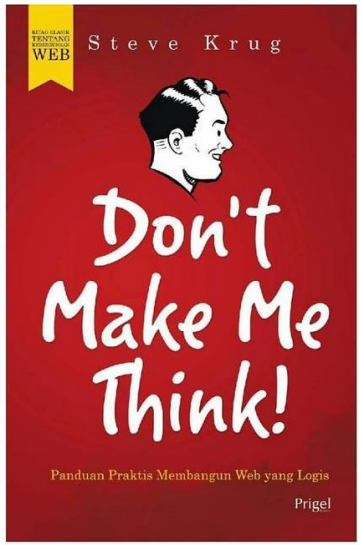


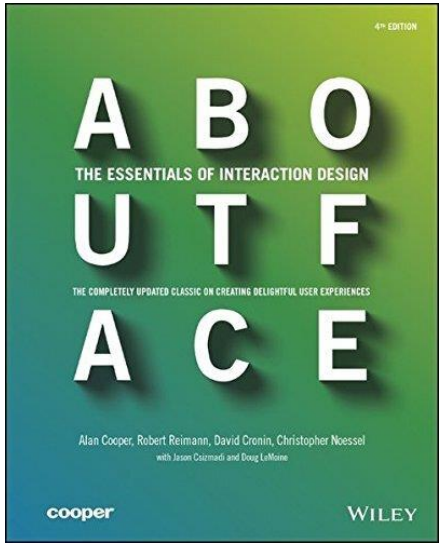
Основные принципы построения интерфейса пользователя

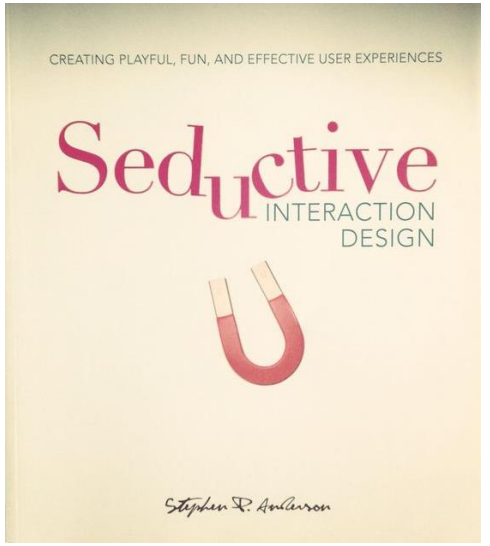
- Современные пользовательские интерфейсы строятся на основе ряда фундаментальных принципов, которые обеспечивают:
  - удобство,
  - интуитивность,
  - эффективность взаимодействия с пользователем.
- Эти принципы сформированы на основе исследований в ряде областей:
  - UX/UI,
  - психология восприятия,
  - технологические возможности (обычные, сенсорные и голографические дисплеи, дополненная и виртуальная реальность, разные способы ввода и т.п.).

UX/UI

- UX-дизайн (User Experience — «пользовательский опыт») отвечает за то, как интерфейс **работает**.
- UI-дизайн (User Interface — «пользовательский интерфейс») отвечает за то, как интерфейс **выглядит**.
- UX дизайнер планирует то, как пользователь будете взаимодействовать с интерфейсом и какие шаги нужно предпринять, чтобы сделать что-то. UI дизайнер придумывает, как каждый из этих шагов будет выглядеть.



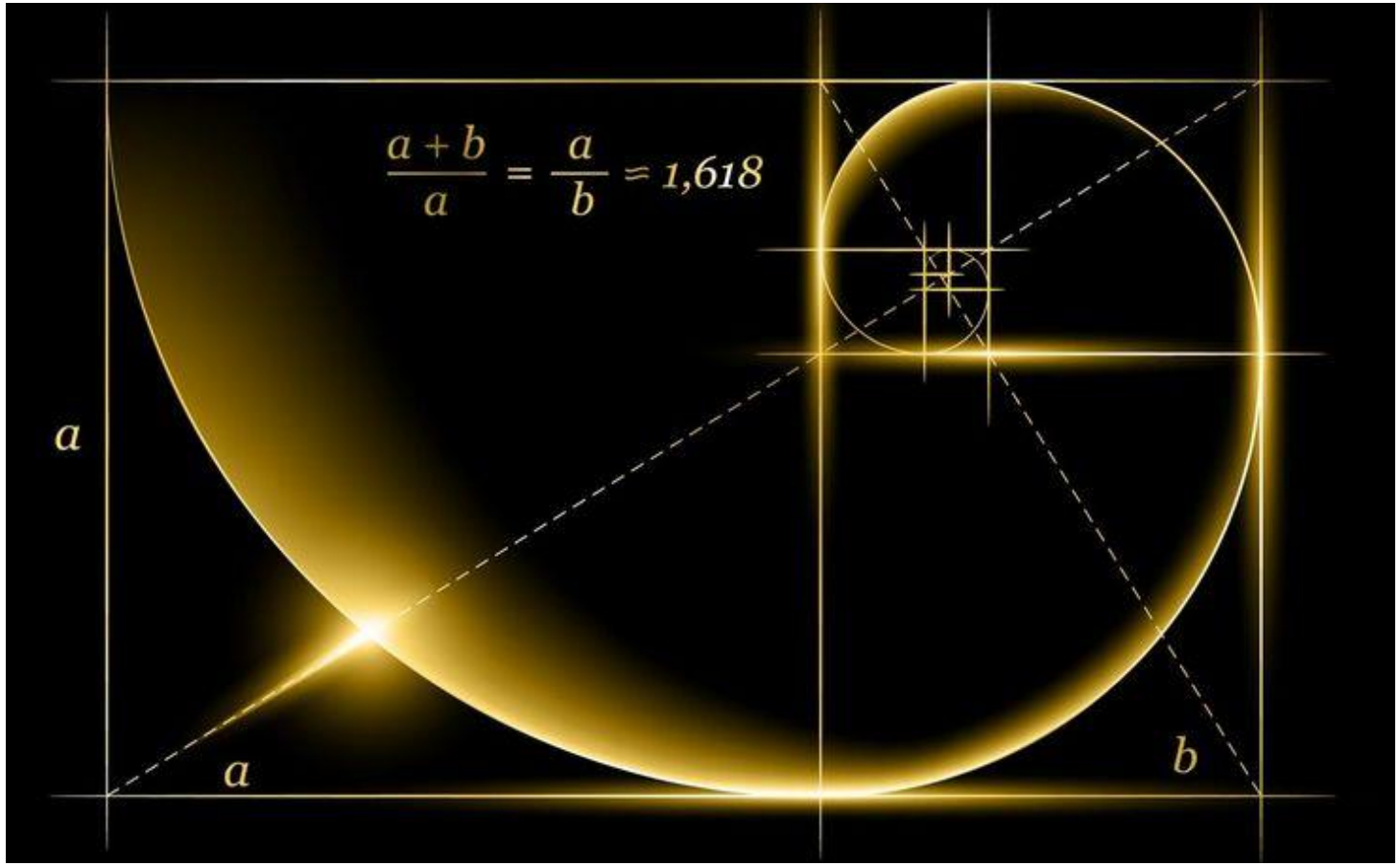




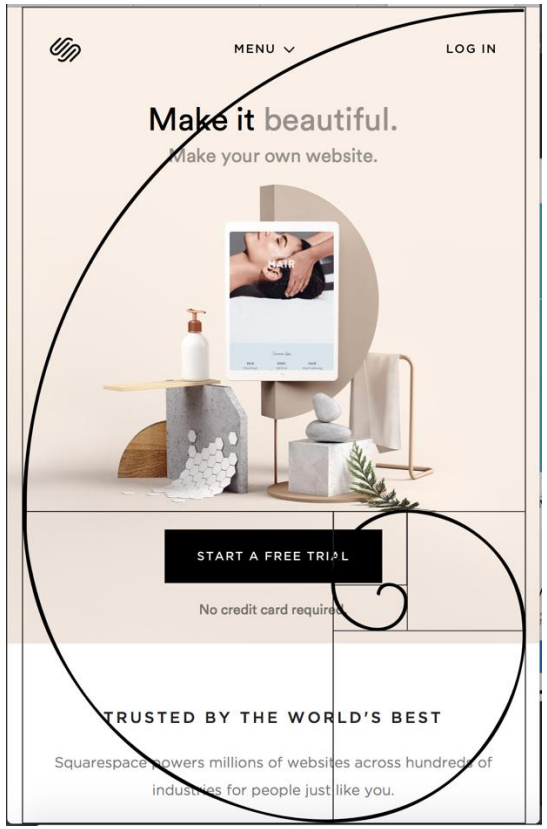
Золотое сечение

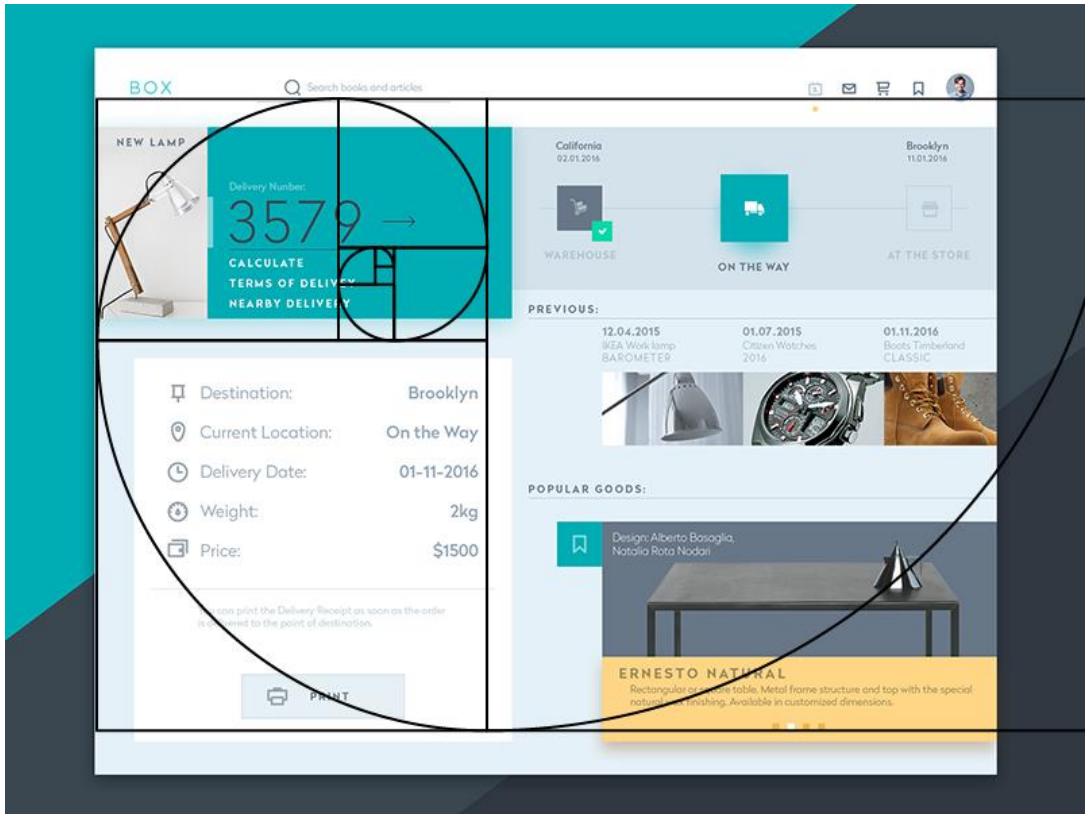
- «Золотое сечение» — это соотношение элементов разного размера, которое считается самым эстетически привлекательным для человеческого глаза. «Золотое сечение» равняется 1:1.618, и оно часто иллюстрируется спиралями в форме раковины, которые вы, возможно, видели в Интернете.

- https://uxgu.ru/golden-ratio-in-ui-design/



Золотое сечение



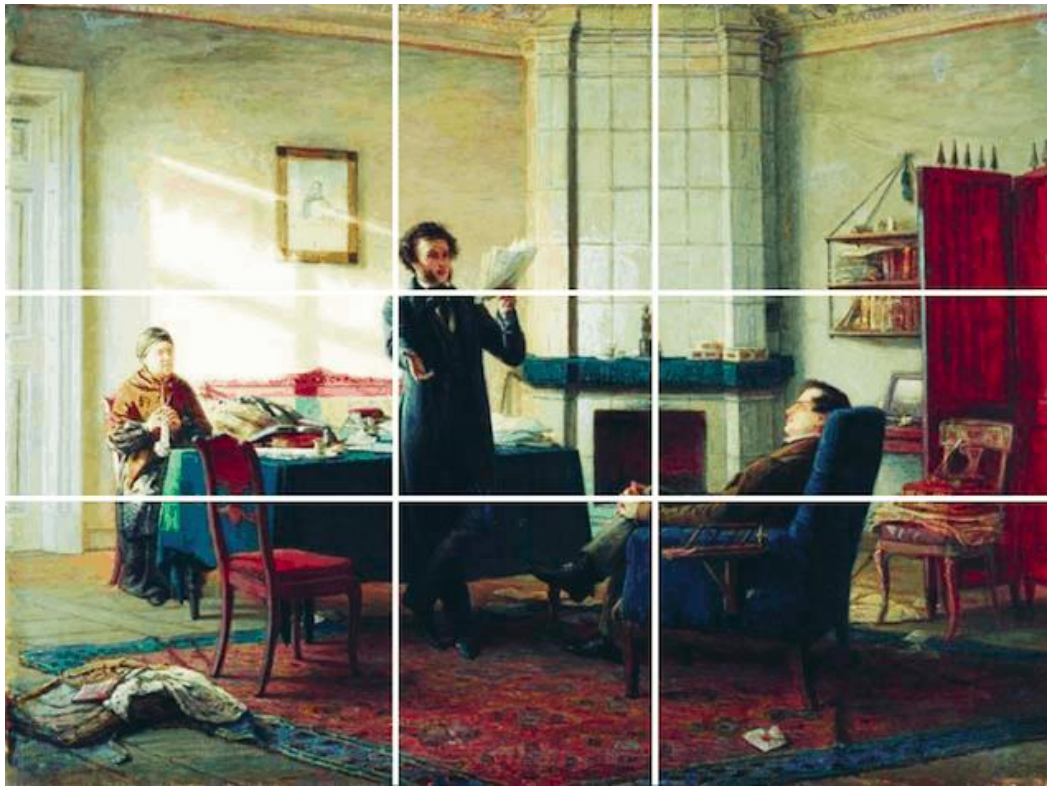


Композиция в фотографии

Анри Картье-Брессон

https://www.fotosklad.ru/expert/articles/vse-cto-vam-nuzno-znat-pro-zolotoe-secenie/

Композиция в живописи

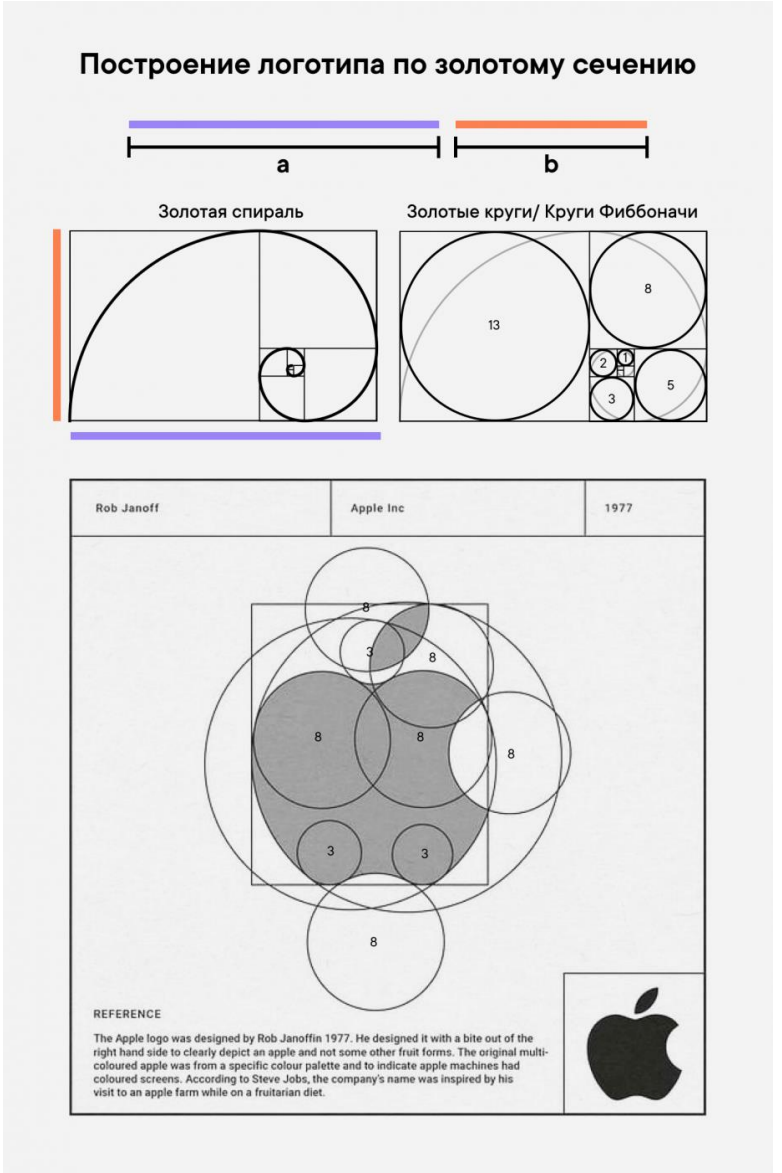


https://izo-life.ru/pravilo-zolotogo-secheniya/

Золотые круги

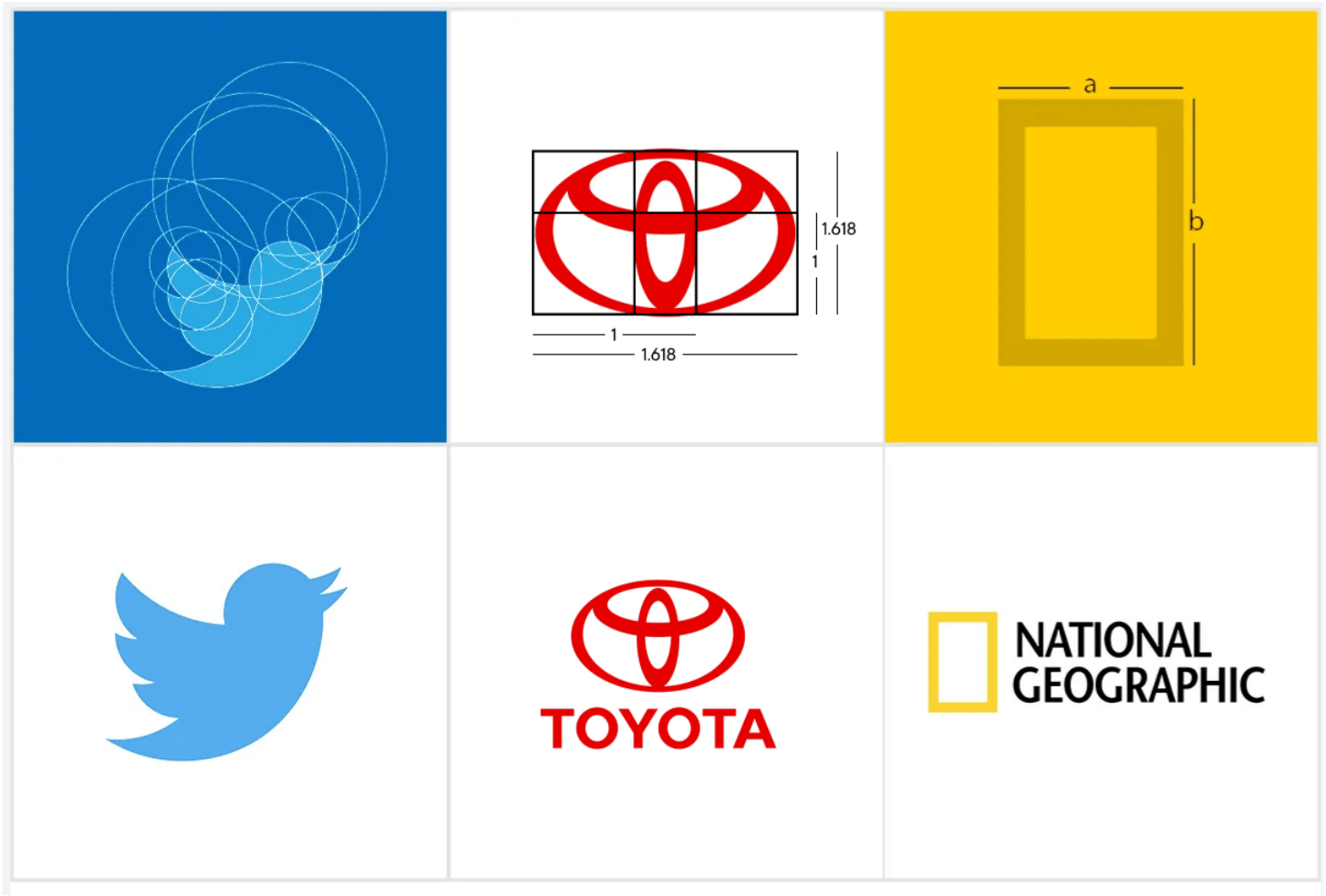
Если в каждый квадрат, полученный при построении золотой пропорции, вписать круг, то получится последовательность кругов в золотой пропорции по отношению друг к другу, или Золотых кругов — их также применяют для построения логотипов.

https://ux-journal.ru/zolotoe-sechenie-v-web-designe.html



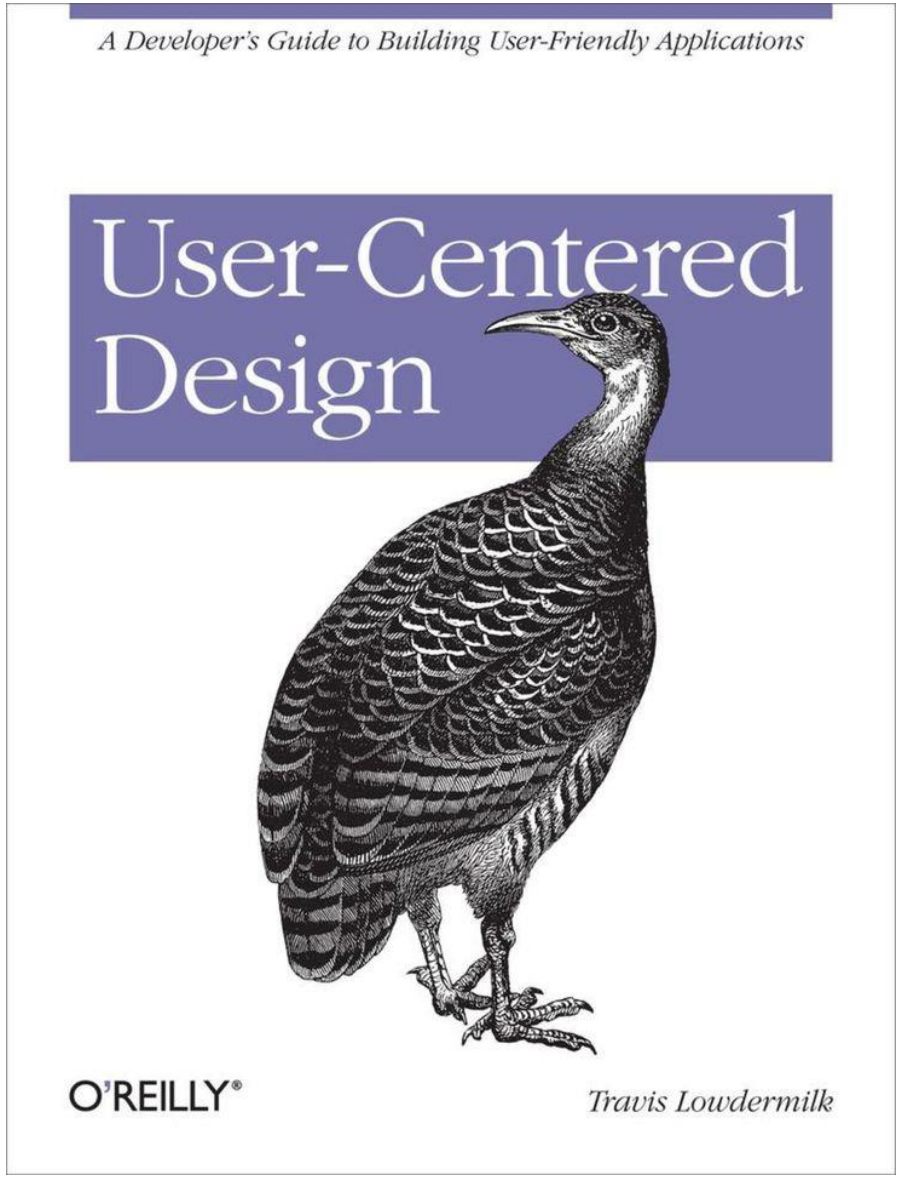
Примеры логотипов

- https://ux-journal.ru/zolotoe-sechenie-v-web-designe.html



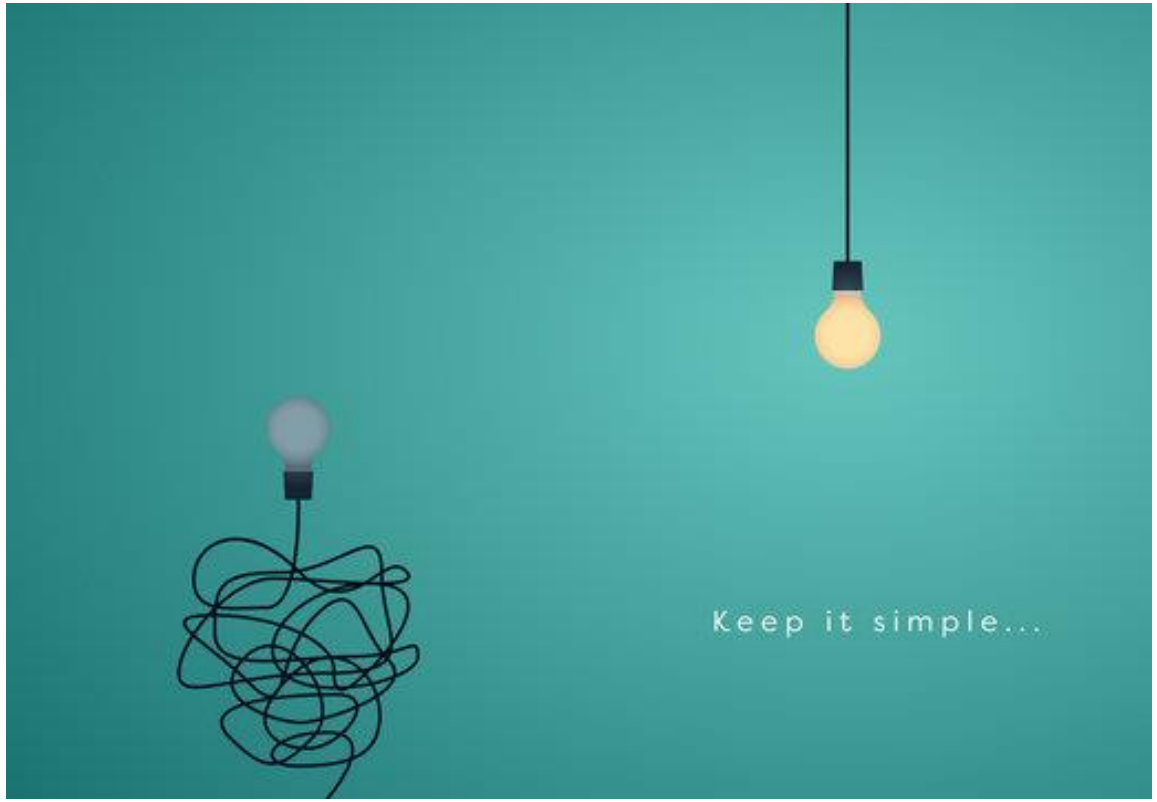
Пользовательская ориентированность (User-Centered Design)

- Интерфейс создается с учетом потребностей, целей и контекста использования целевой аудитории.
  - Исследование пользователей (интервью, опросы, анализ поведения).
  - Создание моделей типичных пользователей.
  - Тестирование прототипов для выявления проблем



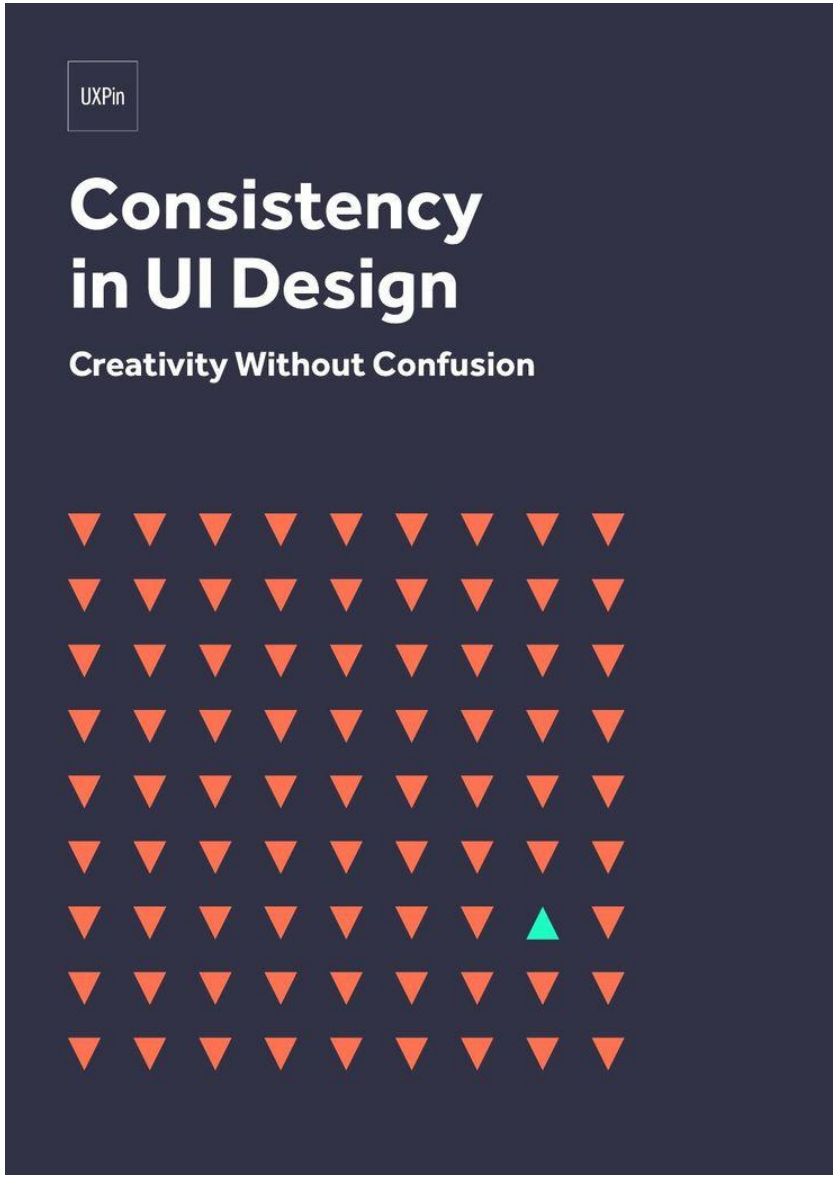
Простота и минимализм (Simplicity)

- Интерфейс должен быть понятным и не перегруженным лишними элементами.
  - Удаление ненужных функций или визуальных деталей (принцип "less is more").
  - Использование знакомых паттернов (например, кнопка "корзина" для покупок).
  - Четкое выделение главного действия (CTA — Call to Action).



Консистентность (Consistency)

- Элементы интерфейса должны быть единообразными по стилю, поведению и расположению.
  - Единый дизайн-язык (шрифты, цвета, иконки).
  - Повторяющиеся паттерны взаимодействия (например, одинаковые жесты для прокрутки).
  - Соблюдение стандартов платформы (iOS Human Interface Guidelines, Material Design для Android).



Интуитивность (Intuitiveness)

- Пользователь должен сразу понимать, как взаимодействовать с интерфейсом, без долгого обучения.
  - Использование метафор из реального мира (например, значок лупы для поиска).
  - Подсказки (например, "Введите запрос" в строке поиска).
  - Постепенное раскрытие функционала (progressive disclosure), чтобы не перегружать новичков.

Доступность (Accessibility)

- Интерфейс должен быть удобен для всех пользователей, включая людей с ограниченными возможностями.
  - Поддержка средств для чтения текста (например, VoiceOver для iOS).
  - Высокая контрастность цветов (WCAG 2.1).
  - Возможность управления жестами, нейроинтерфейс и т.п.

Обратная связь (Feedback)

- Пользователь должен получать немедленное подтверждение своих действий (вспомните требование про 10000 клиентов и возможные задержки)
  - Анимации (например, кнопка "пульсирует" при нажатии).
  - Сообщения об успехе/ошибке
  - Индикаторы загрузки для длительных операций.

Гибкость и адаптивность (Flexibility and Adaptability)

- Интерфейс должен работать на разных устройствах и в разных контекстах.
  - Адаптивный дизайн (responsive design) для экранов разных размеров.
  - Поддержка различных способов ввода (клавиатура, тачскрин, мышь, голос, нейроинтерфейс).
  - Персонализация (например, настройка темы — светлая/темная).

Эффективность (Efficiency)

- Интерфейс должен минимизировать усилия для достижения цели, а не как у вас обычно =)
  - Сокращение количества кликов/шагов для выполнения задачи.
  - Автодополнение и умные подсказки.
  - Горячие клавиши

Эстетика (Aesthetics)

- Визуально привлекательный дизайн улучшает восприятие и вызывает доверие.
  - Гармоничная цветовая палитра и типографика.
  - Использование микроанимаций для "живого" эффекта.
  - Баланс между функциональностью и красотой.
- Это немного не наша тема, это скорее к художникам и дизайнерам. Не если кто-то из вас занимается живописью, дизайном или художественной фотографией – то почему бы и нет.

Предсказуемость (Predictability)

- Пользователь должен предугадывать поведение интерфейса на основе предыдущего опыта.
  - Соблюдение общепринятых стандартов (например, красный цвет для "удалить").
  - Последовательная логика навигации.
  - Минимизация неожиданных изменений в интерфейсе.

Обучаемость (Learnability)

- Новые пользователи должны быстро осваивать интерфейс, а опытные — находить продвинутые возможности.
  - Встроенные подсказки или туториалы для новичков.
  - Постепенное усложнение функций.
  - Документация или справка для сложных систем.

Устойчивость к ошибкам (Error Tolerance)

- Интерфейс должен предотвращать ошибки и помогать их исправлять.
  - Подтверждение опасных действий (например, "Вы уверены, что хотите удалить?").
  - Возможность отмены действий (undo).
  - Четкие сообщения об ошибках с инструкциями.

Современные тенденции

- Голосовые интерфейсы: С ростом популярности ассистентов (таких как Алиса или Siri) UI адаптируется под голосовое управление.
- ИИ и персонализация: интерфейсы становятся "умнее", предлагая контент на основе поведения пользователя.
- Иммерсивный дизайн (дизайн с погружением): AR/VR требуют новых подходов к взаимодействию, учитывающих 3D-пространство.
- Эмоциональный дизайн: Интерфейсы стремятся вызывать эмоции (например, через анимации или эмпатичные тексты).
- Нейрокомпьютерный интерфейс (Brain Computer Interface) — это устройство, которое позволяет человеческому мозгу взаимодействовать с внешним программным или аппаратным обеспечением.

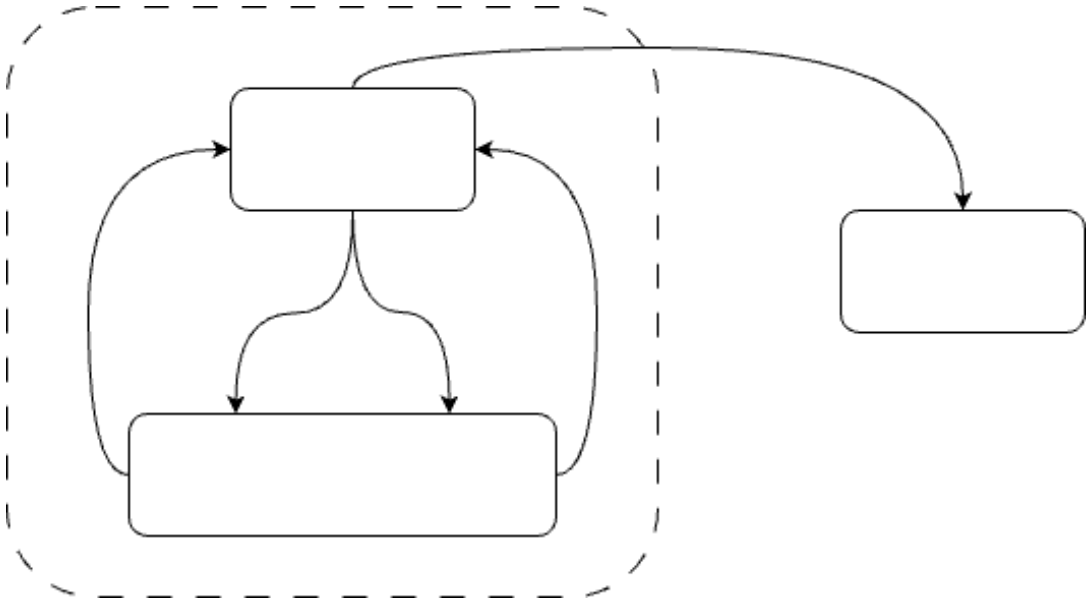
Архитектурные паттерны UI

Паттерны проектирования интерфейса пользователя

- Существует множество различных паттернов проектирования мобильных приложений: MVC, MVP, MVVM, MVI, Viper, Flux(Facebook), Riblets (Uber), Clean-Swift и т.п. Зачем они нужны?
- Разделение программы на уровни
  - Раздельная работа с каждым уровнем (у каждого уровня своя специфика)
  - Упрощение приложения => упрощение отладки и уменьшение числа ошибок и сроков разработки
  - Упрощение создания тестов
  - Упрощение расширения
- Без уровней, без понимания архитектуры программист (особенно начинающий) сделает монолитное приложение, а монолит в сложной программе это всегда плохо (взрыв сложности). Но, чем больше опыт, знания и умения программиста, тем больший монолит он может создать.
- Если программу пишет один программист, если это прототип который делается на коленке и который не нужно сопровождать, то монолит вполне приемлем. В остальных случаях (как это обсуждалось ранее) – монолит это зло.

Как додумались до архитектурных паттернов?

- Сильно связанные объекты лучше объединить, а слабо связанные разделить
- Удачные решения по организации программ, которые можно повторять в разных проектах превращаются в паттерны проектирования.

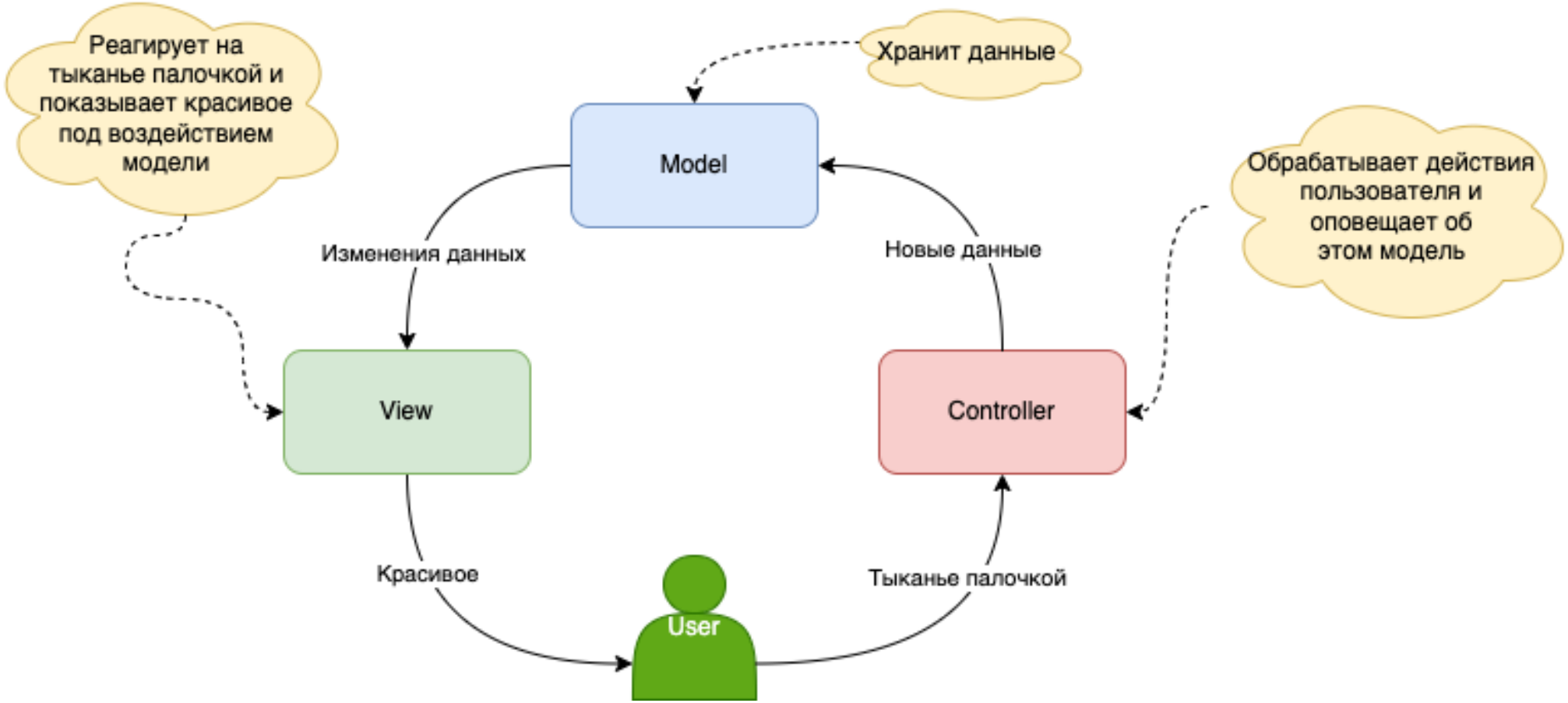


Паттерн проектирования Model View Controller

Модель придумана в Xerox в 1978 году (Smalltalk)

- **View** - представление интерфейса пользователя
- **Controller** - логика работы приложения (не регламентировано), возможно что это контроллер, поддерживающий реализацию какого-нибудь выпадающего списка (например, сортировку данных).
- **Model** - состояние приложения (в смысле состояния конечного автомата), но там может быть и логика приложения.

Model-View-Controller



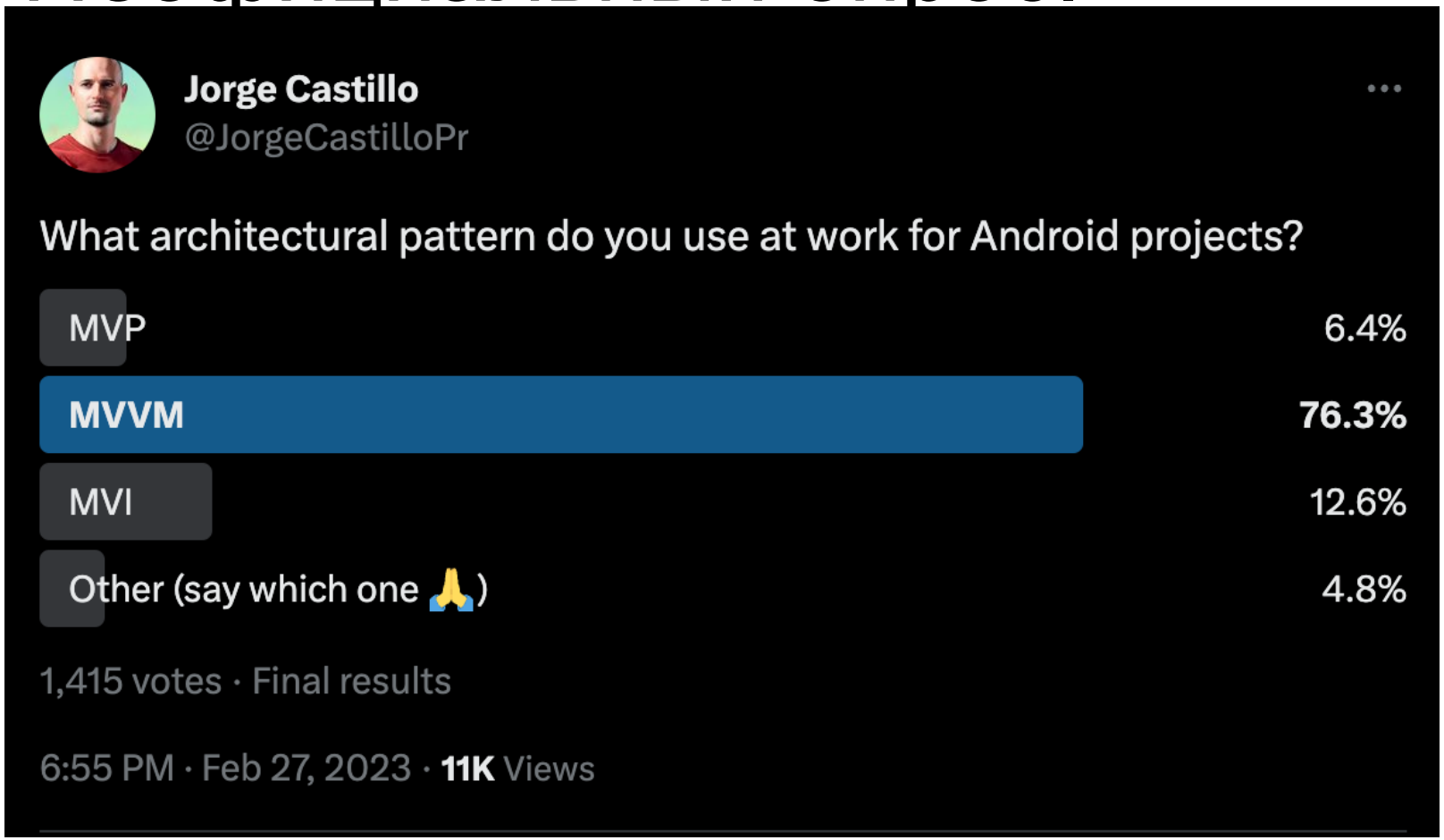
Работа MVC

- Действия пользователя производимые в View приводят к вызову методов контроллера
- Контроллер изменяет модель
- По факту своего изменения, модель вызывает коллбэки View
- View подписывается на изменения модели и меняется, при ее изменении.
- Контроллер также подписывается на модель
- View подписывается на контроллер

Проблемы MVC

- Казалось бы самая простая и ходовая модель из учебника, однако каждый трактует MVC по своему. При подготовке к лекции нашел в Интернете кучу различных вариантов.
- Лично мне не очень нравится, что три различных уровня общаются друг с другом напрямую.
- Нет четкого понимания, что должно быть в модели, а что в контроллере. Если делать то так, то эдак, программа превратится в непонятного монстра очень быстро. Она так обязательно сделает, если ее пишет несколько программистов.
- MVC скорее всего подходит только для совсем уж простых проектов, какими наверно были проекты разрабатываемые в Xerox на языке SmallTalk в начале 80-х годов.

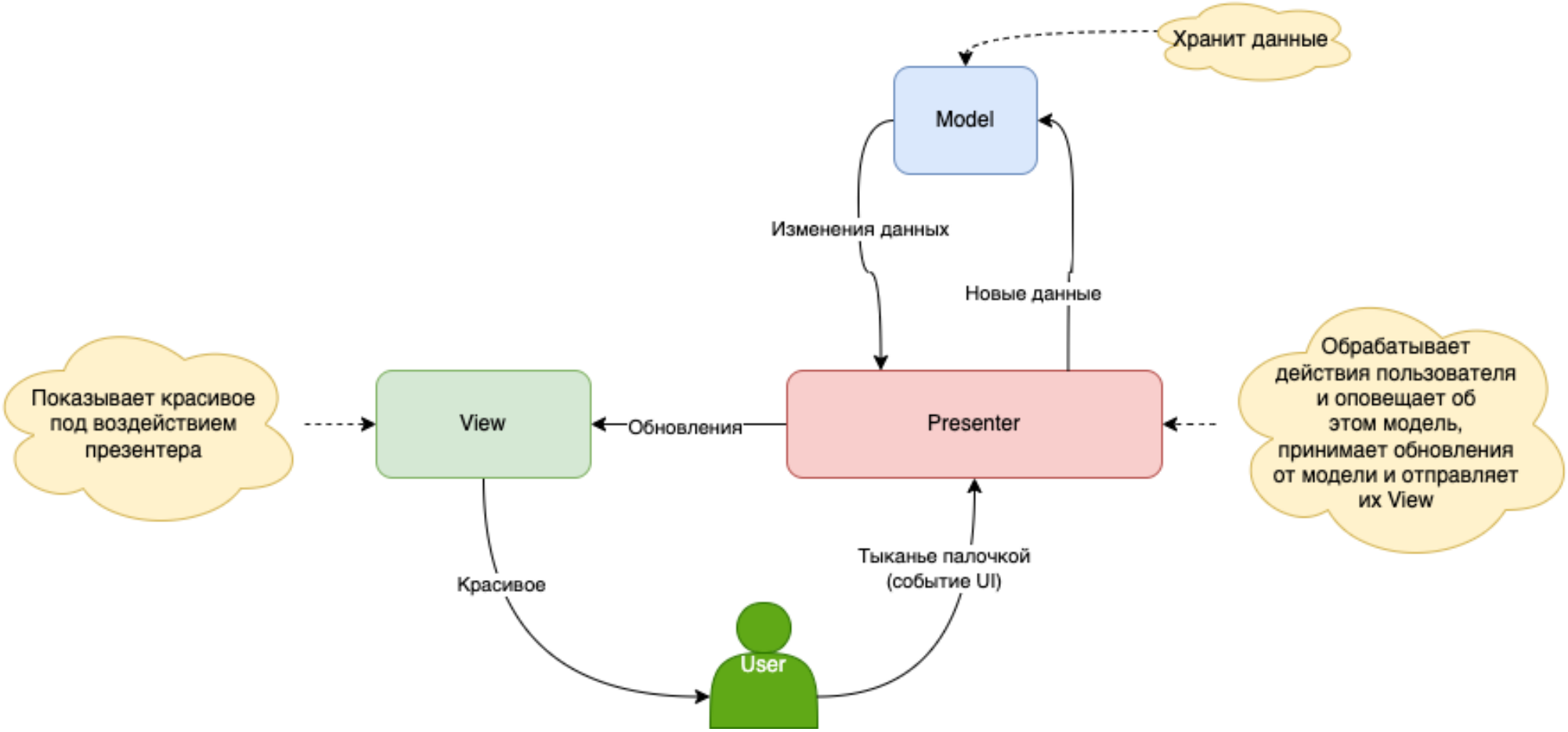
Какие архитектурные паттерны UI чаще всего используются в Jetpack Compose? Неофициальный опрос.



Model-View-Presenter (MVP)

- Model - состояние приложения (как состояние конечного автомата)
- View - отображение элементов интерфейса пользователя
- Presenter - обеспечивает связь между Model и View и содержит в себе прикладную логику приложения.

Model-View-Presenter



Использование MVP

- Windows Forms, ASP.NET
- Java AWT, Swing, SWT
- Android (в том числе JetPack Compose)
- iOS

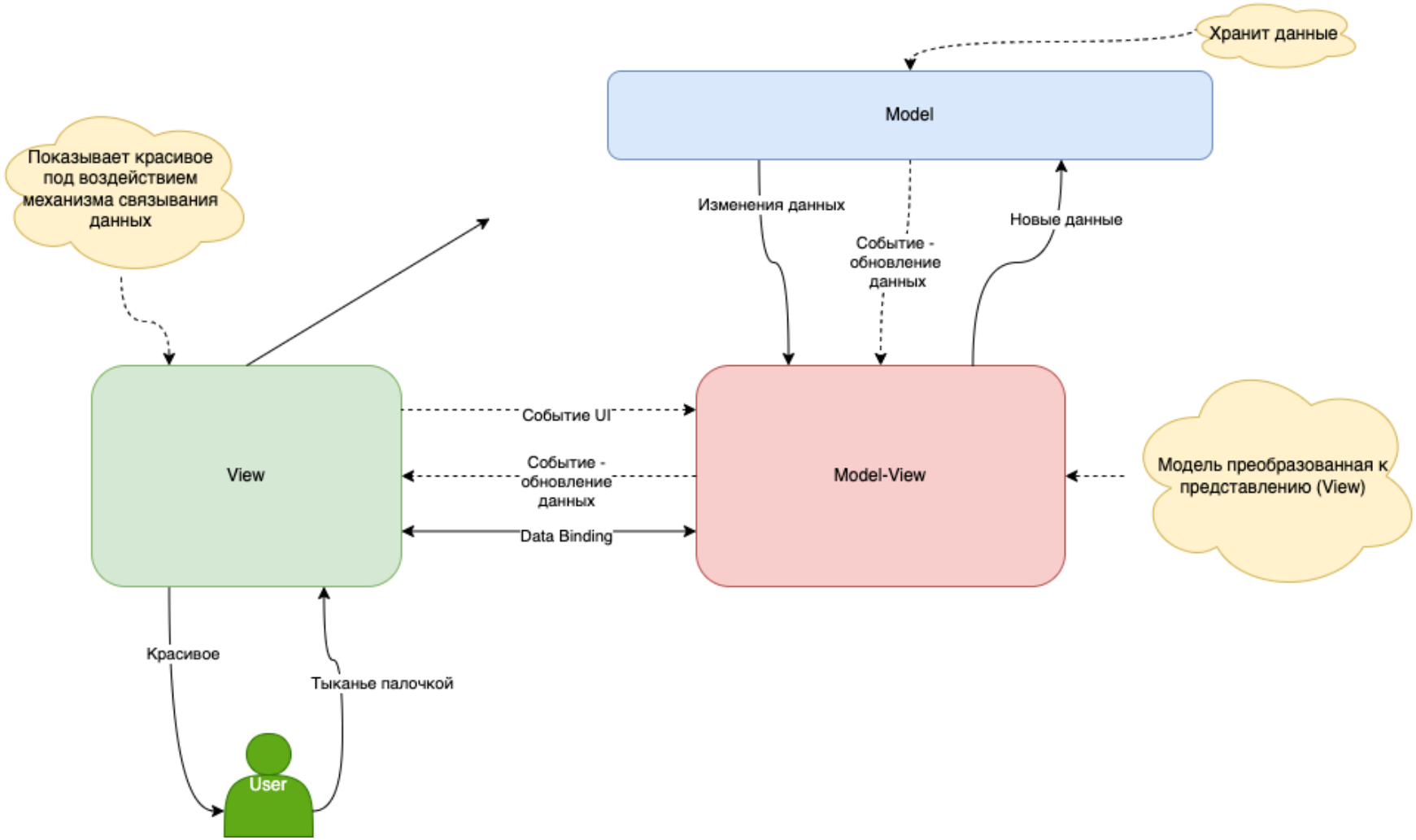
Model View View-Model

- Model - логика работы с данными
- View - элементы GUI, подписываются на события ViewModel
- ViewModel - промежуточное представление данных, приближенное к View

Использование Model View View-Model

- Windows Presentation Foundation (WPF)
- Android JetPack Compose
- **Using Jetpack Compose with MVVM**

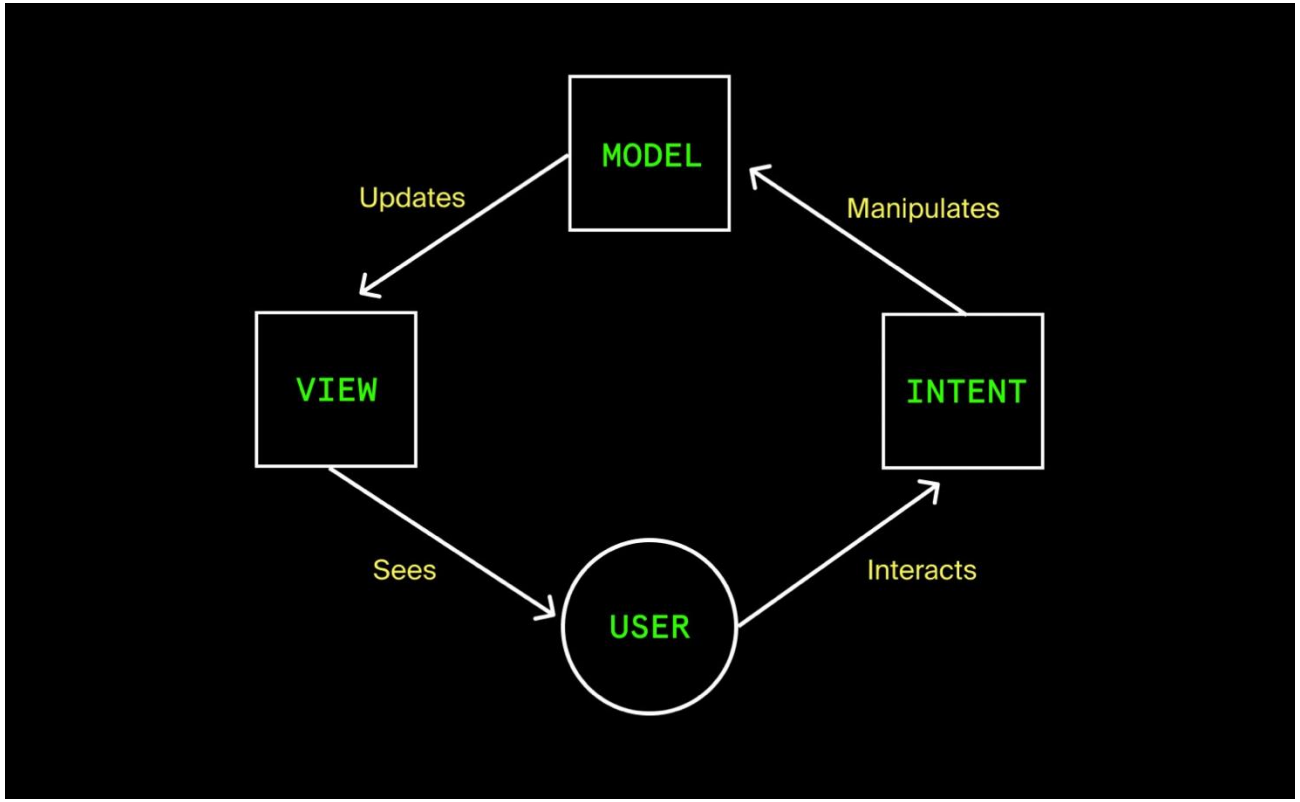
Model View View-Model

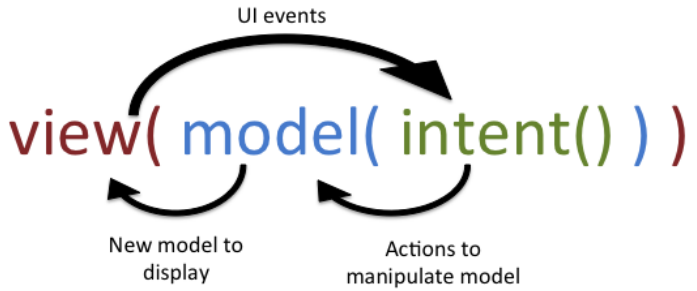


Модель MVI

- **Model-View-Intent** (MVI) впервые описана Андре Штальтцем (André Staltz) для JavaScript-фреймворка cycle.js
- **intent** (намерение) - функция, принимающая входные данные от пользователя (например, onClick) и отправляющая данные в модель. **Не имеет отношения к Android Intent!**
- **model** - функция, которая использует выходные данные из функции intent в качестве входных данных для работы с моделью. Результат работы этой функции – новая модель (с измененным состоянием).
- **view** - функция, которая получает на входе модель от model() и просто отображает ее.

Модель MVI

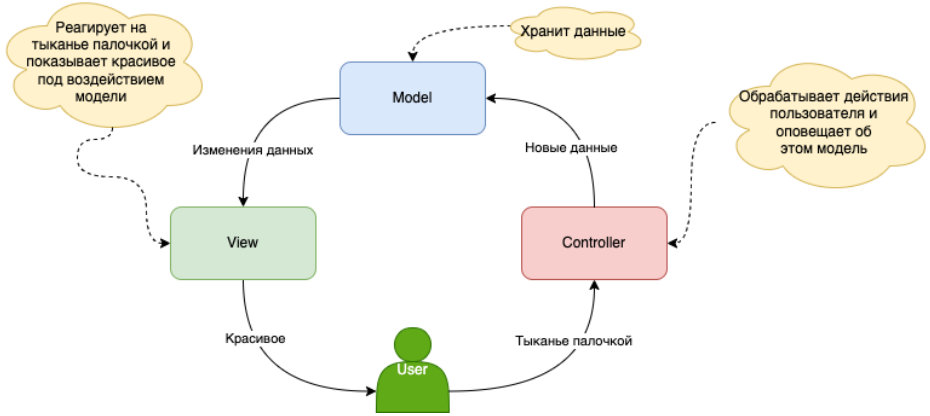

MODEL
Updates
Manipulates
VIEW
INTENT
Sees
Interacts
USER

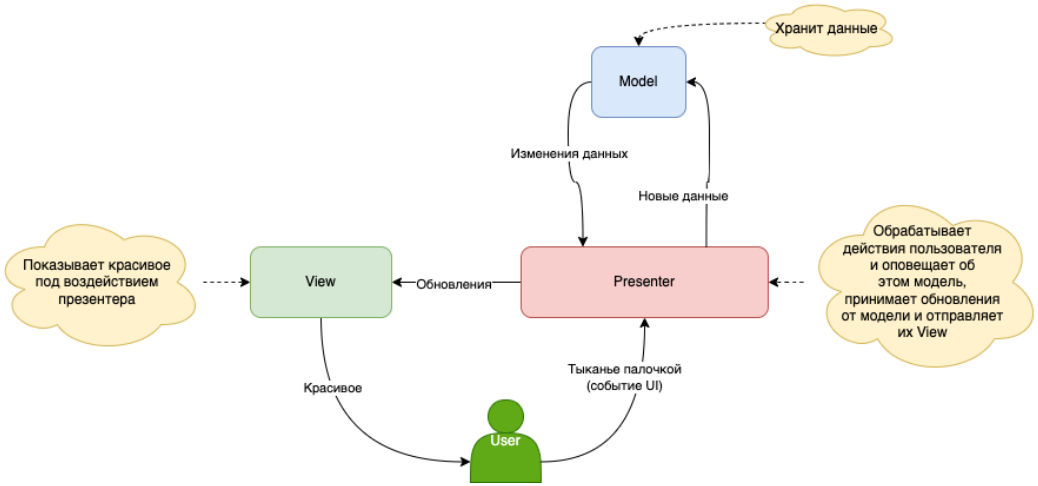

UI events
view(model(intent()))
New model to display
Actions to manipulate model

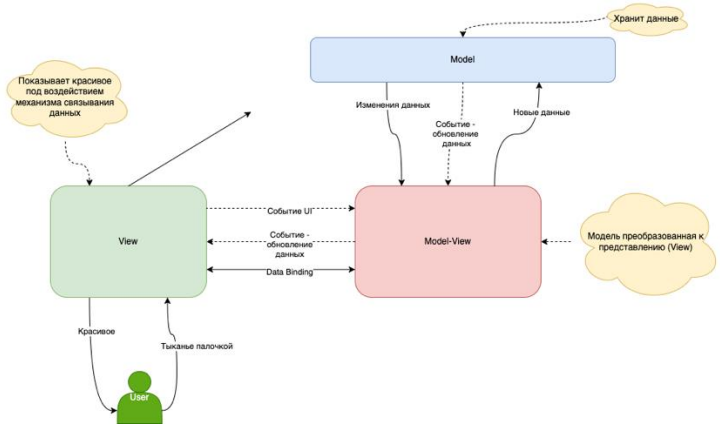
Model View Intent

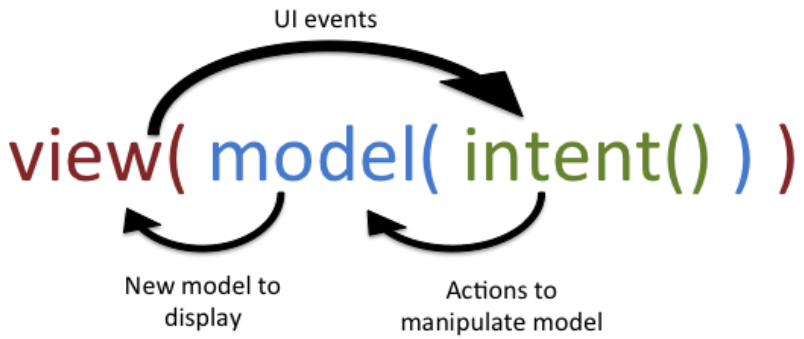
- Хорошо работает с RxJava
- JS
- Jetpack Compose
  - MVICore, MVIKotlin, …
- SwiftUI (Apple)
- См. Почему так удобно использовать паттерн MVI в KMM

Что объединяет эти картинки?

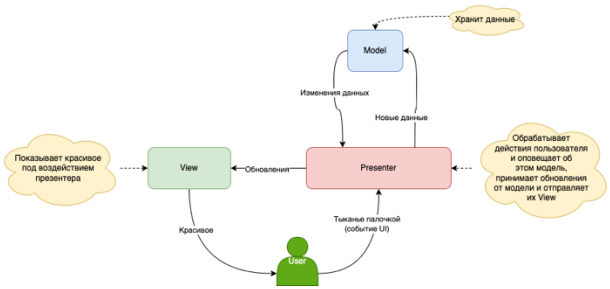


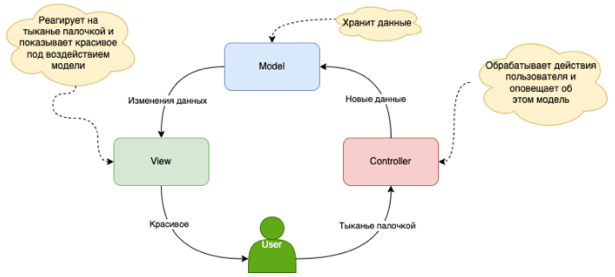






Семантика картинки





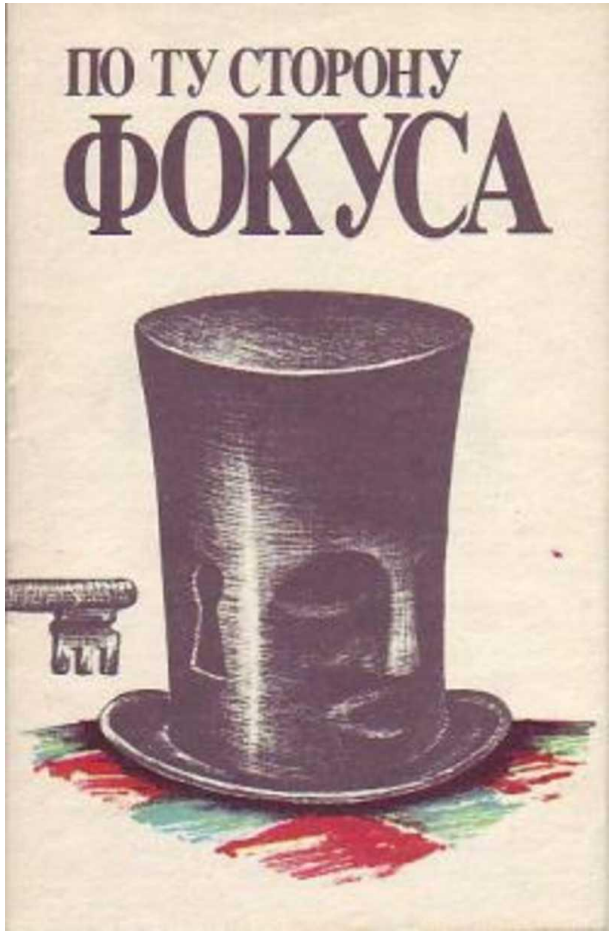
- На этих картинках изображены объекты, которые как-то общаются друг с другом.
- Что такое объект – не сказано (ну наверно, объект противоположность субъекта, но это не точно, так как там часто изображают пользователя)
- Как, когда и каким образом они общаются - непонятно

Суровая реальность

- Архитектура каждой реализации UI зависит от окружения и является разной на разных платформах.
- В реальности, архитектурное описание сложнее, чем это показано в так называемых паттернах UI.
- На досуге попробуйте нарисовать архитектуру UI для Jetpack Compose

Книжка про фокусы, связь между объектами и уровни абстракции

- Представьте, что вы читаете книжку про то как делать фокусы
- В книжке написано «Возьмите тарелку и поставьте ее перед собой на столе»
- Вы берете из шкафа тарелку и ставите ее перед собой на стол
- Автор книги по фокусам, написанной много лет назад заставил вас встать с дивана и поставить перед собой тарелку.
- Какая связь между двумя объектами: автором книги и тарелкой, стоящей перед вами?



Компоненты Android

Основные компоненты Android

- Activity – предназначены для организации связи с человеком
- Content provider – предназначен для обеспечения единообразного, безопасного централизованного доступа к различным ресурсам, таким как медиаресурсы, телефонная книга и т.д.
- Services – предназначены для организации фоновой работы не требующей реакции пользователя.

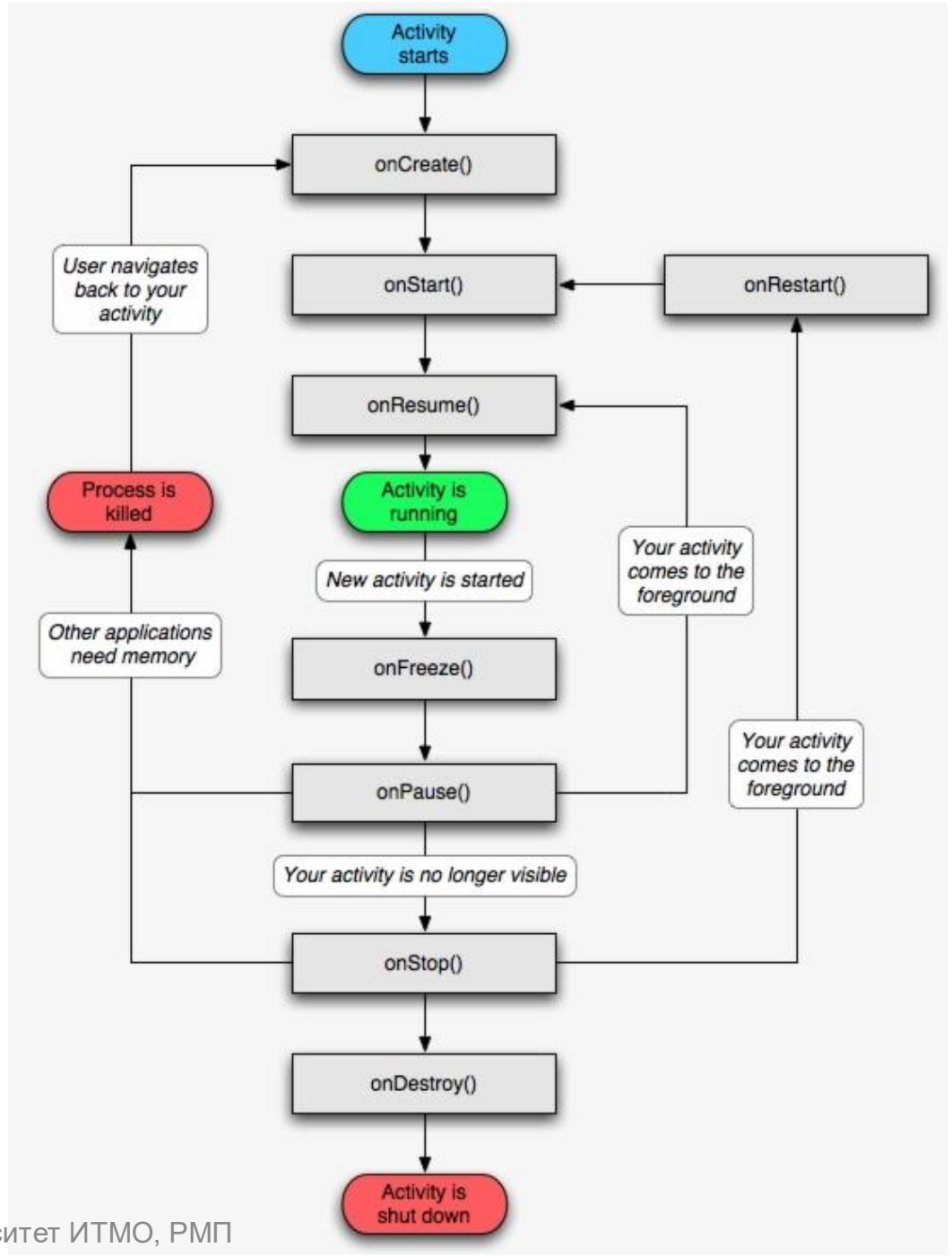
Интерфейс пользователя в Android

- Способ реализации
  - View-based
  - Jetpack Compose
- Дизайн интерфейса – material design

Activity

- Activity - экран, реализующий концепцию UI для Android
- При создании нового Activity мы создаем класс, наследуя от какого-нибудь производного класса Activity, например от AppCompatActivity

Жизненный цикл Activity



Основные методы Activity

- **onCreate()** вызывается при создании Activity.
- **onStart()** вызывается когда Activity отрисована и видима пользователю.
- **onResume()** вызывается перед тем как Activity станет доступна для взаимодействия с пользователем.
- **onPause()** – метод симметричный onResume(). Пользователь больше не может взаимодействовать с Activity, но Activity частично видна пользователю.
- **onStop()** – метод симметричный onStart(). Вызывается, когда Activity больше не видна пользователю.
- **onDestroy()** – метод симметричный onCreate(). Вызывается перед тем, как Activity будет уничтожена системой.

Jetpack Compose

```
MaterialTheme(colors = darkColors()) {
    LazyColumn(modifier = Modifier.fillMaxSize(), state = lazyColumnListState) { this: LazyListScope
        items(count = logText.size) { this: LazyItemScope  it: Int
            if( logText.size > it) {
                Text( text: "$it ${logText[it]}", fontFamily= FontFamily.Monospace)

                composableScope.launch { this: CoroutineScope
                    lazyColumnListState.animateScrollToItem( index: logText.size - 1)
                }
            }
        }
    }
}
```

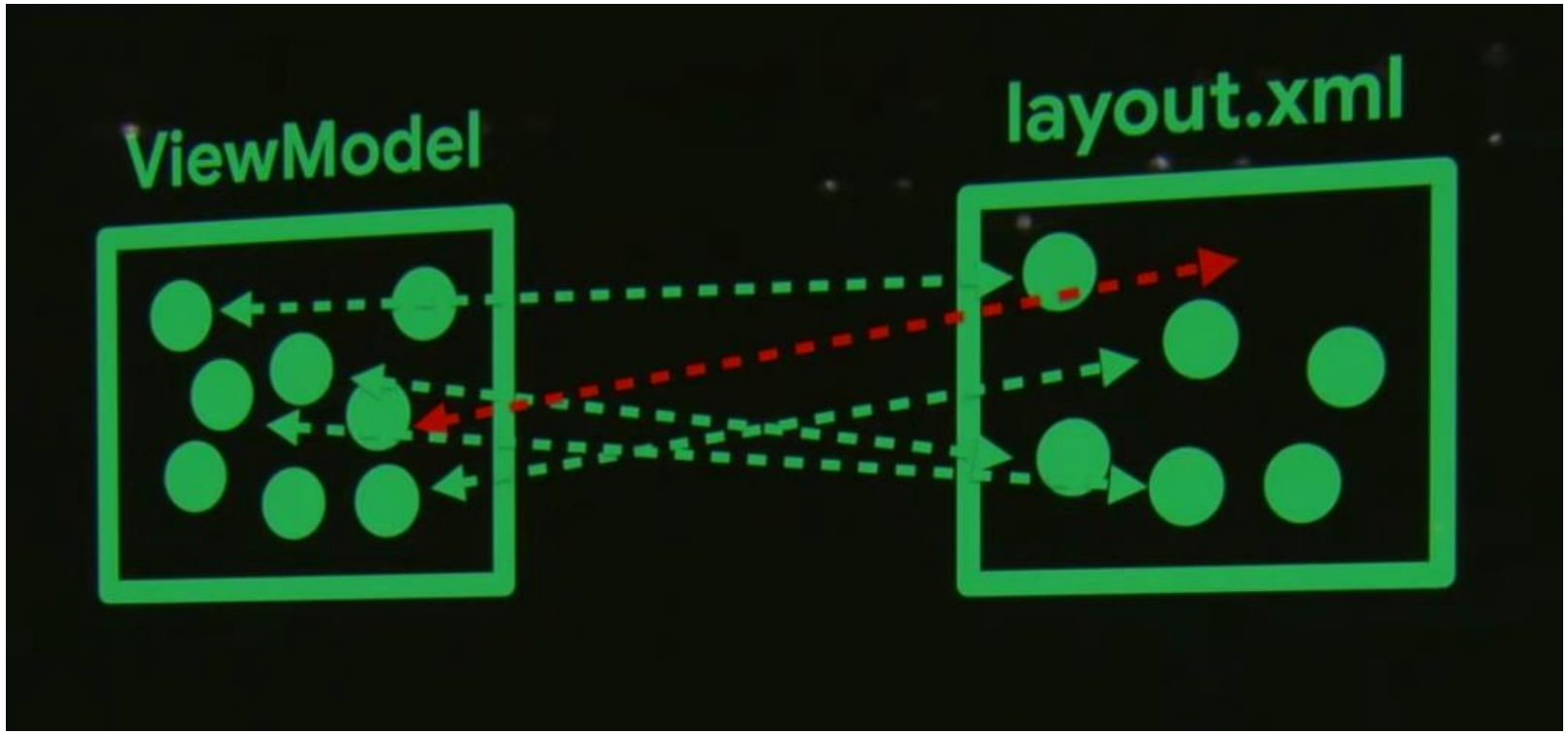
Jetpack Compose

- Декларативный UI фреймворк
  - Слово «декларативный» означает, что в первую очередь мы делаем описание того, что должен делать интерфейс, а не как этого добиться.
  - Вместо физического удаления, добавления или изменения правил видимости компонентов мы описываем каким будет интерфейс при определенном состоянии. Функция может быть представлена или нет в композиции с помощью банального оператора ветвления (if, else).
- Есть версия Compose Multiplatform (iOS, Android, Desktop, Web)
  - Я активно использую десктопный UI на базе Jetpack Compose Multiplatform для реализации системы интеграционного тестирования бэкэнда на базе микросервисов. С точки зрения размера порога вхождения это оказалось достаточно просто, хотя там есть моменты не очень понятные и однозначные (например, каким образом заменить Android Intent, чтобы обновлять данные в разных окнах).

Причины разработки Jetpack Compose

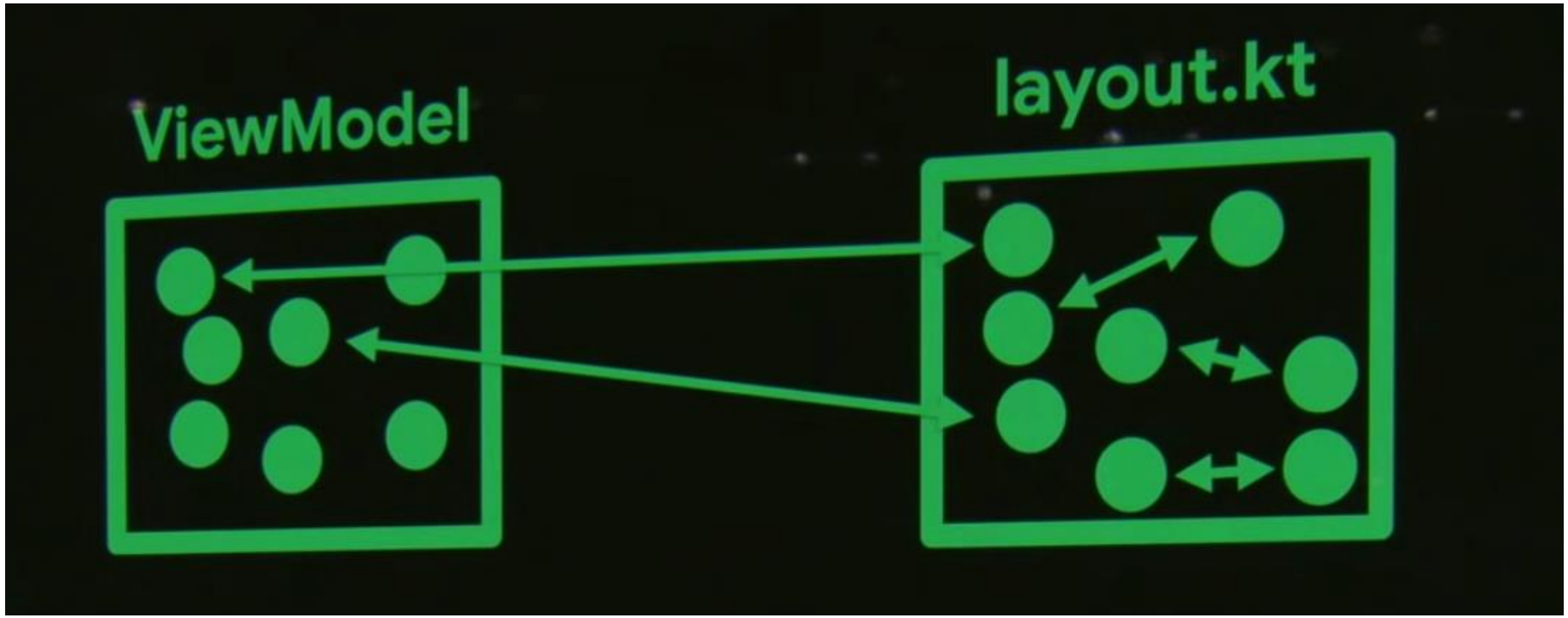
- **Независимость от платформы**. Jetpack Compose не зависит от новых релизов Android, iOS, Mac OS X, Linux или Windows.
- **Минимизация стека технологий** приводит к ускорению процесса обучения, простоте кода, меньшему количеству ошибок. В старой системе есть разделение на ViewModel и XML Layout (поиск findViewById и т.д.), происходит описание задачи на принципиально разных языках Java/Kotlin и XML, при использовании динамического создания компонентов отображения получается жуткая мешанина.
- **Простое управление состояниями и обработкой событий.** Нет ручного обновления.
- **Меньшее количества кода в проекте.**
- Уменьшение количества ошибок.

Старое описание UI на двух языках: Java/Kotlin и XML



[Understanding Compose (Android Dev Summit '19)]

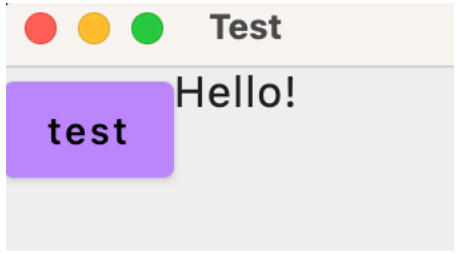
Jetpack compose

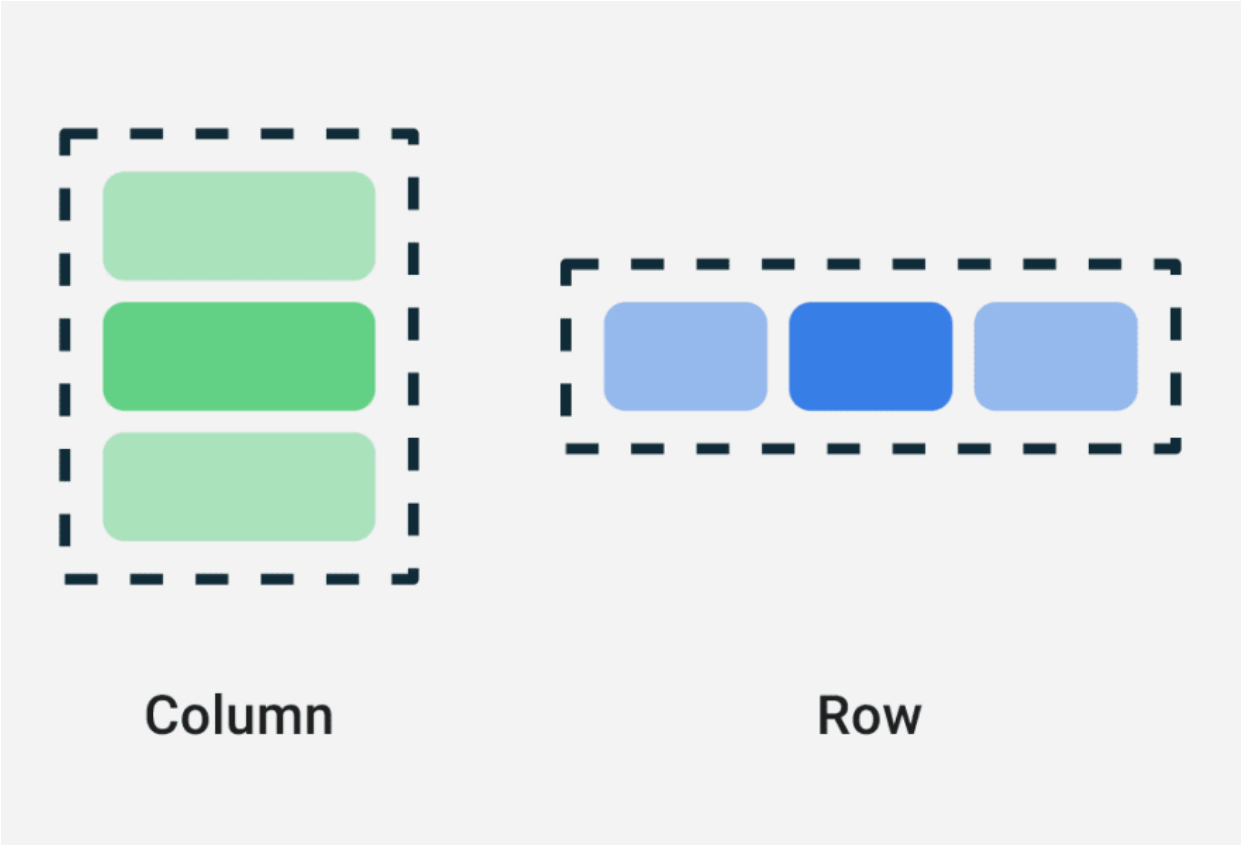


[Understanding Compose (Android Dev Summit '19)]

Группировка элементов

```
@Composable
@Preview
fun  screenTest() {
    MaterialTheme(colors = darkColors()) {
        Row { this: RowScope
            Button( onClick = {
                }) { this: RowScope
                Text( text: "test")
            }
            Text( text: "Hello!")
        }
    }
}
```





В чем проблема?

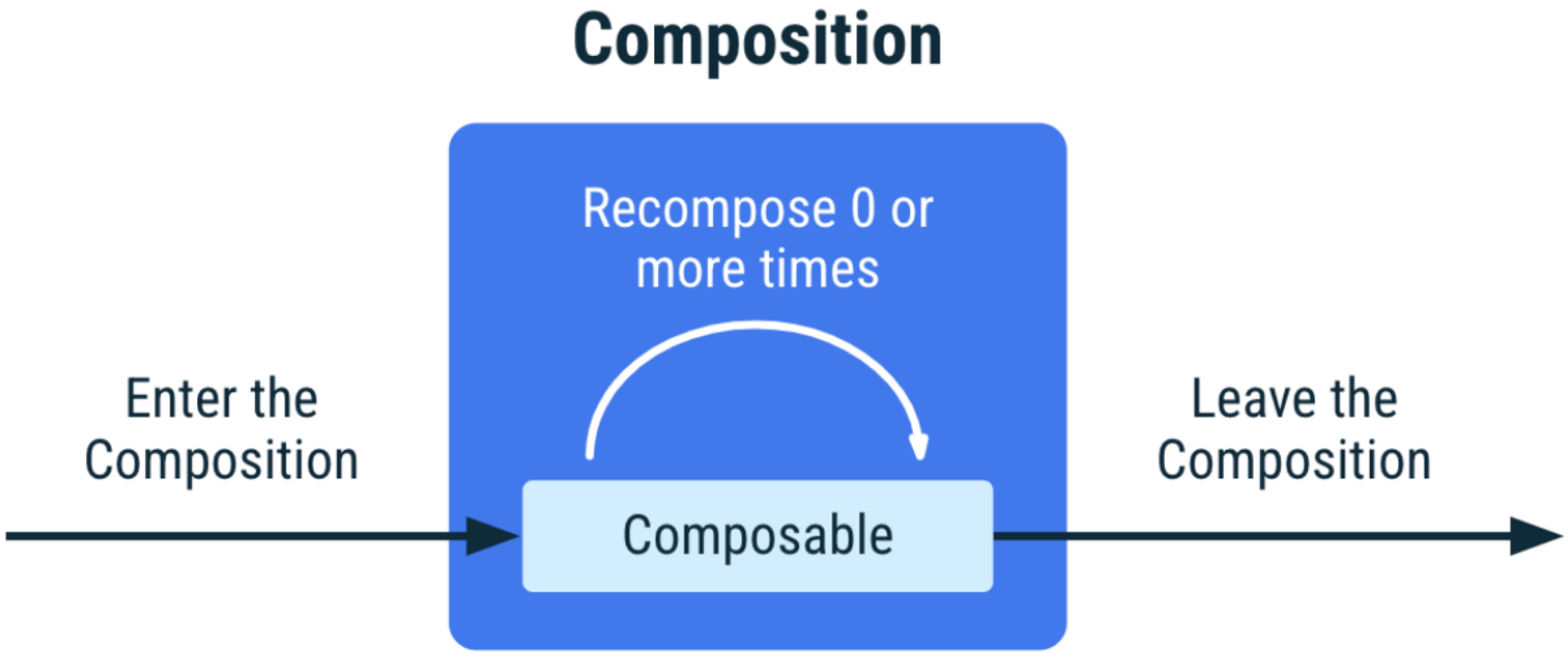
- Мы пишем код на одном языке Kotlin и нам нужно приложить некоторые усилия, чтобы не перемешать макет (layout) и логику приложения.
- К счастью фреймворк Jetpack Compose написан достаточно хорошо и разделять макет и логику было достаточно просто.
- В качестве бонуса у нас появляется возможность делать динамические макеты, которые создаются «на лету» с использованием привычных нам условных ветвлений, циклов и встраивания вложенных @Composable функций.

Что такое композиция?

1. Композиция (Composition): описание пользовательского интерфейса, построенного Jetpack Compose, созданного путем выполнения composable-функций.
2. Изначальная композиция (Initial Composition): первоначальное создание композиции, т.е. когда Jetpack Compose выполняет composable-функции в первый раз.
3. Перекомпозиция (Re-composition): повторный запуск или выполнение composabl’ов с целью обновления композиции.

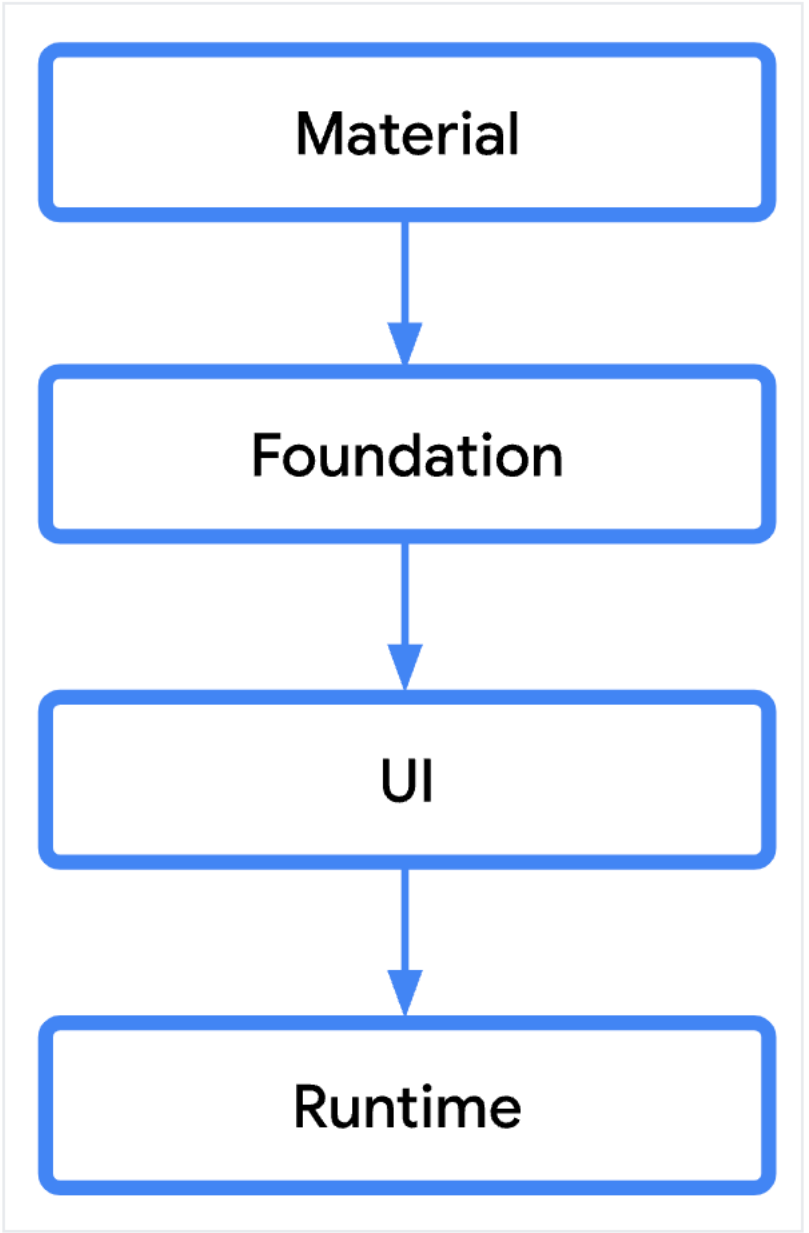
- См. Погружаемся в Compose-Verse — руководство по Jetpack Compose для начинающих: управление состоянием
- Understanding Compose (Android Dev Summit '19)

Композиция


Composition
Recompose 0 or more times
Enter the Composition
Composable
Leave the Composition

Уровни Jetpack Compose

- Runtime - предоставляет основы среды выполнения Compose, такие как remember, mutableStateOf аннотации @Composable и т.д.
- UI – ui-text, ui-graphics, ui-tooling, LayoutNode и т.д. Поля для ввода, вывод графики и т.п.
- Foundation - Row, Column и т.д. Организация элементов на экране.
- Material – темы, стили Material Design&



Как устроено приложение Jetpack Compose?

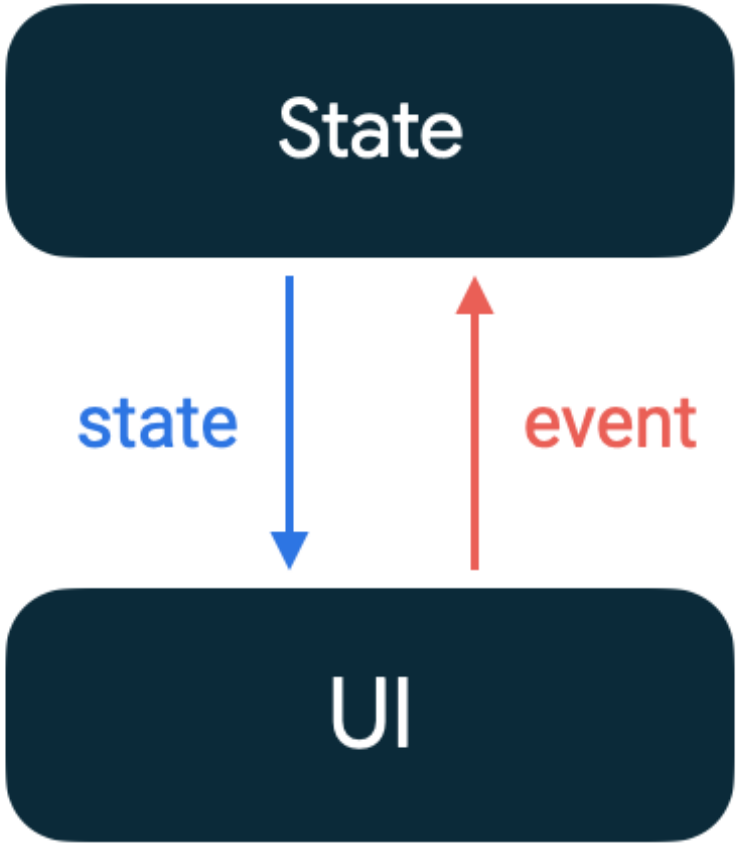
- Внутри приложение представляет собой конечный автомат (FSM), у которого как обычно есть состояния и переходы между ними.
- Переход между состояниями осуществляется под воздействием событий.
- Рекомпозиция (или перекомпоновка) происходит когда нужно показывать пользователю новую информацию, если состояние изменилось.
- Например, чтобы текстовое поле отобразило новое значение, к этому полю нужно привязать специальную переменную, состояние которого будет отслеживаться.
- Для наблюдения за состояниями используются интерфейсы State и MutableState

Что мне напоминает Jetpack Compose?

- Вы читали книжки про магию? Да, это оно.
- Магия, это когда берется стройная картина мира и в этой стройной картине мира для удобства делается дыра или она (картина мира) складывается пополам, потому что это удобно программисту.
- Это как открыть дверцу шкафа, войти в него и выйти через 1000 км от вашего дома чтобы не тратиться на авиа-билеты, когда вам нужно ехать в командировку. Так удобнее, не правда ли?
- Удобно, но не очень понятно и не всегда логично. Главное понять, что дверь в ваш офис в другом городе находится почему-то в шкафу…
- Для понимания этих странностей документации явно недостаточно, мне пришлось лезть в исходные тексты. В документации написано, что дверь в шкафу это само-собой разумеется и это нормально, у всех есть такая дверь (у меня вот нет такой двери, наверно я плохо себя вел). Почему дверь находится именно в шкафу – никто, включая авторов Jetpack Compose, не рассказывает.
- Расширения языка Kotlin с помощью аннотаций и сладкий синтаксический сироп глубиной 5 километров затрудняют понимание семантики конструкций. Но это плата за универсальность языка и его расширяемость. Другими словами язык в основе которого лежит императивная модель Фон-Неймана не очень соответствует решаемой задаче и поэтому приходится придумывать различные костыли (там под «капотом» все состоит из костылей).
- Зато если вы выучили (и приняли как данность) все эти «заклинания», программирование станет для вас легким и удобным.

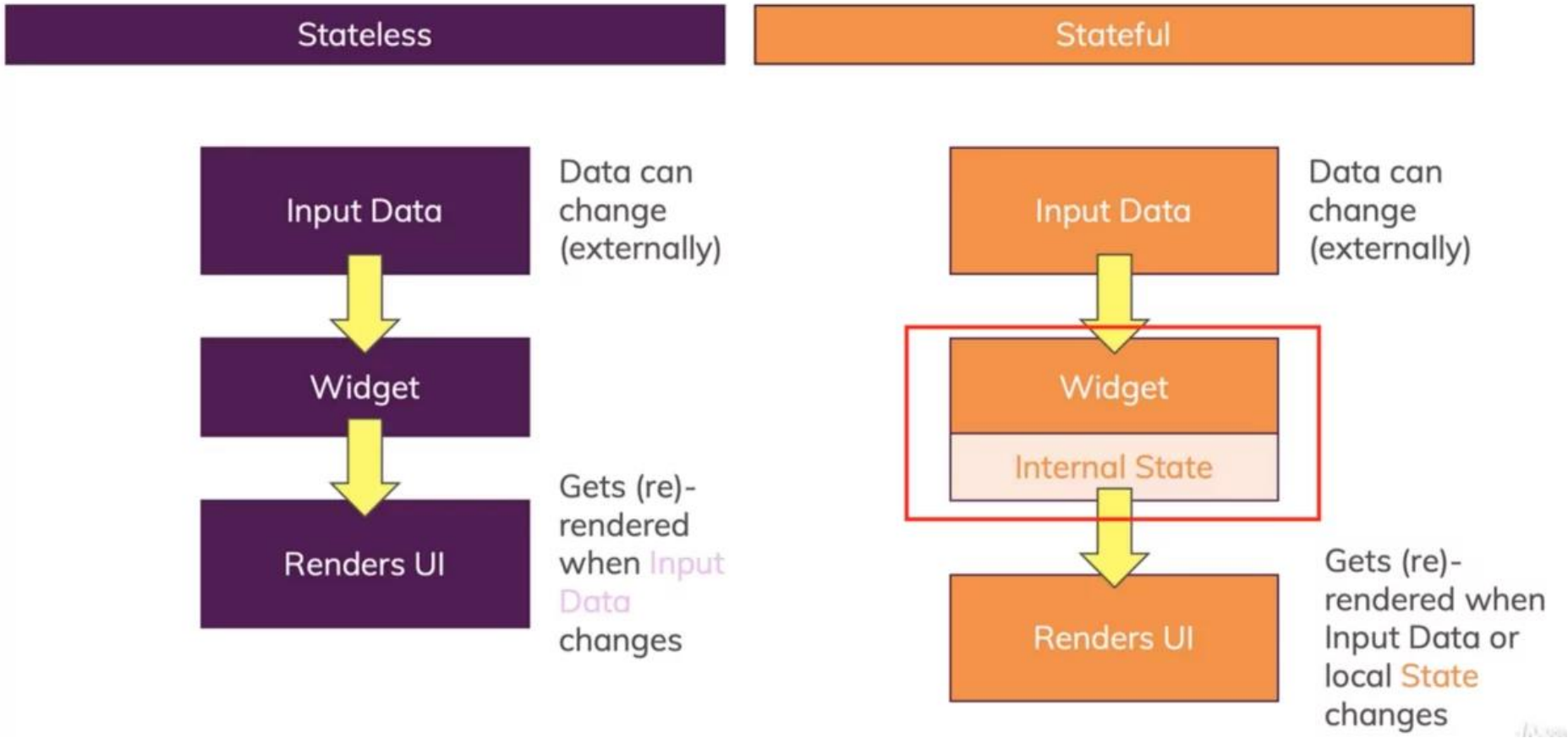
Состояние composable объектов

- Состояние лучше хранить где-то отдельно, наверху и передавать его вниз.
- События от UI должны подниматься наверх.



[How to handle state in Jetpack Compose]

Stateless vs Stateful



[How to handle state in Jetpack Compose]

Компоненты с отслеживанием состояния (Stateful)

- Компоненты с сохранением состояния имеют внутреннее состояние, которое может изменяться в течение срока службы компонента.
- Эти компоненты создаются с использованием аннотации @Composable и ключевого слова (функции) ***remember***.
- ***remember*** используется для объявления переменной, которая может быть обновлена внутри компонента, и ее значение будет сохраняться при повторных составлениях.

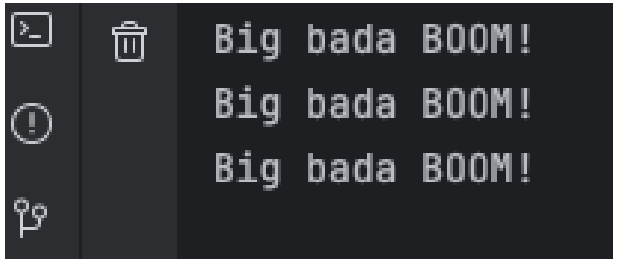
Компоненты без состояния (Stateless)

- Компоненты без сохранения состояния не имеют никакого внутреннего состояния и являются функцией от входных данных. Эти компоненты также создаются с использованием аннотации @Composable, но в них не используется ключевое слово ***remember***.
- Примерами компонентов без сохранения состояния являются кнопки, текстовые надписи и иконки.

Stateless

```
@Composable
fun Button1(txt : String, onClick: (String) -> Unit ) {
    Button( onClick = {
        onClick("Big bada BOOM!")
    }) { this: RowScope
        Text(txt)
    }
}
```


```
@Composable
@Preview
fun  screenTest() {
    MaterialTheme(colors = darkColors()) {
        Row { this: RowScope
            Button1( txt: "test1") { it: String
                println(it)
            }
```

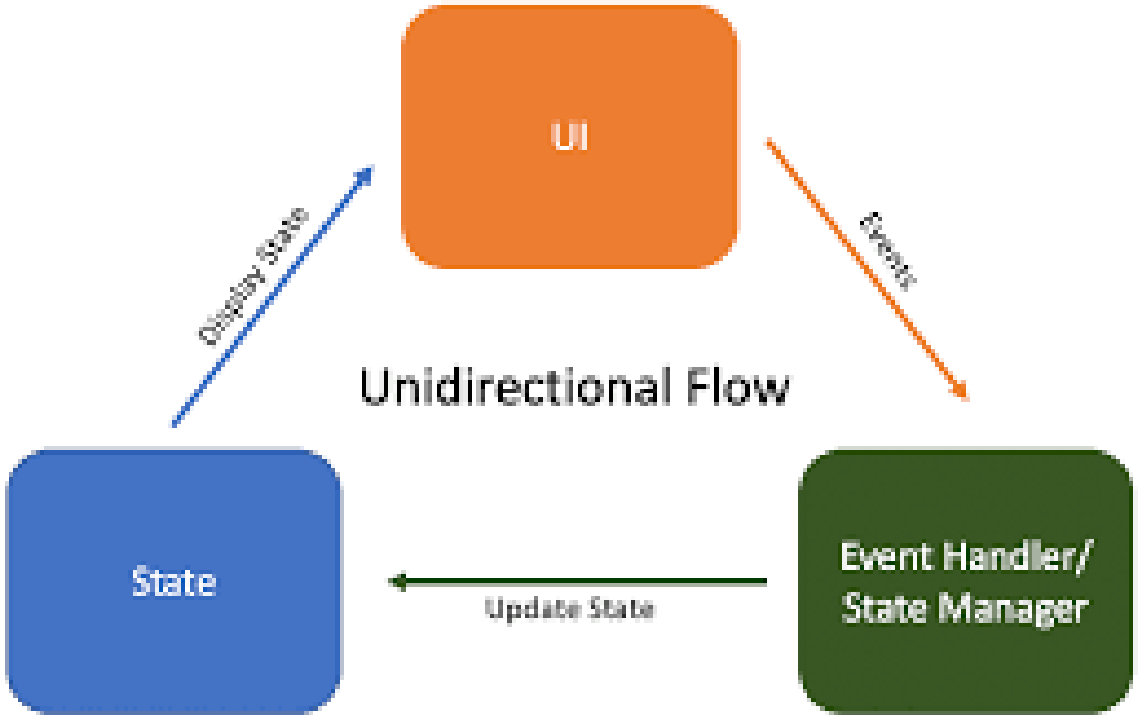


Stateful vs Stateless

- Компоненты с сохранением состояния могут обновлять свое внутреннее состояние, в то время как компоненты без сохранения состояния - нет.
- Компоненты с отслеживанием состояния полезны для создания сложных пользовательских интерфейсов, требующих динамического поведения, такого как обработка пользовательского ввода и реагирование на изменения в данных.
- Компоненты без сохранения состояния, полезны для создания простых пользовательских интерфейсов, которые не требуют какого-либо динамического поведения.

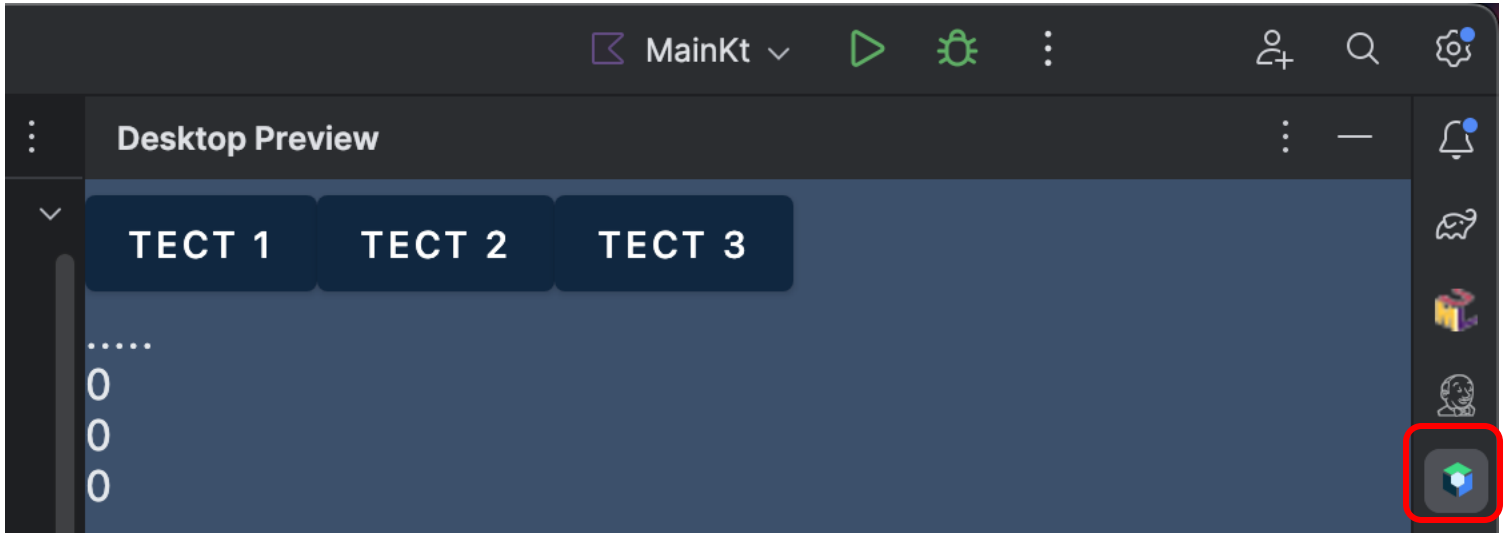
Однонаправленный поток (UDF)

- В шаблоне однонаправленного потока данных данные передаются в одном направлении через пользовательский интерфейс, от компонуемой функции верхнего уровня вниз к компонуемым функциям низкого уровня.
- Данные передаются вниз по иерархии компонуемых функций в качестве параметров функции, любые изменения в пользовательском интерфейсе инициируются изменениями этих данных.



@Preview – предварительный просмотр компонента

```
@Composable
@Preview
fun screenTest() {
    var screenState by remember { mutableStateOf(ScreenState( state: ".....")) }
    var counter1 by remember { mutableStateOf( value: "0") }
    var counter2 by remember { mutableStateOf( value: "0") }
    var counter3 by remember { mutableStateOf( value: "0") }
```



Хранение состояния в Jetpack Compose

- var name **by** **remember** { **mutableStateOf**("")}
- val logText = ***remember*** **{** ***mutableStateListOf***<String>() **}**
- val composableScope = ***rememberCoroutineScope***()
- val lazyColumnListState = ***rememberLazyListState***()

- *Что такое **by**? **by** это про делегирование. Например, val говорит нам о том, что у переменной есть геттер, var говорит нам о том, что у переменной есть геттер и сеттер, by говорит нам о том, что геттер и сеттер нам предоставляет кто-то другой, в данном случае функция **remember**. Шаблон делегирования может использоваться как альтернатива наследованию (Kotlin же не поддерживает множественного наследования).*
- *Для наблюдения за состоянием (переменная value внутри State) используется механизм подписки.*

Функция remember

- Так как отображение элемента интерфейса (например текстового поля или кнопки) производится с помощью специальной функции @Composable, необходимо где-то хранить состояние. Дело в том, что при вызове функции композиции (@Composable) элемент будет отрисовываться по новому и если состояние этого элемента нигде не хранить, то мы каждый раз будем видеть значение по умолчанию.
- Функция remember нужна чтобы возвращать в функцию композиции значение после каждой перерисовки.
  - val mutableState = remember { mutableStateOf(значение) }
  - var value by remember { mutableStateOf(значение) }

remember

```
package androidx.compose.runtime

  Remember the value produced by calculation. calculation will only be evaluated during the
  composition. Recomposition will always return the value produced by composition.

@Composable
inline fun <T> remember(crossinline calculation: @DisallowComposableCalls () -> T): T =
    currentComposer.cache( invalid: false, calculation)
```

- @DisallowComposableCalls – защита от повторного вхождения
- currentComposer.cache – собственно та штука, которая хранит данные

Почему нужна функция remember?

- Экран перерисовывается постоянно при изменении данных (и вообще по любому чиху, например, когда вы свернули окно), что вызывает создание всех локальных переменных функций Composable переменных по новой.
- Remember вместе с MutableSet обеспечивает нечто типа статических переменных в контексте одной Composable функции

Создание компонентов

- name – текст на кнопке
- command – команда, передаваемая в обработчик
- executor – обработчик события onClick (чтобы не загромождать код компонента). Вместо команды command можно просто подсовывать разные классы на базе интерфейса IButtonExecutor.

```
@Composable
fun TestButton(name : String, command : String,
               executor : IButtonExecutor,
               onClick: (String) -> Unit) {
    Button( onClick = {
        onClick( executor.start(command) )
    }) { this: RowScope
        Text(name)
    }
}
```

```
interface IButtonExecutor {
    fun start(command : String) : String
}
```

Передача события наверх через функцию обратного вызова (callback)

```
@Composable
fun TestButton(name : String, command : String,
               executor : IButtonExecutor,
               onClick: (String) -> Unit) {
```

```
    Button( onClick = {
        onClick( executor.start(command) )
```

```
TestButton( name: "ТЕСТ 1",  command: "test1", TestButtonExecutor(), onClick = { screenState = screenState.copy(it) })
```

Состояние экрана

```
data class ScreenState(var state : String)
```

```
@Composable
@Preview
fun  screenTest() {
    var screenState by remember { mutableStateOf(ScreenState( state: ".....")) }
    var counter1 by remember { mutableStateOf( value: "0") }
    var counter2 by remember { mutableStateOf( value: "0") }
    var counter3 by remember { mutableStateOf( value: "0") }
```

Работа с очередями, rememberCoroutineScope

```
@Composable
fun screenDispatcher(onEvent: (String) -> Unit, screenDispatcher : IScreenDispatcher ) {
    val composableScope = rememberCoroutineScope()
    var started by remember { mutableStateOf( value: false) }

    if( !started ) {
        started = true
        composableScope.launch { this: CoroutineScope
            screenDispatcher.start(onEvent)
        }
    }
}
```

```
class ScreenDispatcher(val initialValue : Int = 42, val period : Long = 1000) : IScreenDispatcher {
    override suspend fun start(onEvent: (String) -> Unit) {
        var counter = initialValue
        while(true) {
            delay(period)
            onEvent(counter++.toString())
        }
    }
}
```

Спасибо за внимание!