Волновая архитектура на практике

Let's make Mobile Development great again! 🧢

Юрий Дубовой | Mobius 2025 Autumn

Искромётная shoot-ка

никаких шуток

Искромётная shoot-ка

О докладе и докладчике

- Помимо ПК Мобиус участвую в ряде проектов, в основном в роли архитектора. 15 лет пишу коммерческий код руками, потому что истина — в коде =)
- Самым визуально наглядным проектом является Делимобиль, который я развивал на дистанции почти в 4 года. Накоплена объемная статистика и практический опыт, поэтому и в докладе приводятся примеры из сферы каршеринга.
- Приложение Делимобиль: 7 лет истории коммитов, 200 экранов, 400k строк кода, 100%-е покрытие бизнес-логики тестами, crash-free 99.99%
- Так было не всегда: на определенном этапе проект был в режиме «из 10 коммитов 9 с пометкой bugfix». Для мобильных приложений этот порог — примерно 100k строк кода. Эдакая долина смерти. Мы сумели ее пройти и сегодня я поделюсь опытом решения проблем мобильной разработки =)

~~42~~ 2 проблемы мобильной разработки

- Mobile Development в целом – это про извлечь данные и вывести их на экран: «*JSON парсим, кнопки красим!*».
- Два (2!) вопроса:

  1. Где размещать/выполнять сетевые запросы?
  2. Как готовить/тестировать UI?

- Эти два вопроса связаны. Докажем. Закройте глаза, представьте свой архитектурный стек. Сетевые вызовы — кто точно знает, где их размещать/выполнять? А кто точно уверен, что они будут там же в новом проекте на новом месте работы?
- Кому фичи приносят в виде детализированных спецификаций? А кому в виде Figma-макетов и портянки текста ~~как в этом докладе~~ в Confluence?
- Все просто: мы, человеки, любим наглядность, ибо 80% инфы получаем визуально.

~~42~~ 2 проблемы мобильной разработки

- Мы все совершаем одну и ту же mindset-ошибку: мыслим экранами. Экран — это статичная структура. Как диаграмма классов в UML — для первичного погружения достаточно, но для создания динамической, т.е. действующей системы потом нужна
еще и диаграмма активности и/или диаграмма последовательности.

  1) Как теперь доказать, что закодированное соответствует нарисованному
  в разных контекстах?
  2) Как доказать, что запросы выполняются в правильном порядке?

- Архитектура: хотим, чтобы была наглядная, и вынуждены, чтобы была тестируемая. Проблем вообще всегда только две:

«*There are 2 hard problems in computer science:*
cache invalidation, naming things, and off-by-1 errors.»

Leon Bambrick

Декомпозиция против Дракона Сложности

- Дракон Сложности — часть природы мира, и с ним бесполезно сражаться, его невозможно приручить. А если и кому и удастся, тот сам станет Драконом. Дракон Сложности живет в состоянии, потому что именно в состоянии хранится контекст. Для управления и описания используем машину состояний.

- Всякая логическая развилка в машине состояний удваивает минимально необходимое количество тестов. Но ведь, можно же протестировать только тот участок, что изменился? Верно, но для этого нужно как-то вырезать этот участок, чтобы протестировать его отдельно!

- Выход: декомпозировать большую машину состояний на части поменьше, в каждой из которых квадратичный закон не успел набрать мощь.

Декомпозиция против Дракона Сложности

- У нас снова два (2!) вопроса:

  1. Где граница разумной декомпозиции?

  2. Как протестировать согласованную работу разделенных частей? Как готовить/тестировать UI?

- Для ответа вспомним, что правильная ИС — это отражение бизнес-процессов. А они меняются вслед за рынком, поэтому и мы должны быть готовы в любой момент дополнительно декомпозировать или объединить части. Сохраняя при этом наглядность и тестируемость.

- Ключ — в самоподобии. Хорошая архитектура: наглядная, тестируемая и **фрактальная**. Мы декомпозируем состояние не в плоскую, а в иерархическую структуру.

- Как это сделать? Кое-кто считает, что мобильная разработка — это тот же backend, только 10 лет назад. Ок, заглянем по ту сторону API Gateway в мир бекенда.

Фрактальные (микро) сервисы

- Представим REST-запрос от мобильного приложения к API Gateway: один запрос под капотом порождает целый каскад обращений к микросервисам. Обращения распространяются в вычислительном пространстве от одного сервиса к другому, подобно волне на поверхности озера.

- Исходный бекенд-монолит обычно разбивают на микросервисы, именно чтобы контролировать сложность. Но кто сказал, что за API некоторого микросервиса не скрывается каскад других сервисов? Он может быть вовсе не «микро»!

- Таким образом, сеть микросервисов формирует фрактальную структуру. Каждый из узлов при обработке запроса:
  а) ожидает получить в запросе контекст (параметры запроса), а в ответ на запрос всегда возвращает результат (значение либо ошибку);
  б) скрывает от внешнего мира внутреннее состояние, чтобы изолировать свою сложность.

- Дадим определение такому сервису.

Сервис (CROSService-F)

- Сервис — это интерфейс в вычислительном пространстве, предоставляющий доступ к мутабельному состоянию.
- Мы не знаем, как сервис устроен, но точно знаем, что мы можем что-то ему передать и что-то от него получить. Стейт-машину внутри сервиса мы описываем диаграммой состояний.
- Иерархия сервисов с вершиной в API Gateway дает наглядность, тестируемость и фрактальность, ее мы можем описать структурной диаграммой (классов/компонентов/развертывания).
- Это описание структуры, но мы ведь строим динамическую систему! Где описание правильной последовательности внешних запросов к «главному» сервису — API Gateway? Мы должны сначала зарегистрировать пользователя, а потом позволить ему выполнить поездку, но не наоборот!
- В UML помимо диаграммы классов есть как минимум еще диаграммы активности и/или последовательности!

Где живет бизнес-логика?

- Описание правильной последовательности запросов — это фактически описание причинно-следственных связей между запросами, т.е. описание бизнес-логики.

«There are only two hard problems in distributed systems:
2. Exactly-once delivery
1. Guaranteed order of messages
2. Exactly-once delivery.»

Mathias Verraes

- Логика этих связей содержится не в коде бекенда, она живет в коде фронтенда. В нашем случае — в коде мобильного приложения. Нас учили тому, что ее нужно отделять от провайдеров данных и UI-слоя, чтобы уметь в тестируемость, заменяемость и переиспользуемость слоев.

Последовательность

- Возвращаемся к первому вопросу мобильной разработки: в какой вью-модели, в каком интеракторе/презентере/координаторе они — запросы — будут инициироваться?
- Цепочка запросов — это бизнес-логика, как в таком случае ее вообще отделить от запросов? Очевидного ответа нет, поэтому тут мы ломаем копья о шкуру дракона. Дракона нельзя победить и приручить, но… можно обойти.
- С точки зрения мобильного приложения всякий запрос к серверу — это атомарная операция, поскольку приложение гарантированно получает код ответа (за исключением клиентского таймаута) и ничего не знает про API Gateway и/или каскад микросервисов за ним. Для приложения бекенд — это интерфейс, сервис.
- Дадим определение такой атомарной операции.

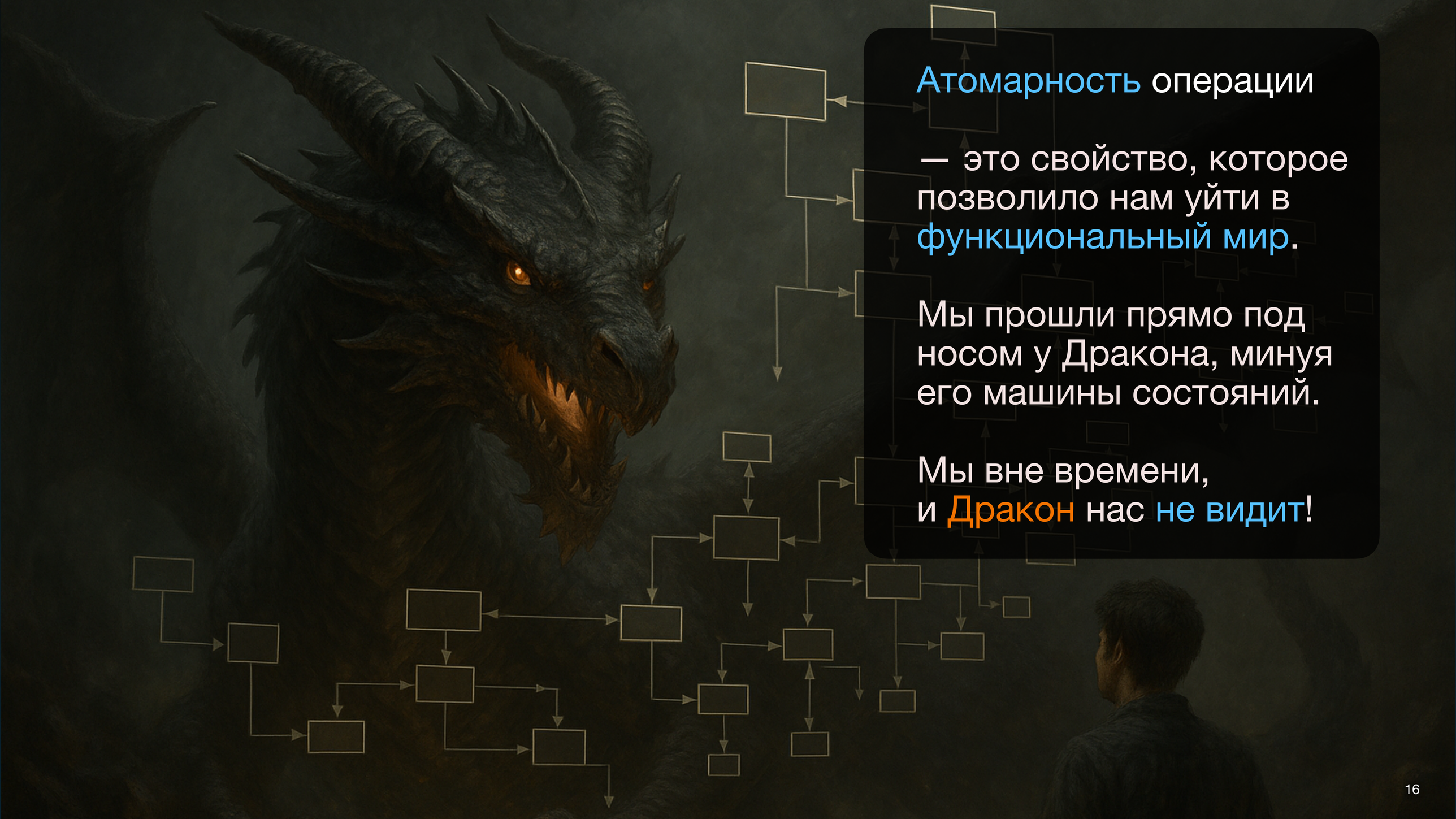
Операция (CROperationSS-F)

- Операция — это атомарный акт доступа к состоянию, гарантированно возвращающий результат.
- Операция похожа на REST-запрос, но не ограничивается только сетевым слоем. Запрос операции может выполняться и к внутреннему хранилищу приложения, и к системной службе геопозиции, и к платежному SDK.
- Операция НЕ похожа на Rx-оператор: у операции нет семантики ошибки. Обработка ошибки – уже логическая развилка, уже код обработки результата.
- Атомарность означает, что результат операции нельзя через if-else-switch обрабатывать внутри другой операции. А где тогда? Цепочка операций и логика их вызова — логика причинно-следственных связей — где они живут?
- Операции живут в сценариях!

Сценарий (CROSScenario-F)

- Сценарий — это функция, которая последовательно и синхронно выполняет вызов и обработку результата для двух и более операций.
- Сценарий, состоящий из одной операции — не сценарий!
- Операция не знает ошибок. Сценарий решает, интерпретировать ли результат как ошибку!
- Следовательно, логические развилки бизнес-логики тоже живут в сценариях.
- Состояние все еще в сервисах, но операции атомарны, поэтому сценарий становится «чистой функцией» вне времени. Легко тестировать.

Атомарность



Атомарность операции

— это свойство, которое позволило нам уйти в функциональный мир.

Мы прошли прямо под носом у Дракона, минуя его машины состояний.

Мы вне времени,
и Дракон нас не видит!

Сценарий (CROSScenario-F)

Ок, бизнес-логика в функциональном мире, что дальше?

1. Подобно операции, сценарий должен завершаться и уметь об этом сообщать. Нужен якорь, нить Ариадны, которая позволит нам каким-то образом вернуться в наш императивный мир.

2. В отличие от операции, сценарий может запрашивать в ходе работы вызов других сценариев. Так мы реализуем фрактальную переиспользуемость.

   Поэтому всякому сценарию для запуска на вход нужно передать контекст.

Контекст (ContextROSS-F)

- Контекст — это структура, которая содержит параметры и функции обратного вызова, необходимые для работы сценария.
- Callbacks вызываются в логических развилках внутри сценария, но определяются снаружи. Это позволяет сценарию вызывать другие сценарии, но ничего не знать ни о сценариях, ни о деталях их реализации.
- Callback-функция *onFinish* является обязательной. Так сценарий уведомляет вызывающую среду о своем завершении.
- А кто именно формирует контекст? Где это происходит?

Контекст (ContextROSS-F)

- Контексты для сценариев формируют контекстные функции.
- В них формируются callbacks и data-параметры, необходимые для работы сценария. Запуск сценария осуществляется там же.
- Контекстная функция — это чистая функция, которая получают на вход фабрику сценариев и... другой, внешний по отношению к этой функции контекст.
- Каждый контекст обязательно предоставляет onFinish, поэтому вместо линейной цепочки вложенных функций мы можем сформировать древовидную фрактальную структуру.

Fractal (CROSS-Fractal)

- Это дерево больше похоже на кактус. У него нет листьев, потому что мы всегда возвращаемся назад.
- Так мы реализуем функциональное программирование с применением чистых нелинейных функций высшего порядка.
  Звучит сложнее, чем выглядит.

Контекстные функции с обязательным onFinish формируют фрактальный «кактус»

- Чистота: если функция высшего порядка A принимает аргумент-функцию B, то A «чистая», если B — тоже чистая. Контекстные функции самоподобны, формируют фрактальную структуру, поэтому они все — чистые.

onFinish

Вся магия — в обязательном коллбеке возврата *onFinish*. Это нить Ариадны, которая позволяет обойти Дракона через функциональное пространство и вернуться назад в императивный мир.

Fractal (CROSS-Fractal)

- А где самая первая, самая главная контекстная функция-патриарх? Эта функция:
  — либо не является чистой,
  — либо вообще без контекста,
  — либо контекст для нее задал кто-то другой, за пределами нашего кода,
  — либо все сразу.

  Которая растет из земли, где корень кактуса. Мы все ее знаем, это...
  функция main!

- Контекст ее запуска формирует OS (аргументы функции main), а фабрикой сценариев для нее служит OS API и стандартная библиотека выбранного ЯП.

- Мы обошли дракона. Для backend-разработки на этом можно было бы и закончить. Но мы должны идти дальше, потому что в мобильной разработке есть второй вопрос — UI.

Кольцо

- Каждый, кто разрабатывал консольные приложения, знает, что при завершении работы функции main завершается и работа приложения. Мы прошли по всем отросткам кактуса и вернулись назад.
- Чтобы приложение не закрылось и работало интерактивно, нам нужен бесконечный цикл, кольцо. Такой цикл есть в любом UI-приложении.
- В мобильных приложениях для iOS это @UIApplicationMain + Run Loop, а для Android это ActivityThread + Looper.

Кольцо

- ОС постоянно регистрирует события всяких датчиков: геопозиция, касания на экране, пакеты данных из сети и т.п.:

- ОС помещает события в буферы данных. Событие (event), записанное в буфер, будем называть артефактом.
- Event Loop на каждой итерации просматривает буферы и извлекает/обрабатывает артефакты.
- Артефакт сразу обрабатывается в теле цикла либо как есть передается в поле того контекста, который далее будет передан в некоторую контекстную функцию.

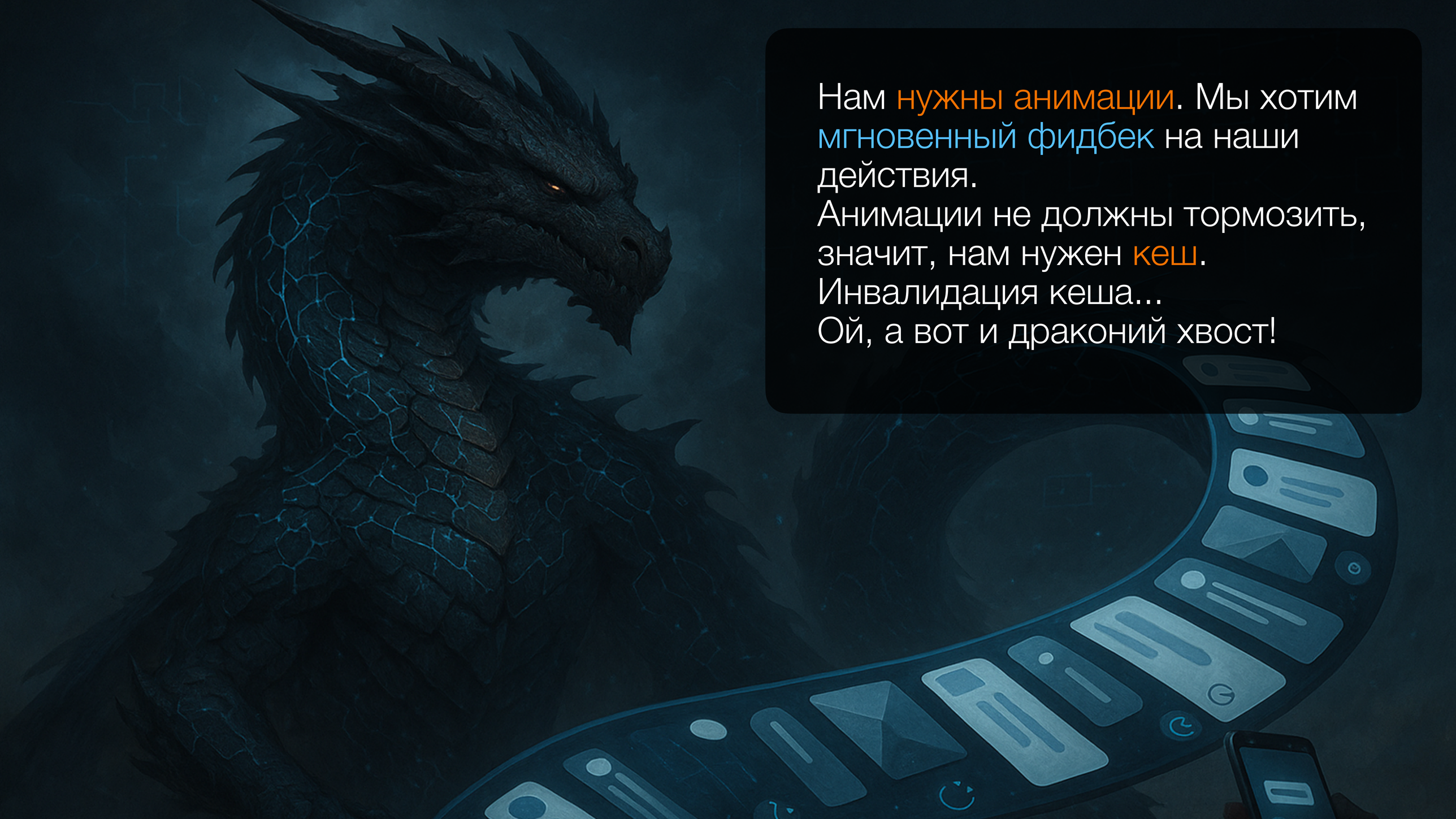
Кольцо

- Касание на экране — тоже артефакт. Но изображение на экране — это состояние. И здесь Дракон приготовил новую ловушку. Мы помним, что его оружие — это состояние. А в UI (User Interaction) очень, очень много состояния!

- Так устроен наш мозг: рубильник нагляднее кнопки, потому что рубильник сигнализирует от своем состоянии, а кнопка — нет.

- Мы, мобильные разработчики мыслим экранами, потому что каждый экран одновременно и service, и цикл обработки касаний.

UI



Нам нужны анимации. Мы хотим мгновенный фидбек на наши действия.
Анимации не должны тормозить, значит, нам нужен кеш.
Инвалидация кеша...
Ой, а вот и драконий хвост!

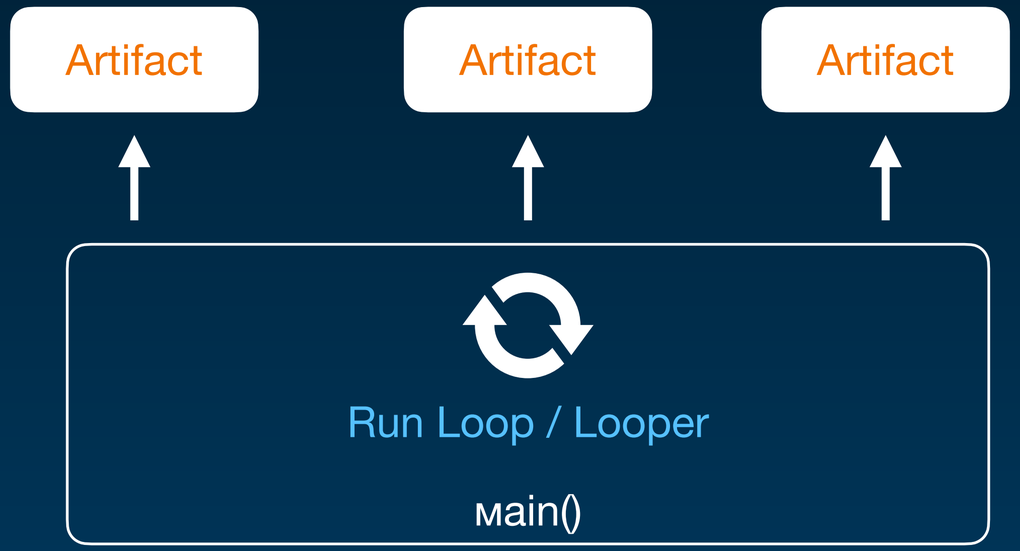
Кольцо

- Итак, UI содержит состояние, а для реализации UI нам нужен бесконечный цикл.
- Как обойти Дракона Сложности?
- Убрать цикл в сервис нельзя, потому что сервис – это интерфейс доступа, а не активный участник.
- Воспользоваться предыдущим способом (уходом в функциональный мир) мы не можем. Контекстная функция должна быть чистой, что исключает использование бесконечного цикла.
- Мы смиряемся с тем, что функция main чистой не является и убираем такой цикл в отдельный объект, который мы будем называть реле.

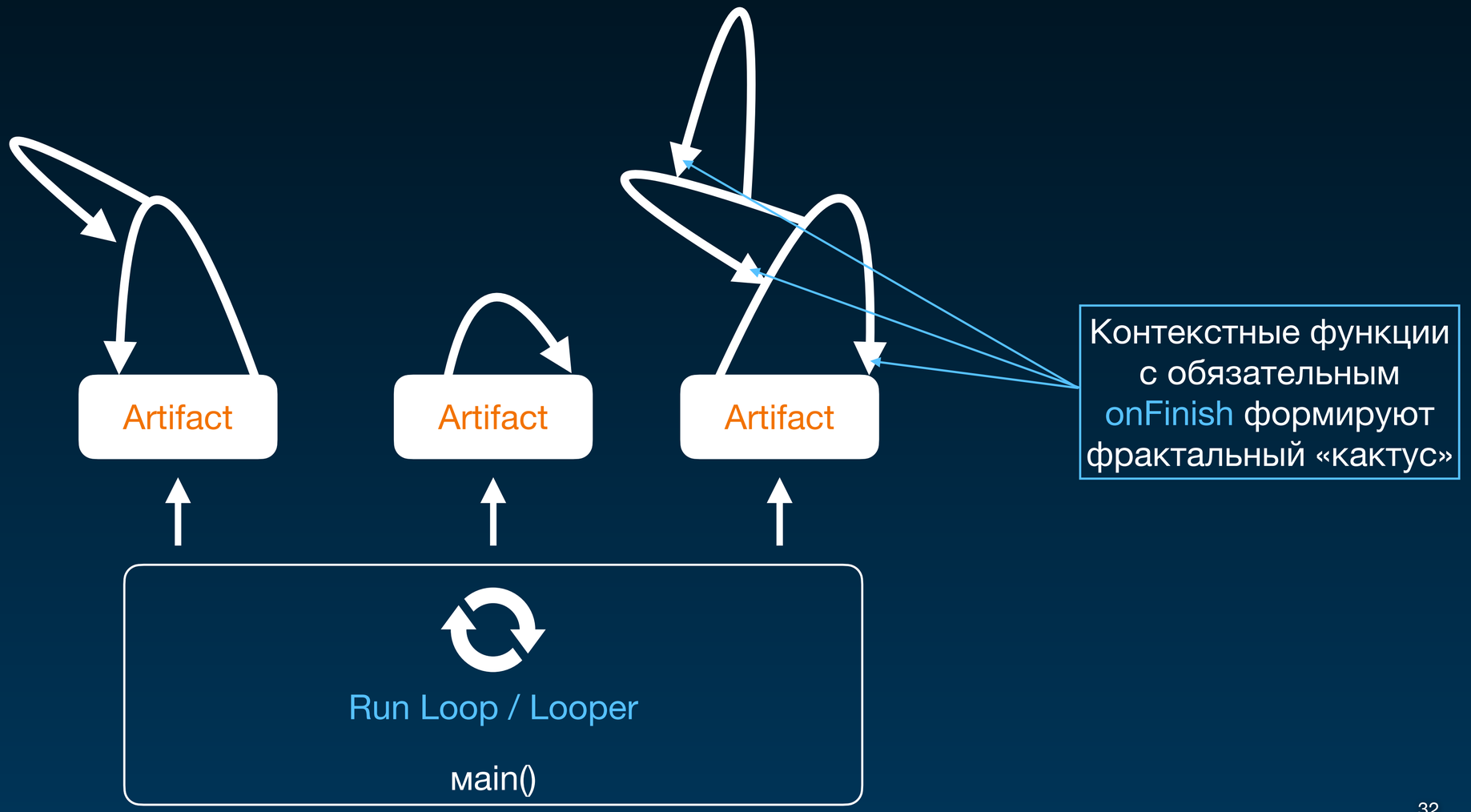
Реле (CRelayOSS-F)

- Реле — это объект, который циклически улавливает поступающие извне артефакты и преобразует их в контексты для сценариев.
- Набор реле формирует «глаза и уши» приложения. Всякое реле так или иначе использует бесконечный цикл, который крутится где-то в API операционной системы и скрыт от глаз разработчика.
- При инициализации/параметризации реле пробрасывает в эти API функции обратного вызова, после чего ожидает поступления событий-артефактов.
- Датчик геопозиции обслуживает LocationRelay.
  Пуш-уведомления обслуживает PushesRelay.
  TCP-сокет обслуживает ChatEventsRelay.
  А экран телефона обслуживает особое реле — UIRelay!

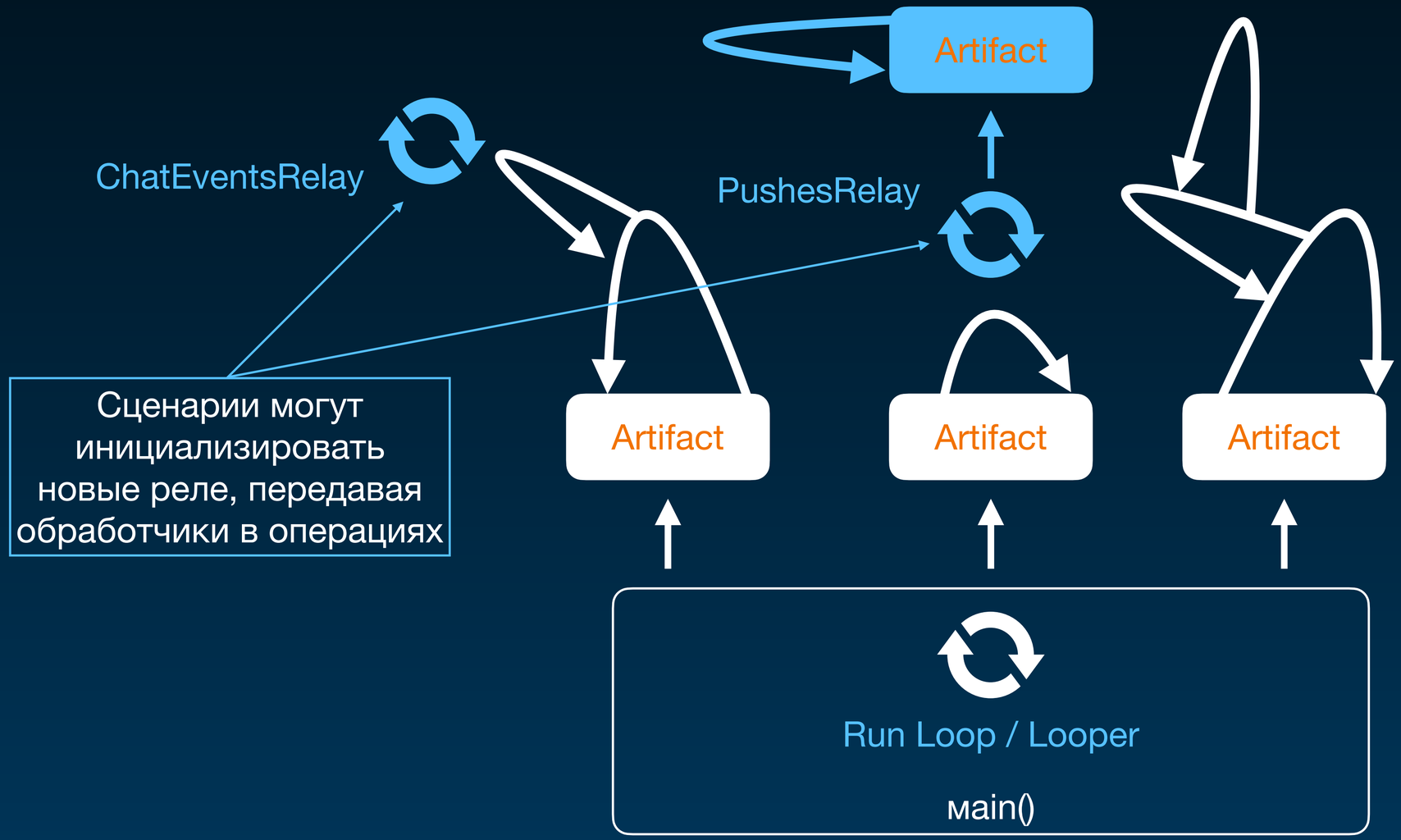
Реле (CRelayOSS-F)



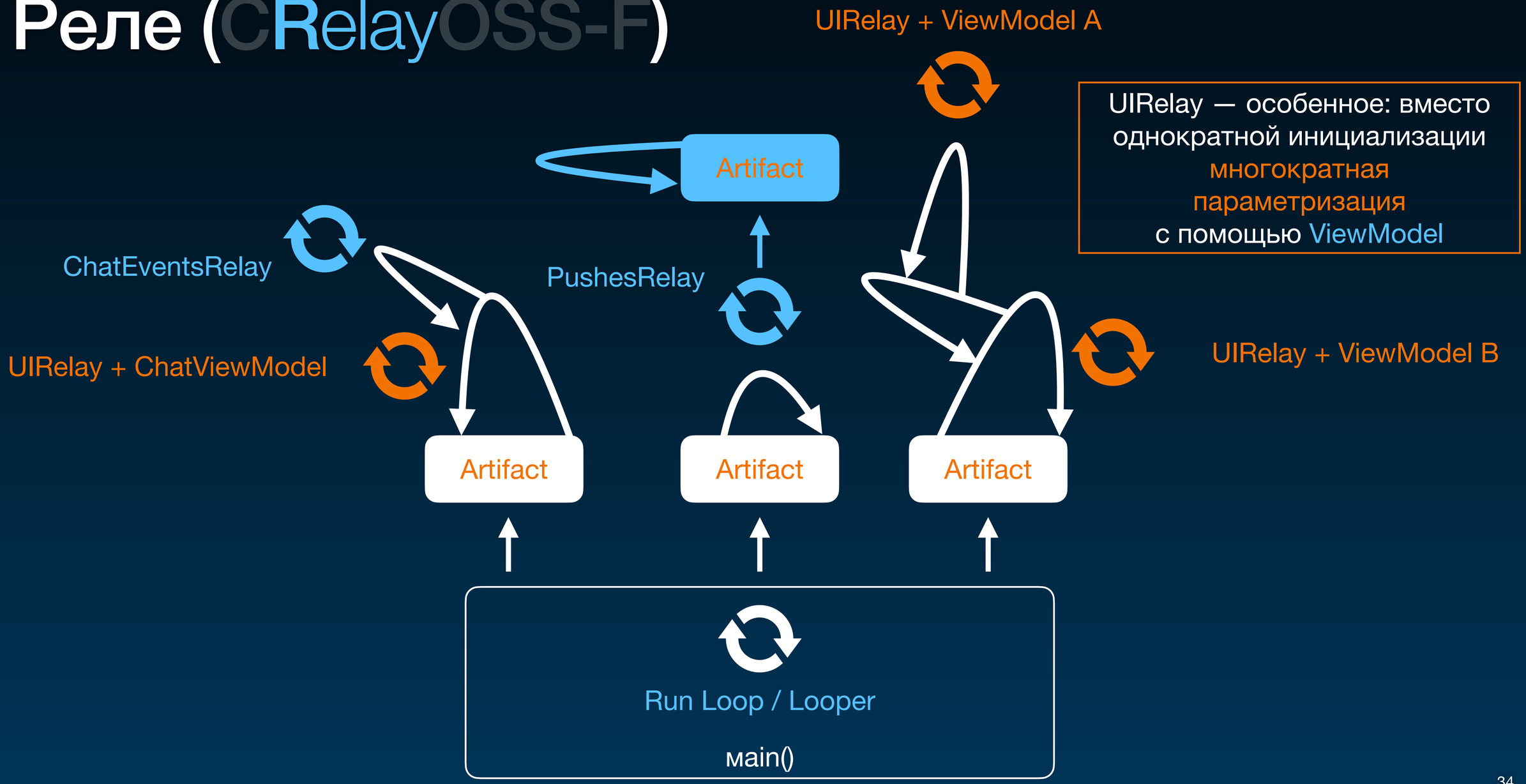
Реле (CRelayOSS-F)



Реле (CRelayOSS-F)



Реле (CRelayOSS-F)



Мобильная разработка в естественной среде обитания

Мобильная разработка 10 лет спустя

CROSS-F — ЭТО:

1. Атомарность операций
2. Обязательный onFinish в контекстах
3. Дуализм UI

CROSS-F. Зачем?

- Зачем нужна ~~еще одна~~ новая архитектура?
- Потому что мир изменился. Мобильная разработка стала взрослой: в стартапах разработчиков все меньше, а в корпоративных супераппах их все больше. За прошедшие полгода тренд только усилился. Большинству из нас придется действовать в условиях корпоративных кросс-функциональных команд.
- Мы теперь знаем, что мобильная разработка — это кактус. Любую веточку кактуса можно легко оторвать, дождаться, пока нарастет пленочка и посадить в другой горшок.
- Горшок, который поливает отдельная команда =) Легче всего отделяется и укореняется ветка, которая сидит на реле.

CROSS-F. Зачем?

- Для крупных проектов все то же самое. Приложение Делимобиль на момент этого доклада насчитывает 429 операций, 104 сценария, 13 реле и около 200 нативных UI-экранов. Разработчики, действующие в разных контекстных функциях, почти не пересекаются в конфликтах на PR/MR в Git.
  Контекстную функцию легко скопипастить и пометить копию как *deprecated*. В нее легко добавить фичефлаг.

- Такая независимость неслучайна, потому что в основана на фундаменте из мира физики. CROSS-F является волновой архитектурой.

Волна

- Волна («Wave») — это каузально упорядоченная последовательность операций в пределах одного локального вычислительного пространства.
- Волна-последовательность порождается артефактом, попадающим в это пространство извне.
- Всякая волна по своей природе не зависит от других волн, поскольку взаимодействует только со средой.
- Вокруг независимых источников артефактов и их волн строим независимые кросс-функциональные команды.
- Независимость команд ➡ горизонтальное масштабирование. Мы можем строить системы произвольной сложности.

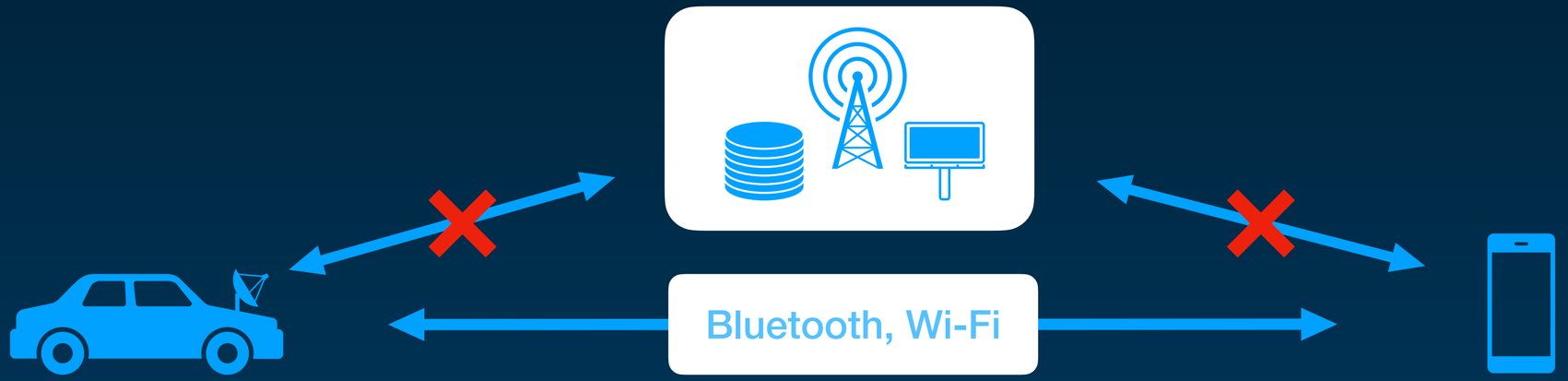
CROSS-Functional

IOT, который мы не ждали

- Мир изменился не только экономически. Сложность пришла, откуда не ждали. Если раньше мы жили в парадигме клиент-сервер, причем мобильный клиент тонкий, у которого всегда есть интернет, то теперь интернет — это опция.
- На примере Делимобиль: в условиях отсутствия связи ни у телефона, ни автомобиля интернета нет. А ехать надо!

IOT, который мы не ждали

- Мир изменился не только экономически. Сложность пришла, откуда не ждали. Если раньше мы жили в парадигме клиент-сервер, причем мобильный клиент тонкий, у которого всегда есть интернет, то теперь интернет — это опция.
- На примере Делимобиль: в условиях отсутствия связи ни у телефона, ни автомобиля интернета нет. А ехать надо. Ну и кто теперь сервер?



IOT, который мы не ждали

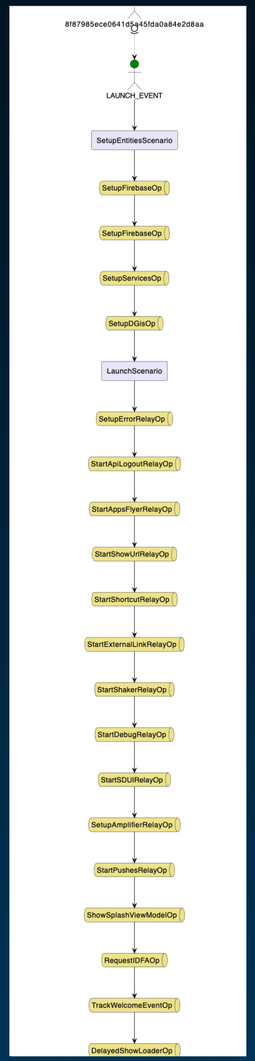
- Сервер — кто угодно. И автомобиль, и мобильное приложение. Но лучше приложение, потому что у смартфона обычно есть: а) мощный процессор б) достаточный объем памяти.
- Мы вступили в эру децентрализованных вычислений.
- Волна информационных взаимодействий теперь может протекать вне традиционного сервера, но как тогда ее анализировать и контролировать?
- Во-первых, нам нужна аналитика на клиенте — оффлайновый сбор логов в приложении-сервере.
- Во-вторых, нам нужна персонализация на клиенте — тоже оффлайн, без доступа к статистике.

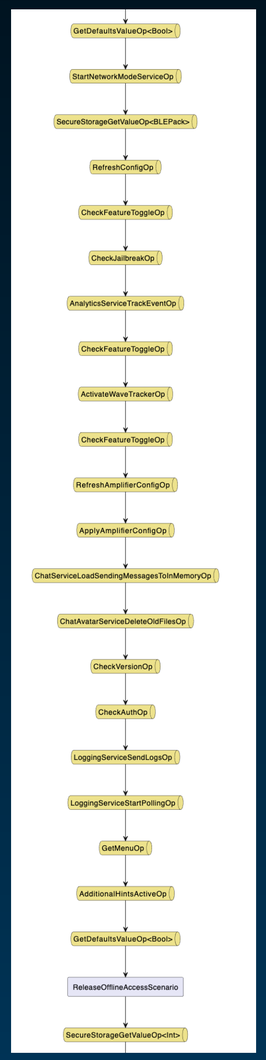
Без сервера

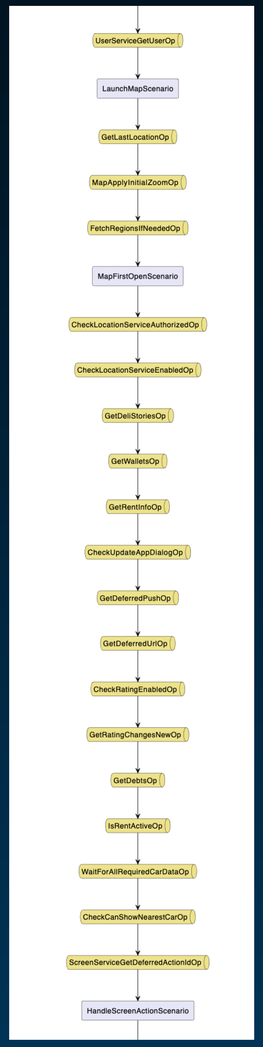
Волновой трейс

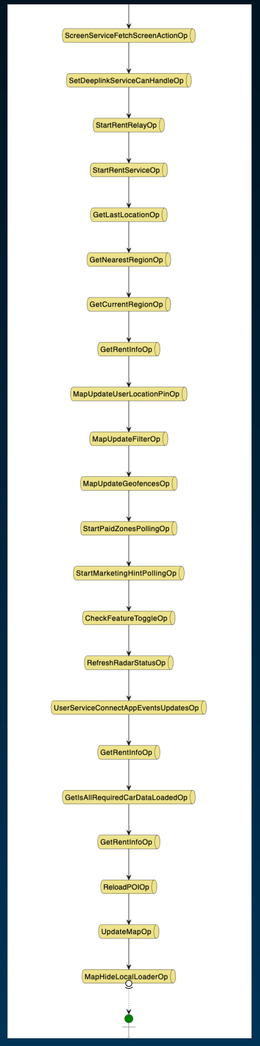
- Волна — это каузально упорядоченная последовательность операций. А последовательность волн — это путь пользователя в системе. Путь, который мы можем и детектировать на клиенте, и анализировать на клиенте. Этот путь будем называть волновым трейсом.
- Для принятия решений по персонализации мы не можем полагаться на серверную статистику по ивентам. Все, что у нас есть на оффлайновом клиенте — это волновой трейс текущего пользователя и... набор шаблонов поведения — референсных трейсов, загруженных на клиент предварительно.
- Сравнивая текущий трейс с референсным, мы принимаем решение, какую фичу показать, а какую скрыть.

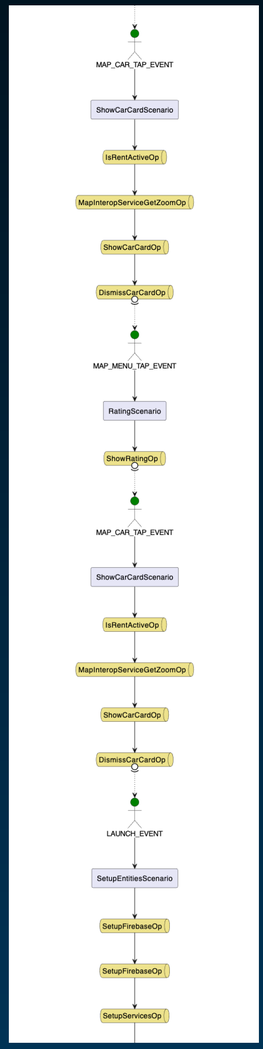
Волновой трейс

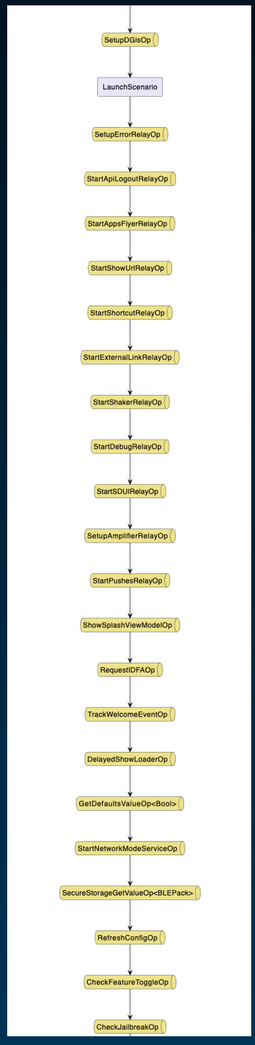


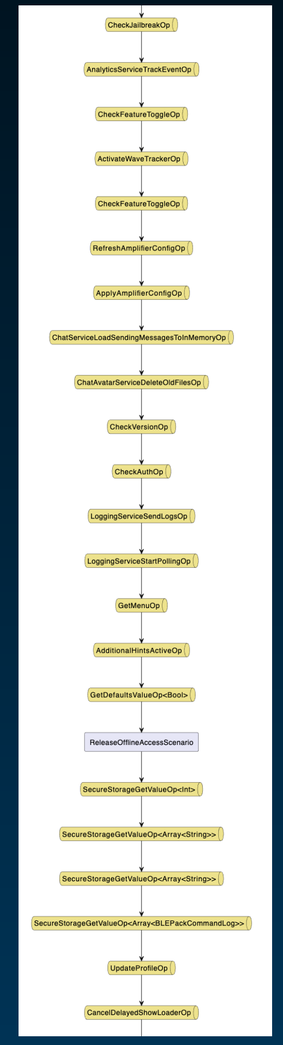








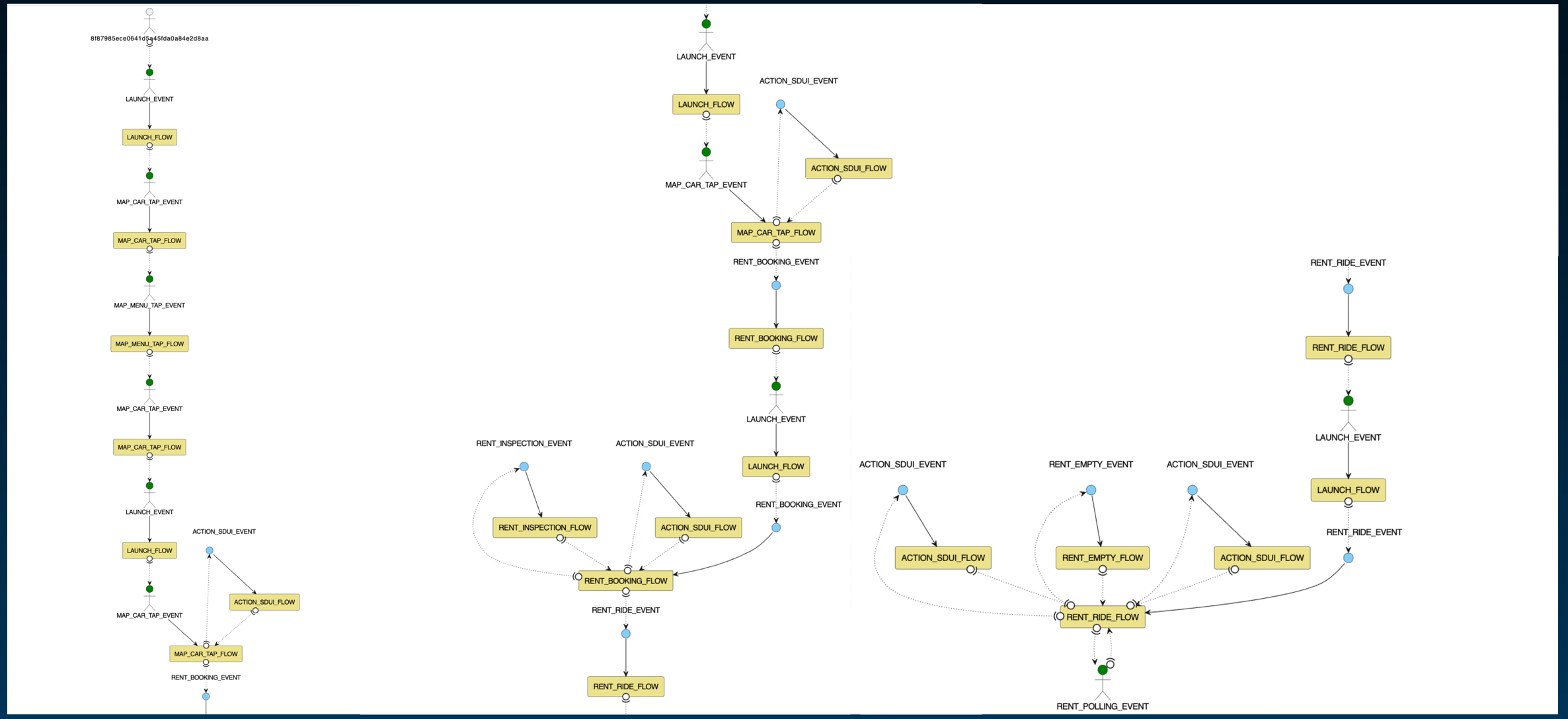




Волновой трейс

- В приложении Делимобиль минимальная пользовательская поездка — это 2-4 запуска приложения, около 20 артефактов (т.е. волн) и 600 операций. А ведь бывают еще длительные поездки/аренды. Неделя ежедневных поездок — 10k событий, месяц — 40k.
- Анализ всех «подстрок» в трейсе носит квадратичный характер: 10k^2=100k операций сравнения. Для реалтайма многовато, даже на топовых смартфонах. Кроме того, полного сходства цепочек все равно не будет (есть обработка ошибок и ретраи).
- Но у нас есть супер-оружие — волновая модель. Мы можем «зумить» путь пользователя, опираясь на артефакты.

Волновой трейс + «волновой зум»



Фрактальный зум

- Если показать только волны, то 600 событий теперь помещаются на одном экране.

- Высокоуровневый шаблон тоже содержит только волновые артефакты. Сравнение текущего трейса в «сколлапсированном» виде с шаблоном получается гораздо
  1) устойчивее к ошибкам и ретраям;
  2) вычислительно дешевле.
  Мы получаем искомую персонализацию на лету.

Фрактальный **зум**

Фрактальный зум

- А если нам нужно обработать несколько поездок? Легко: при детекции поездки (у нас есть шаблон детектирования) мы добавляем в трейс мета-артефакты, обозначающие поездку.

- И снова применяем «фрактальный зум». Шаблон 2-го уровня применяется к дважды сколлапсированному трейсу: «Ага, детект 4-х поездок! Можешь воспользоваться расширенной зоной парковки.»
  И все это в оффлайне!

- Волновая модель — это ответ на изменения в окружающем нас мире. Причинно-следственных связей становится слишком много, их трудно удержать в голове. Даже в этом докладе их чересчур много. Давайте выполним «фрактальный зум» этого доклада!

Эпилог

- **Атомарная операция** позволяет разделить состояние на небольшие части и перенести бизнес-логику в функциональный мир. Небольшие стейт-машины в отдельных сервисах нам не страшны.
- **Обязательный onFinish у контекста**, передаваемого в сценарий и/или контекстную функцию, формирует «нить Ариадны». Она позволяет фрактализировать бизнес-логику и вернуться в императивный мир. Мы получаем логарифмическую сложность вместо квадратичной. Кактус обошел Дракона.
- **UI — это параметризируемое реле**, потому что здесь приложение сталкивается с человеком. В одном месте содержатся и элементы управления, и состояние. ViewModel — это параметризирующий контекст для UIRelay и одновременно провайдер состояния для UIService.
- Волновая модель — это **основа для кросс-функциональных команд**. Волны независимы, поэтому и команды, их реализующие, тоже могут работать независимо.
- В оффлайне каждый сам себе сервер. Статистических данных в доступе больше нет, поэтому **в оффлайне для персонализации используется волновой трейс**.
- Фрактальный зум заменяет квадратичную сложность логарифмической. Мы можем децентрализованно **детектировать поведение и персонализировать сервис прямо на клиенте**, на лету.

Эпилог

- **Атомарная операция** позволяет разделить состояние на небольшие части и перенести бизнес-логику в функциональный мир. Небольшие стейт-машины в отдельных сервисах нам не страшны.
- **Обязательный onFinish у контекста**, передаваемого в сценарий и/или контекстную функцию, формирует «нить Ариадны». Она позволяет фрактализировать бизнес-логику и вернуться в императивный мир. Мы получаем логарифмическую сложность вместо квадратичной. Кактус обошел Дракона.
- **UI — это параметризируемое реле**, потому что здесь приложение сталкивается с человеком. В одном месте содержатся и элементы управления, и состояние. ViewModel — это параметризирующий контекст для UIRelay и одновременно провайдер состояния для UIService.
- Волновая модель — это **основа для кросс-функциональных команд**. Волны независимы, поэтому и команды, их реализующие, тоже могут работать независимо.
- В оффлайне каждый сам себе сервер. Статистических данных в доступе больше нет, поэтому **в оффлайне для персонализации используется волновой трейс**.
- Фрактальный зум заменяет квадратичную сложность логарифмической. Мы можем децентрализованно **детектировать поведение и персонализировать сервис прямо на клиенте**, на лету.

Эпилог

- **Атомарная операция** позволяет разделить состояние на небольшие части и перенести бизнес-логику в функциональный мир. Небольшие стейт-машины в отдельных сервисах нам не страшны.
- **Обязательный onFinish у контекста**, передаваемого в сценарий и/или контекстную функцию, формирует «нить Ариадны». Она позволяет фрактализировать бизнес-логику и вернуться в императивный мир. Мы получаем логарифмическую сложность вместо квадратичной. Кактус обошел Дракона.
- **UI — это параметризируемое реле**, потому что здесь приложение сталкивается с человеком. В одном месте содержатся и элементы управления, и состояние. ViewModel — это параметризирующий контекст для UIRelay и одновременно провайдер состояния для UIService.
- Волновая модель — это **основа для кросс-функциональных команд**. Волны независимы, поэтому и команды, их реализующие, тоже могут работать независимо.
- В оффлайне каждый сам себе сервер. Статистических данных в доступе больше нет, поэтому **в оффлайне для персонализации используется волновой трейс**.
- Фрактальный зум заменяет квадратичную сложность логарифмической. Мы можем децентрализованно **детектировать поведение и персонализировать сервис прямо на клиенте**, на лету.

Эпилог

- **Атомарная операция** позволяет разделить состояние на небольшие части и перенести бизнес-логику в функциональный мир. Небольшие стейт-машины в отдельных сервисах нам не страшны.
- **Обязательный onFinish у контекста**, передаваемого в сценарий и/или контекстную функцию, формирует «нить Ариадны». Она позволяет фрактализировать бизнес-логику и вернуться в императивный мир. Мы получаем логарифмическую сложность вместо квадратичной. Кактус обошел Дракона.
- **UI — это параметризируемое реле**, потому что здесь приложение сталкивается с человеком. В одном месте содержатся и элементы управления, и состояние. ViewModel — это параметризирующий контекст для UIRelay и одновременно провайдер состояния для UIService.
- Волновая модель — это **основа для кросс-функциональных команд**. Волны независимы, поэтому и команды, их реализующие, тоже могут работать независимо.
- В оффлайне каждый сам себе сервер. Статистических данных в доступе больше нет, поэтому **в оффлайне для персонализации используется волновой трейс**.
- Фрактальный зум заменяет квадратичную сложность логарифмической. Мы можем децентрализованно **детектировать поведение и персонализировать сервис прямо на клиенте**, на лету.

Эпилог

- **Атомарная операция** позволяет разделить состояние на небольшие части и перенести бизнес-логику в функциональный мир. Небольшие стейт-машины в отдельных сервисах нам не страшны.
- **Обязательный onFinish у контекста**, передаваемого в сценарий и/или контекстную функцию, формирует «нить Ариадны». Она позволяет фрактализировать бизнес-логику и вернуться в императивный мир. Мы получаем логарифмическую сложность вместо квадратичной. Кактус обошел Дракона.
- **UI — это параметризируемое реле**, потому что здесь приложение сталкивается с человеком. В одном месте содержатся и элементы управления, и состояние. ViewModel — это параметризирующий контекст для UIRelay и одновременно провайдер состояния для UIService.
- Волновая модель — это **основа для кросс-функциональных команд**. Волны независимы, поэтому и команды, их реализующие, тоже могут работать независимо.
- В оффлайне каждый сам себе сервер. Статистических данных в доступе больше нет, поэтому **в оффлайне для персонализации используется волновой трейс**.
- Фрактальный зум заменяет квадратичную сложность логарифмической. Мы можем децентрализованно **детектировать поведение и персонализировать сервис прямо на клиенте**, на лету.

Эпилог

- **Атомарная операция** позволяет разделить состояние на небольшие части и перенести бизнес-логику в функциональный мир. Небольшие стейт-машины в отдельных сервисах нам не страшны.
- **Обязательный onFinish у контекста**, передаваемого в сценарий и/или контекстную функцию, формирует «нить Ариадны». Она позволяет фрактализировать бизнес-логику и вернуться в императивный мир. Мы получаем логарифмическую сложность вместо квадратичной. Кактус обошел Дракона.
- **UI — это параметризируемое реле**, потому что здесь приложение сталкивается с человеком. В одном месте содержатся и элементы управления, и состояние. ViewModel — это параметризирующий контекст для UIRelay и одновременно провайдер состояния для UIService.
- Волновая модель — это **основа для кросс-функциональных команд**. Волны независимы, поэтому и команды, их реализующие, тоже могут работать независимо.
- В оффлайне каждый сам себе сервер. Статистических данных в доступе больше нет, поэтому **в оффлайне для персонализации используется волновой трейс**.
- Фрактальный зум заменяет квадратичную сложность логарифмической. Мы можем децентрализованно **детектировать поведение и персонализировать сервис прямо на клиенте**, на лету.

Let's make Mobile Development ~~great~~ fun again!

Юрий Дубовой | Mobius 2025 Autumn