Javanese Online

(https://vk.com/javanese.online) (http://android-developers.ru/)

Главная (/) / Статьи (/%D1%81%D1%82%D0%B0%D1%82%D1%8C%D0%B8/)

Hype-driven Android-development, или как инженерная специальность превращается в маркетинг

Я привык считать программирование инженерной специальностью. То есть такой, где нужно проектировать, рассчитывать, измерять, а выбор обосновывать составленными списками плюсов и минусов. Где лучший специалист — тот, кто пишет максимально простой для понимания и внесения изменений код; кто способен спроектировать надёжную и производительную систему, разработать уникальную функциональность, проработать все угловые случаи; сделать пользователя довольным. При таком подходе основной критерий при выборе инструментов — баланс проблем, которые он решает, против неудобств, которые приносит.

Разберём в деталях, что происходит вместо этого в экосистеме Android-разработки.

ConstraintLayout

Теперь в свежесозданном проекте сразу подключена эта библиотека, а корневым контейнером заготовки вёрстки стал `ConstraintLayout` вместо `RelativeLayout`. Constraint — отличный нишевый инструмент для решения очень узкого круга задач: позволяет плоско сверстать сложные отношения, расставив вью, например, по окружности, в вершинах звезды или на кончиках усов котёнка. Только это не значит, что он везде должен собой заменить `RelativeLayout`, `FrameLayout` и `LinearLayout`. Многим проектам он не нужен вовсе, потому что способен улучшить вёрстку одного редко посещаемого экрана, ухудшив остальные. Самое неприятное, что `RelativeLayout` теперь переехал во вкладку Legacy в палитре компонентов визуального редактора (а, всё равно все редактируют XML руками) (очевидно, в Android SDK ничего нет ему на замену).

Преимущества

— в некоторых случаях уменьшает вложенность вёрстки, тем самым ускоряя `onMeasure` (https://cloud.tencent.com/developer/article/1365408) и сокращая количество потребляемой памяти

— поддерживает редактирование мышью (в том самом превью, которое почти не работает)

Недостатки

— непредсказуемая (https://medium.com/@krpiotrek/constraintlayout-performance-c1455c7984d7) производительность (https://android.jlelse.eu/constraint-layout-performance-870e5f238100), ставящая под вопрос (https://blog.usejournal.com/constraintlayout-v1-1-2-performance-test-1c3abfc3a7d9) целесообразность всей затеи (https://medium.com/@fchristysen/android-layout-layout-time-comparison-9096cc37e37a)
— констреинты (https://android.googlesource.com/platform/frameworks/opt/sherpa/+/studio-3.0/constraintlayout/src/main/java/android/support/constraint/Constraints.java), направляющие (https://android.googlesource.com/platform/frameworks/opt/sherpa/+/studio-3.0/constraintlayout/src/main/java/android/support/constraint/Guideline.java), а также хелперы (https://android.googlesource.com/platform/frameworks/opt/sherpa/+/studio-3.0/constraintlayout/src/main/java/android/support/constraint/ConstraintHelper.java) — группа (https://android.googlesource.com/platform/frameworks/opt/sherpa/+/studio-3.0/constraintlayout/src/main/java/android/support/constraint/Group.java) и барьер (https://android.googlesource.com/platform/frameworks/opt/sherpa/+/studio-3.0/constraintlayout/src/main/java/android/support/constraint/Barrier.java) — полноценные вьюхи с большим retained size и толстым конструктором, что снова ставит под вопрос целесообразность плоской вёрстки
— Placeholder (https://android.googlesource.com/platform/frameworks/opt/sherpa/+/studio-3.0/constraintlayout/src/main/java/android/support/constraint/Placeholder.java) — очень удобный и мощный инструмент, если бы не его бесполезность вне констреинта
— неудобство переиспользования/композирования отдельных частей вёрстки
— сложность чтения вёрстки без IDE/Preview — всё сваливается в неиерархическую кучу
— трудночитаемые diff'ы в контроле версий, особенно после редактирования мышью
— как и любая библиотека, увеличивает размер APK; как и любые классы, требует загрузки в оперативную память, верификации, JIT-компиляции

RxJava 2 (https://github.com/ReactiveX/RxJava)

Библиотека для «реактивщины» — так сейчас модно называть асинхронную обработку событий. Является одной из реализаций (худших реализаций) парадигмы ФРП (функциональное реактивное программирование) (на самом деле нет).

Решает лишь те проблемы, которые уже давно решены с помощью `java.util.stream` и `java.util.concurrent`, от `Executors` до `CompletableFuture`.

Проблемы реализации

— невозможность передачи null-значений. Сам факт наличия null в Java — это ошибка, но исправить её уже не суждено. В JDK, Android и многих популярных библиотеках полно

нуллов, так что практические решения обязаны смириться и поддержать это. Для симуляции нуллов можно использовать Maybe, но оно может заворачивать не только значение, но и асинхронные вычисления, так что использование `blockingGet` рисковано
— обилие разных интерфейсов (технически это ещё и абстрактные классы) для похожих задач: `Single != Observable`, `Completable != Single<Void>`
— горы операторов для обработки ошибок вместо использования result type
— необходимость ручной отписки (плохо ложится на парадигму автоматического управления памятью)
— очень много «операторов» (т. е. методов, продублированных внутри `Observable`, `Single`, `Flowable`) — значительно больше, чем, например, в перегруженных `Collection` и `Stream` из JDK. SRP нарушен дичайшим образом
— все «операторы» — это member methods, что никак не поддаётся расширению. Вот этот hello world

```
stream
    .filter(it -> it > 10)
    .map(it -> 10*it)
    .reduce((a, b) -> a + b)
```

мог бы выглядеть так, если бы все операторы были реализованы снаружи:

```
stream
    .compose(filter(it -> it > 10))
    .compose(map(it -> 10*it))
    .compose(reduce((a, b) -> a + b))
```

такой стиль склоняет к более модульному коду и слабому связыванию.

— объём: занимает более 10 тысяч методов, более 1 МБ DEX
— отдельные «операторы» `subscribeOn` и `observeOn` выглядят не особо удачно: обычно автор задачи знает, где она должна выполняться (пример из Java Core: `CompletableFuture.supplyAsync(supplier, executor)`), а потребитель — куда приносить результат (`completableFuture.thenAcceptAsync(consumer, executor)`); сложно представить, как будет вести себя `observable.subscribeOn(sch1).subscribeOn(sch2).observeOn(sch3).observeOn(sch4)` — а значит, нужно запретить это

Проблемы применения

— с обработкой коллекций в Java отлично справляется Solid — минималистичный порт стримов (https://github.com/konmik/solid), в Kotlin — stdlib
— для одноразовых асинхронных операций в Java с незапамятных времён есть `ExecutorService`

(https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html) и `Future` (https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html).

```
Future<A> aFuture = ioExecutor.submit(() -> {
    Thread.sleep(500); // типа, сходили в интернет
    return new A(/* типа, вернули данные */);
});
Future<B> bFuture = ioExecutor.submit(() -> {
    Thread.sleep(500);
    return new B(...);
});
ioExecutor.execute(() -> {
    doSomethingWith(aFuture.get(), bFuture.get());
});
```

Здесь get — блокирующее ожидание результата. В UI-потоке, конечно, так делать нельзя, ждать придётся на бэкграунде:

```
/**
 * Executes {@param supplier} on the {@param worker},
 * passes returned value to {@param consumer} on the {@param target}.
 * Ignores error handling problem.
 */
public static <R> Future<?> executeAsync(
        final ExecutorService worker, final Callable<R> supplier,
        final Handler target, final Consumer<R> consumer
) {
    return worker.submit(() -> {
        final R value = supplier.call();
        target.post(() -> consumer.accept(value));
        return null; // conform Callable
    });
}

void sample() {
    Future<A> aFuture = ...;
    Future<B> bFuture = ...;
    executeAsync(
            ioExecutor, () -> combineSomehow(aFuture.get(), bFuture.get()),
            new Handler(Looper.getMainLooper()), result -> {
                someTextView.setText(result.textValue());
            }
    );
}
```

Более современный и похожий на pipeline путь — CompletableFuture (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html), который появился в Java 1.8 (2014), но был бэкпортирован как минимум дважды: stefan-zobel/streamsupport (https://github.com/stefan-zobel/streamsupport), retrostreams/android-retrofuture (https://github.com/retrostreams/android-retrofuture), (заметка о лицензии) (https://github.com/stefan-zobel/streamsupport/issues/3).

```
supplyAsync(() -> new A(...), ioExecutor)
.thenCombineAsync(
        supplyAsync(() -> new B(...), ioExecutor),
        ::combineSomehow,
        ioExecutor
)
.thenAcceptAsync(
        result -> someTextView.setText(result.textValue()),
        uiExecutor
)
```

В качестве `uiExecutor` не удастся передать экземпляр `Handler`, а вот лямбду `runnable -> handler.post(runnable)` — вполне.

— когда дело касается обработки потока данных, всё зависит от того, откуда берётся этот поток, какие там данные и что с ними нужно сделать. Например, debounce текста из поля ввода тривиально реализовывается с помощью `Handler`:

```
public final class Debounce extends SimpleTextWatcher
        implements Runnable, View.OnAttachStateChangeListener {

    /**
     * Invokes {@param consumer} on {@param handler} passing text from {@param sourc
     * at most one time in {@param delayMillis}
     * and only if {@param source} view is attached to window.
     */
    public static void debounced(
            Handler handler, TextView source, int delayMillis, Consumer<String> cons
    ) {
        Debounce d = new Debounce(handler, source, delayMillis, consumer);
        source.addTextChangedListener(d);
        source.addOnAttachStateChangeListener(d);
    }

    private final Handler handler;
    private final TextView source;
    private final int delayMillis;
    private final Consumer<String> consumer;

    private Debounce(
            Handler handler, TextView source, int delayMillis, Consumer<String> cons
    ) {
        this.handler = handler;
        this.source = source;
        this.delayMillis = delayMillis;
        this.consumer = consumer;
    }

    @Override public void afterTextChanged(@NotNull Editable s) {
        if (source.isAttachedToWindow()) {
            handler.removeCallbacks(this);
            handler.postDelayed(this, delayMillis);
        }
    }

    @Override public void run() {
        consumer.accept(source.getText().toString());
    }

    @Override public void onViewAttachedToWindow(View v) {
    }
    @Override public void onViewDetachedFromWindow(View v) {
        handler.removeCallbacks(this);
    }

}
```

В этом коде примечательно, что не нужно отписываться, — `Debounce` станет недостижимым вместе с иерархией вью. А при необходимости несложно добавить пару усовершенствований: отправку первого события сразу же при создании; отправку события при `onAttachedToWindow`, если предыдущая отправка была отменена из-за `onViewDetachedFromWindow`

— в определённых случаях для обработки событий подходит «коробочный» инструмент из Java 1.8 — `Streams`. Они значительно отличаются от RxJava (https://stackoverflow.com/a/35759458/3050249), но пересечение функциональности достаточно большое. Для Android имеется бэкпорт — Stream Support (https://github.com/stefan-zobel/streamsupport)

Преимущества

— очень развитая экосистема — RxBindings, поддержка в Retrofit, GreenDAO, Room, Realm

Только вот разработчикам библиотек стоит учитывать, что если они хотят поддержать реактивность, нужно добавлять в зависимости не RxJava, а Reactive Streams, чтобы позволить клиенту выбирать реализацию самостоятельно. Ведь есть ещё как минимум Project Reactor.

DI-фреймворки

Из-за фундаментальных ошибок проектирования Android (http://javanese.online/%D1%81%D1%82%D0%B0%D1%82%D1%8C%D0%B8/%D1%84%D1%83%D0%BD%D0%B4%D0%B0%D0%BC%D0%B5%D0%BD%D1%82%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D0%BF%D1%80%D0%BE%D0%B1%D0%BB%D0%B5%D0%BC%D1%8B_android/) даже такая простая задача, как создание графа объектов и удовлетворение их зависимостей, превращается в кошмар. Что могут предложить нам DI-контейнеры (https://ru.wikipedia.org/wiki/%D0%92%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D0%B5_%D0%B7%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B8) и Service Locator'ы (https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%BA%D0%B0%D1%82%D0%BE%D1%80_%D1%81%D0%BB%D1%83%D0%B6%D0%B1) вроде Dagger 2 (https://github.com/google/dagger), Toothpick (https://github.com/stephanenicolas/toothpick), Koin (https://github.com/InsertKoinIO/koin), KoDeIn (https://github.com/Kodein-Framework/Kodein-DI)?

Решаемые проблемы

— собственно, передача необходимых объектов от `Application` или `static` к компонентам приложения

Привносимые проблемы

— создаёт свой метаязык (зачастую из аннотаций): знания языка программирования и платформы для поддержки такого кода недостаточно

— унося контроль из классов, также изымает его из рук разработчиков — зависимости контролируются опосредованно
— обычно тормозит компиляцию кодогенерацией или рантайм — рефлексией
— зачастую не даёт compile-time гарантий существования зависимостей
— провоцирует сильное связывание (делает проблематичным инжект разных реализаций одного интерфейса (/%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F/%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%BD%D1%8B%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B/monomorphic_interface/), тем самым нарушает DIP (https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%B8%D0%BD%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D0%B8_%D0%B7%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B5%D0%B9)) и делает инверсию контроля бессмысленной
— скоупы — это глобальное изменяемое состояние, а их закрытие сродни ручному управлению памятью
— усложняет навигацию даже в IDE

Вот и выходит, что забрать зависимости из static или Application — это более прямолинейно, менее многословно и не тормозит ни компиляцию, ни рантайм, а отличия в коде минимальные.

При выборе архитектурного решения можно задавать вопрос: как это будет работать в двух экземплярах? Например: смогут ли соседствовать два фрагмента, которые обмениваются данными с активити через синглтон? если они общаются через интерфейс, сможет ли активити отличить один от другого? если два фрагмента открыли один и тот же скоуп, не столкнутся ли их зависимости? можно ли добавить на экран два фрагмента одного класса, передав им разные реализации презентера/вьюмодели?

Если ответы — «нет», «вроде можно, но это не точно», «со скрипом» или «ой, ну сделаем, если надо будет», то выбранное решение ведёт к сильному связыванию и мешает повторному использованию кода.

MVP

Model-View-Presenter — порт неудачного server-side паттерна Model-View-Controller на client-side. Презентер отличается от контроллера тем, что не только передаёт данные во View, но и слушает события из View и реагирует на них.

Решаемые проблемы

— помогает хоть как-то отделить логику от UI и избежать совесем уж монолитного кода
— тестируемость

Собственные проблемы

— императивность: принимает события и отдаёт команды, что приводит к сложным, трудновоспроизводимым состояниям
— состояние живёт в презентере и во вью — сложно его отделить и сохранить
— бессмысленность тестирования: при тестировании легко доказать, что презентер в определённых обстоятельствах вызывает нужные методы вью, но нет никакой гарантии, что вью обрабатывает их правильно — в каком состоянии оно будет после `showProgress(); showData(list)`? ProgressBar+List или только List?

Moxy (https://github.com/Arello-Mobile/Moxy)

Moxy — реализация MVP, которая сохраняет презентер между сменами конфигурации, а после пересоздания View восстанавливает его состояние, передоставляя события.

Восстановление View работает только при смене конфигурации (что довольно бесполезно, ведь в Activity можно поставить `configChanges="orientation|screenSize"` и не терять вью, а в retain-фрагментах можно хранить состояние в полях, и его никто не украдёт). В случае рестарта процесса никак не поможет. Даже наоборот: рекомендуется «очищать» состояние вью в `onFirstViewAttach()`, потому что после пересоздания процесса создастся новый презентер, а состояние View восстановится встроенными средствами Android.

То есть единственная задача, с которой справляется Moxy, — протаскивание багов мимо тестирования в релиз (ну, пока отдел тестирования не прочитает про Don't Keep Activities). Всё это — ценой замедленной компиляции (apt/kapt), добавления больших объёмов сгенерированного кода, замусоривания результатов поиска классов и символов в IDEA/AS.

AAC ViewModel

Не предлагает вообще ничего, не помогает никак. Положить `HashMap<Class<T>, T>` в `nonConfigurationInstance` можно и без помощи библиотеки.

Не имеет никакого отношения к ViewModel из MVVM, правильное название — `RetainObject` или `NonConfigurationInstance`.

Отнимает у разработчика доступ к конструктору (ну, как обычно).

Таки можно создавать экземпляры через нормальный конструктор посредством фабрики (https://github.com/googlesamples/android-architecture-components/blob/d31d2bb632467051c2fddd3dc493cc480c44ada0/BasicSample/app/src/main/java/com/example/android/persistence/viewmodel/ProductViewModel.java#L90), но у вьюмоделей остаётся семантика синглтонов: нет внятного способа создать вьюмодели одного класса с разными зависимостями.

Retrofit

Retrofit представляет удалённые HTTP-эндпоинты в виде Java-методов. Считаю, что это очень уместная абстракция. Однако, имеются проблемы:

— метаязык из аннотаций `@POST("smth/{wtf}") Call<T> smth(@Path("wtf") String path)` вместо явного `new Endpoint(Method.POST, "smth/{wtf}", new PathParameter("wtf"))`
— CallAdapter.Factory (https://square.github.io/retrofit/2.x/retrofit/retrofit2/CallAdapter.Factory.html) и Converter.Factory (https://square.github.io/retrofit/2.x/retrofit/retrofit2/Converter.Factory.html) регистрируются в билдере, а «используются» в интерфейсе. До запуска приложения нет простого способа узнать, присутствуют ли все необходимые для работы конвертеры; есть ли неиспользуемые конвертеры. На последний вопрос также не в состоянии ответить ProGuard, т. к. поддерживает reflection в очень ограниченном виде.

На пересечении Retrofit и ProGuard возникает ещё одна проблема: т. к. ProGuard не предоставляет возможности сохранить определённые аннотации, вырезав остальные, приходится писать `-keepattributes *Annotation*`, сохраняя все. Когда добавляется Kotlin, в конечное приложение попадает ощутимое количество мусора в виде `@kotlin.Metadata`.

Clean architecture

Вероятно, здесь я буду критиковать не саму clean architecture Дядюшки Боба, а распространённую в русскоязычном сообществе интерпретацию. В качестве примера разберём теоретический материал по основам архитектуры (https://github.com/AndroidArchitecture/AndroidArchitectureBook/blob/master/theory/Theory_article.md) на ломаном русском.

Решаемые проблемы

— много слоёв проще редактировать большой командой — меньше шансов получить merge conflict
— интеракторы и презентеры легко покрываются тестами (правда, баги всё равно обычно возникают в других местах)

Привносимые проблемы

— провоцирует так называемый пахлава-код — код, состоящий из большого количества несамостоятельных слоёв, ни один из которых ни за что конкретное не отвечает
— чтобы отредактировать одну фичу, нужно затронуть несколько слоёв — отличный подход с точки зрения удержания рабочего места
— большинство интерфейсов пишутся, потому что надо (http://javanese.online/%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F/%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%BD%D1%8B%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B/monomorphic_interface/).

— классы предлагается группировать по слоям, то есть в одном пакете находятся классы, решающие разные продуктовые задачи (фичи)

В продолжение этой темы некоторые разделяют проект на модули. Но, опять же, вместо того чтобы отделять фичи (что бывает полезно), они отделяют слои. Смысл этого костыля в том, что kapt меньше тормозит, если Room, Moxy, Dagger находятся каждый в своём модуле.

Конференции

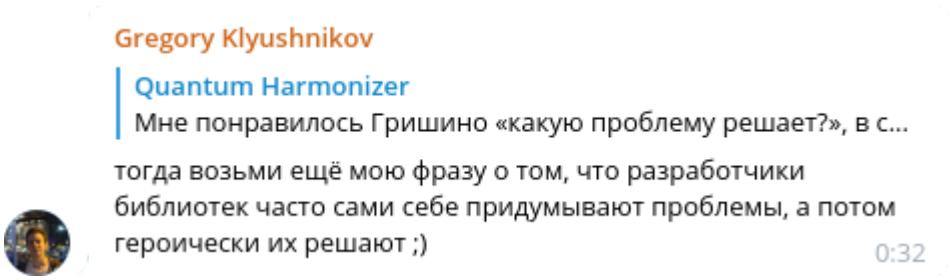
Множество докладов заключается в пересказе документации очередной модной библиотеки или изложении очередного модного подхода. Главный вопрос — какая при этом решается проблема — остаётся без ответа.

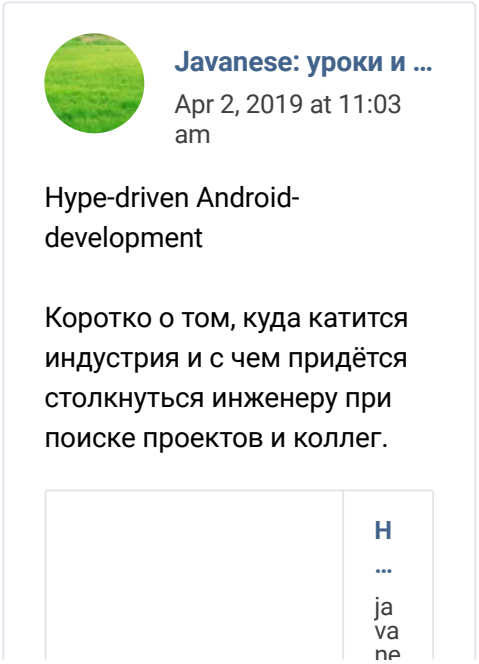
Трудоустройство

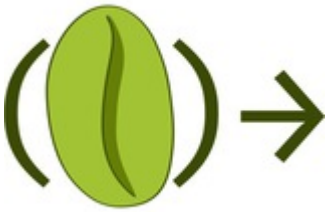
Казалось бы: пусть сходят с ума как хотят. Кому какое дело? Проблема в том, что мусорные библиотеки стали стандартом и насквозь пронизали индустрию. Хорошую работу со своевременной «белой» зарплатой и адекватным руководством найти непросто, а тут ещё и технологии, которые делят индустрию на несколько лагерей.

Я вижу такой (неутешительный) вывод: на рынке труда лучше себя чувствует тот разработчик, который владеет модными инструментами, может найти им применение там, где это не нужно, сумеет разобраться в «современном» коде. А инженерные вопросы стали своего рода «вопросами религии» и могут спровоцировать неиллюзорные «холивары». Я уже предвижу в комментариях религиозных фанатиков, адептов превеликого Dagger и святой RxJava, которые абсолютно необходимы в любом современном CRUD-приложении.

Вместо завершения







Javanese Online
Hype-driven Android-development

Коротко о том, куда катится индустрия и с чем придётся столкнуться инженеру при поиске проектов и коллег.

Комментарии к статье

GitHub / Nik-Gleb
Miha! :) Большое тебе спасибо за такую содержательную статью, такой глубокий анализ. На самом деле твои мысли - имеют схожее отражение у 25% наших коллег, и возможно еще у 25% - это молчаливый глас совести. Очень рад что кто-то взялся за оформление этих позиций в тезисы и словесную форму. Более того - статья получилась, по-настоящему техническая, беспристрастная, с анализом, плюсами и минусами. Приятно, что она в итоге ни к чему не клонит, не призывает, не снабжена приглашением на очередной доклад или репозиторий на гитхабе. Абсолютно согласен с общим посылом в целом и особенно с абзацем-заключением про "влияние на трудоустройство и рынок труда". Всё прям точно сказано! Я бы сказал - "как с уст снял)"

09:42 01.04.2019

GitHub / andrewgrow
Отличная статья, спасибо! Вот прямо мои мысли оформил в текст. Сохранил в закладки, а то в спорах я быстро думать не умею и потому проигрываю обсуждения. А тут вот прямо готовый меч-кладенец)

11:31 02.04.2019

GitHub / Semper-Viventem
Миша, спасибо за статью! Только про Rx хочу уточнить.

> *"Решает лишь те проблемы, которые уже давно решены с помощью java.util.stream и java.util.concurrent, от Executors до CompletableFuture."*

Так реактивные стримы вроде CompletableFeature и появились только в Java 8 (2014 год), а первый коммит в RxJava был в 2012 (https://github.com/ReactiveX/RxJava/commit/697fd66aae9beed107e13f49a741455f1d9d8dd9 (https://github.com/ReactiveX/RxJava/commit/697fd66aae9beed107e13f49a741455f1d9d8dd9)), а это несколько раньше получается. Затем вышла вторая версия RxJava, которая является идейным продолжением, и исправляет некоторые проблемы, и предлагает новые решения для работы с реактивными потоками.

14:47 02.04.2019

GitHub / Miha-x64

Думаю, что создатели `CompletableFuture` вдохновлялись именно идеями RxJava. Но сейчас уже нет большой разницы, что появилось раньше: просто есть разные инструменты и выбор между ними, а CF — один из примеров возможного выбора.

(А мне обычно хватает `Executors`.)

15:21 02.04.2019

GitHub / alaershov

Было бы классно предложить альтернативы по каждому из перечисленных вами пунктов. Вы назвали популярные решения для некоторых задач. Да, решения не идеальные, но ничего ведь не происходит на ровном месте, в том числе и перечисленные вами библиотеки. Если вы не используете ни одно из них, и утверждаете, что это всё не нужно, и при этом нашли свой подход, который решает все задачи лучше, чем "хайповые" решения, то, может быть, поделитесь этим подходом с массами?

15:27 02.04.2019

GitHub / Miha-x64

Часть проблем считаю совсем надуманными. Часть решаю «коробочными» инструментами. Ещё для части разрабатываю библиотеку (http://github.com/Miha-x64/reactive-properties/), про которую буду рассказывать на AppsConf (http://appsconf.ru/moscow/2019/abstracts/4652).

16:48 02.04.2019

GitHub / alaershov

Круто, ждём доклада)

08:11 03.04.2019

GitHub / Yukooo

Автор, не путай людей, применения любого инструменты в работе требует ясного понимание, что это за инструмент и как с ним правильно обращаться. PS. Наберись опыта и поучаствуй в больших продуктовых проектах (от 10 человек).

18:14 02.04.2019

GitHub / Miha-x64

Аргументов, конечно, не будет.

20:29 02.04.2019

GitHub / techyourchance

В деталях я не согласен по нескольким пунктам (особенно с тем что касается Dependency Injection), но в целом, конечно, все верно, и от этого очень грустно. Особенно огорчает ситуация с конфами (хотя мне кажется, что в русскоязычном сообществе проблема не так остра как везде) и трудоустройством. Спасибо за статью.

06:56 03.04.2019

GitHub / ilyamodder

TL;DR: все хайповые инструменты мне непонятны, а значит, не нужны. Что мне привычнее, то и надо использовать

14:23 11.04.2019

GitHub / Miha-x64

tl;dr в статью не вникал, аргументов не будет.

14:01 14.04.2019

GitHub / syndarin

Похоже на очередной плач Ярославны, если честно.

1. ConstraintLayout - возражу с помощью всего лишь двух аргументов: Chains & Guidelines. Сделать интерфейс, элементы которого равномерно расположены по экрану, либо же занимают фиксированную его долю всегда было большой головной болью, которую эти фичи успешно решают. Тем паче, что костыли с LinearLayout & weights работают далеко не во всех случаях.

— "неудобство переиспользования/композирования отдельных частей вёрстки" - что, include уже запретили? Если Вы не об этом, то тогда вообще не вполне понятно, как Вы привыкли её переиспользовать (копипаст не считается).

2. RxJava:

— "невозможность передачи null-значений" - строго говоря, воспринимая любой rx-овый тип как поток значений, не вполне понятно, зачем нужны отсутствующие? Если они таки нужны, то, имхо, это проблемы с дизайном Вашей архитектуры. Ну, в крайнем случае, используйте Optional, кто мешает?

— "обилие разных интерфейсов" - обусловлено разным поведением, не находите?.

— "необходимость ручной отписки (плохо ложится на парадигму автоматического управления памятью)" - за отсутствием memory leaks тоже приходится следить вручную, Вас это не напрягает?

— "возможность возникновения OnErrorNotImplementedException" - простите, как разумный аргумент воспринимать не могу.

И да, StreamAPI не замена Rx. По крайней мере, если не использовать Rx исключительно для операций над коллекциями. Из часто используемых примеров - zip, combineLatest, withLatestFrom.

3. DI-фрейморки

Вступление уже прекрасно: "Из-за фундаментальных ошибок проектирования Android даже такая простая задача, как создание графа объектов и удовлетворение их зависимостей, превращается в кошмар."

Это действительно причина рассматривать DI-фреймворки как просто очередной хайповый инструмент?

14:36 11.04.2019

GitHub / Miha-x64

> *"Сделать интерфейс, (...) всегда было большой головной болью, которую эти фичи успешно решают."*

К такому применению никаких претензий нет. Я же говорю о ситуации, когда констреинт используется вместо всего.

> *"что, include уже запретили?"*

Если не рассматривать вариант инклуда констреинта в констреинт, то все include-вёрстки должны содержать layout-параметры именно для констреинта. Тогда такие «вставки» гораздо меньше похожи на самостоятельные компоненты и сильно зависят от родительского контейнера.

> *"Если они [null-значения] таки нужны, то, имхо, это проблемы с дизайном Вашей архитектуры."*

Согласен. Мне вообще приятно писать на языках без нуллов. А кому-то это зачем-то нужно (https://stackoverflow.com/a/45371201/3050249)

> *"в крайнем случае, используйте Optional, кто мешает?"*

В Java/Kotlin встроен бесплатный `Optional`. `null` называется.

> *"(...) OnErrorNotImplementedException" - простите, как разумный аргумент воспринимать не могу."*

Согласен, я плохо сформулировал свою идею. В JVM-мире нет подходящего решения проблемы, лучше удалю этот пункт.

> *"И да, StreamAPI не замена Rx."*

Да, ничто не замена Rx, кроме Rx. Для разных применений я предложил разные замены.

> *"Это действительно причина рассматривать DI-фреймворки как просто очередной хайповый инструмент?"*

Нет, это вступление, в котором я утверждаю, что проблема зависимостей в Android стоит действительно остро. А вот DI-фреймворки — неудачное её решение.

15:32 14.04.2019

GitHub / KuVaUgU
Почти со всем согласен. Вы работу не ищите?

14:43 11.04.2019

GitHub / Miha-x64
Короткий ответ — всё сложно. Можно обсудить в Телеграме (https://t.me/Harmonizr)

16:12 14.04.2019

Чтобы оставить комментарий, зайдите через GitHub (/OAuth/GitHub/?to=%2F%25D1%2581%25D1%2582%25D0%25B0%25D1%2582%25D1%258C%25D0%25B8%2Fhype-driven_android-development%2F%23comments_section).