Реактивная обработка ошибок своими руками

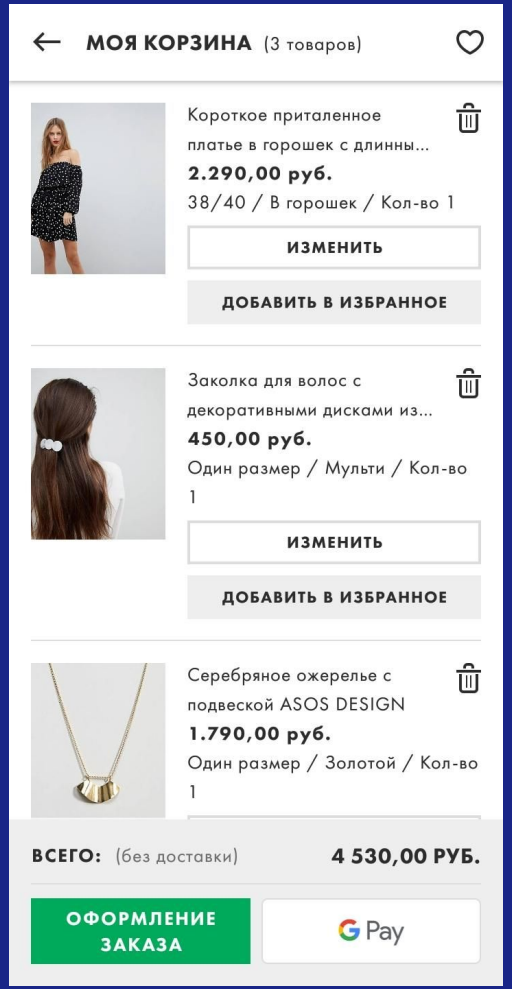
Константин Цховребов
@Redmadrobot SPb

Константин Цховребов

http://telegra.ph/Kto-ya-11-26

- с 2010 в Android-разработке
- архитектор в Redmadrobot SPb
- Cicerone, GitFox
- KUG, GDG

https://www.asos.com/



Обработка ошибок

Обработка ошибок

“разбираемся на месте”

Обработка ошибок

“разбираемся на месте”

- показываем диалог или другое сообщение

Обработка ошибок

“разбираемся на месте”

- показываем диалог или другое сообщение
- возвращаем элемент на место

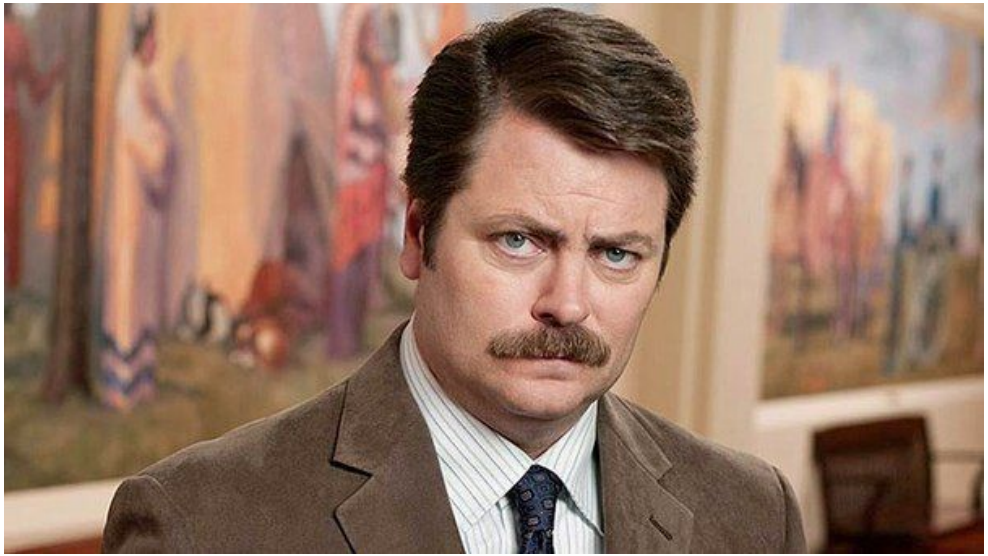
Обработка ошибок

“разбираемся на месте”

- показываем диалог или другое сообщение
- возвращаем элемент на место
- ...

А давайте “retry”?

Я человек простой...



Сохраняем данные запроса в презентере и при показе диалога с ошибкой позволяем повторить запрос.

Я человек простой...

- А если “retry” надо сделать на всех экранах?

Я человек простой...

- А если “retry” надо сделать на всех экранах?
- А если мы удалили подряд несколько элементов?

Я человек простой...

- А если “retry” надо сделать на всех экранах?
- А если мы удалили подряд несколько элементов?

Надо что-то более общее.

Больше абстракций богу абстракций!

- Обернем все запросы в Task.

Больше абстракций богу абстракций!

- Обернем все запросы в Task.
- У Task будет состояние и ID.

Больше абстракций богу абстракций!

- Обернем все запросы в Task.
- У Task будет состояние и ID.
- Можно будет следить за процессом и повторять при необходимости.

Больше абстракций богу абстракций!

- Обернем все запросы в Task.
- У Task будет состояние и ID.
- Можно будет следить за процессом и повторять при необходимости.

Сработает, но... надо перевернуть **все** с ног на голову, а потом еще всех учить новому фреймворку

Архитектура слоя исполнения асинхронных задач. Степан Гончаров. AppsConf 2018

Можно пойти еще дальше...

- Offline-first приложение
  Поддержка оффлайна в мобильном приложении. Синхронизация. DevFest 2018 Руслан Калбаев

Можно пойти еще дальше...

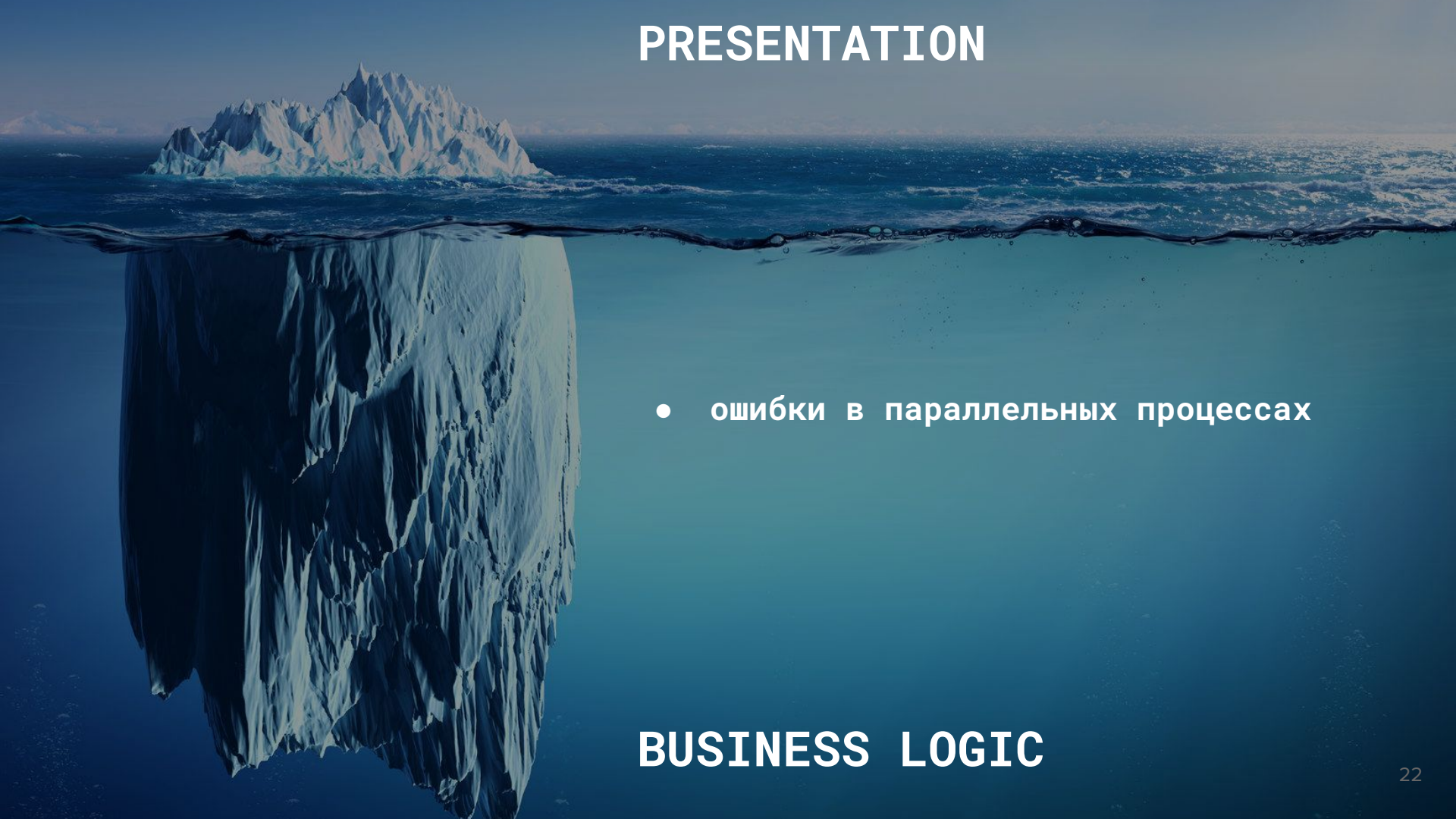
- Offline-first приложение
  Поддержка оффлайна в мобильном приложении. Синхронизация. DevFest 2018 Руслан Калбаев
- Лог запросов к серверу и отдельный сервис, гарантирующий их выполнение
  Upload в Одноклассниках. Mobius 2018. Кирилл Попов

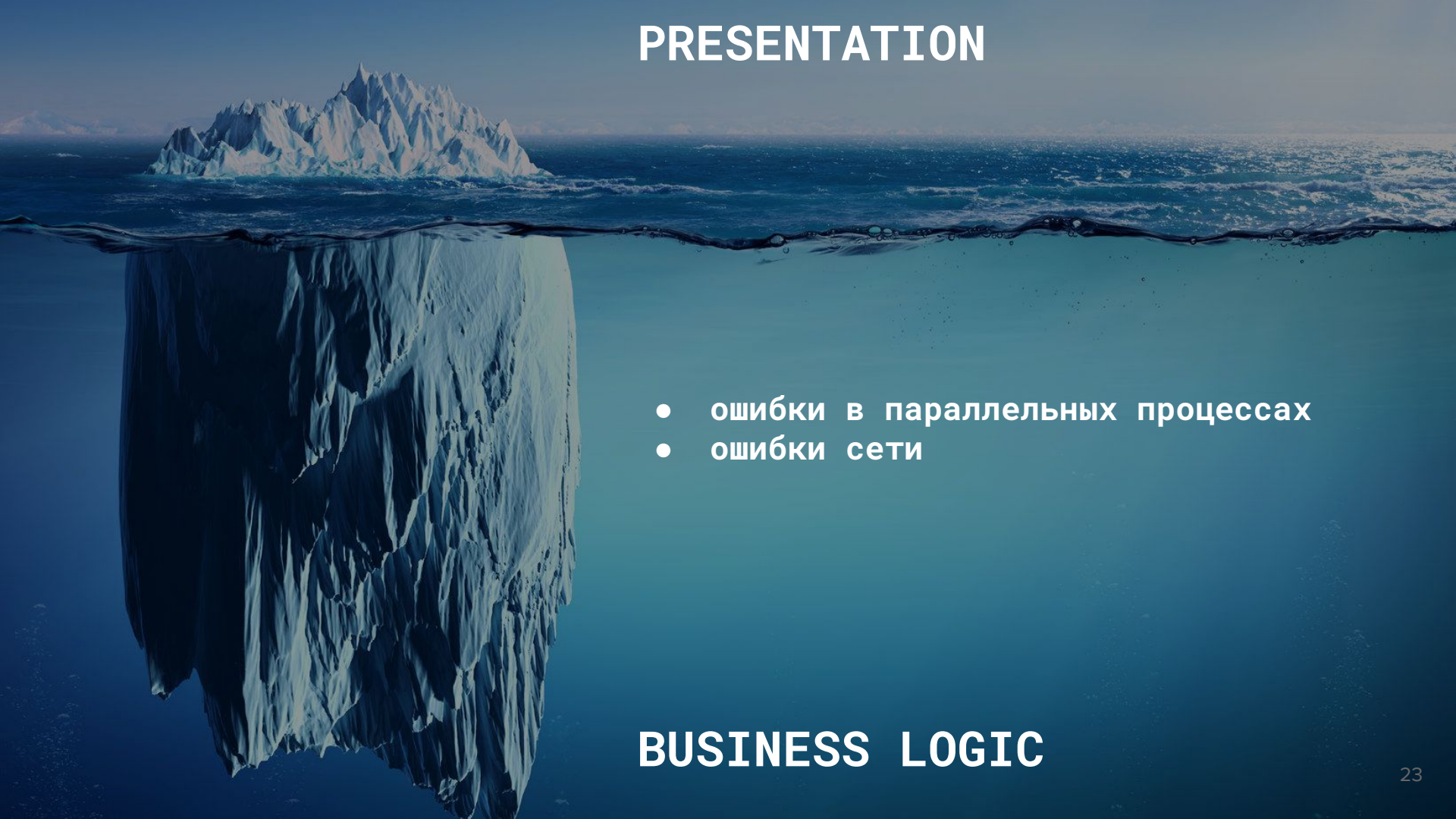
Можно пойти еще дальше...

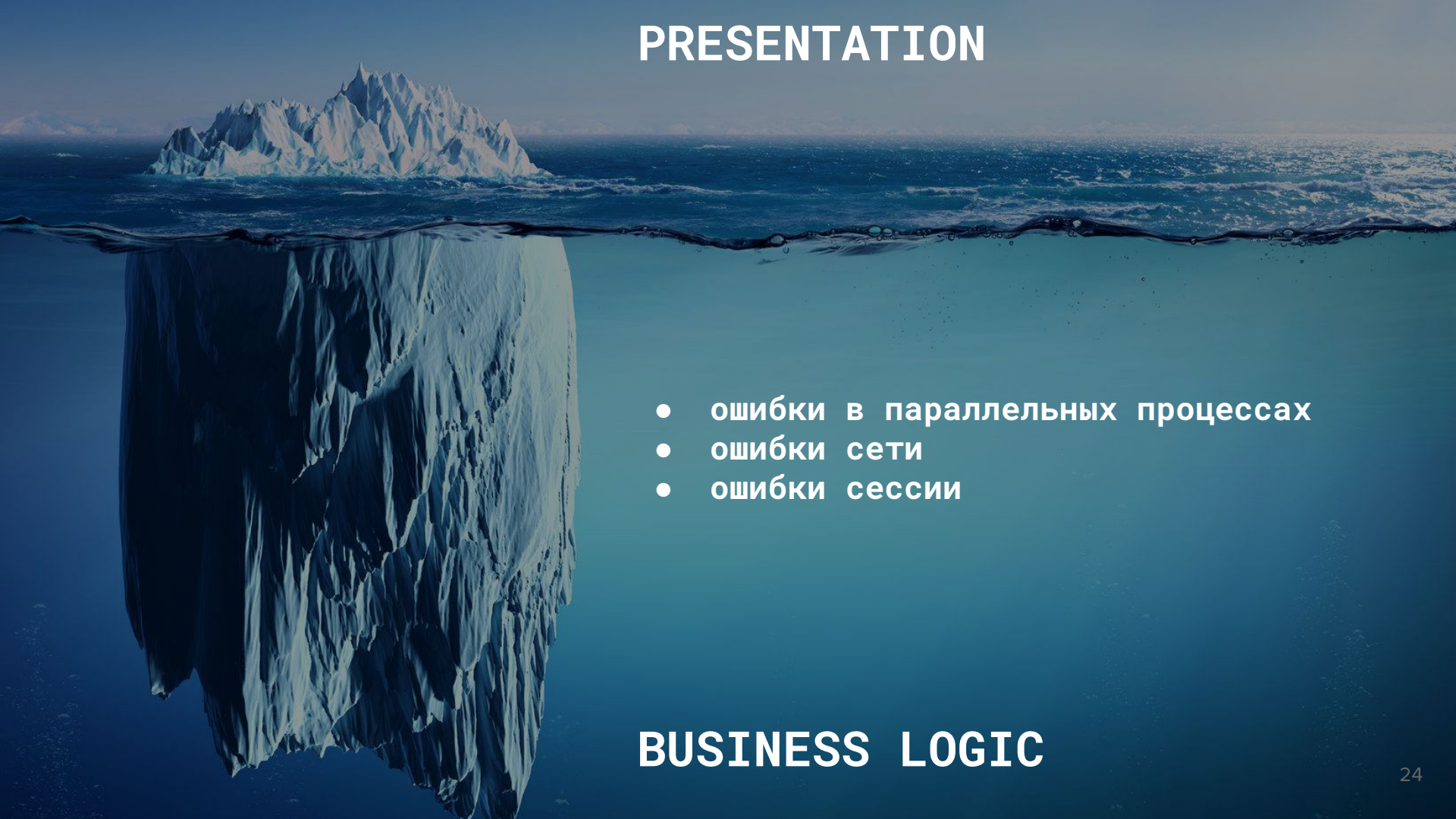
- Offline-first приложение
  Поддержка оффлайна в мобильном приложении. Синхронизация. DevFest 2018 Руслан Калбаев
- Лог запросов к серверу и отдельный сервис, гарантирующий их выполнение
  Upload в Одноклассниках. Mobius 2018. Кирилл Попов

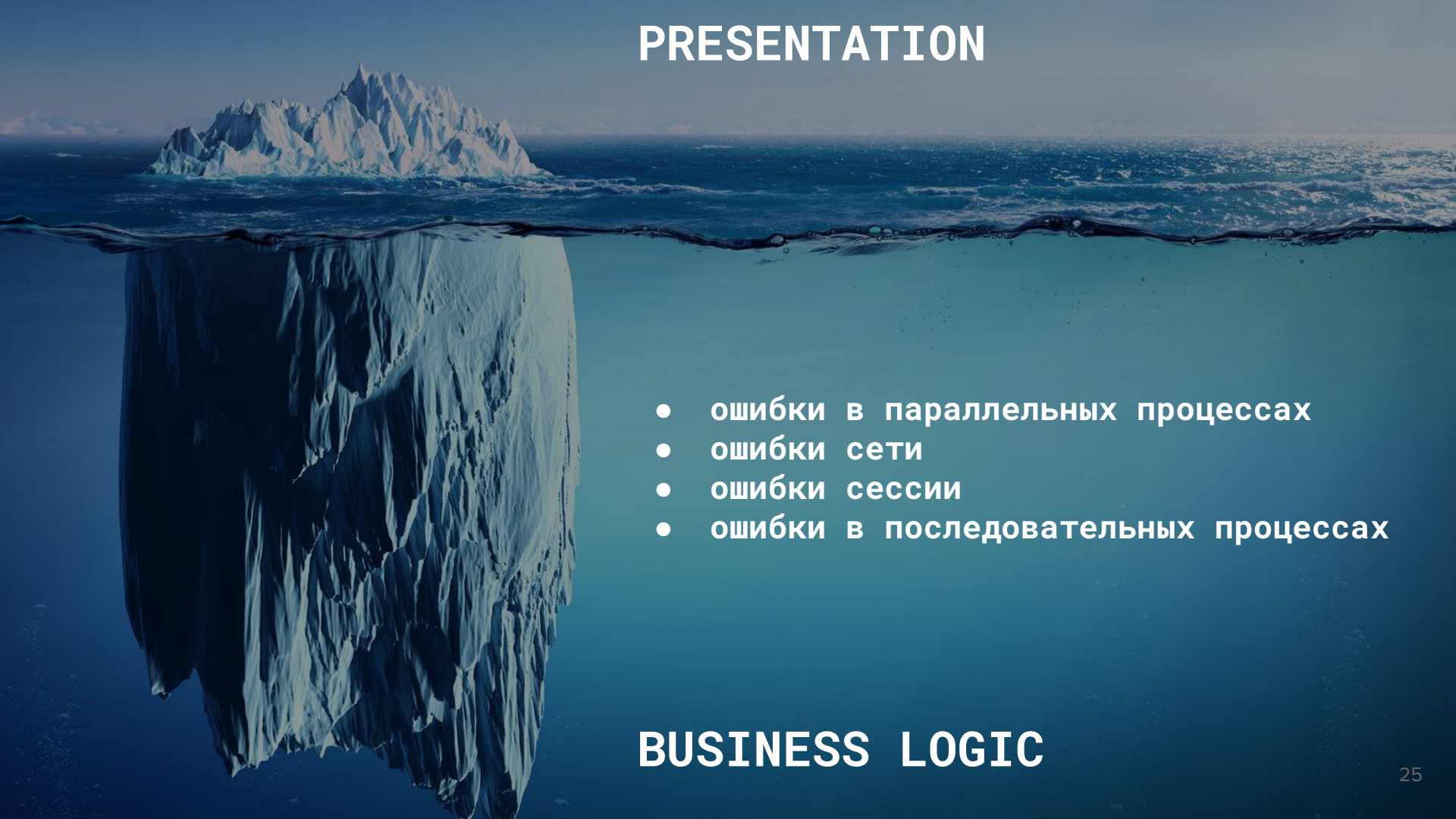
Но релиз уже близко, а такая “**простая**” вещь, как кнопочка **retry** требует дописать или переделать очень много!


PRESENTATION
BUSINESS LOGIC


PRESENTATION
ошибки в параллельных процессах
BUSINESS LOGIC


PRESENTATION
● ошибки в параллельных процессах
● ошибки сети
BUSINESS LOGIC


PRESENTATION
ошибки в параллельных процессах
ошибки сети
ошибки сессии
BUSINESS LOGIC


PRESENTATION
ошибки в параллельных процессах
ошибки сети
ошибки сессии
ошибки в последовательных процессах
BUSINESS LOGIC

“Дело было вечером”

Как бы я хотел решать такую задачу?

Как бы я хотел решать такую задачу?

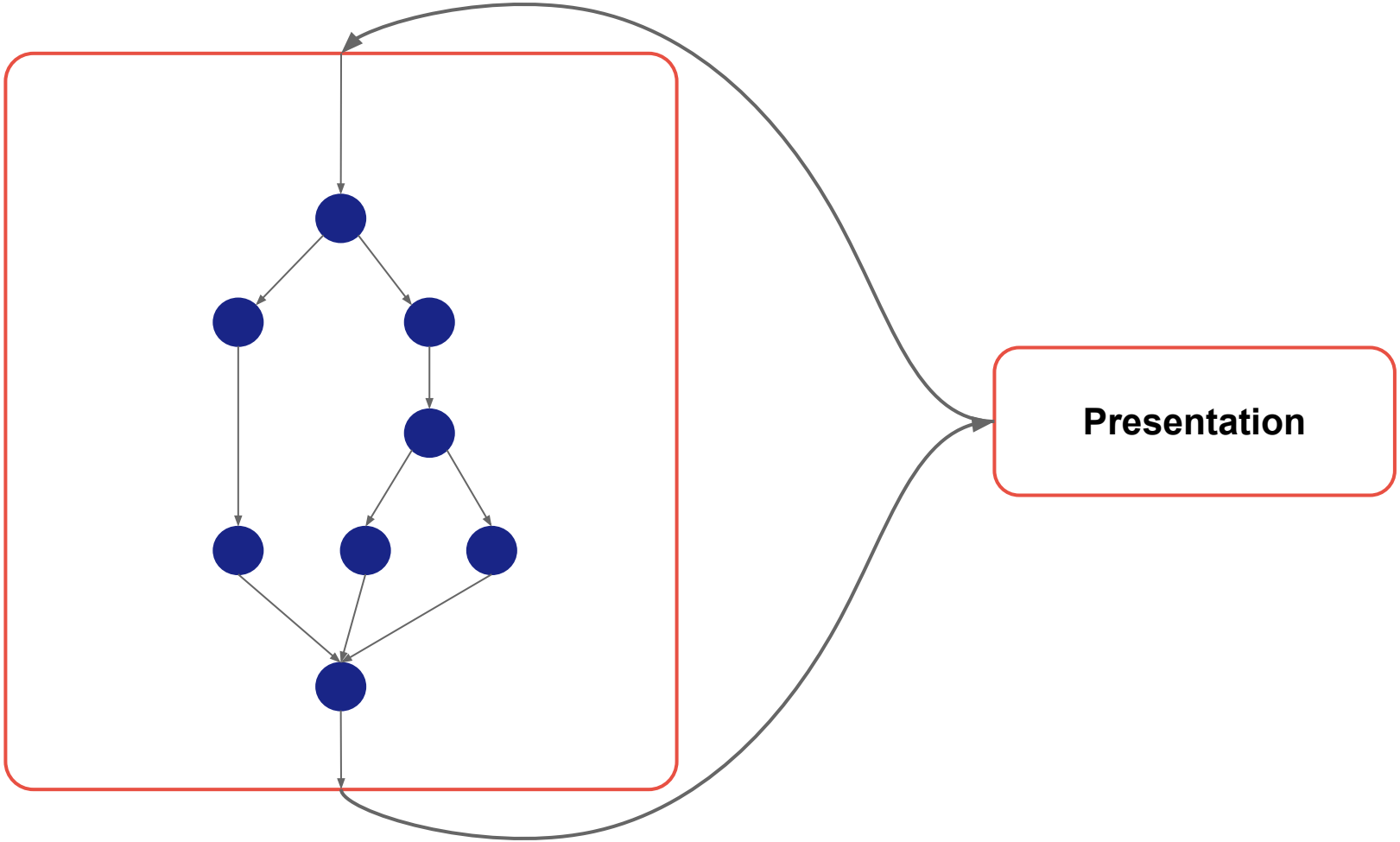
- **не трогать код презентера**, если ошибка к нему не относится

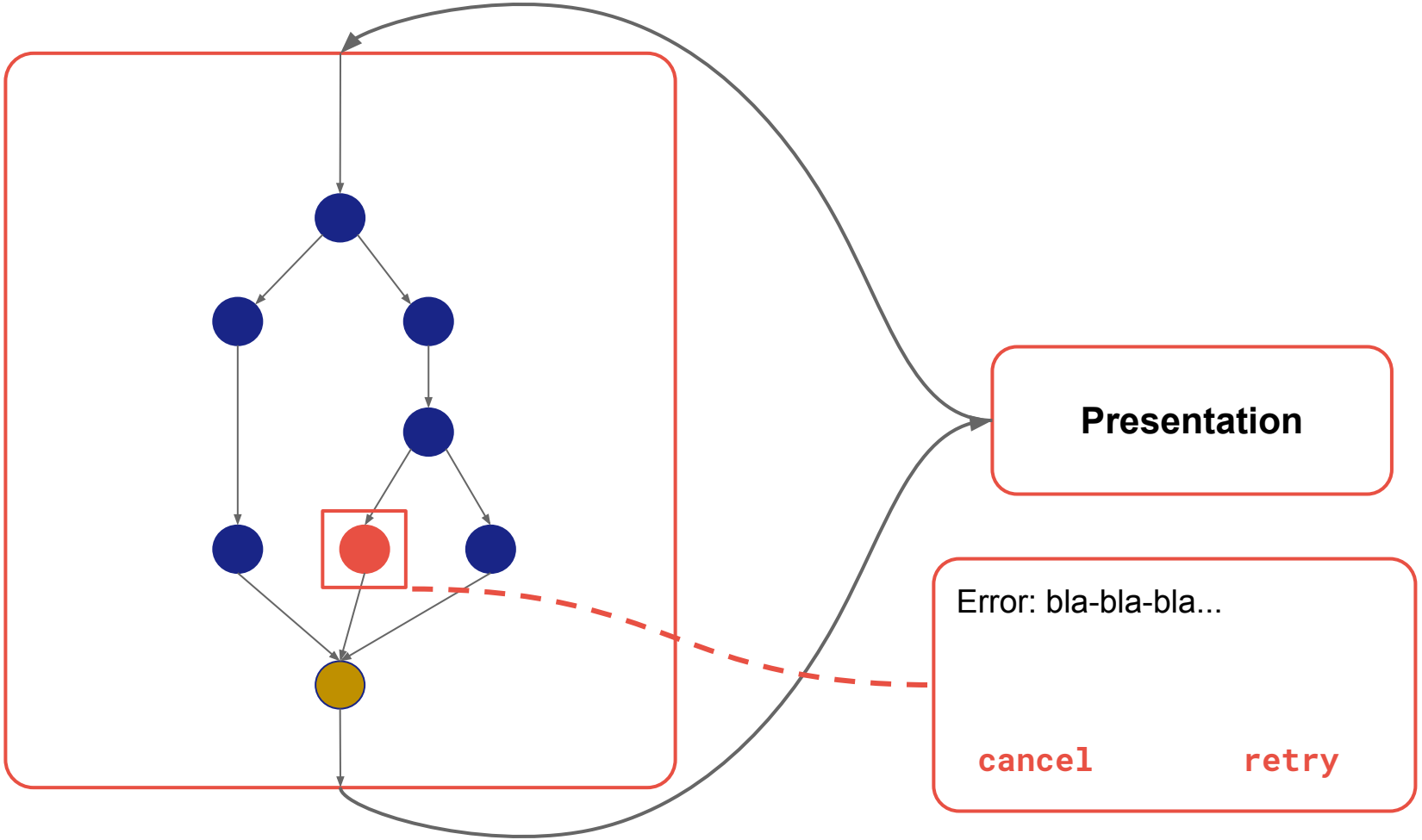
Как бы я хотел решать такую задачу?

- **не трогать код презентера**, если ошибка к нему не относится
- разобраться с общей обработкой ошибок и без проблем добавлять “туда” возможность повтора

Как бы я хотел решать такую задачу?

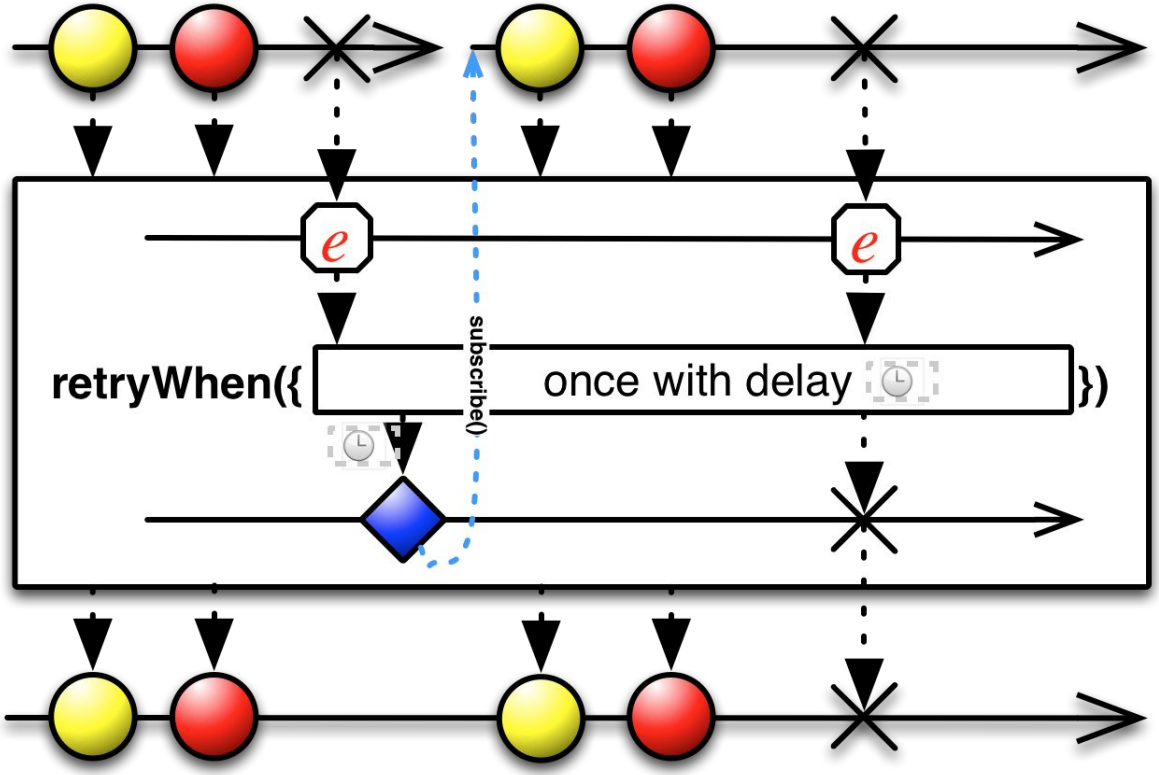
- **не трогать код презентера**, если ошибка к нему не относится
- разобраться с общей обработкой ошибок и без проблем добавлять “туда” возможность повтора
- “снаружи” даже не знать, что произошла ошибка, если ее можно “исправить” внутри


Presentation


Presentation
Error: bla-bla-bla...
cancel
retry

Решение “на коленке”

Rx-оператор **retryWhen**



Реактивный показ диалога

```
class BaseActivity : Activity() {

    fun showRetryDialog(e: Throwable, callback: (result: Boolean) -> Unit) {
        AlertDialog.Builder(this)
            .setTitle("Error!")
            .setMessage(e.toString())
            .setPositiveButton("retry") { _, _ -> callback(true) }
            .setNegativeButton("cancel") { _, _ -> callback(false) }
            .create()
            .show()
    }

}
```

Скрестим ежа с ужом

```
object ErrorHandler {
    var activity: BaseActivity? = null

    fun proceed(e: Throwable) = Completable.create { source ->
        activity?.let { a ->
            a.showRetryDialog(e) { result ->
                if (result) source.onComplete()
                else source.onError(e)
            }
        } ?: run {
            source.onError(e)
        }
    }
}
```

Подсластим код

```
fun <T> Flowable<T>.userRetry() =
    this.retryWhen { errors ->
        errors
            .observeOn(AndroidSchedulers.mainThread())
            .flatMapSingle { e ->
                ErrorHandler.proceed(e).toSingleDefault(Unit)
            }
    }
```

Оно работает!

Проблемы

- ЖЦ приложения

Проблемы

- ЖЦ приложения
- одновременные одинаковые ошибки

Проблемы

- ЖЦ приложения
- одновременные одинаковые ошибки
- протухшие диалоги

Проблемы

- ЖЦ приложения
- одновременные одинаковые ошибки
- протухшие диалоги
- зависшие процессы

Проблемы

- ЖЦ приложения
- одновременные одинаковые ошибки
- протухшие диалоги
- зависшие процессы
- все на RxJava

Проблемы

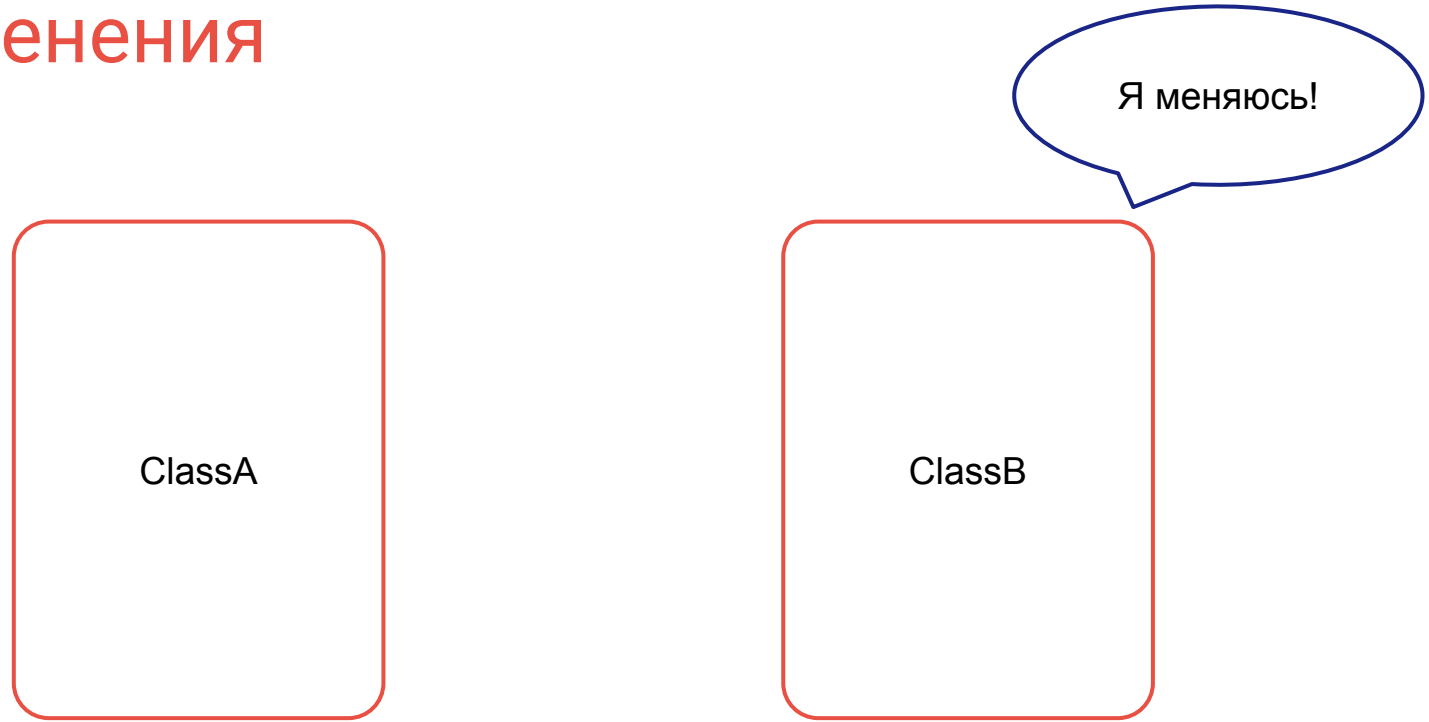
- ЖЦ приложения
- одновременные одинаковые ошибки
- протухшие диалоги
- зависшие процессы
- все на RxJava
- утечки и многое другое

Реактивная парадигма

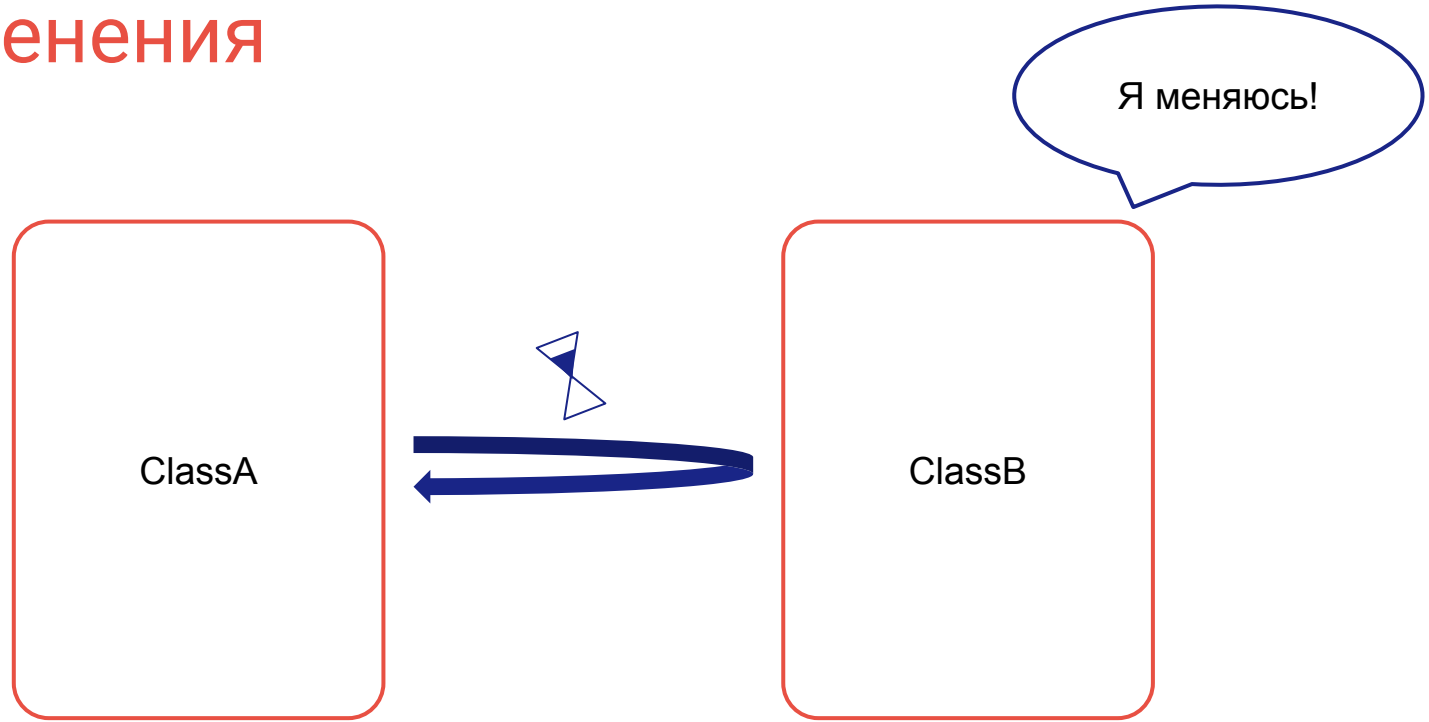
Реактивное программирование

Wikipedia: парадигма программирования, ориентированная на потоки данных и распространение изменений.

Изменения

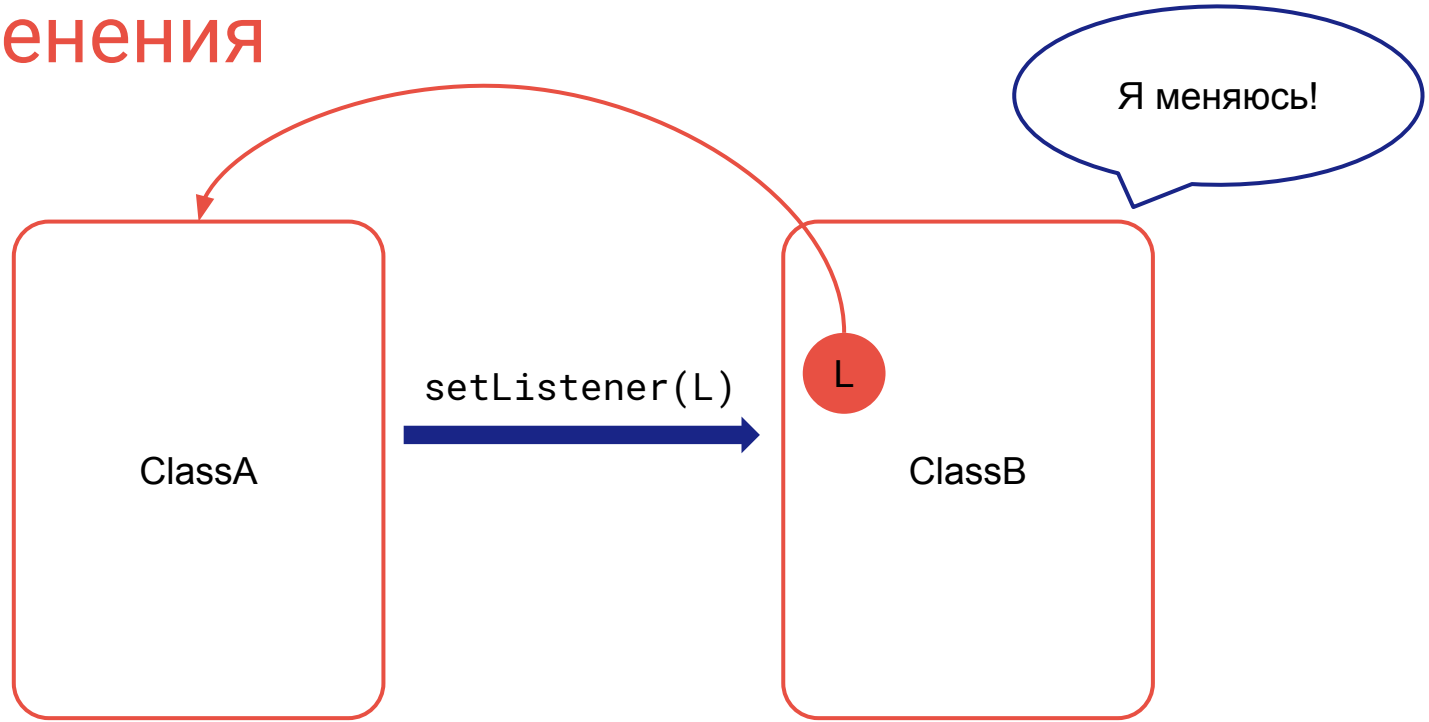


Изменения



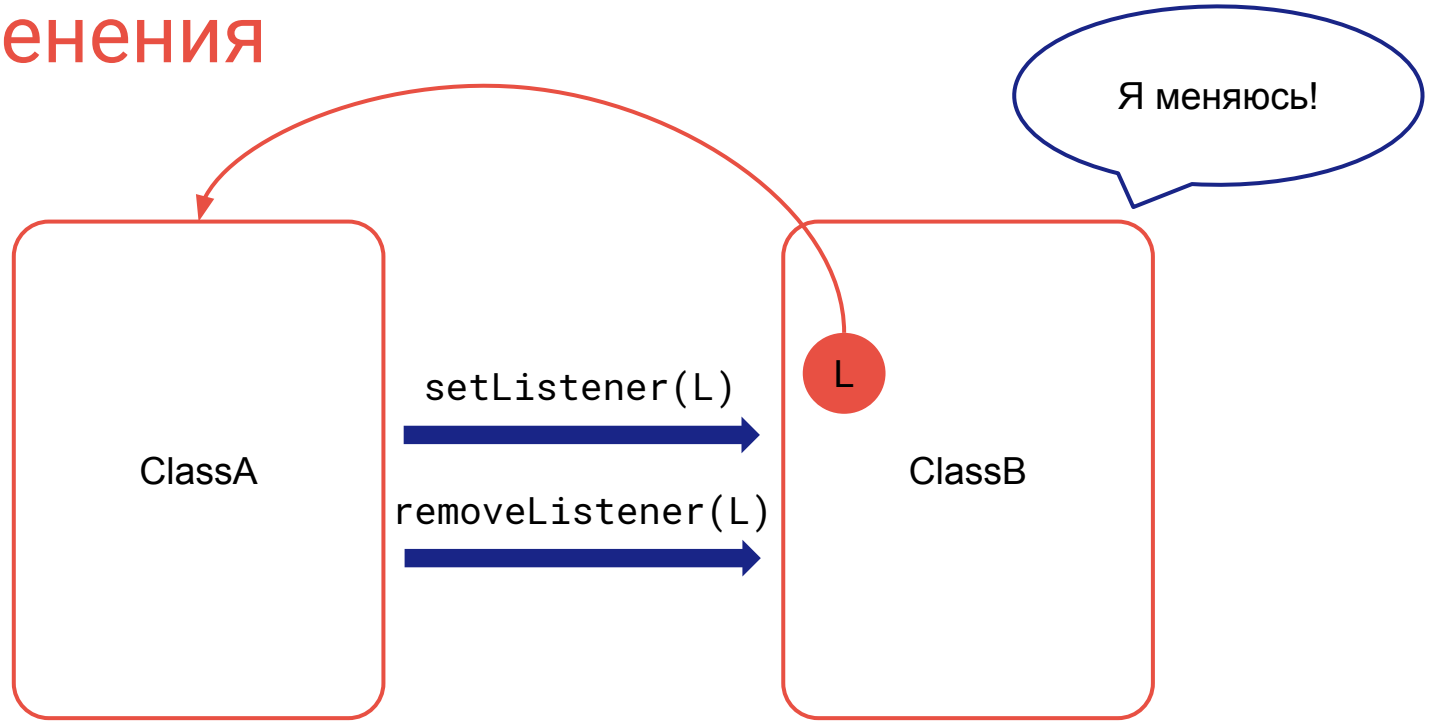
Не реактивно!

Изменения



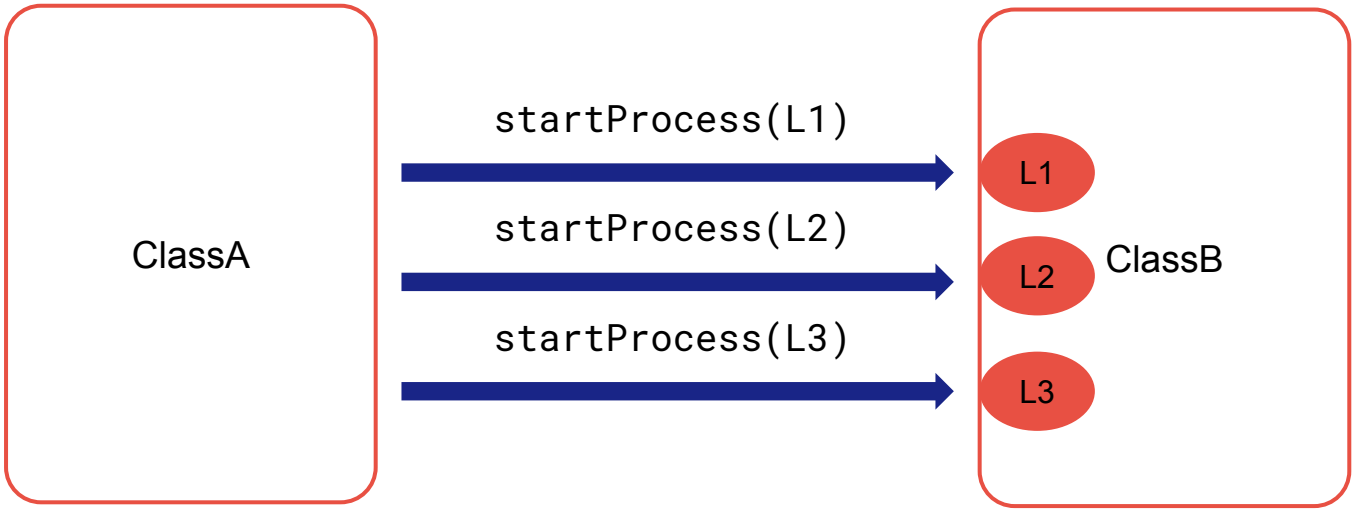
Глобальный слушатель!

Изменения



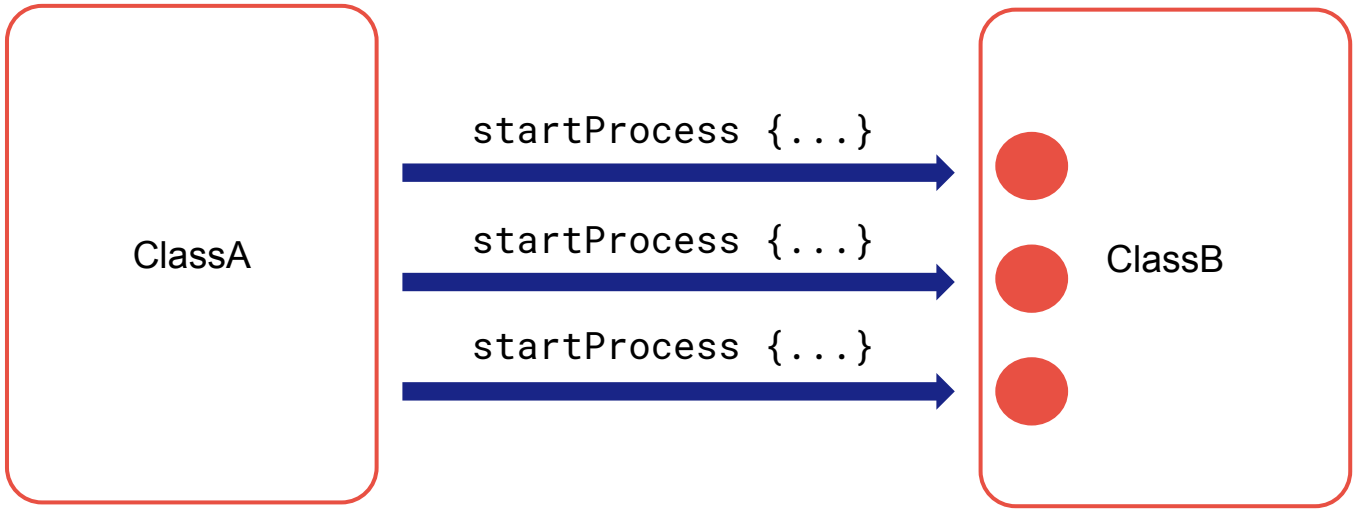
Глобальный слушатель!

Изменения



Локальные слушатели!

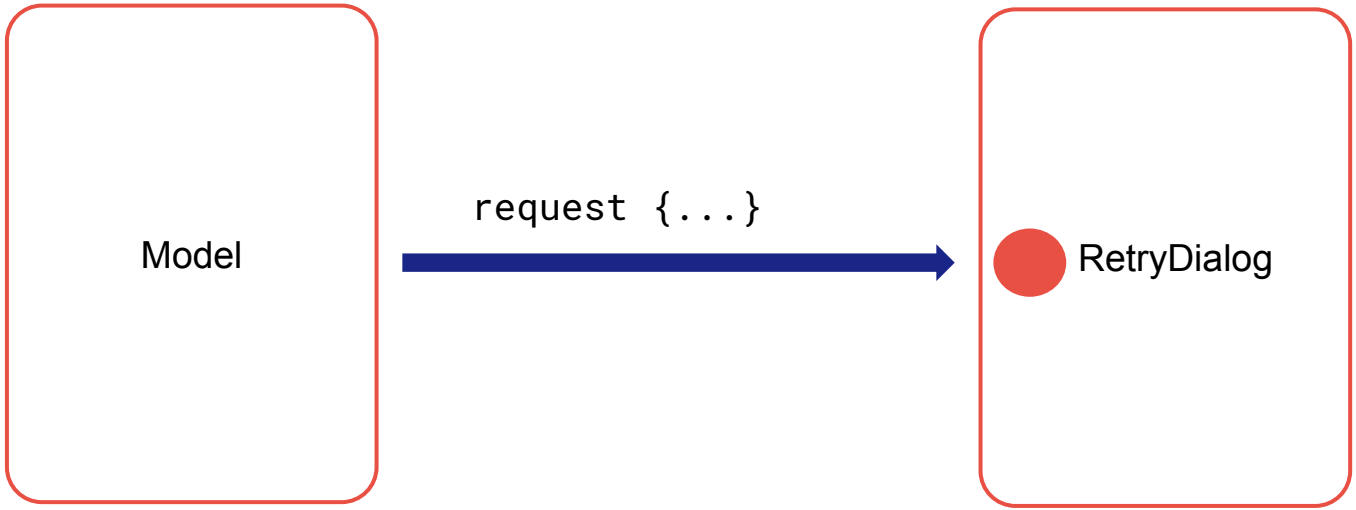
Изменения



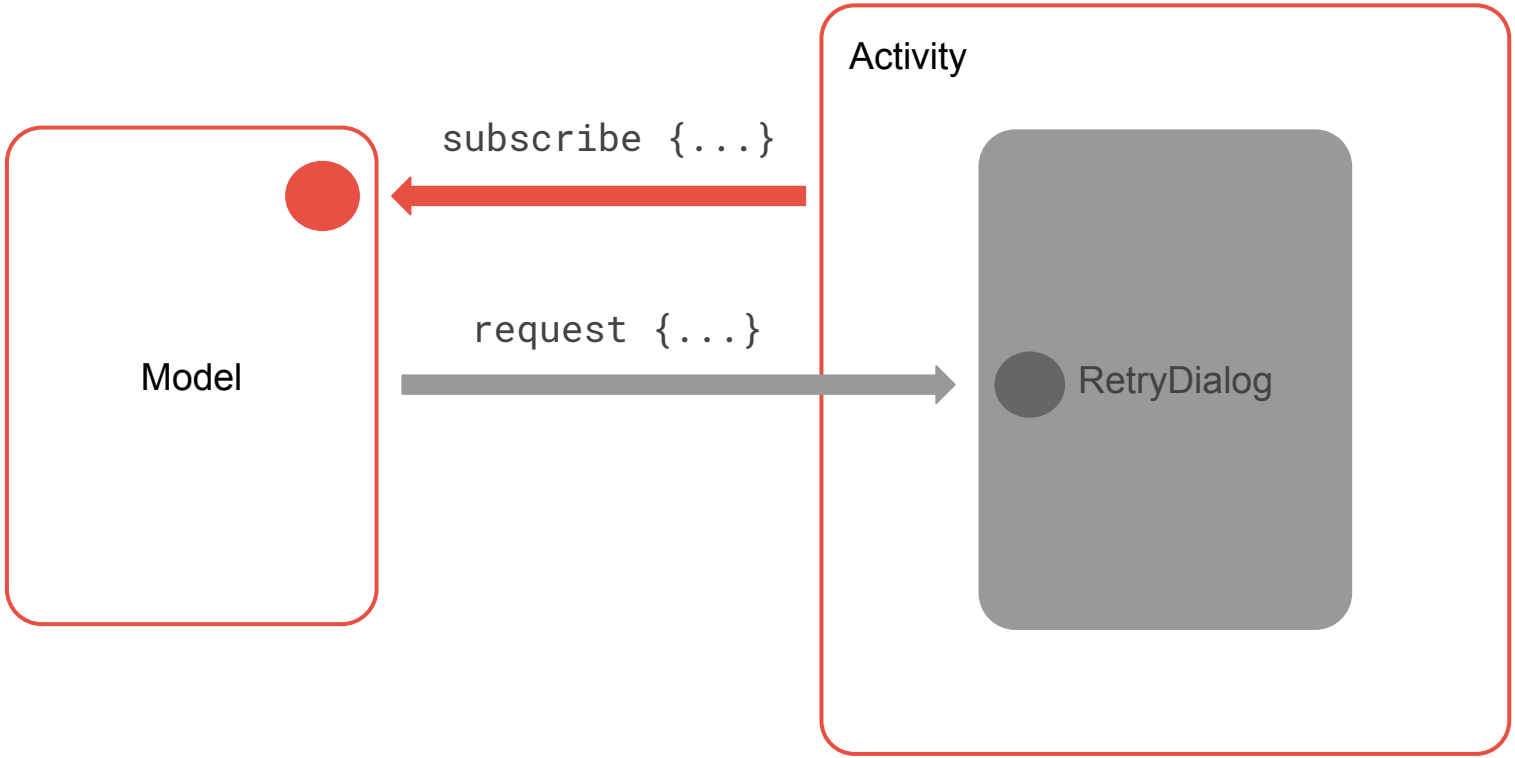
Лаконичный код!

Где изменения в нашем случае?

Ожидание ответа пользователя



Ожидание запроса, чтобы показать диалог

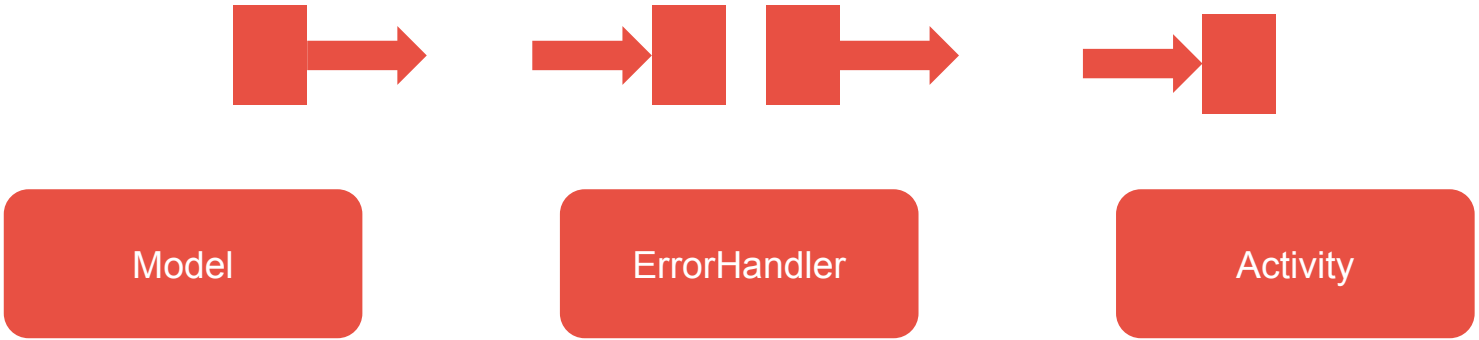


Emitter и Subscriber

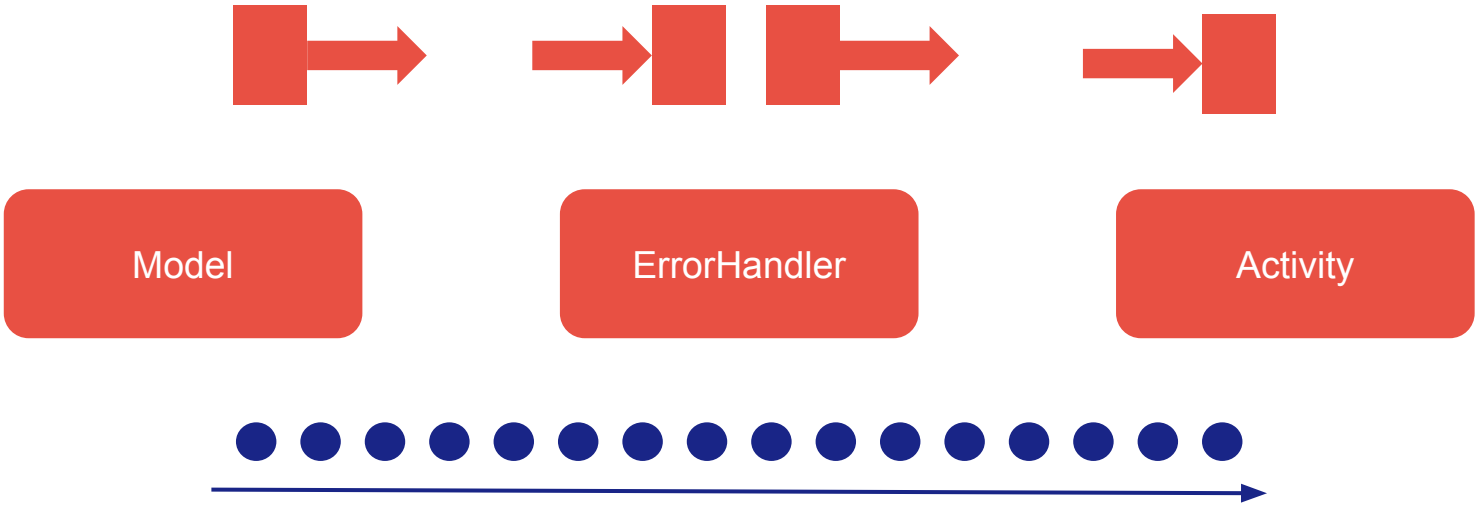
Emitter и Subscriber

```
interface Emitter {
    fun subscribe(subscriber: Subscriber)
    fun unsubscribe()
}

interface Subscriber {
    fun onNext(
        error: Throwable,
        callback: (retry: Boolean) -> Unit
    )
}
```

Emitter и Subscriber

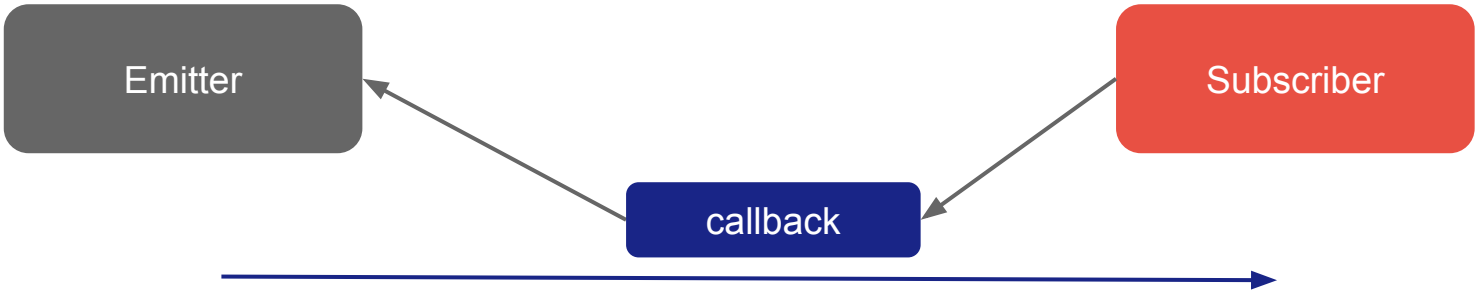


Emitter и Subscriber




```
Событие = данные + замыкание (лямбда)
```

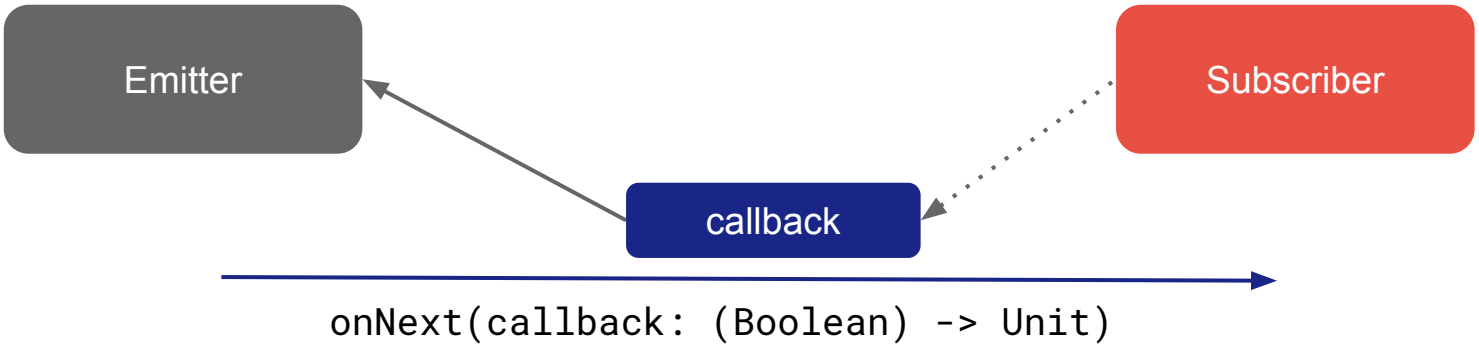
Борьба с утечкой ссылки




```
onNext(callback: (Boolean) -> Unit)
```

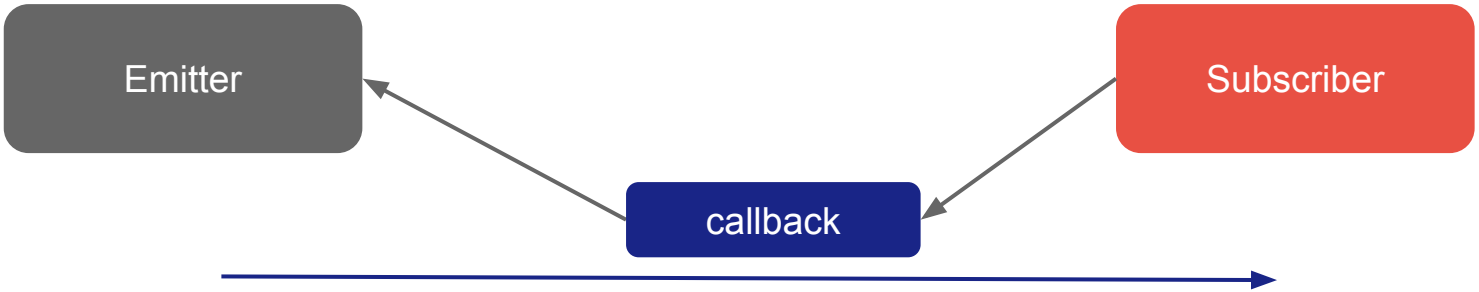
Борьба с утечкой ссылки

Вариант 1: сохранять ссылку на лямбду как слабую



Борьба с утечкой ссылки

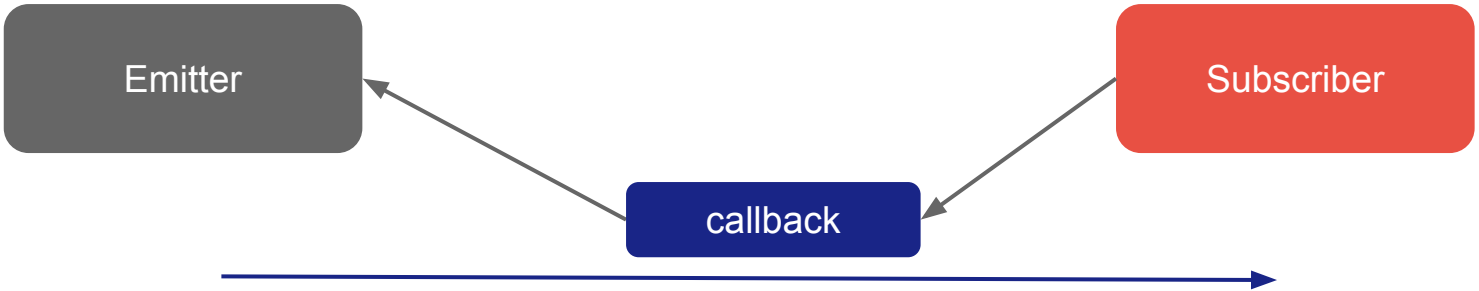
Вариант 2: отдельный метод для завершения процесса




```
onNext(callback: (Boolean) -> Unit)

cancel(callback: (Boolean) -> Unit)
```

Борьба с утечкой ссылки

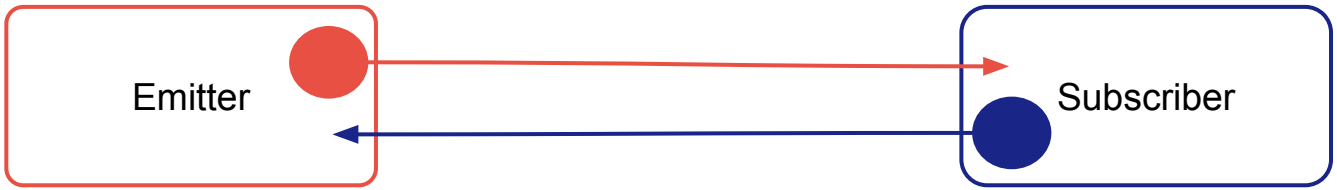
Вариант 2б: использовать код запроса




```
onNext(callback: (Boolean) -> Unit): Long

cancel(id: Long)
```

Борьба с утечкой ссылки

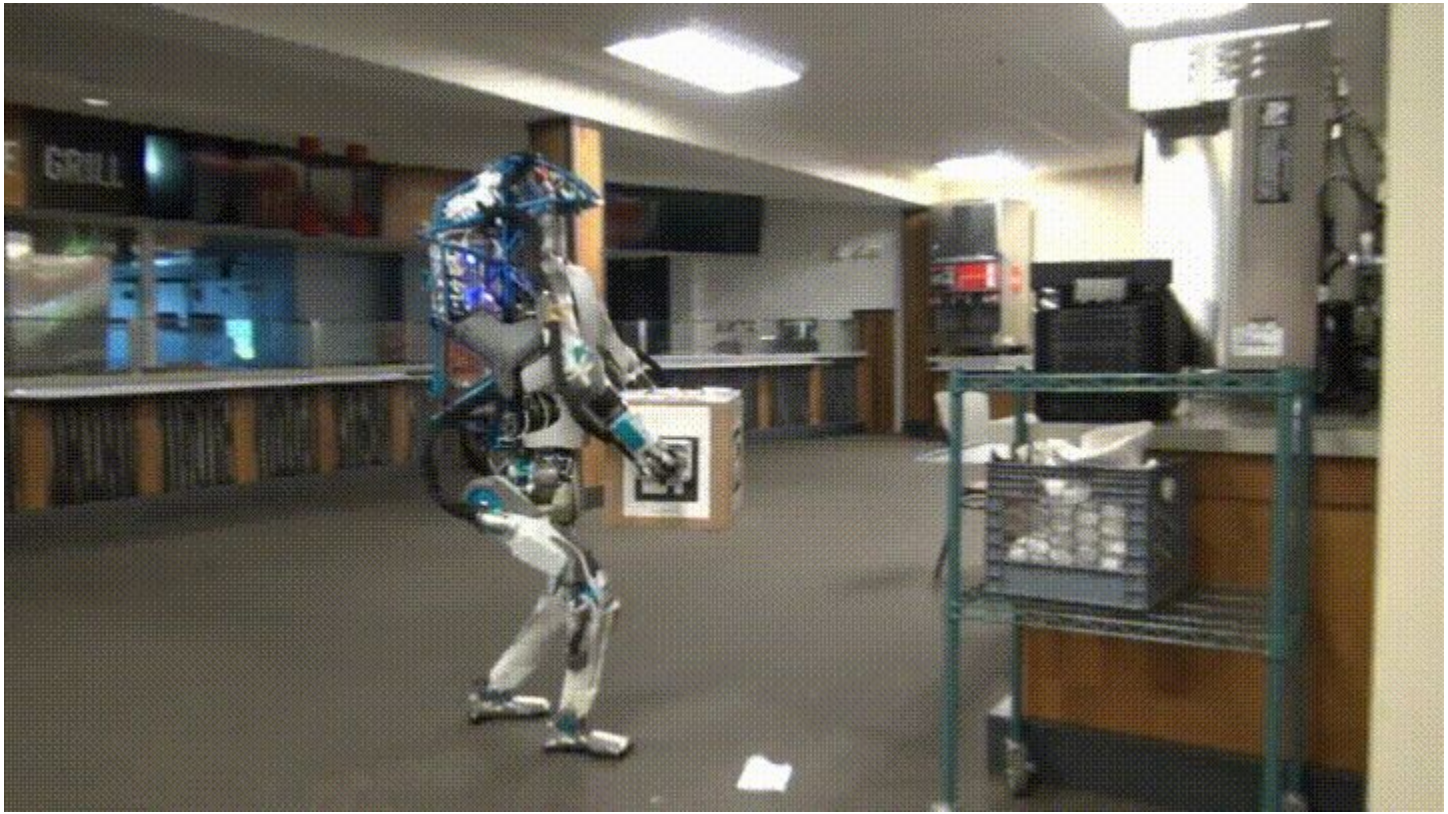
Вариант 3: использовать еще одну лямбду!




```
onNext(callback: (Boolean) -> Unit): Disposable
```

Disposable

```
interface Disposable {
    fun dispose()
}

interface Subscriber {
    fun onNext(
        error: Throwable,
        callback: (retry: Boolean) -> Unit
    ): Disposable
}
```



почти все хорошо, но

RetryDialogSubscriber

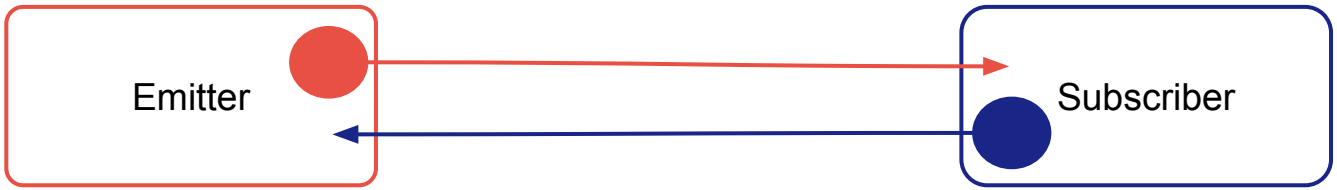
```
override fun onNext(
  error: Throwable, callback: (retry: Boolean) -> Unit
): ScaremongerDisposable {

   AlertDialog.Builder(activity).apply {
       setTitle(titleText)
       setMessage(msgCreator(error))
       setPositiveButton(retryButtonText) { _, _ -> callback(true) }
       setNegativeButton(cancelButtonText) { _, _ -> callback(false) }
       setCancelable(false)
   }.create().show()

}
```

RetryDialogSubscriber

```
override fun onNext(
  error: Throwable, callback: (retry: Boolean) -> Unit
): ScaremongerDisposable {

   val d = AlertDialog.Builder(activity).apply {...}.create()
   d.show()

   return object : ScaremongerDisposable {
       override fun dispose() {
           d.dismiss()
       }
   }

}
```

Борьба с утечкой ссылки

Вариант 3: использовать еще одну лямбду!

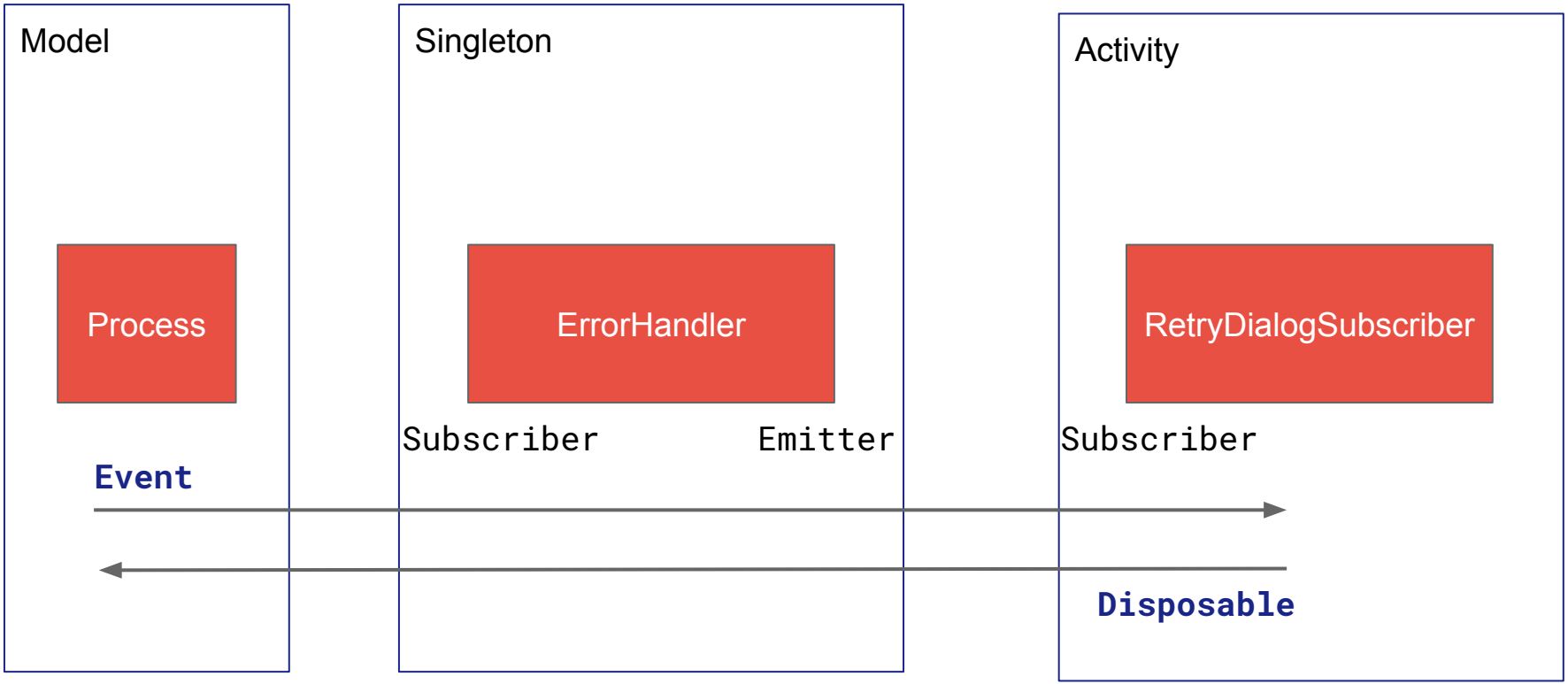


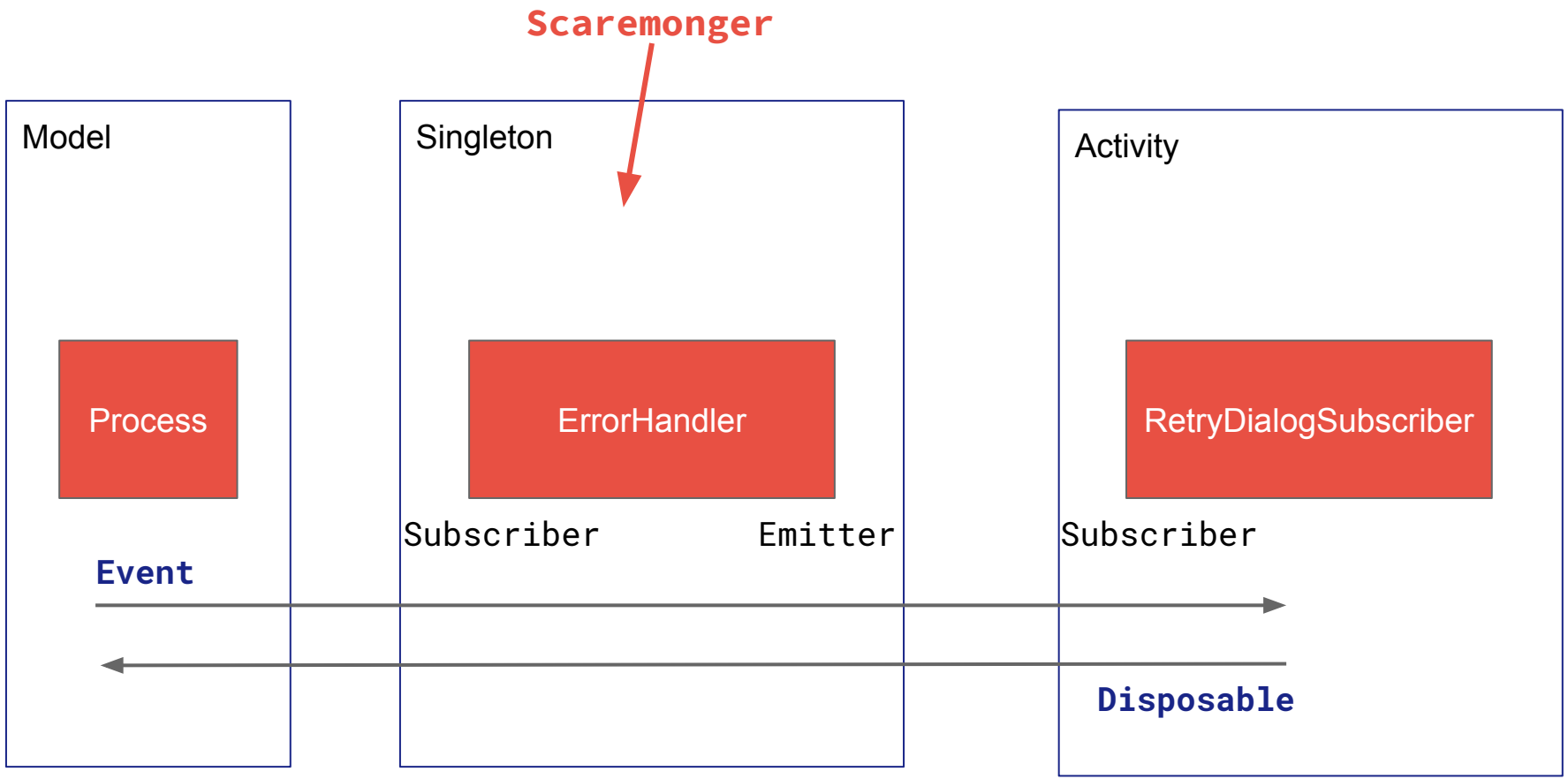

```
onNext(callback: (Boolean) -> Unit): Disposable
```

Disposable

```
class Disposable(private val block: () -> Unit) {
    var isDisposed: Boolean = false

    fun dispose() {
        if (!isDisposed) block()
    }
}
```

Избавились от RxJava


Model
Singleton
Activity
Process
ErrorHandler
RetryDialogSubscriber
Subscriber
Emitter
Subscriber
Event
Disposable


Scaremonger
Model
Singleton
Activity
Process
ErrorHandler
RetryDialogSubscriber
Subscriber
Emitter
Subscriber
Event
Disposable

Scaremonger

```
object Scaremonger : ScaremongerEmitter, ScaremongerSubscriber {

}
```

Scaremonger

```
object Scaremonger : ScaremongerEmitter, ScaremongerSubscriber {

    private var subscriber: ScaremongerSubscriber? = null

    override fun subscribe(subscriber: ScaremongerSubscriber) {
        this.subscriber = subscriber
    }

    override fun unsubscribe() {
        this.subscriber = null
    }

}
```

Scaremonger

```
object Scaremonger : ScaremongerEmitter, ScaremongerSubscriber {
    override fun onNext(
        error: Throwable,
        callback: (retry: Boolean) -> Unit
    ): ScaremongerDisposable {
        subscriber?.let { s ->
            return s.request(error, callback)
        } ?: run {
            callback(false)
            return FakeDisposable
        }
    }
}
```

RXx = RX extensions

```
private fun Flowable<Throwable>.handleErrors() =
    this
        .observeOn(AndroidSchedulers.mainThread())
        .flatMapSingle<Unit> { err ->
            Single.create<Unit> { s ->
                Scaremonger.onNext(err) { retry ->
                    if (retry) s.onSuccess(Unit) else s.onError(err)
                }
            }
        }
```

RXx = RX extensions

```
private fun Flowable<Throwable>.handleErrors() =
    this
        .observeOn(AndroidSchedulers.mainThread())
        .flatMapSingle<Unit> { err ->
            Single.create<Unit> { s ->
                Scaremonger.onNext(err) { retry ->
                    if (retry) s.onSuccess(Unit) else s.onError(err)
                }
            }
        }
```

RXx = RX extensions

```
Single.create<Unit> { s ->
    Scaremonger.onNext(err) { retry ->
        if (retry) s.onSuccess(Unit) else s.onError(err)
    }
}
```

RXx = RX extensions

```
Single.create<Unit> { s ->
    val disposable = Scaremonger.onNext(err) { retry ->
        if (retry) s.onSuccess(Unit) else s.onError(err)
    }

    s.setDisposable(object : Disposable {
        private var disposed = false
        override fun isDisposed() = disposed

        override fun dispose() {
            disposed = true
            disposable.dispose()
        }
    })
}
```

RXx = RX extensions

```
fun <T> Flowable<T>.scaremonger() = retryWhen { it.handleErrors() }
```

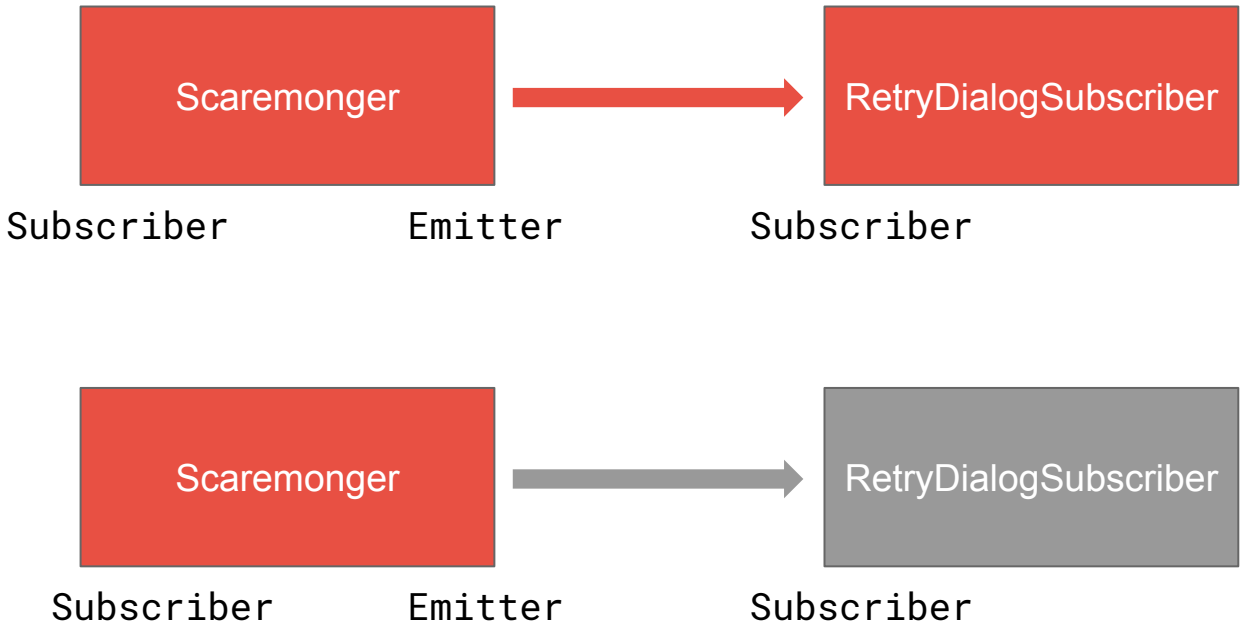
Проблемы

- ЖЦ приложения
- одновременные одинаковые ошибки
- ~~протухшие диалоги~~
- зависшие процессы
- ~~все на RxJava~~

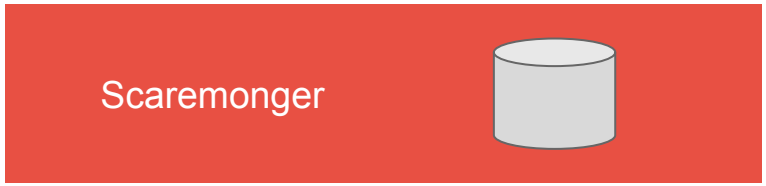
Обработаем ЖЦ

Проблема с ЖЦ

Проблема с ЖЦ

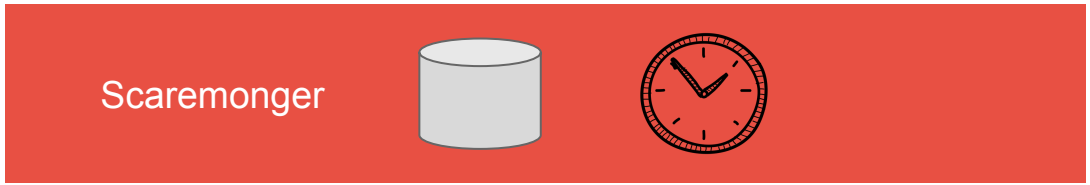


Проблема с ЖЦ



`Subscriber` `Emitter`

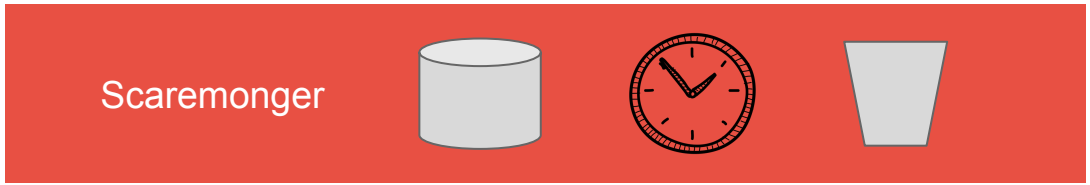
Зависшие процессы



Subscriber

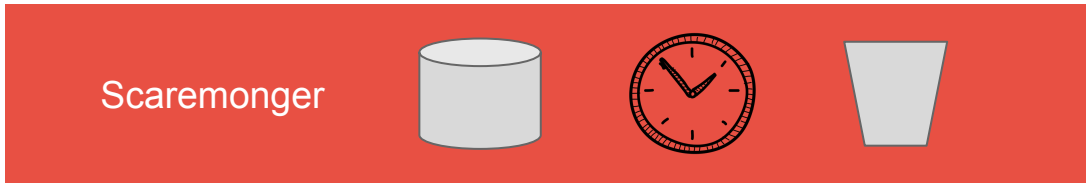
Emitter

Одинаковые ошибки

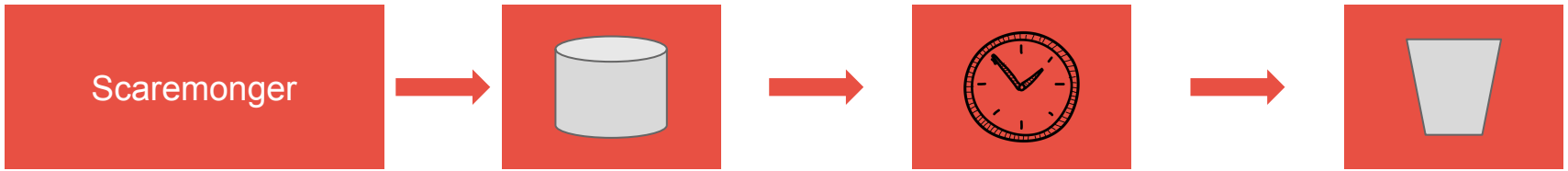


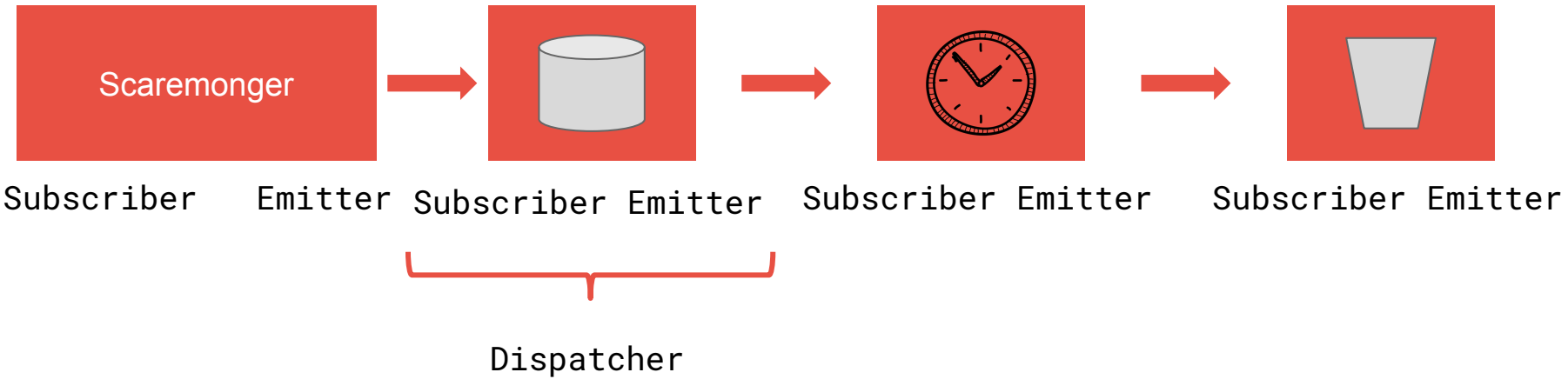
`Subscriber` `Emitter`

Одинаковые ошибки



`Subscriber` `Emitter`

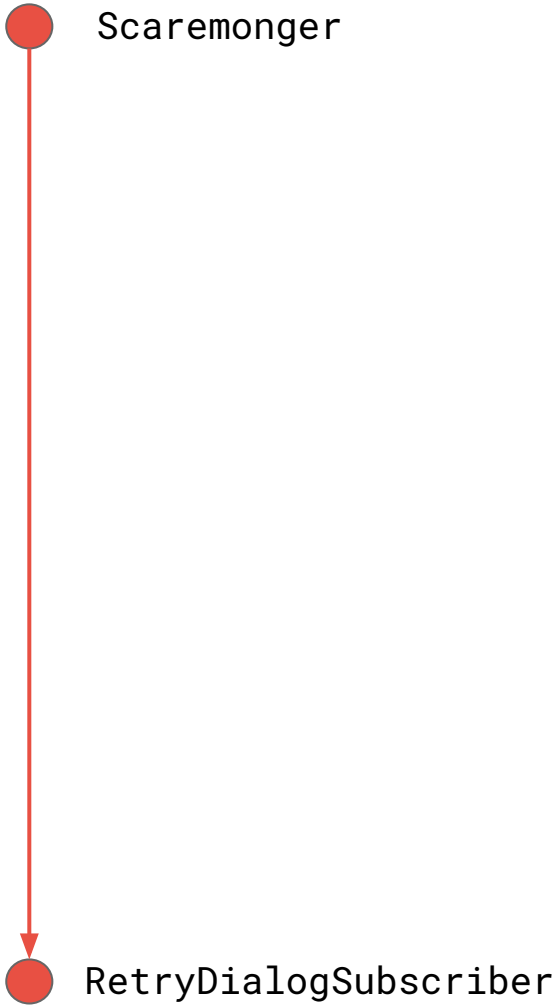

Scaremonger
Subscriber
Emitter
Subscriber
Emitter
Subscriber
Emitter
Subscriber
Emitter


Scaremonger
Subscriber
Emitter
Subscriber Emitter
Dispatcher
Subscriber Emitter
Subscriber Emitter

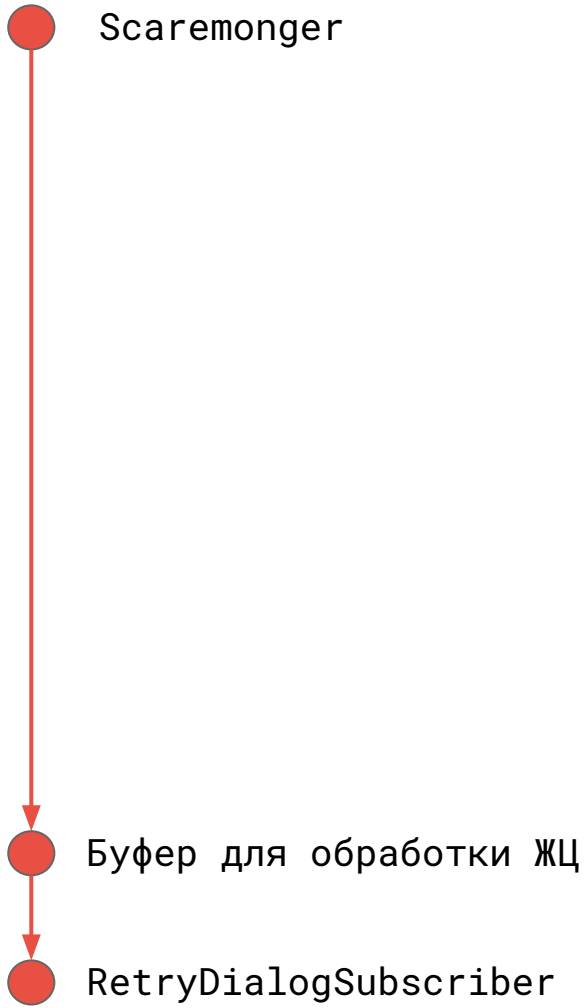
Dispatcher

```
interface ScaremongerDispatcher : ScaremongerEmitter, ScaremongerSubscriber
```

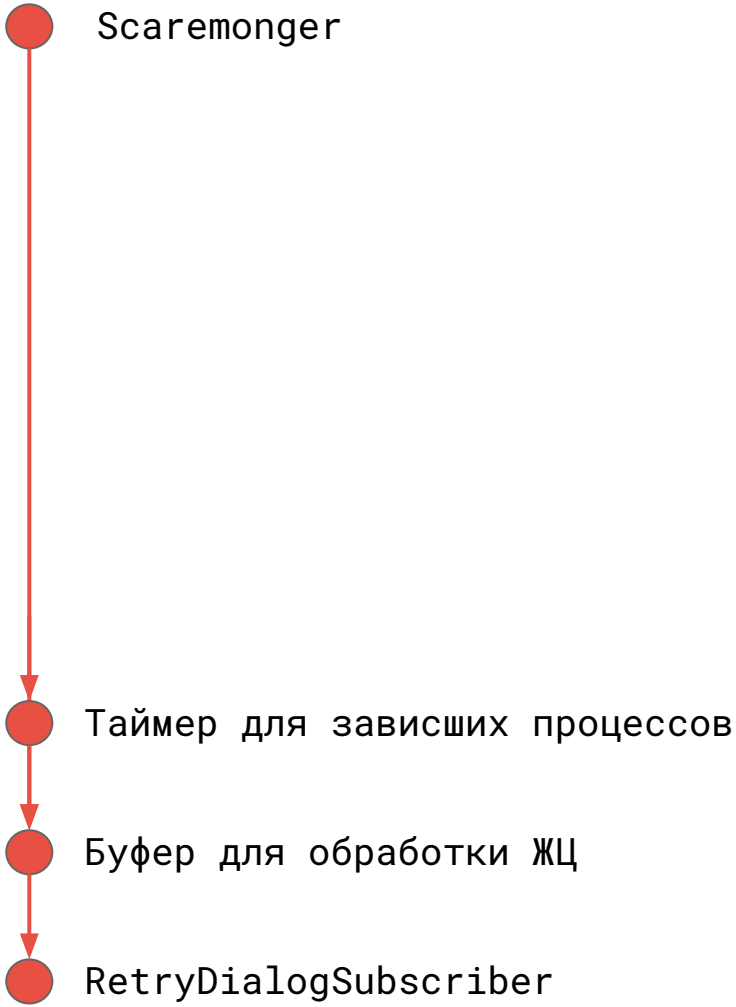
Dispatcher



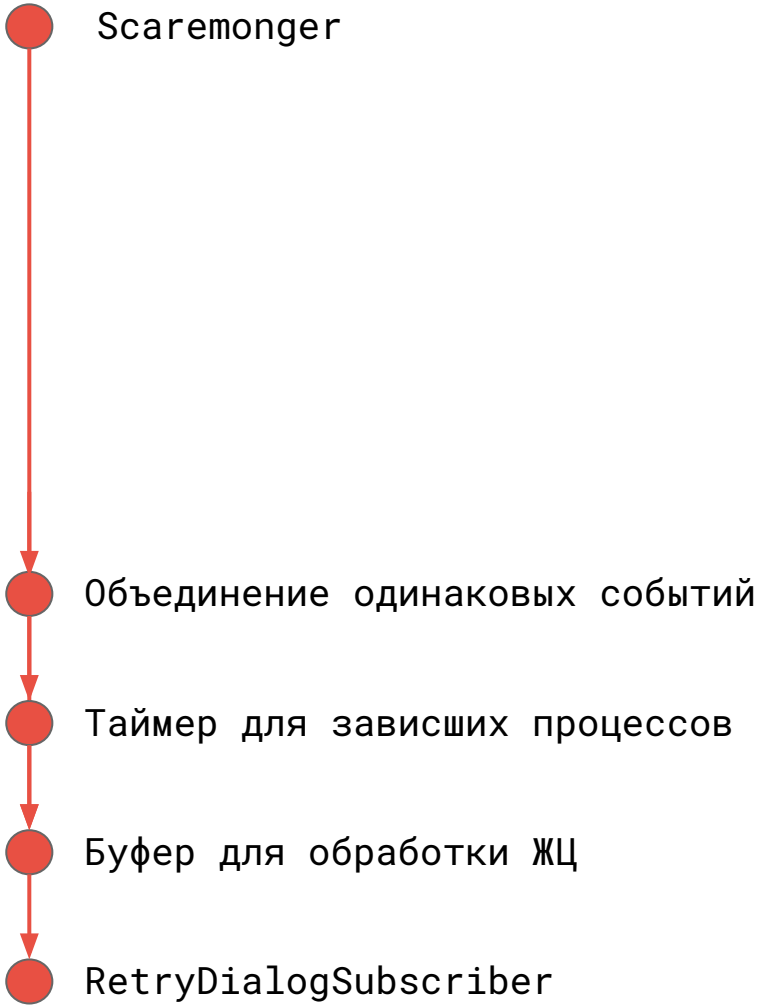
Dispatcher


Scaremonger
Буфер для обработки ЖЦ
RetryDialogSubscriber

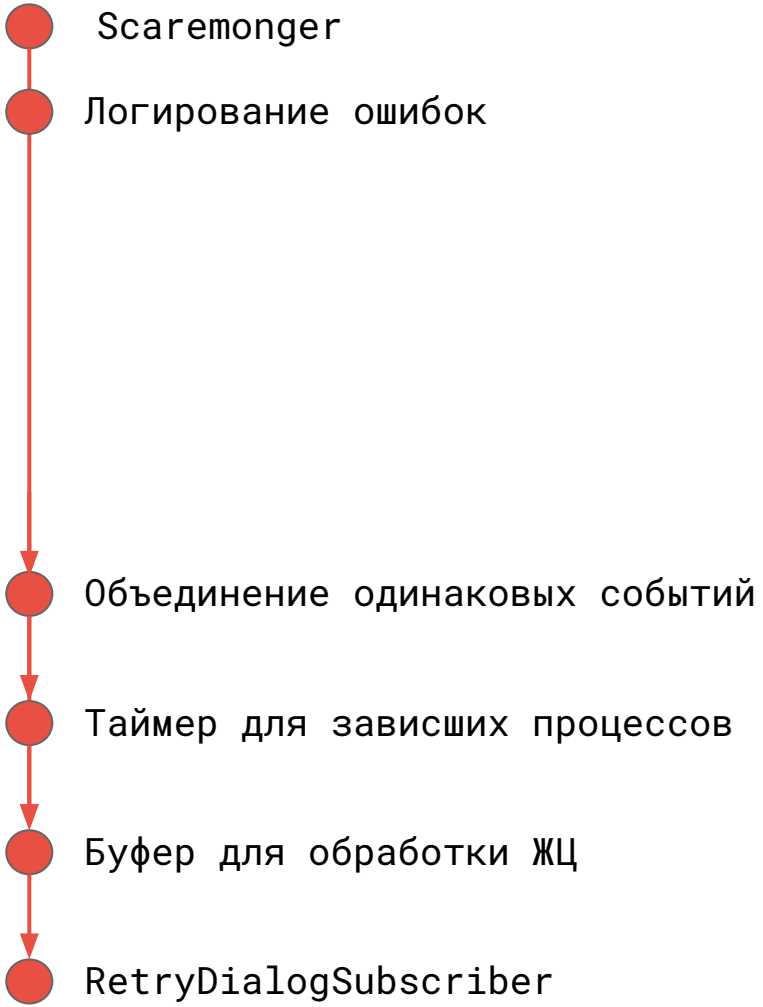
Dispatcher


Scaremonger
Таймер для зависших процессов
Буфер для обработки ЖЦ
RetryDialogSubscriber

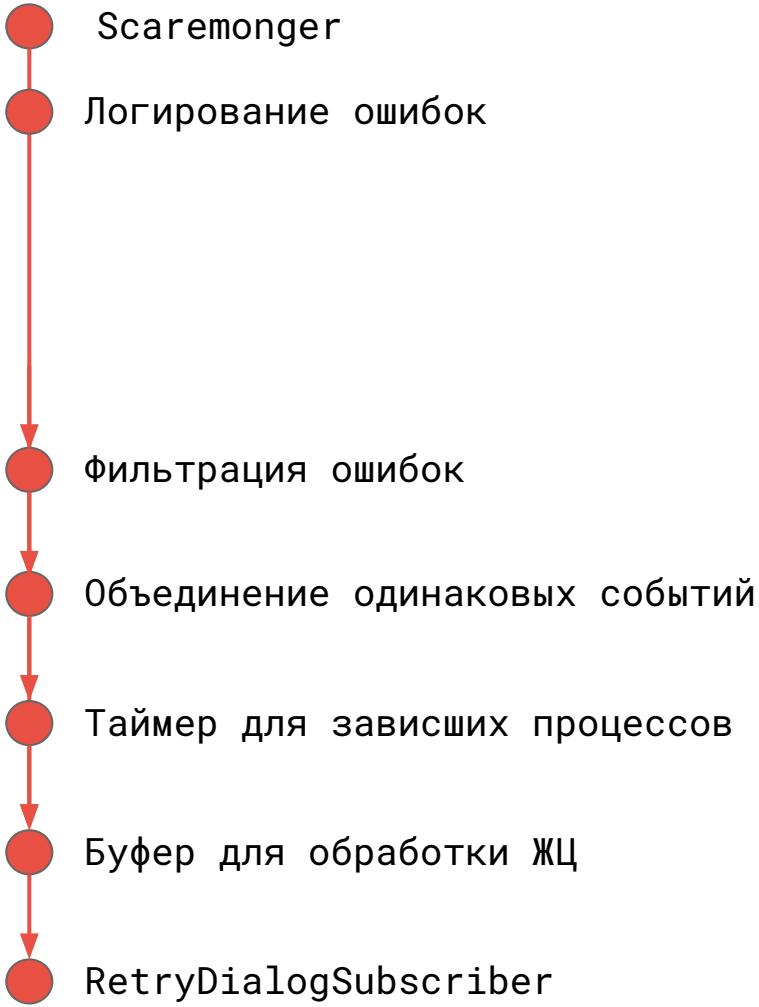
Dispatcher


Scaremonger
Объединение одинаковых событий
Таймер для зависших процессов
Буфер для обработки ЖЦ
RetryDialogSubscriber

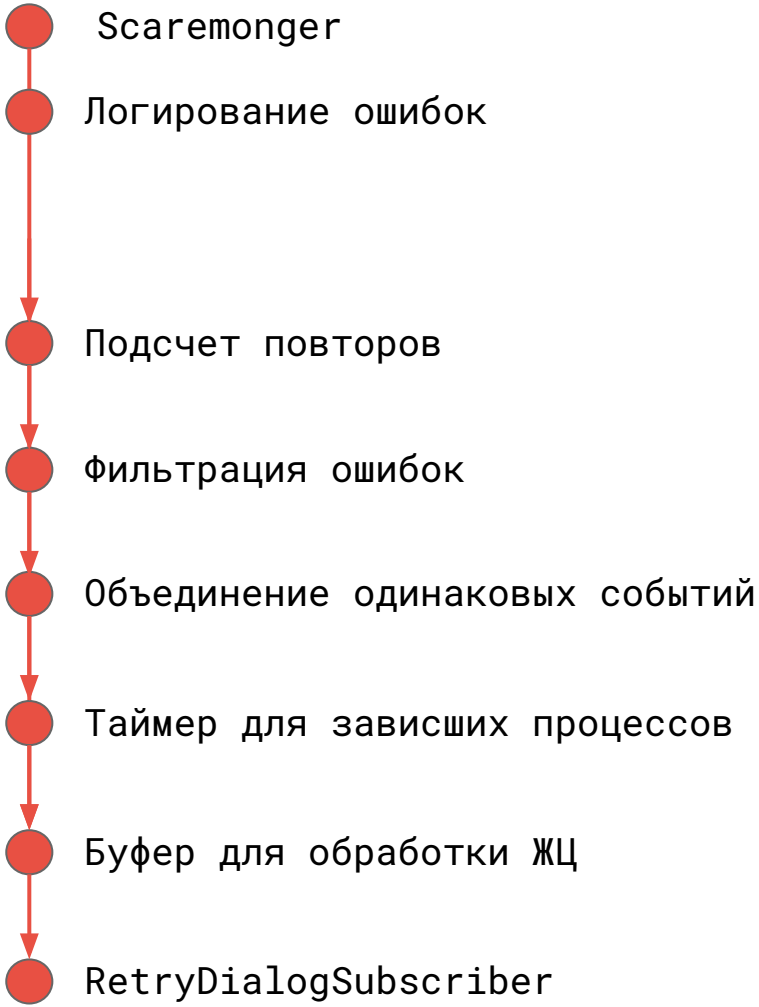
Dispatcher


Scaremonger
Логирование ошибок
Объединение одинаковых событий
Таймер для зависших процессов
Буфер для обработки ЖЦ
RetryDialogSubscriber

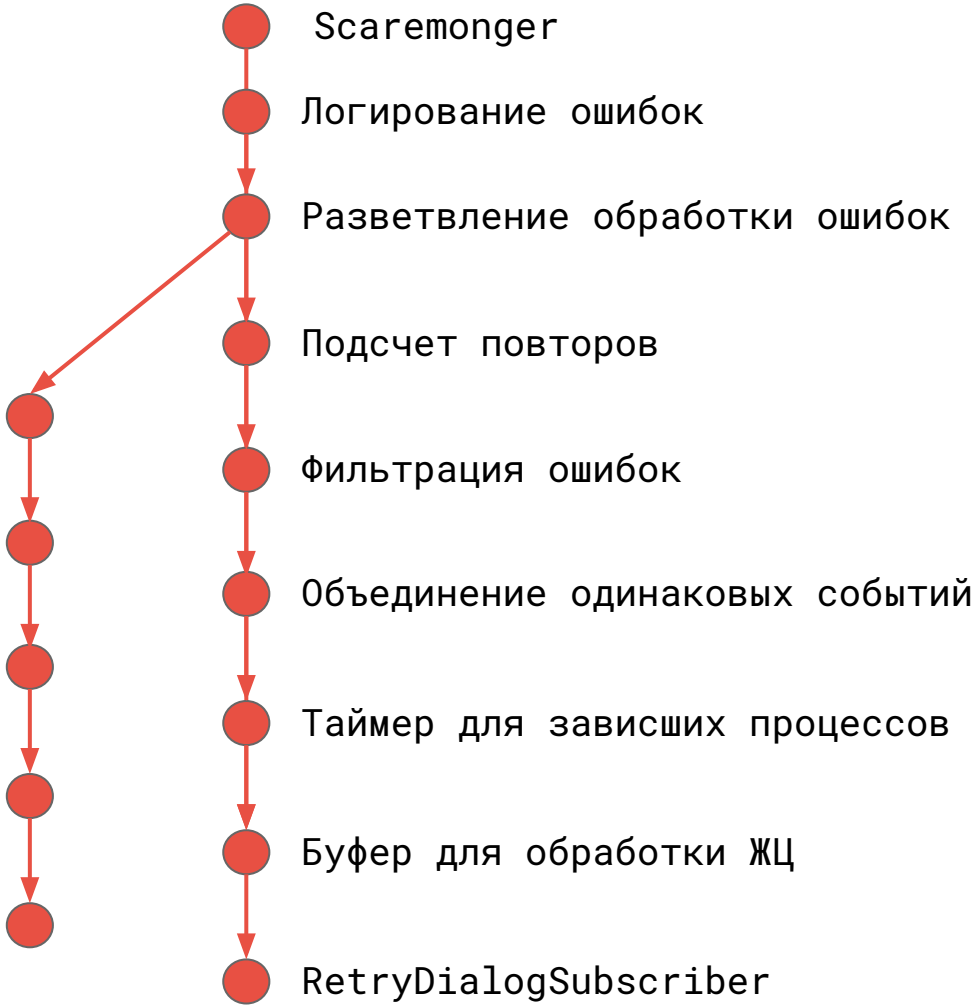
Dispatcher


Scaremonger
Логирование ошибок
Фильтрация ошибок
Объединение одинаковых событий
Таймер для зависших процессов
Буфер для обработки ЖЦ
RetryDialogSubscriber

Dispatcher

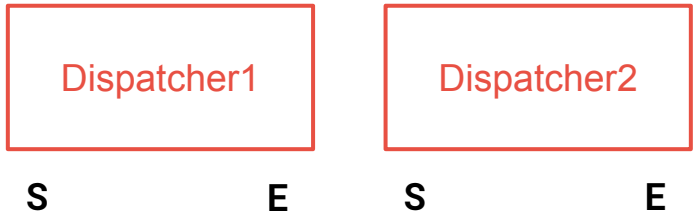

Scaremonger
Логирование ошибок
Подсчет повторов
Фильтрация ошибок
Объединение одинаковых событий
Таймер для зависших процессов
Буфер для обработки ЖЦ
RetryDialogSubscriber

Dispatcher

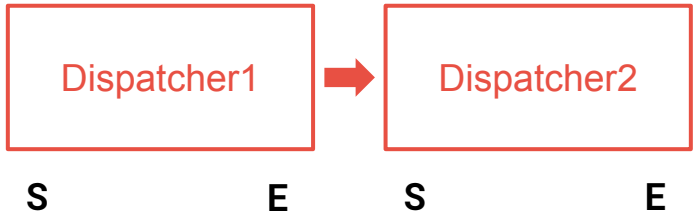

Scaremonger
Логирование ошибок
Разветвление обработки ошибок
Подсчет повторов
Фильтрация ошибок
Объединение одинаковых событий
Таймер для зависших процессов
Буфер для обработки ЖЦ
RetryDialogSubscriber

Соединение диспетчеров

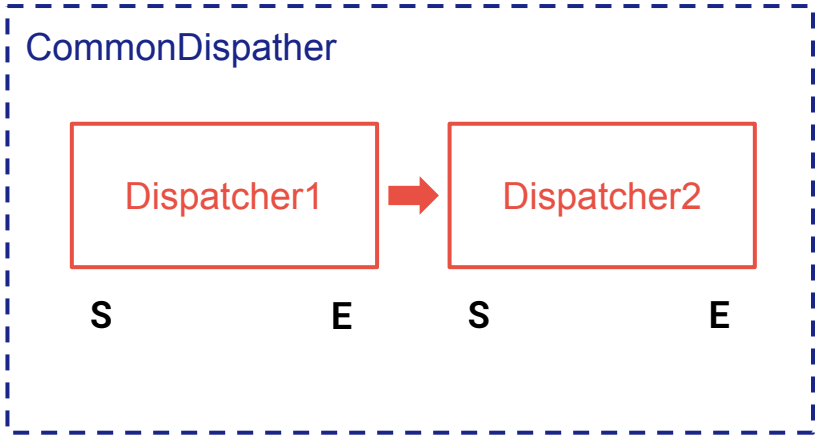
Соединение диспетчеров



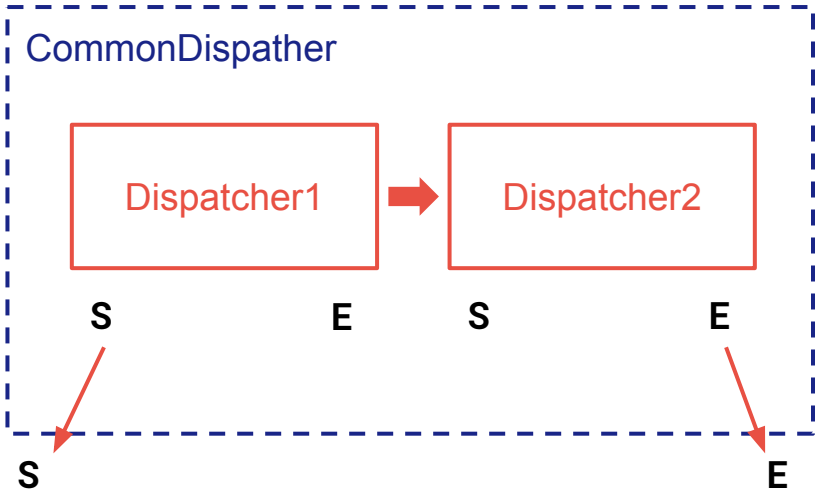
Соединение диспетчеров



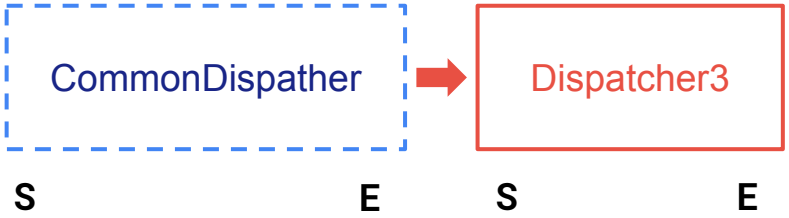
Соединение диспетчеров



Соединение диспетчеров



Соединение диспетчеров



Dispatcher

```
interface ScaremongerDispatcher : ScaremongerEmitter, ScaremongerSubscriber {

    fun connect(other: ScaremongerDispatcher): ScaremongerDispatcher {
        subscribe(other)
        return object : ScaremongerDispatcher,
            ScaremongerEmitter by other,
            ScaremongerSubscriber by this {}
    }

}
```

Цепочки диспетчеров

```
Scaremonger
    .connect(LoggingDispatcher({ e -> Timber.e(e) }))
    .connect(FilterDispatcher({ e -> e is MyError }))
    .connect(BufferedDispatcher())
```

LoggingDispatcher

```
class LoggingDispatcher(
    private val logger: (error: Throwable) -> Unit
) : ScaremongerDispatcher {









}
```

LoggingDispatcher

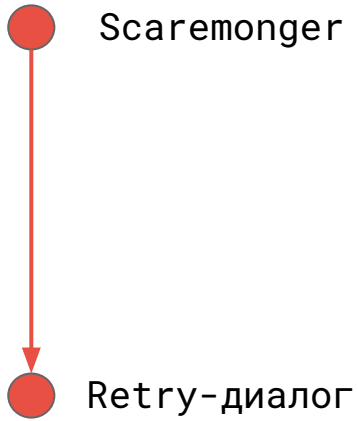
```
class LoggingDispatcher(
    private val logger: (error: Throwable) -> Unit
) : ScaremongerDispatcher {

    private var subscriber: ScaremongerSubscriber? = null

    override fun subscribe(subscriber: ScaremongerSubscriber) {
        this.subscriber = subscriber
    }

    override fun unsubscribe() {
        this.subscriber = null
    }
}
```

LoggingDispatcher

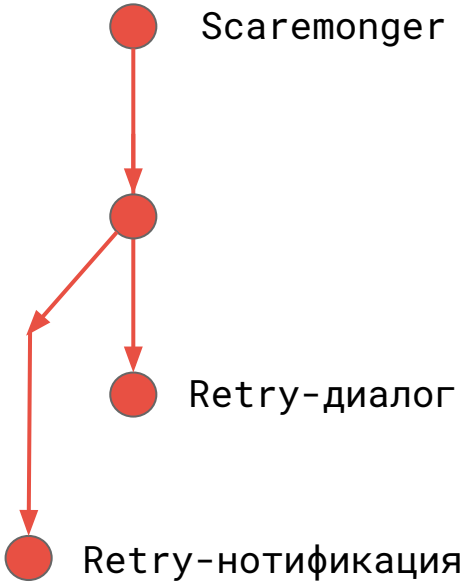
```
class LoggingDispatcher(
    private val logger: (error: Throwable) -> Unit
) : ScaremongerDispatcher {

    override fun onNext(
      error: Throwable, callback: (retry: Boolean) -> Unit
    ): ScaremongerDisposable {
        logger(error)
        subscriber?.let { s ->
            return s.request(error, callback)
        } ?: run {
            callback(false)
            return ScaremongerDisposable()
        }
    }
}
```

Сила подписчиков

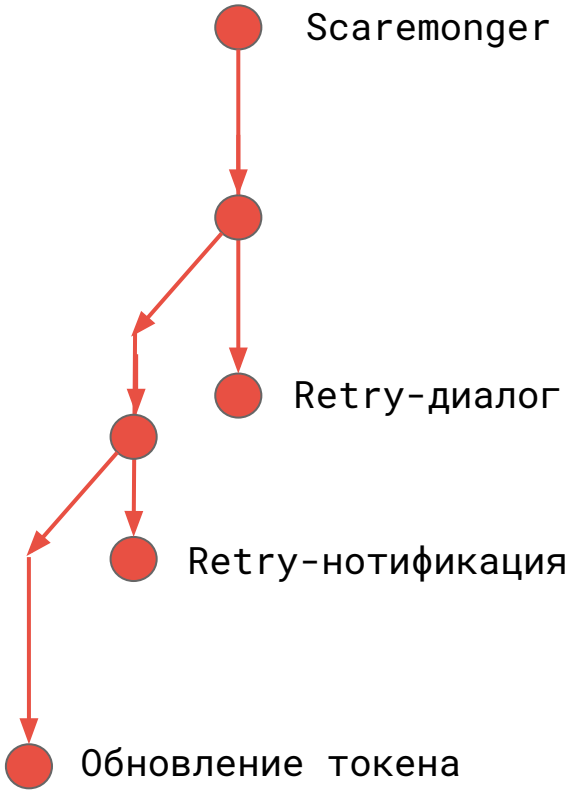
Subscriber



Dispatcher



Dispatcher


Scaremonger
Retry-диалог
Retry-нотификация
Обновление токена

Dispatcher


Scaremonger
Retry-диалог
Retry-нотификация
Обновление токена
Запрос системных разрешений!

Итоги

как это безумие выглядит для пользователя

Как используется такое решение

1. В Application-классе один раз настраивается цепочка диспетчеров

Как используется такое решение

1. В Application-классе один раз настраивается цепочка диспетчеров
2. Во всех Rx-цепочках, где необходима обработка ошибок, добавляется оператор **.scaremonger()**

Как используется такое решение

1. В Application-классе один раз настраивается цепочка диспетчеров
2. Во всех Rx-цепочках, где необходима обработка ошибок, добавляется оператор **.scaremonger()**
3. В Activity создается **RetryDialogSubscriber** и ему передается жизненный цикл (resume/pause)

Совсем не обязательно только RxJava

Можно приспособить для Корутин или к любому другому асинхронному процессу.

Философский вопрос

Объединение обработки ошибок в одно место - это правильно или нет?

Спасибо

https://t.me/terrakok

Спасибо

https://t.me/terrakok

https://github.com/terrakok/Scaremonger