Готовим конфигурацию Android-приложения с помощью Dynamic Proxy и рефлексии

Гусев Анатолий

Android разработчик

RuStore

О чем будет говорить

— Зачем нужна система конфигов;

— Проблемы конфигов в многомодульном проекте;

— База системы конфигов;

— Экран конфигурации для дебажных сборок;

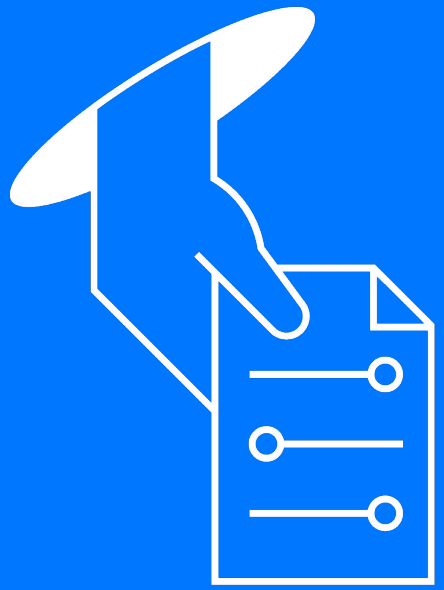
— Работаем с конфигами в UI тестах.

Немного контекста

* Большое приложение

* Много фич и бизнес логики

* Многомодульность с разделением на api/impl



Зачем нужна конфигурация

* Холодные
  тогглы

---

* Интервалы/длительность
  фоновых процессов

---

* Отдельные хосты
  с которыми работают фичи

Дополнительные требования

* Простая и удобная работа с конфигами

* Редактирование конфигов на отладочном экране

Прямой подход

* Выделяем один модуль для всех конфигов

* Потребители получают конфиг из единой сущности

Пример простого конфига

```
data class Configuration(

    val loggerEnabled: Boolean,

    val syncInterval: Long,

    val someFeatureEnabled: Boolean

    ...
)
```

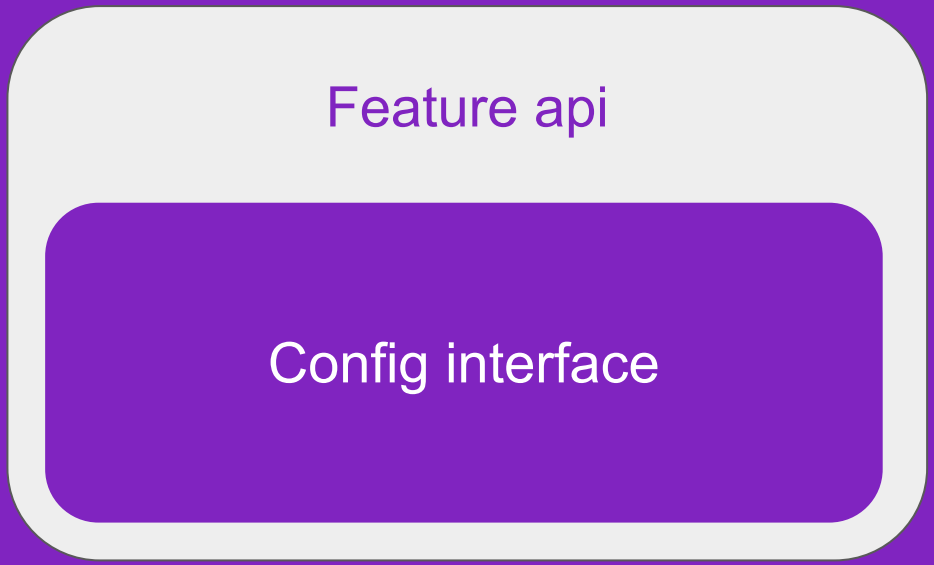
Проблемы прямого подхода

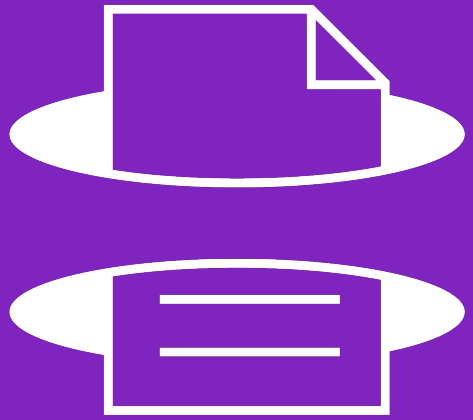
* Утекает бизнесовый контекст

* Мерж-конфликты

* Легко оставить старые поля

Конфиг в многомодульном приложении





Выделение интерфейса

* Стало сильно больше кода

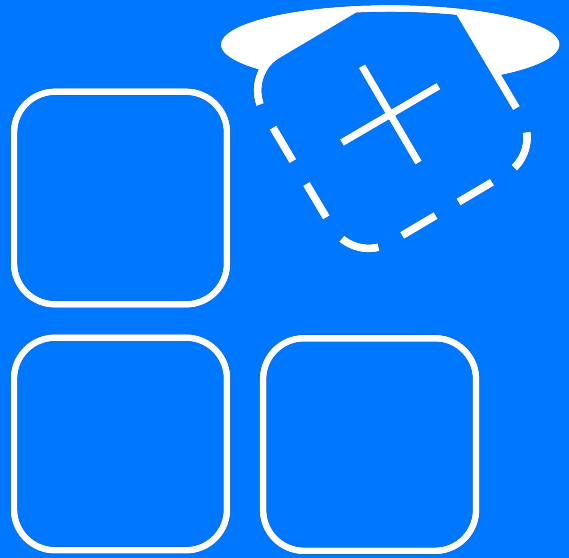
* Непонятно куда поместить дебажную реализацию

Dynamic proxy

* Может создать инстанс по переданному интерфейсу

* Позволяет избавиться от релизной реализации конфига

* Значения будем подставлять из аннотаций над полями интерфейса



Пример простого конфига фичи

```
interface GreetingConfig : ConfigMarker {

    @get:ConfigStringValue("World")
    val target: String
}
```

Интерфейс провайдера конфигов

```
interface Configs {

    operator fun <T : ConfigMarker> get(configType: Class<T>): T
}
```

Интерфейс провайдера конфигов

```
interface Configs {

    operator fun <T : ConfigMarker> get(configType: Class<T>): T
}
```

Реализация провайдера конфигов

```
class ConfigsImpl @Inject constructor(): Configs {

    override operator fun <T : ConfigMarker> get(configType: Class<T>): T {
        val types = arrayOf(configType)
        val invocationHandler = ConfigProxyInvocationHandler()
        return Proxy.newProxyInstance(javaClass.classLoader, types, invocationHandler) as T
    }
}
```

Реализация провайдера конфигов

```
class ConfigsImpl @Inject constructor(): Configs {

    override operator fun <T : ConfigMarker> get(configType: Class<T>): T {
        val types = arrayOf(configType)
        val invocationHandler = ConfigProxyInvocationHandler()
        return Proxy.newProxyInstance(javaClass.classLoader, types, invocationHandler) as T
    }
}
```

Реализация провайдера конфигов

```
class ConfigsImpl @Inject constructor(): Configs {

    override operator fun <T : ConfigMarker> get(configType: Class<T>): T {
        val types = arrayOf(configType)
        val invocationHandler = ConfigProxyInvocationHandler()
        return Proxy.newProxyInstance(javaClass.classLoader, types, invocationHandler) as T
    }
}
```

InvocationHandler

```
internal class ConfigProxyInvocationHandler: InvocationHandler{
    override fun invoke(instance: Any, method: Method, args: Array<out Any>?): Any {
        var valueFromAnnotation:Any? = null

        when{
            method.isAnnotationPresent(ConfigStringValue::class.java) ->{
                valueFromAnnotation = method.getAnnotation(ConfigStringValue::class.java)?.releaseValue
            }
            ...
        }

        return valueFromAnnotation ?: throw IllegalStateException("All properties must be annotated")
    }
}
```

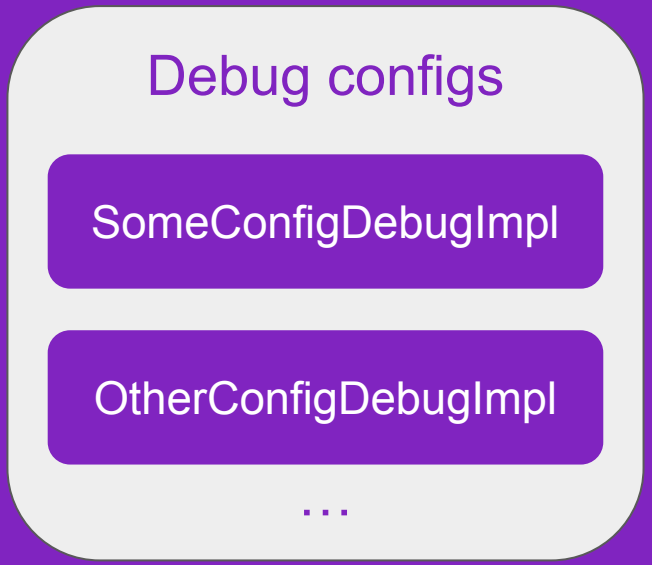
InvocationHandler

```
internal class ConfigProxyInvocationHandler: InvocationHandler{
    override fun invoke(instance: Any, method: Method, args: Array<out Any>?): Any {
        var valueFromAnnotation:Any? = null

        when{
            method.isAnnotationPresent(ConfigStringValue::class.java) ->{
                valueFromAnnotation = method.getAnnotation(ConfigStringValue::class.java)?.releaseValue
            }
            ...
        }

        return valueFromAnnotation ?: throw IllegalStateException("All config properties must be annotated")
    }
}
```

Дебажные реализации конфигов



Пример

```
data class GreetingDebugConfig(
    override val target: String = "Test",
) : GreetingConfig
```

Пример

```
class ConfigInMemoryDataSource @Inject constructor() {

    private val configs: MutableMap<Class<out ConfigMarker>, ConfigMarker> = mutableMapOf()

    fun <T : ConfigMarker> get(configType: Class<T>): T =
        configs[configType] as? T ?: error("No debug config")

    fun set(newConfigs: Map<Class<out ConfigMarker>, ConfigMarker>) {
        configs.clear()
        configs.putAll(newConfigs)
    }
}
```

Пример

```
class ConfigInMemoryDataSource @Inject constructor() {

    private val configs: MutableMap<Class<out ConfigMarker>, ConfigMarker> = mutableMapOf()

    fun <T : ConfigMarker> get(configType: Class<T>): T =
        configs[configType] as? T ?: error("No debug config")

    fun set(newConfigs: Map<Class<out ConfigMarker>, ConfigMarker>) {
        configs.clear()
        configs.putAll(newConfigs)
    }
}
```

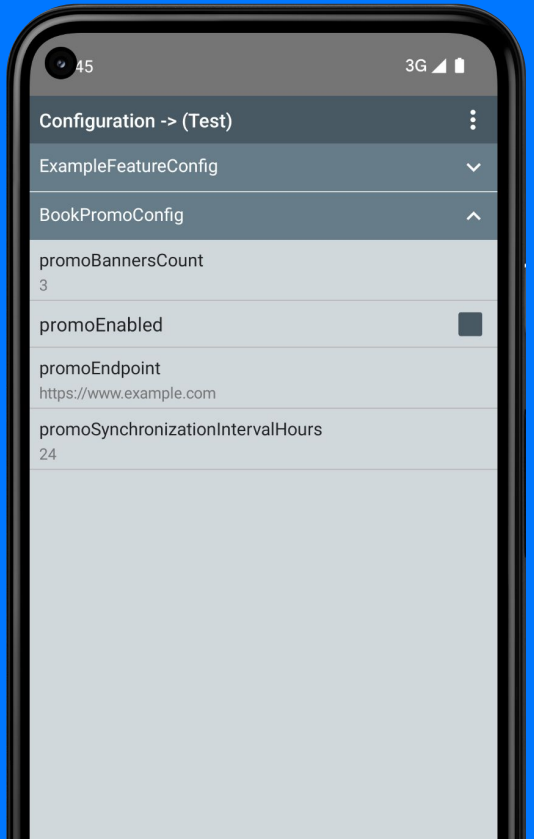
Отладочный экран

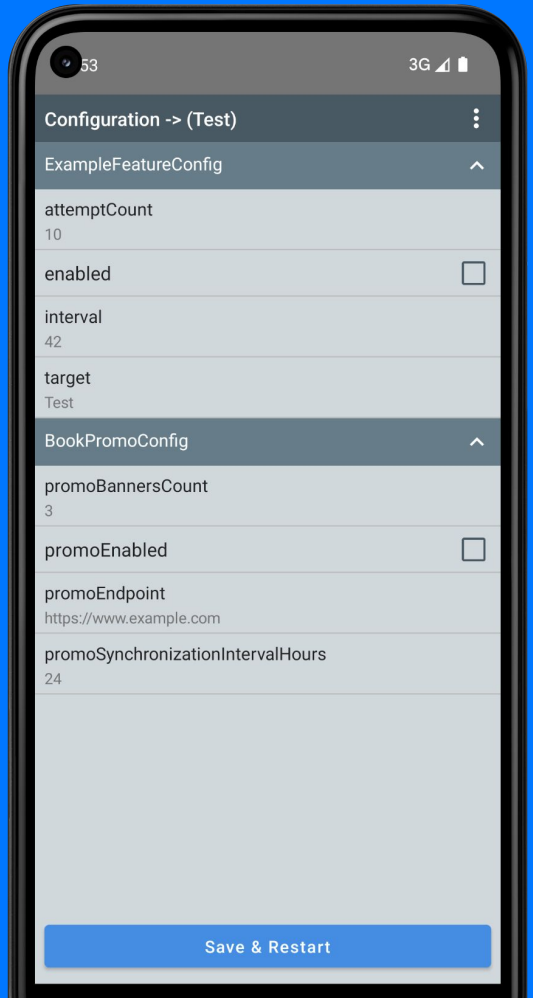
* Нужно хранить локальные изменения

* Нужно иметь дефолтные пресеты конфигов

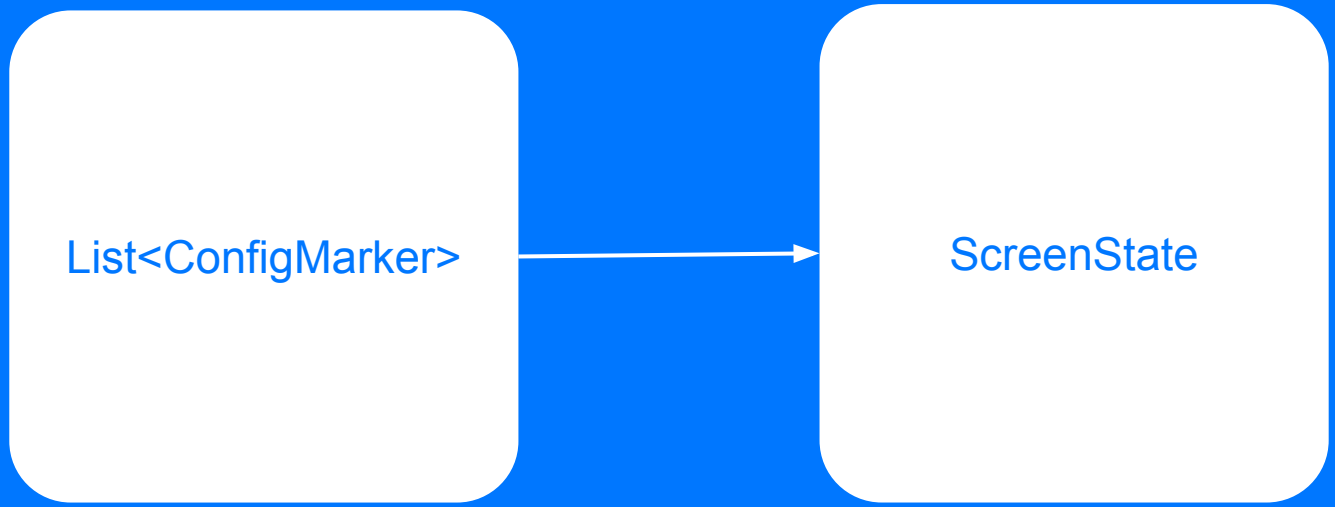
* Широко используем рефлексию

Отладочный экран





Нужно конвертировать конфиги в стейт



Маппинг в стейт

```
fun map(configType: Class<out ConfigMarker>, config: ConfigMarker): ConfigState {
    val properties = config.javaClass.kotlin.memberProperties

    val propertiesData = properties.associate { property ->
        val propertyValue = property.getter.call(config)

        val fullName = "${configType.name}.${property.name}"
        fullName to toPropertyModel(fullName, property, propertyValue)
    }
    return ConfigState(propertiesData)
}
```

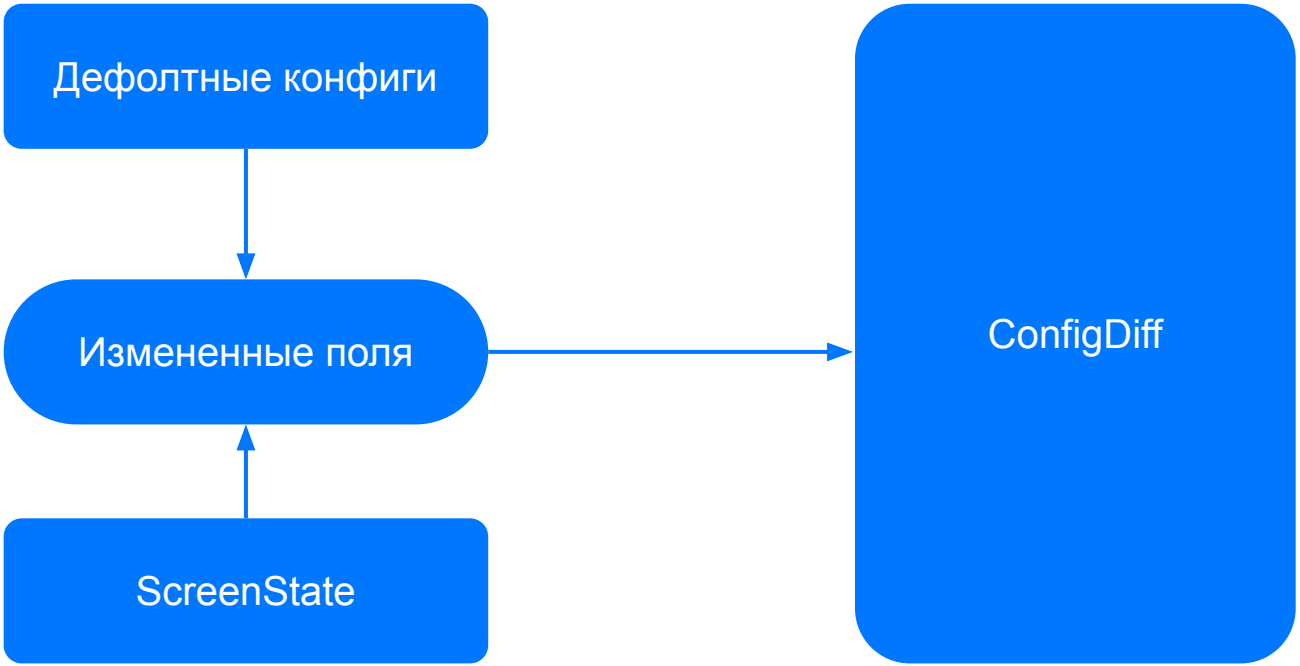
Отладочный экран

* Храним в preferences

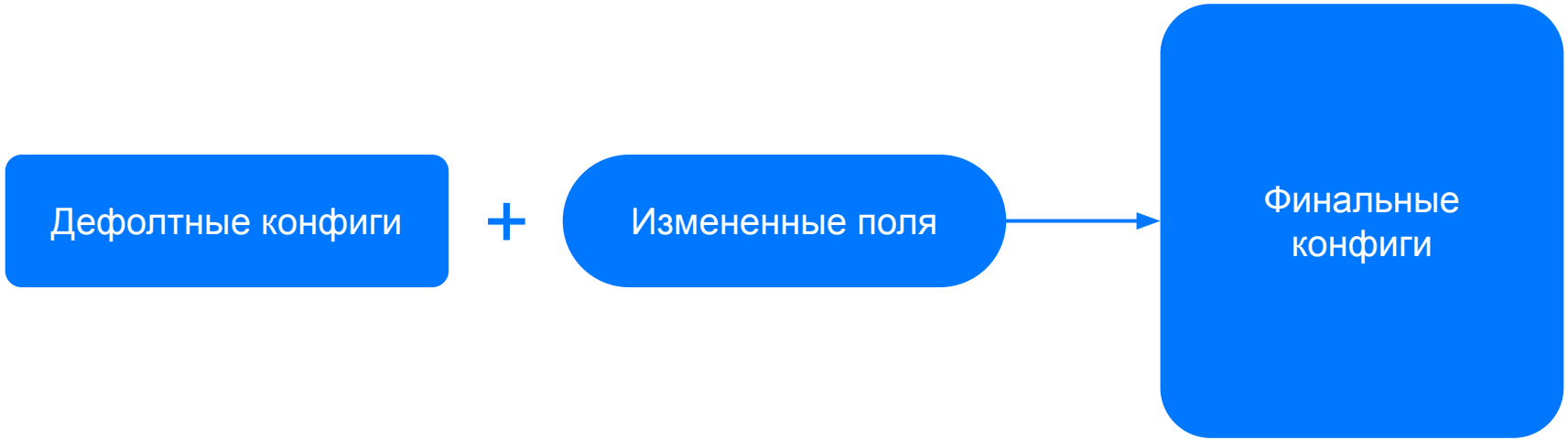
* Храним как пару ключ-значение

* Храним только отредактированные поля

Хранение изменений



Применяем изменения



Применяем сохраненные значения

```
private fun applyToConfig(configKey: Class<out ConfigMarker>, config: ConfigMarker, changed: Map<String, String>) {
    val properties = config.javaClass.kotlin.memberProperties

    properties.forEach { property ->
        val fullName = "${configKey.name}.${property.name}"

        if (changed.contains(fullName)) {
            property.javaField?.isAccessible = true
            when {
                isPropertiesAssignableFrom(property, Boolean::class.java) -> {
                    property.javaField?.set(config, changed[fullName]?.toBoolean())
                }
                ...
            }
        }
    }
}
```

Применяем сохраненные значения

```
private fun applyToConfig(configKey: Class<out ConfigMarker>, config: ConfigMarker, changed: Map<String, String>) {
    val properties = config.javaClass.kotlin.memberProperties

    properties.forEach { property ->
        val fullName = "${configKey.name}.${property.name}"

        if (changed.contains(fullName)) {
            property.javaField?.isAccessible = true
            when {
                isPropertiesAssignableFrom(property, Boolean::class.java) -> {
                    property.javaField?.set(config, changed[fullName]?.toBoolean())
                }
                ...
            }
        }
    }
}
```

Инициализация

```
internal class ConfigInitializer : Initializer<Unit> {

    @Inject
    internal lateinit var initConfigsUseCase: InitConfigsUseCase

    override fun create(context: Context) {
        //inject dependency

        initConfigsUseCase()
    }
}
```

Пример

```
object ConfigInMemoryDataSource constructor() {

    private val configs: MutableMap<Class<out ConfigMarker>, ConfigMarker> = mutableMapOf()

    fun <T : ConfigMarker> get(configType: Class<T>): T =
        configs[configType] as? T ?: error("No debug config")

    fun set(newConfigs: Map<Class<out ConfigMarker>, ConfigMarker>) {
        configs.clear()
        configs.putAll(newConfigs)
    }
}
```

Добавление нового конфига

* Добавить интерфейс

---

* Добавить дебажную реализацию

---

* Дебажную реализацию добавить в дефолтные конфиги

Пример

```
interface GreetingConfig : ConfigMarker {

    @get:ConfigStringValue("World")
    val target: String
}
```

Пример

```
class GreetingDebugConfig : GreetingConfig {
    override val target: String =
        "Test"
}
```

Пример

```
class TestConfigsDataSource @Inject constructor() {
    fun get(): Map<Class<out ConfigMarker>, ConfigMarker> =
        mapOf(GreetingConfig::class.java to GreetingDebugConfig())
}
```

Пример

```
class TestConfigsDataSource @Inject constructor() {
    fun get(): Map<Class<out ConfigMarker>, ConfigMarker> =
        mapOf(GreetingConfig::class.java to GreetingDebugConfig())
}
```

Пример

```
val greetingConfig = configs[GreetingConfig::class.java]
```

Конфиги в UI тестах

* Передаем конфиги через статику

* Для удобства добавим компактный DSL

Пример dsl в UI тесте

```
edit(BookPromoConfig::class) {
    on { enabled } returns true
}
```

Proxy для перехвата обращения к полю

```
class ConfigProxy<Config : ConfigMarker>(
    private val configClass: KClass<Config>
) : InvocationHandler {

    fun getProxyInstance(): Config =
        Proxy.newProxyInstance(this.javaClass.classLoader, arrayOf(configClass.java), this) as Config

    override fun invoke(instance: Any, method: Method, args: Array<out Any>?): Any? {
        val getterName = method.name
        val filedName = getterName.drop(3).replaceFirstChar { it.lowercase(Locale.ENGLISH) }
        throw ConfigUsageException(filedName = filedName)
    }

    class ConfigUsageException(val filedName: String) : IllegalStateException("Config only accessible in on block")
}
```

Proxy для перехвата обращения к полю

```
class ConfigProxy<Config : ConfigMarker>(
    private val configClass: KClass<Config>
) : InvocationHandler {

    fun getProxyInstance(): Config =
        Proxy.newProxyInstance(this.javaClass.classLoader, arrayOf(configClass.java), this) as Config

    override fun invoke(instance: Any, method: Method, args: Array<out Any>?): Any? {
        val getterName = method.name
        val filedName = getterName.drop(3).replaceFirstChar { it.lowercase(Locale.ENGLISH) }
        throw ConfigUsageException(filedName = filedName)
    }

    class ConfigUsageException(val filedName: String) : IllegalStateException("Config only accessible in on block")
}
```

Редактор полей

```
class FieldEditor<Config : ConfigMarker>(
    private val configProxy: ConfigProxy<Config>
) {
    private val editedFields: MutableMap<String, Any> = mutableMapOf()

    fun on(lambda: Config.() -> Unit): FieldValueEditor {
        val fieldName = try {
            configProxy.getProxyInstance().lambda()
            throw IllegalStateException("Property must be called in on block")
        } catch (exception: ConfigProxy.ConfigUsageException) {
            exception.filedName
        }

        editedFields[fieldName] = NOT_INITIALIZED
        return FieldValueEditor(fieldName)
    }
    ...
}
```

Редактор полей

```
class FieldEditor<Config : ConfigMarker>(
    private val configProxy: ConfigProxy<Config>
) {
    private val editedFields: MutableMap<String, Any> = mutableMapOf()

    fun on(lambda: Config.() -> Unit): FieldValueEditor {
        val fieldName = try {
            configProxy.getProxyInstance().lambda()
            throw IllegalStateException("Property must be called in on block")
        } catch (exception: ConfigProxy.ConfigUsageException) {
            exception.filedName
        }

        editedFields[fieldName] = NOT_INITIALIZED
        return FieldValueEditor(fieldName)
    }
    ...
}
```

Редактор полей

```
class FieldEditor<Config : ConfigMarker>(
    private val configProxy: ConfigProxy<Config>
) {
    ...
    inner class FieldValueEditor(private val fieldName: String) {
        infix fun returns(fieldValue: Any) {
            editedFields[fieldName] = fieldValue
        }
    }
    ...
}
```

Редактор конфигурации

```
class ConfigEditor {
    private val configEditedFields: MutableMap<KClass<out ConfigMarker>, Map<String, Any>> = mutableMapOf()

    fun <Config : ConfigMarker> edit(configType: KClass<Config>, edit: FieldEditor<Config>.(Config) -> Unit) {
        val configProxy = ConfigProxy(configType)
        val fieldEditor = FieldEditor(configType, configProxy)
        fieldEditor.edit(configProxy.getProxyInstance())

        val editedFields = fieldEditor.getEditedFields()

        configEditedFields[configType] = editedFields
    }

    internal fun getConfigsEditedFields(): Map<KClass<out ConfigMarker>, Map<String, Any>> =
        configEditedFields
}
```

Профиты

* Простое добавление конфига
* Невозможно забыть остатки устаревшего конфига
* Конфиг автоматически добавляется на отладочный экран
* Конфиги можно редактировать в UI тестах

Спасибо за внимание!

RuStore