Kotlin DSL for Business Processes Automation in Telecom and Underlying Engine for their Execution

Kirill D. Kreslavski

Academic Supervisor: Marina G. Ivanova, lead engineer at Nexign

St. Petersburg School of Physics, Mathematics, and Computer Science

National Research University Higher School of Economics – Saint Petersburg

Introduction to domain

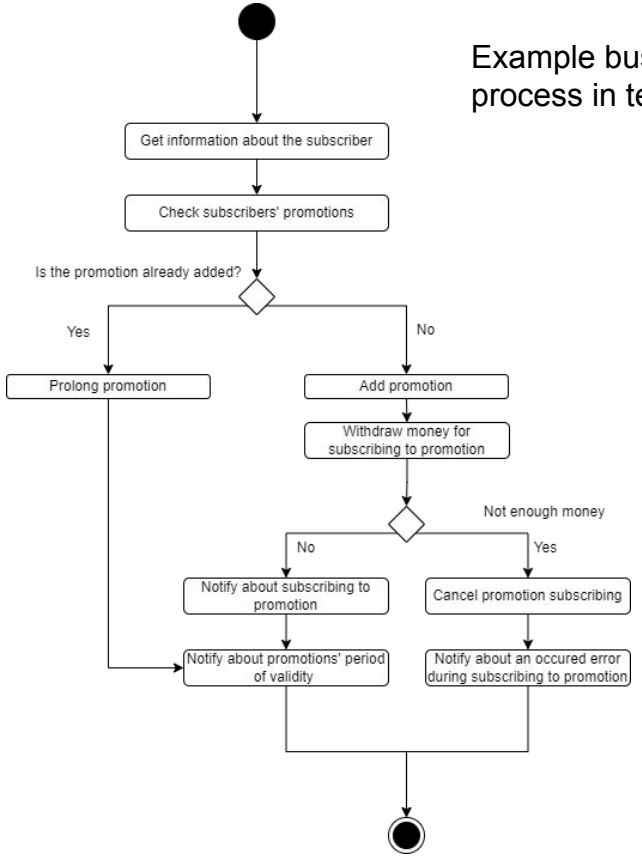
DSL – Domain Specific Language, in this case basically a small programming language for our specific domain inside another programming language - Kotlin.

Great Kotlin DSL examples can be found here github.com/zsmb13/VillageDSL

```
val v = village {
    house {
        +Person("Emily", 31)
        +Person("Hannah", 27)
        +Person("Alex", 21)
        +Person("Daniel", 17)
    }
}
```

```
val v = village containing houses {
    house with people {
        "Emily" age 31
        "Hannah" age 27
        "Alex" age 21
        "Daniel" age 17
    }
}
```

Example business process in telecom



Motivation

Nexign — russian IT company, OSS/BSS software and business digitalization solutions developer, Russia's first telecom billing software developer.

Nexign has a product called CRAB, which basically is a high-load business scenario execution system and the problem is that Scenario describing and developing is not fast and the result is long and messy piece of code with the description of all transitions, conditions, states and procedures.

What is desired is a Kotlin DSL, that would help structurize and simplify the development of Scenarios and reduce an amount of Java boilerplate and human-factor errors.

Goals and Tasks

Global goal: develop effective Kotlin DSL for short, readable scenarios description and a compact engine for their execution, reducing amount of “human factor” errors (e.g. typos, types compatibility errors, etc.)

- Research and analyze existing analogues, their advantages and disadvantages
- Develop DSL and add more new functionality
- Develop Production ready and Testing engines
- Autotests cover
- Proof of the concept, functionality check. Get user experience feedback

Comparison with analogues 1

Frameworks / libraries and products for business processes/scenarios descriptioning and development

Cadence – **Golang** BPMN engine with libraries in Go, **Java**, Python and Ruby. **Open source**. Gigantic, covers the whole BPMN functionality, therefore is very complex. Lots of options and preferences for all and everything. Overexpandable for almost anything. Shortly said – overkill.	Temporal – BPMN framework, based on Cadence, easier in exploitation and customization, also **Open source,** also **Java**.	iWF – BPMN framework based on Cadence, even easier than Temporal.

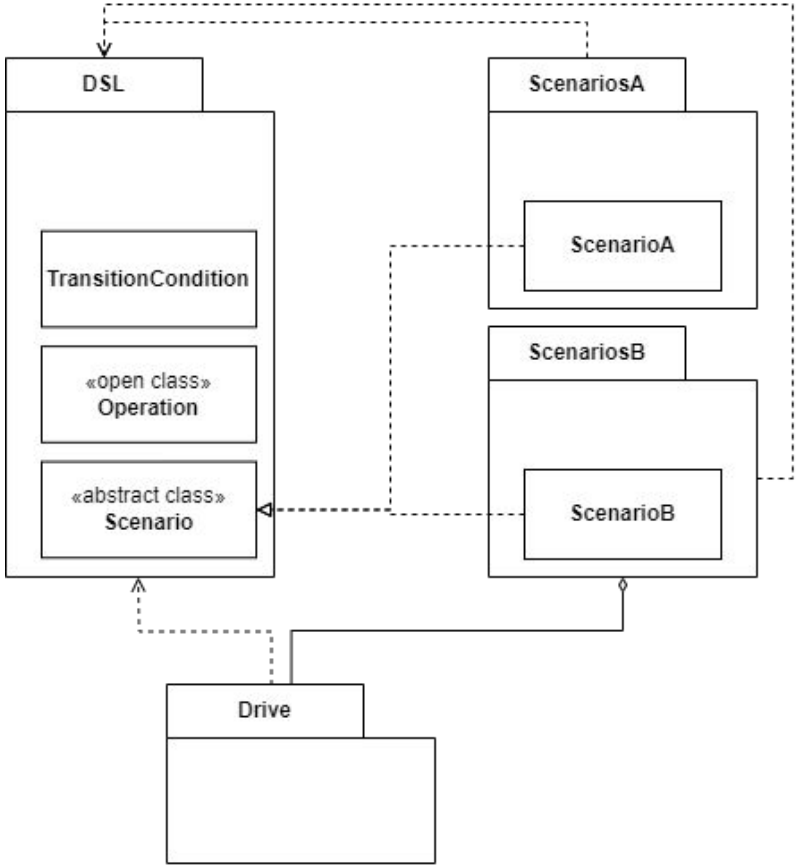
Common issue – transitions definition is smeared in all across the code, no single-point for scenario description, not humanly readable.

Comparison with analogues 2

Frameworks / libraries and products for finite state machines development

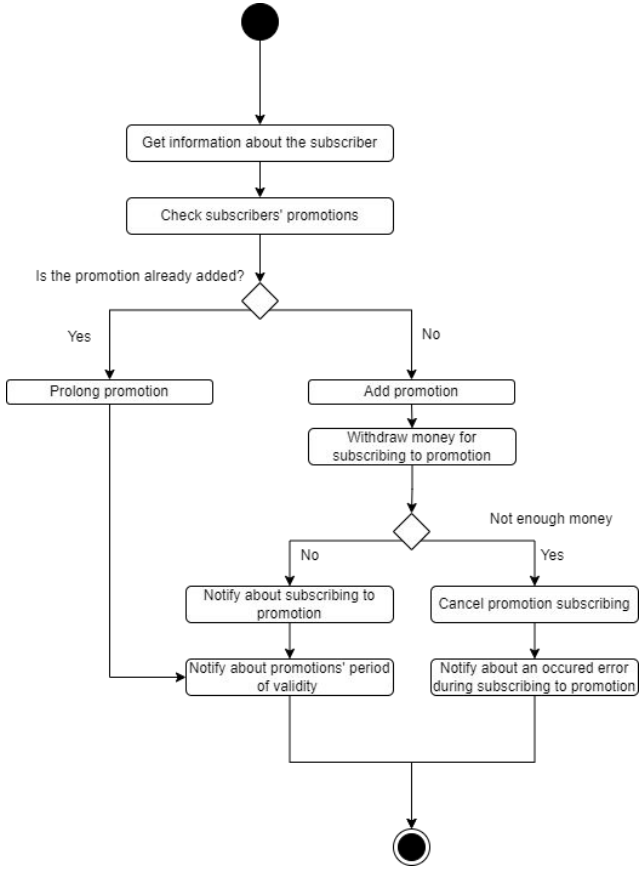
Name	Short description	Has an execution engine	Automatic picture generation from code
KStateMachine	Quite compact framework, but now it is all the way around – all the operations, what they do, their code must be declared in one place, which is bad for decomposition	+/-	-
StateMachine (tinder.com)	Pure State Machine Builder, useful job can only be done via side effects from transitions, the states are just states. It also becomes a quite long spaghetti.	-	-
KFSM	Decent framework, provides api as exactly Kotlin DSL, but describing even a tiny State Machine takes a lot of code and space for no meaningful reason.	-	+

Our solution



Scenario packages require only to know how to describe the scenarios - our DSL. And they totally shouldn't care how the Engine insides work. The Engine itself should only know how to interpret the scenarios - our DSL. It can consume the Scenario classes as already prepared JARs.

Our solution




```
class ExampleScenario(override val input: ExampleScenarioInput) : Scenario(input)  {

    override val specification: Specification = specification { this: Specification
        routing = routing { this: RoutingMap.RoutingBlockBuilder
            -getSubscriberInfo
            -checkSubscriberPromotions binary { this: RoutingMap.BinaryChoice
                yes = route { this: RoutingMap.RoutingBlockBuilder
                    -prolongPromotion
                    -notifyAboutPromotionTimePeriod
                    -end
                }
                no = route { this: RoutingMap.RoutingBlockBuilder
                    -activatePromotion
                    -writeOffMoney multiple { this: RoutingMap.MultipleChoiceBuilder
                        +(YES to route { this: RoutingMap.RoutingBlockBuilder
                            -cancelPromotionActivation
                            -NotifyAction("error when activating promotion")
                            -end
                        })
                        +(NO to route { this: RoutingMap.RoutingBlockBuilder
                            -NotifyAction("promotion activation")
                            -notifyAboutPromotionTimePeriod
                        })
                    }
                }
            }
        } errorRouting { this: RoutingMap.ErrorRoutingBuilder
            listOf(activatePromotion, cancelPromotionActivation)  with ActionProblemsETC             togetherRoutesTo specialErrorHandling
            OperationDefault                                      with SomethingUnexpectedHappened   routesTo defaultErrorHandling
        }
    }
```

Solved tasks

- Developed a quite comprehensive DSL
  - Created a Formal Language structure architecture for DSL
  - As much uncluttered descriptions for Operations as possible
  - Concise Transitions description
  - Description for high-level error handling
- Separated DSL, Engine entities and Scenarios packages
  - The Engine is built standalone and uses DSL as a library dependency
  - DSL knows nothing about internals of the Engine
  - For descriptioning and development of Scenarios only DSL is needed
- Automatic human-readable Scenario description generation (as text in english) from code
- Human-readable scenario execution logs

In Progress

- Engine enhancements and improvements
  - Try to create a way to provide external dependencies via Kotlin Context (experimental)
  - Automatic generation of Scenarios description as xml specifications and as schematic graph images
  - The same for Scenario execution log
  - Add REST API service
  - Improve multi-threading
- Write auto-tests and add testing to CI/CD
- Write more documentation

Results

repository