Разработка аспектно-ориентированного расширения для языка Kotlin

Борис Скрипаль
Санкт-Петербургский политехнический
университет Петра Великого
Email: skripal@kspt.icc.spbstu.ru

Владимир Ицыксон
Санкт-Петербургский политехнический
университет Петра Великого
Email: vlad@icc.spbstu.ru

***Аннотация*—В статье описывается разработка аспектно-ориентированного расширения для языка Kotlin. Kotlin — молодой мультипарадигменный язык программирования, быстро набирающий популярность. Однако для полноценной конкуренции с существующими языками необходима реализация многих возможностей, уже существующих для традиционных языков. Одна из таких возможностей — аспектно-ориентированное программирование. В статье на основе анализа существующих аспектно-ориентированных расширений для различных объектно-ориентированных языков разрабатывается программный прототип, позволяющий использовать аспектно-ориентированный подход при написании программ на языке Kotlin. Для этого проектируется язык описания аспектов, реализуются методы внедрения сквозной функциональности. Созданный прототип протестирован на серии примеров. Тестирование показало корректность предложенного подхода и работоспособность прототипа.**

I. Введение

Аспектно-ориентированный подход (АОП) был предложен как решение проблемы описания сквозной функциональности в объектно-ориентированных программах. Впервые подход был представлен в 1997 году Грегором Кичалесом в работе «Aspect-oriented programming» [1]. В предложенном подходе сквозная функциональность описывается отдельно от объектно-ориентированного кода программы и внедряется на этапе компиляции. Такое разделение позволяет не только компактно описывать сквозную функциональность, но и делать её внедрение прозрачным для программиста.

Парадигма АОП предложила элегантное решение для ряда задач, как, например, протоколирование и трассировка программ, работа с транзакциями, обработка ошибок, а также некоторых других. АОП стал интенсивно развиваться и, в настоящий момент, аспектно-ориентированные расширения созданы для большинства объектно-ориентированных языков программирования, как, например, C++ [2], C# [3], Java [4], [5], Python [6] и т.п.

В данной статье мы представляем аспектно-ориентированное расширение для нового языка программирования Kotlin. Язык Kotlin — молодой мультипарадигменный язык программирования, разрабатываемый компанией JetBrains. Основная цель языка Kotlin — быть компактной, выразительной и надежной заменой языка Java. При этом язык обеспечивает полную совместимость с программами на языке Java.

Наличие существенного числа новых конструкций не позволяет использовать уже готовые АОП-расширения для JVM-совместимых языков, поэтому аспектно-ориентированное расширение для языка Kotlin необходимо разрабатывать, специально ориентируясь на особенности языка Kotlin.

Оставшаяся часть статьи организована следующим образом. В втором разделе приведено общее описание аспектно-ориентированной парадигмы, в третьем разделе представлены актуальные реализации аспектно-ориентированных расширений. Четвертый раздел посвящен описанию разрабатываемого расширения для языка Kotlin. В пятом разделе проводится тестирование расширения на реальных проектах. В заключении приводятся направления дальнейшего развития.

II. Аспектно-ориентированный подход

При использовании АОП, сквозная функциональность инкапсулируется в отдельные сущности, называемые *аспектами* (aspects). Каждый аспект состоит из *советов* (advices) и *срезов* (pointcuts). Совет — сущность, содержащая в себе непосредственно сквозную функциональность (код совета), а также способ её внедрения в программный код. Срез — описание множества *точек внедрения* советов. Типовыми способами внедрения сквозной функциональности являются:

- вставка совета до точки внедрения;
- вставка совета после точки внедрения;
- вставка совета после возникновения исключительной ситуации;
- вставка совета вместо точки внедрения.

Вставка кода совета может производиться статически и динамически. При статическом способе внедрения советов, сквозная функциональность внедряется или на этапе сборки (модификация исходных кодов программы и модификация промежуточного представления в процессе компиляции), или же, сразу после сборки, например, путем модификации байт-кода (для программ, работающих поверх JVM). Основным преимуществом данного подхода является отсутствие затрат на внедрение советов во время выполнения программы [7], однако, при изменении одного совета, будет необходима повторная сборка всего проекта. В связи с этим, статический способ внедрения используется как правило в

тех случаях, когда сквозная функциональность необходима для корректной работы программы.

При динамическом внедрении сквозной функциональности код советов применяется непосредственно в процессе выполнения программы. Одним из способов динамического внедрения советов является создание прокси-объектов [8], которые перехватывают исполнение программы, позволяя выполнять код совета до, после или вместо точки внедрения. Такой способ внедрения аспектов не требует повторной сборки всего проекта после изменения аспектов, однако имеет меньшую производительность.

III. Обзор существующих аспектно-ориентированных расширений

Перед разработкой АОП-расширения для языка Kotlin рассмотрим существующие на рынке АОП-решения и их основные особенности.

A. AspectJ

AspectJ — первое аспектно-ориентированное расширение для языка Java, оно было представлено в 2003 году [9] и развивается до сих пор[1]. AspectJ расширяет грамматику языка Java, предоставляя дополнительные языковые конструкции для описания аспектов. В отличие от классов описание аспекта в AspectJ начинается с ключевого слова *aspect*. Сам аспект имеет уникальное имя и состоит из тела, которое содержит описание срезов и советов, а также может использовать переменные и методы языка Java. Описания срезов могут задаваться как отдельно от совета (в таком случае им необходимо задавать идентификатор, уникальный в рамках аспекта), так и, непосредственно, при описании совета. Способ внедрения кода совета относительно точки внедрения задается при описании совета с помощью ряда ключевых слов: *before* (после точки внедрения), *after* (перед точкой внедрения) и т.д. Также в AspectJ существует возможность описания аспектов при помощи специальных аннотаций над классами языка Java. При таком подходе в качестве аспекта выступает класс, снабженный аннотацией *@Aspect*, а срезами и советами выступают методы данного класса, имеющие аннотации *@Pointcut* или же специальными аннотациями советов (например, *@Before*) соответственно. Описание срезов также задается внутри аннотаций.

AspectJ поддерживает статический и динамический способы внедрения аспектов [4]. Статическое внедрение аспектов может производиться, как на уровне исходных кодов, так и сразу после компиляции программы (внедрение кода советов в байт-код скомпилированной программы). Динамическое внедрение аспектов производится непосредственно перед тем, как JVM загружает файл класса, что позволяет избежать временных затрат на внедрение аспектов во время выполнения программы.

[1]На момент написания статьи последняя доступная версия 1.8.10 датирована 12 декабря 2016 г.

B. SpringAOP

Другим популярным АОП-расширением для языка Java является SpringAOP — расширение, входящее в состав фреймворка «Spring Framework». Первая версия данного расширения была представлена в 2005 году, последняя (5.0.0.M5) — в феврале 2017 года. Для описания аспектов в SpringAOP используются специальные аннотации, размечающие классы и их методы, как аспекты, советы и срезы [5]. Также, как и в AspectJ, аспект может содержать не только описание срезов или советов, но и обычные в понимании языка Java переменные и методы.

SpringAOP поддерживает только динамическое внедрение аспектов в код. Основным способом применение аспектов в SpringAOP является связывание при помощи прокси-объектов.

Начиная с версии 5.0, Spring Framework добавил поддержку языка Kotlin [10], что позволяет учитывать execution функции, а также проверку на nullability при использовании SpringAOP.

C. AspectC++

Третьим рассматриваемым АОП-расширением является AspectC++ [2] — АОП-расширение для программ на языке C++. Первая реализация AspectC++ была представлена в 2001 году, последняя — в марте 2017 года. В качестве срезов, в AspectC++ могут выступать как статические сущности (классы, структуры, функции и т.д.), так и динамические сущности (вызов и выполнение функций, создание и удаление объектов) [11]. В данном расширении, аспект выступает как отдельная сущность, обозначающая ключевым словом *aspect* и содержащая в себе описание советов и именованных срезов. Основным способом внедрения аспектов, используемым в AspectC++ является статическое внедрение аспектов на уровне исходных кодов [12].

D. PostSharp

Еще одной реализацией АОП является PostSharp — АОП-фреймворк для программ, написанных на языке C#. PostSharp было представлен в 2004 году [3], как свободная библиотека для языка C#, однако в 2009 году данное расширение стало проприетарным. На момент написания статьи последняя стабильная версия расширения 4.3.29 была представлена в феврале 2017 года. Для описания аспектов в PostSharp существует ряд классов, обеспечивающих внедрения кода совета в программу. Каждый из таких классов определяет некоторый набор способов включения сквозной функциональности относительно точки внедрения [13]. Для реализации аспектов, необходимо переопределить соответствующие методы этого класса, а также прописать правила формирования среза внутри специальных аннотаций данного класса.

PostSharp поддерживает как статический, так и динамический способы внедрения аспектов. Для динамического применения аспектов могут использоваться как прокси-объекты, так и перехват момента загрузки файла в память,

после чего аспекты применяются непосредственно к бинарному файлу. Статическое внедрение аспектов может производиться как на уровне исходных кодов, так и во время компиляции. При применении аспектов во время компиляции используется MSIL — промежуточное представление, создаваемое в процессе компиляции программ для платформы .NET. MSIL является независимым от процессора набором инструкций, который впоследствии преобразуется в набор кодов для конкретного процессора. Такой подход позволяет удобно анализировать целевую программу, а также избавляет от необходимости поддерживать корректность исходных кодов программы после внедрения аспектов, что необходимо делать при модификации на уровне исходных кодов.

По результатам анализа ряда популярных АОП-расширений можно сделать вывод о том, что не смотря на схожую функциональность, расширения сильно отличаются как способами описания аспектов, так и способами их внедрения. Данные различия обусловлены не только особенностями языка программирования, для которого предназначено расширение, но и стремлением сделать синтаксис описания аспектов как можно более удобным и понятным для разработчика.

IV. Разработка аспектно-ориентированного расширения для языка Kotlin

При реализации аспектно-ориентированного расширения необходимо сформировать синтаксис аспектов и определить механизмы внедрения аспектов.

A. *Разработка синтаксиса описания аспектов*

Описание аспектов является расширением языка программирования и должно позволять в удобной и понятной форме задавать все сущности АОП: аспекты, советы, срезы и т.п. В этой работе мы приняли решение не разрабатывать специализированный синтаксис аспектов для языка Kotlin, а воспользоваться синтаксисом описания аспектов, используемый в AspectJ для языка Java, расширив его необходимыми для языка Kotlin конструкциями. Такой выбор был сделан по нескольким причинам:

- синтаксис описания аспектов, используемый в AspectJ, является очень удобным для программиста;
- грамматика AspectJ, описанная на языке ANTLR4, находится в свободном доступе и может быть использована для разработки языка аспектов для языка Kotlin;
- AspectJ является одним из самых популярных аспектно-ориентированных расширений для языка Java [14] и знаком многим разработчикам.

Учитывая, что в AspectJ описание аспектов может производится двумя способами: при помощи специальных аннотаций над классами и как отдельные сущности, было решено использовать второй способ. Такой выбор был сделан ввиду того, что такой способ описания аспектов, по мнению авторов, является менее избыточным, по сравнению с аннотациями, а также нагляднее отделяет объектно-ориентированную функциональность от сквозной.

Для адаптации AspectJ к языку Kotlin в синтаксис AspectJ были внесены следующие изменения:

- способ описания методов был изменен в соответствии с правилами языка Kotlin;
- стандартные типы языка Java были заменены на стандартные типы языка Kotlin;
- в соответствии с правилами языка Kotlin изменены модификаторы полей и методов;
- добавлена возможность задавать атрибуты аргументов методов;
- добавлена поддержка функций-расширений (extension functions) и подставляемых функций (inline functions).

Описание аспекта начинается с ключевого слова *aspect* и состоит из идентификатора аспекта и тела аспекта. Тело аспекта может содержать в себе описание срезов и советов. Описание среза начинается с ключевого слова *pointcut* и состоит из идентификатора среза и правила, в соответствии с которым производится формирования среза.

Правило формирования среза по сути является описанием множества точек внедрения, для задания которого используются логические операции и специальные ключевые слова, например, *execution* или *call*, указывающие на определенную группу мест в программном коде. Допускается описывать одни срезы с использованием других.

Для указания множества методов, входящих в срез, используются следующие механизмы:

- модификаторы видимости с логическими операторами (например, можно выделить все не public методы);
- класс, к которому принадлежит метод;
- имя метода или его часть;
- список аргументов метода с атрибутами;
- тип возвращаемого значения.

Описание совета состоит из задания способа применения совета (например, *before* или *after*), правила формирования среза и кода совета.

Пример описания аспекта приведен в листинге 1.

```
aspect A {
    pointcut fooPC(): execution(fun Foo.*())
    pointcut printPC(): call(public fun kotlin.io.
    pri*(!Any))

    after(): fooPC() && printPC() {
        println("Hello after!!")
    }

    before(): fooPC() && printPC() {
        println("Hello before!!")
    }
}
```

Листинг 1. Пример описания аспекта

Приведенный в листинге аспект содержит описание двух срезов: fooPC и printPC. Срез fooPC включает в себя все места вызовов функций и инициализации переменных внутри всех методов класса Foo. Срез printPC включает в себя все места вызовов публичных методов пакета kotlin.io, имя

которых начинается с pri, принимающих один аргумент, тип которого отличается от Any. Далее описываются два совета: первый вставляет код совета (вызов метода println) до всех точек внедрения, которые принадлежат одновременно к двум срезам fooPC и printPC, второй — после. После применения совета к программе во всех местах вызовов методов, удовлетворяющих следующим условиям:

- вызов происходит внутри методов, принадлежащих классу Foo;
- вызываемый метод имеет модификатор public и принадлежит к пакету kotlin.io;
- имя вызываемой метода начинается с pri;
- метод принимает единственный аргумент, тип которого отличается от «Any».

до вызова метода будет выведена на печать строка «Hello after!!», а после — «Hello before!!».

На момент написания статьи реализованы следующие возможности, существующие в AspectJ:

- Структуры, используемые при описании срезов:
  - *call* — вызов заданного метода;
  - *execution* — выполнение заданного метода;
  - *target* — вызов метода у экземпляра указанного класса;
- Способы внедрения советов:
  - *before* — вставка кода совета перед точкой внедрения;
  - *after* — вставка кода совета после точки внедрения;
  - *around* — вставка кода совета до и после точки внедрения.

B. Внедрение аспектов

Язык Kotlin имеет ряд оригинальных языковых конструкций (extension функции, специфичные лямбда-функции и т.п.), при этом основной целевой платформой компиляции для языка Kotlin является JVM. Как результат — все специальные конструкции Kotlin преобразуются в стандартный байт-код, который имеет одинаковую структуру и для Java-, и для Kotlin-программ. Из-за этого поиск некоторых структур языка Kotlin в байт-коде становится затруднительным и, как следствие, динамическое внедрение кода советов при загрузке файлов в JVM становится практически невозможным. По той же причине статическое внедрение советов в байт-код программы также является очень сложной задачей.

Таким образом, единственным разумным способом внедрения аспектов является внедрение аспектов в исходный код программы или в модель программы во время компиляции.

Разработчики языка Kotlin предусмотрели специальную структуру данных для работы с программным кодом — Program Structure Interface (PSI). В нашем проекте мы используем PSI для внедрения объектов на этапе компиляции проекта.

Внедрение аспектов происходит в несколько этапов, схематически изображенных на рисунке 1.

Первым этапом является построение PSI — промежуточного представления проекта, состоящего из набора виртуальных файлов, соответствующих исходными файлам, а также из прочей информации о проекте (конфигурация, путь до JDK и т.п.). Каждый из этих файлов содержит дерево разбора программного кода и информацию о файле. Каждый элемент дерева разбора содержит в себе текст соответствующего элемента, ссылки на потомков и различную сопровождающую информацию. Одним из таких полей является «userMap» типа *KeyFMap* — структура в, которую могут быть записаны различные пользовательские данные. На рисунке 1 схематически представлен процесс внедрения аспектов в программный код.

Вторым этапом является чтение файлов с описаниями аспектов и формирование модели аспектов, состоящих из срезов и советов. Каждый экземпляр совета и среза содержит:

- уникальный идентификатор, используемый для разметки PSI;
- дерево разбора логического выражения, используемое при анализе принадлежности точки срезу.

Также экземпляр совета содержит в себе код совета, приведенный к виду промежуточного представления для более удобной модификации PSI. Дерево разбора в качестве нетерминальных узлов содержит в себе логические операции «и», «или», «не». Терминальными же узлами выступают или сигнатуры, используемые для описания срезов, или же идентификаторы других срезов.

Формирование набора точек внедрения также происходит в несколько шагов. На первом шаге, каждый узел PSI проверяется на соответствие каждому из отдельно описанных внутри совета срезов. В случае, если узел подходит данному срезу, то в структуру userMap добавляется идентификатор этого среза. На втором шаге, после того, как дерево было размечено идентификаторами срезов, для каждого совета проверяется, можно ли его применить к данному узлу дерева. Если можно, то происходит внедрение кода совета в соответствующую позицию относительно точки внедрения.

При внедрении в PSI код советов оборачивается в лямбда функцию *run*, что позволяет разрешать сложные случаи, например, при последовательном вызове нескольких функций без создания большого числа дополнительных промежуточных переменных. Для наглядности, рассмотрим участок кода в листинге 2.

```
...
val res = A.foo().bar().baz()
...
```

Листинг 2. Пример целевой точки внедрения

При необходимости применить совет непосредственно после вызова функции *bar()* целевая функция оборачивается в метод *run*, значение, возвращаемое функцией присваивается некоторой промежуточной переменной, после чего вставляется код совета и затем из блока возвращается перемен-

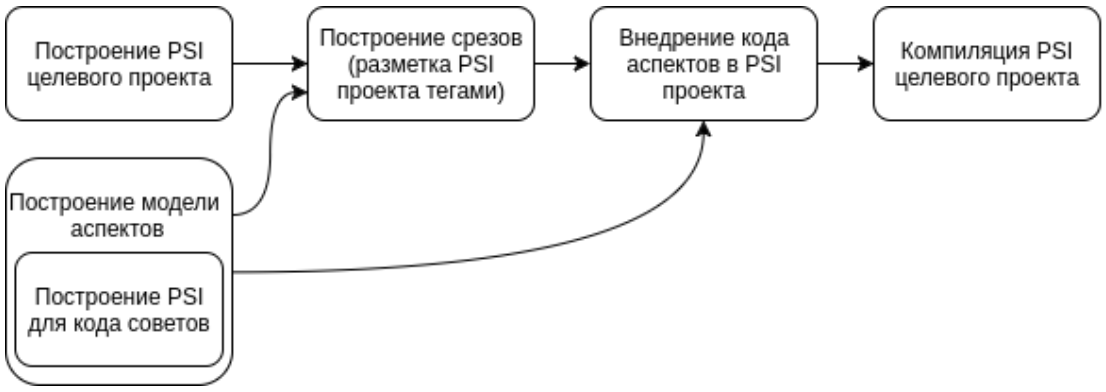

Рис. 1. Процесс внедрения аспектов в программный код при компиляции

ная, содержащая результат, возвращенный функцией *bar*, к которой был применен совет. Результат преобразования представлен в листинге 3.

```
...
val res = A.foo().run{
        val buf = bar()
        //advice code
        buf
    }.baz()
...
```

Листинг 3. Пример внедрения кода с использованием функции run

Данный подход решает проблему разнесения вложенного вызова методов, при котором пришлось бы заводить множество временных переменных, а также контролировать возвращаемые результаты методов. К недостаткам данного подхода можно отнести то, что при изменении аспекта, требуется полная перекомпиляция проекта. Однако, при таком подходе, нет необходимости производить дополнительные действия при запуске для корректной работы полученного jar-файла.

V. Тестирование разработанного прототипа

Для того чтобы убедиться в работоспособности изложенного выше подхода, был реализован прототип АОП-расширения для языка Kotlin. Проверка работоспособности разработанного подхода проводится с помощью тестирования. Сначала проверяется корректность реализации отдельных функций, затем проводится проверка приложений, к которым применили аспекты.

Проверка корректности реализации отдельных функций включает в себя:

- тестирование разбора логического выражения, описывающего срез;
- проверка правильности выделения точек внедрения;
- проверка внедрения кода советов в PSI целевой программы.

Из-за того, что сама по себе проверка PSI является сложной задачей, было решено проверить корректность работы на уровне функций следующим образом:

1) составить набор аспектов, которые будут применяться к целевой программе;
2) составить набор эталонных файлов, представляющих из себя исходные файлы целевой программы, к которым вручную применены описанные выше аспекты;
3) модифицировать PSI целевой программы, а именно произвести составление модели аспектов, разметку PSI и применить их к PSI проекта;
4) сформировать на основе модифицированных виртуальные файлов исходные файлы на языке Kotlin;
5) сравнить на уровне токенов полученные файлы с эталонными исходными файлами, в которых аспекты применены вручную.

После успешной проверки корректности работы на уровне функций, необходимо убедиться в том, что после применения аспектов программа остается корректной и работает согласно нашим ожиданиям. Это можно сделать, например, поместив в код совета вывод отладочных сообщений, и после выполнения программы необходимо сравнить результат работы программы с ожидаемым.

На начальных этапах работы, для тестирования были созданы простые примеры программ (несколько классов с 2-3 функциями в каждом из них, суммарный объем — несколько десятков строк кода). Впоследствии была проведена проверка на программах студентов, изучающих язык Kotlin (суммарный размер каждого проекта достигает несколько сотен строк кода). Время внедрения аспектов к программам размером в несколько сотен строк кода составило около 1-2 секунд.

В ходе тестирования ошибок обнаружено не было, что свидетельствует о работоспособности подхода и прототипа.

Рассмотрим пример протоколирования программы при помощи разработанного прототипа. В качестве целевой программы было выбрано приложение, вычисляющее значение арифметического выражения, заданного в файле. В качестве входных данных приложение принимает имя файла, в котором записано выражение, а также список значений параметра x, используемые при вычислении выражения. Результатом работы является ассоциативный массив в котором ключем является значение параметра, а значением — результат вы-

числения выражения при подстановке данного значения в качестве значения параметра.

Программа работает по следующему алгоритму: первым вызывается метод *pExpr*, принимающий имя файла в котором записано выражение и список значений параметра. Внутри метода *pExpr* в цикле вызывается метод *calc*, принимающий значение параметра и возвращающий вычисленное значение выражения. После чего значение параметра и результат вычисления записывается в ассоциативный массив, возвращаемый функцией *pExpr* в качестве результата. Для того, чтобы зафиксировать в журнале время начала и завершения работы метода *calc* воспользуемся аспектом, представленным в листинге 4.

```
aspect A {
  pointcut pExprPC() : execution(fun pExpr(String,
    List<Int>): Map<Int, Int> )
  pointcut calcPC() : call(fun Expression.calc(Int):
     Int)

  before(): pExprPC() && calcPC() {
    val l = java.util.logging.Logger.getLogger("ALog
    ")
    l.info("Start ${System.currentTimeMillis()}")
  }

  after(): pExprPC() && calcPC() {
    val l = java.util.logging.Logger.getLogger("ALog
    ")
    l.info("Finish ${System.currentTimeMillis()}")
  }
}
```

Листинг 4. Пример аспекта, используемого для логирования вызова метода calc

После применения аспекта, место вызова метода *calc*, представленное в листинге 5 будет преобразовано к виду, приведенному в листинге 6.

```
for (value in values) {
  result[value] = expr.calc(value)
}
```

Листинг 5. Место вызова метода calc до применения аспекта

```
for (value in values) {
  result[value] = expr.run{
    val ____a = run{
      val l = java.util.logging.Logger.getLogger("
    ALog")
      l.info("Start ${System.currentTimeMillis()}")
      calc(value)}
  val l = java.util.logging.Logger.getLogger("ALog")
  l.info("Finish ${System.currentTimeMillis()}")
  ____a}
}
```

Листинг 6. Место вызова метода calc после применения аспекта

Как видно из листинга 6, вызов метода был помещен внутрь лямбда функции *run* согласно ожиданиям. После запуска, в журнал выводятся сообщения о времени начала и окончания работы метода *calc*, что соответствует ожиданиям.

VI. Заключение

В ходе данной работы был разработан подход, позволяющий использовать аспектно-ориентированную парадигму при написании программ на языке Kotlin. На основе AspectJ было реализовано расширение, которое, используя статическое внедрение советов, модифицирует модель PSI. Работоспособность подхода была показана на ряде примеров.

Направления дальнейшей работы:

- расширение синтаксиса аспектов для поддержки всех возможностей языка Kotlin;
- оптимизация процедур внедрения аспектов;
- более глубокое тестирование разработанного программного обеспечения.

Список литературы

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier и J. Irwin, “Aspect-oriented programming,” *European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[2] (2017). The home of aspectc++. англ., url: http://www.aspectc.org/Home.php.

[3] (2017). Postsharp documentation. англ., url: http://doc.postsharp.net/.

[4] (2017). Aspectj documentation. англ., url: http://www.eclipse.org/aspectj/docs.php.

[5] (2017). Aspect oriented programming with spring. англ., url: https://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html.

[6] (2017). Aspect oriented programming — spring python. англ., url: http://docs.spring.io/spring-python/1.2.x/sphinx/html/aop.html.

[7] M. Forgac и J. Kollar, “Static and dynamic approaches to weaving,” *5th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence and Informatics*, 2007.

[8] W. Gilani и O. Spinczyk, “A family of aspect dynamic weavers,” *Proceedings of the 2003 Dynamic Aspect Workshop (DAW04 2003)*, 2003.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, J. Kersten M. Palm и W. Griswold, “An overview of aspectj,” *European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[10] (2017). Introducing kotlin support in spring framework 5.0. англ., url: https://kotlin.link/articles/Introducing-Kotlin-support-in-Spring-Framework-5-0.html.

[11] O. Spinczyk, “Documentation: Aspectc++ language reference,” *Pure-systems*, 2017.

[12] ——, “Aspectc++ — a language overview,” *Friedrich-Alexander University Erlangen-Nuremberg Computer Science 4*, 2005.

[13] (2017). Postsharp. aspects namespace. англ., url: http://doc.postsharp.net/n_postsharp_aspects.

[14] H. Kurdi, “Review on aspect oriented programming,” *International Journal of Advanced Computer Science and Applications*, 2013.

Aspect-oriented extension for the Kotlin programming language

Boris Skripal, Vladimir Itsykson

This paper present an aspect-oriented extension for the Kotlin programming language. Kotlin is a new multi-paradigm programming language that is rapidly gaining popularity. However, to be competitive with the existing programming languages it is necessary to implement many extensions already existing for the traditional programming languages. One of such extensions is an aspect-oriented approach. In the paper, after the analysis of the aspect-oriented extensions for the different object-oriented languages, we have developed a program prototype, which allows using the aspect-oriented approach with Kotlin programming language. We have developed a grammar for describing aspects and have created extension of Kotlin compiler for weaving cross-cutting functionality. The developed extension has been successfully verified on a test suite.