

Санкт-Петербургский государственный университет

Программная инженерия

Поляков Александр Романович

Декларативный
предметно-ориентированный язык
разработки мобильных приложений

Отчёт о научно-исследовательской практике

Научный руководитель:
старший преподаватель Дмитрий Вадимович Луцев

Консультант:
руководитель направления разработки языков
Huawei R&D Алексей Евгеньевич Недоря

Санкт-Петербург
2021

Оглавление

Введение	3
1. Постановка задачи	5
2. Обзор	6
2.1. Общие сведения	6
2.1.1. Завершающие лямбда-выражения	6
2.1.2. Непрозрачные типы	6
2.1.3. ASSORD: спецификации	7
2.2. Предметная область	7
2.2.1. Предметно-ориентированные языки	7
2.2.2. Подходы к реализации предметно-ориентированных языков	8
2.2.3. Отображение пользовательского интерфейса . . .	12
2.3. Существующие решения	14
3. Требования к спецификации и компилятору языка раз- работки мобильных приложений	19
3.1. Функциональные требования	19
3.2. Нефункциональные требования	19
3.3. Соответствие существующих решений собранным требо- ваниям	19
4. Архитектура и особенности реализации	20
4.1. Архитектура решения	20
4.2. Особенности реализации	20
Приложение А. Рекомендации по выбору подхода к созданию предметно-ориентирован- ного языка	22
Список литературы	23

Введение

Жизнь человека в настоящее время невозможно представить без мобильных устройств, проникших практически во все сферы его жизни. Речь идет о смартфонах, планшетах, умных часах и так далее. При этом человеко-машинное взаимодействие осуществляется через специальный вид программного обеспечения — мобильные приложения. Графический интерфейс является важной компонентой мобильных приложений, поскольку непосредственно влияет на пользовательский опыт.

Рынок мобильных устройств и приложений в последние годы непрерывно растет [12], что является причиной увеличения разнообразия и сложности архитектур процессоров и операционных систем мобильных устройств [2, 7]. Это разнообразие, а также необходимость оптимизировать процесс разработки мобильных приложений стали причиной интенсивного развития средств создания мобильных приложений [4, 8, 9, 13, 17, 18].

Конкуренция и технический прогресс постоянно расширяют и усложняют требования, предъявляемые к средствам разработки мобильных приложений. На сегодняшний день эти средства представляют собой многокомпонентные программные или программно-аппаратные комплексы, включающие среду разработки на некотором языке программирования, компилятор, отладчик и окружение исполнения для программ, написанных на этом языке, подсистему отрисовки графического интерфейса, отладочные платы целевых устройств и их эмуляторы. Таким образом, создавая новое средство разработки мобильных приложений, необходимо спроектировать и создать вышеописанные компоненты с тем, чтобы итоговое средство отвечало как можно большему числу предъявляемых требований.

Создание и отрисовка графического интерфейса приложения являются одними из наиболее сложных задач [X], с которыми сталкиваются разработчики мобильных приложений. Использование декларативных языков для описания интерфейсов и увеличение роли статической компиляции в процессе отображения пользовательского интерфейса на

экране являются тенденциями последних лет.

ASSORD — язык программирования общего назначения, сочетающий элементы компонентного, объектно-ориентированного и функционального программирования. Данный язык разрабатывается в Санкт-Петербургском научно-исследовательском центре компании HUAWEI. На текущий момент язык находится в стадии ранней разработки и не обладает устоявшейся спецификацией. Прототип компилятора языка ASSORD написан на языке программирования Go [X], имеет автоматически генерируемый по формальной контекстно-свободной грамматике в расширенной форме Бэкуса-Наура [X] и ручной LL(1) парсеры, оптимизатор, работающий над высокоуровневым промежуточным представлением программы, кодогенерацию в LLVM IR [X] и байткод некоторой виртуальной машины. В будущем спецификация языка ASSORD и исходный код его компилятора станут открытыми.

Одно из перспективных направлений применения языка ASSORD — разработка мобильных приложений. Данная работа фокусируется на внесении изменений в спецификацию языка ASSORD и его компилятор с тем, чтобы данный язык мог использоваться в качестве предметно-ориентированного языка разработки мобильных приложений, учитывающего современные тенденции данной области.

1. Постановка задачи

Целью данной работы является внесение изменений в спецификацию языка ACCORD, позволяющих декларативно описывать пользовательский интерфейс приложений, а также модификация его компилятора, предоставляющая возможность оптимизации процесса отображения пользовательского интерфейса с помощью информации о программе, доступной во время компиляции приложения. Для достижения этой цели были поставлены следующие задачи:

- выполнить обзор предметной области и существующих решений;
- выполнить сбор и анализ требований к спецификации и компилятору современного языка разработки мобильных приложений;
- предложить изменения, которые необходимо внести в спецификацию языка программирования ACCORD и его компилятор для достижения поставленной цели;
- реализовать данные изменения;
- провести апробацию полученного решения.

2. Обзор

В данном разделе представлен обзор предметной области: основные способы реализации предметно-ориентированных языков; процесс отображения пользовательских интерфейсов; существующие языки программирования общего назначения, предоставляющие пользователям возможность декларативного описания пользовательских интерфейсов с помощью предметно-ориентированных языков.

2.1. Общие сведения

2.1.1. Завершающие лямбда-выражения

Завершающие лямбда-выражения (от англ. *trailing lambdas*) — синтаксические возможности некоторых языков программирования, позволяющие при вызове функции, последним формальным параметром которой является другая функция, передать в качестве последнего аргумента лямбда-выражение, вынесенное за пределы синтаксического контекста остальных аргументов.

На листинге 1 представлено упрощённое синтаксическое правило передачи конечных лямбда-выражений, записанное в нотации инструмента ANTLR4 [1].

```
1 trailingLambda
2   : '{' parameters? ( '→' returnType 'in' )? expressionList? '}'
3   ;
4
5 functionCall
6   : functionName arguments? trailingLambda?
7   ;
```

Листинг 1: Синтаксис завершающих лямбда-выражений

2.1.2. Непрозрачные типы

Непрозрачные типы (от англ. *opaque types*) — механизм типовой системы языка программирования и его компилятора, позволяющий функции или методу с непрозрачным типом возвращаемого значения скры-

вать информацию типе возвращаемого значения. Вместо того, чтобы указывать конкретный тип в качестве типа возвращаемого значения функции, тип возвращаемого значения описывается в терминах интерфейсов, которым он соответствует. В отличие от возврата значения, тип которого является типом интерфейса, непрозрачные типы сохраняют идентичность — компилятор имеет доступ к информации о конкретном типе, в то время как клиенты функции или метода его не имеют.

2.1.3. ASSORD: спецификации

2.2. Предметная область

2.2.1. Предметно-ориентированные языки

Предметно-ориентированный язык (*domain-specific language*, DSL) — это язык программирования с более высоким уровнем абстракции, отражающий специфику решаемых с его помощью задач. Такой язык оперирует понятиями и правилами из определенной предметной области [6].

В отличие от языков программирования общего назначения, таких как C, PYTHON, JAVA, предметно-ориентированные языки предоставляют абстракции, адекватные решаемой проблеме, позволяя выражать решения, написанные с их помощью, кратко и ёмко; причём в некоторых случаях использование DSL не требует квалификации программиста. В качестве примера DSL можно привести SQL — декларативный язык программирования, применяемый для создания, модификации и управления данными в реляционной базе данных. Основным недостатком применения предметно-ориентированных языков является стоимость их разработки, требующая экспертизы как в области разработки языков программирования, так и в целевой предметной области. Это является одной из причин того, что предметные языки редко применяются для решения задач программной инженерии, в отличие от языков программирования общего назначения. Другой причиной отказа от обособленных предметных языков является тот факт, что сочета-

ние программной библиотеки и языка программирования общего назначения может заменять DSL. Программный интерфейс (*Application Programming Interface*, API) библиотеки содержит специфичный для определённой области словарь, образованный именами классов, методов и функций, доступный всем пользователям языков программирования общего назначения, подключившим библиотеку. Однако, вышеприведённый подход проигрывает предметным языкам в следующих аспектах [10, 19]:

- устоявшаяся в области нотация, как правило, выходит за рамки ограниченных механизмов определения пользовательских операторов, предоставляемых языками общего назначения;
- абстракции определённой области не всегда могут быть просто отображены в конструкции языков общего назначения [5];
- использование предметно-ориентированного языка сохраняет возможность анализа, верификации, оптимизации, параллелизации и трансформации в рамках конкретной области, что, в случае работы с исходным текстом языка программирования общего назначения, является более сложной задачей.

2.2.2. Подходы к реализации предметно-ориентированных языков

В последнее время всё больше исследований в области предметно-ориентированных языков направлены на категоризацию предметных языков, а также выработку советов и лучших практик, отвечающих на вопросы "когда и как?" создавать DSL для конкретной области [10, 14, 20].

Препроцессинг DSL-конструкции транслируются в более низкоуровневый программный код базового языка программирования общего назначения.

- *Макрокоманда.* Конструкции предметного языка представлены символическими именами, заменяемыми при обработке препроцессором

на последовательность программных инструкций базового языка.

- *Транспилиция.* Исходный код предметного языка транслируется в исходный код языка общего назначения.
- *Лексическая обработка.* Трансформация предметного языка в язык общего назначения осуществляется на уровне лексем.

Преимуществом данного подхода является простота реализации DSL, поскольку большая часть семантического анализа выполняется средствами базового языка. В то же время, это является и недостатком данного подхода ввиду отсутствия предметно-ориентированного статического анализа, оптимизаций и сообщений об ошибках.

Встраивание в базовый язык В данном подходе конструкции базового языка используются для построения библиотеки предметно-ориентированных операций. С помощью синтаксиса базового языка задаётся диалект, максимально приближенный к определённой предметной области.

Преимуществом данного подхода является полное переиспользование компилятора или интерпретатора базового языка для построения DSL. Основными недостатками являются сообщения об ошибках, соответствующие спецификации базового языка, и ограниченная синтаксическая выразительность, обусловленная существующим синтаксисом базового языка.

Самостоятельный компилятор В данном подходе для создания DSL используются методы построения компиляторов или интерпретаторов. В случае компилятора, конструкции предметного языка транслируются во внутреннее представление компилятора, а статический анализ производится над спецификацией DSL. В случае интерпретатора, конструкции предметного языка распознаются и выполняются в ходе цикла выборки-распознавания-исполнения (fetch-decode-execute cycle).

Преимуществами данного подхода являются приближенные к предметной области синтаксис языка и сообщения об ошибках. Серьёзным недостатком является необходимость создания нового компилятора или интерпретатора предметного языка.

Компилятор компиляторов Данный подход схож с предыдущим за исключением того, что все или некоторые стадии компиляции выполняются с использованием *компилятора компиляторов* — программы, воспринимающей синтаксическое или семантическое описание языка программирования и генерирующей компилятор для этого языка.

Преимуществом подхода является снижение расходов на создание компилятора предметного языка. Ограниченность итогового DSL возможностями используемого компилятора компиляторов, а также сложность проработки предметного языка в деталях, что может быть критично для достижения определённого уровня производительности и близости сообщений об ошибках к предметной области, составляют недостатки данного подхода.

Расширение существующего компилятора Компилятор языка программирования общего назначения расширяется предметно-ориентированными правилами оптимизации и/или генерации кода.

В сравнении с предыдущим, данный подход менее трудоёмок из-за возможности переиспользования частей существующего компилятора. Однако, стоит отметить, что расширение существующего компилятора может оказаться сложной задачей, для выполнения которой необходима поддержка расширений со стороны компилятора языка общего назначения, а также минимизация пересечений синтаксиса и семантики базового и предметного языков.

Использование готовых инструментов Существующие инструменты и нотации адаптируются под конкретную предметную область. Примером такого подхода являются DSL, основанные на нотации XML. В

большинстве случаев предметные языки, полученные данным способом, плохо подходят для их использования людьми в ручном режиме.

DSL as a Library?

2.2.3. Отображение пользовательского интерфейса

В данном разделе представлен один из способов отображения пользовательских интерфейсов, использующийся в популярных средствах разработки приложений [4, 13, 15, 18]. Графическая подсистема языка ACCORD основана на схожем подходе. Стоит также отметить, что далее будут рассмотрены лишь базовые принципы построения и отображения интерфейсов, которых должно быть достаточно для объяснения тех или иных оптимизационных решений, принятых в данной работе.

Задачей построения и отображения пользовательского интерфейса занимается так называемый движок рендеринга или отрисовки (*rendering engine*) — программное обеспечение, получающее изображение по какой-либо модели. Здесь *модель* — это описание любых объектов или явлений на строго определённом языке или в виде структуры данных.

На рисунке 1 представлен схематичный шаг алгоритма рендеринга пользовательского интерфейса. Так, отрисовка кадра N состоит из нескольких последовательных этапов:

1. построение дерева компонентов по текущему состоянию пользовательского интерфейса. *Дерево компонентов* — древовидная структура данных, элементами которой являются высокоуровневые компоненты интерфейса. Примером такого дерева может являться DOM (от англ. *Document Object Model* — объектная модель документа), использующийся в современных веб-браузерах.
2. построение дерева элементов по дереву компонентов. *Дерево элементов* — древовидная структура данных, изоморфная дереву компонентов, элементами которой являются структуры данных движка рендеринга.
3. построение дерева рендеринга по дереву элементов. *Дерево рендеринга* — древовидная структура данных, содержащая в себе только необходимые для непосредственно рендеринга элементы интерфейса. Говоря о переходе между деревом элементов и деревом рендеринга как об отображении, можно сказать, что не все узлы

деревя элементов имеют свой образ в дереве рендеринга. Так же стоит отметить, что узлы дерева рендеринга содержат в себе низкоуровневую информацию, такую как относительные координаты элемента в виртуальном пространстве движка отрисовки.

4. размещение и отрисовка: перевод всех относительных свойств элементов в абсолютные, например, перевод относительных координат элемента в виртуальном пространстве движка отрисовки в координаты элемента на экране реального устройства; формирование запроса к аппаратуре для отрисовки интерфейса.

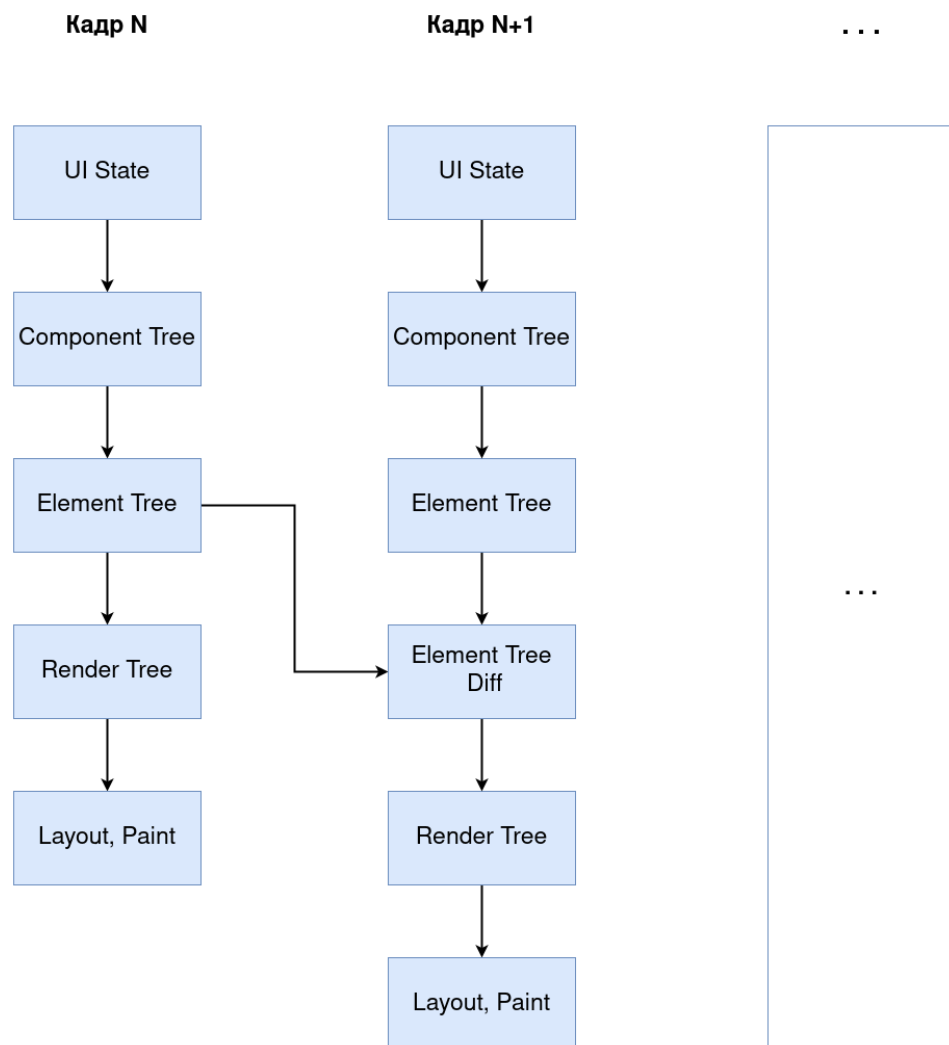


Рис. 1: Схематичный шаг алгоритма рендеринга пользовательского интерфейса

Процесс отрисовки кадра $N+1$, в свою очередь, повторяет выше-описанный алгоритм за исключением следующего: дерево рендеринга перестраивается на основании разницы между деревьями элементов текущего и предыдущего кадров. Подобное дополнение позволяет значительно повысить производительность всего графического конвейера за счёт переиспользования вычислений, хранящихся в дереве рендеринга предыдущего кадра.

2.3. Существующие решения

Dart/Flutter

FLUTTER [4] — язык разработки мобильных приложений, разрабатываемый компанией GOOGLE. По сравнению с остальными решениями, данный язык в меньшей степени является предметно-ориентированным, поскольку реализован в виде библиотеки графических компонентов, создающей словарь предметной области, для языка программирования общего назначения DART [3].

На листинге 2 представлен пример приложения счётчика количества нажатий на кнопку, написанное на языке DART/FLUTTER. Несмотря на то, что структура графического интерфейса описывается декларативно, разработчику приходится сталкиваться с конструкциями, традиционно присущими императивным языкам программирования: `@override`, `return`. Более того, разделение реализации графической компоненты на несколько связанных классов накладывает определённые ограничения на минимальную квалификацию разработчика графического интерфейса приложения, требуя от него наличия базовых навыков программирования.

```
1 import '...'  
2  
3 class CounterApp extends StatelessWidget {  
4   @override  
5   Widget build(BuildContext context) {  
6     return MaterialApp(  
7       home: Counter(someCondition),  
8     );  
9   }  
10 }
```

```

9      }
10   }
11
12   class Counter extends StatefulWidget {
13     final bool condition;
14     const Counter(this.condition);
15
16     @override
17     _CounterState createState() => _CounterState();
18   }
19
20   class _CounterState extends State<Counter> {
21     int _counter = 0;
22
23     void _incrementCounter() {
24       setState(
25         () { _counter++; }
26       );
27     }
28
29     @override
30     Widget build(BuildContext context) {
31       return Column(
32         children: <Widget>[
33           Text('Current count: $_counter'),
34           TextButton(
35             onPressed: _incrementCounter,
36             child: Text('Click on me!')
37           ),
38           widget.condition ?
39             Image.asset('assets/images/image.png') :
40             Text('Text Component')
41         ]
42       )
43     }
44   }

```

Листинг 2: Счётчик нажатия кнопки на языке DART/FLUTTER

Отрисовка графического интерфейса средством разработки мобильных приложений DART/FLUTTER осуществляется по классическому алгоритму, описанному в главе 2.2.3 за исключением некоторых оптимизаций, направленных на уменьшение количества графических компонентов, отслеживание изменений и перерисовку которых необходимо выполнять во время исполнения приложения при каждой смене кадра.

Так, выполняя обход дерева элементов нового кадра и находя разницу между ним и подобным деревом предыдущего кадра, проверка изменения некоторых графических компонентов может быть опущена при условии выполнения определённых условий, например:

- соответствие динамического типа проверяемой графической компоненты некоторому типу, гарантирующему неизменяемость данной компоненты во время исполнения программы;
- наличие у потенциально изменяемой во время исполнения программы графической компоненты выставленного атрибута, указывающего на использование в данной компоненте неизменяемых данных (прим.: компонента `Text`, созданная при помощи строкового литерала).

VueJS

Flow9

Swift/SwiftUI

SWIFTUI [17] — язык разработки мобильных приложений, разрабатываемый компанией APPLE. Данный предметно-ориентированный язык реализован методом встраивания в базовый язык, которым, в данном случае, является язык программирования общего назначения SWIFT [15]. В отличие от языка SWIFT, являющегося открытым и свободным программным обеспечением, исходный код и спецификация SWIFTUI являются проприетарными и закрытыми, что не позволяет провести полноценный анализ данного решения. При этом, на основе наблюдаемого поведения программ, написанных на языке SWIFTUI, и доступной документации можно сделать некоторые выводы о реализации SWIFTUI.

Графический интерфейс приложений на языке SWIFTUI описывается декларативно:

```
1 struct ContentView : View {  
2     var condition: Bool  
3     @State var count: Int = 0  
4 }
```



```

5   var body: some View {
6       VStack {
7           Text("Current count: \(count)")
8           Button(action: {
9               self.count += 1
10          }) {
11               Text("Click on me!")
12          }
13          if condition {
14              Image("path/to/image")
15          } else {
16              Text("text component")
17          }
18      }
19  }
20 }

```

Листинг 3: Счётчик нажатия кнопки на языке SwiftUI

Каждый компонент графического интерфейса, описываемый пользователем, должен явно указывать своё соответствие специальному протоколу [16] `VIEW`. Согласно этому протоколу, тип, соответствующий ему, обязан иметь поле `BODY` соответствующего протоколу `VIEW` типа. На языке SWIFT данный протокол может быть описать следующим образом:

```

1 protocol View {
2     associatedtype Body: View
3     var body: Self.Body { get }
4 }

```

Листинг 4: Реализация протокола VIEW на языке SWIFT

Для достижения декларативности, представленной на листинге 3, разработчикам языка SWIFT потребовалось расширить его спецификацию следующими нововведениями:

- завершающие лямбда-выражения 2.1.1;
- непрозрачные типы 2.1.2;
- неявный возврат значений из функций, состоящих из единственного выражения возврата результата.

Важной особенностью SwiftUI является высокая степень использования информации, доступной во время компиляции программы, для оптимизации процесса отображения пользовательского интерфейса. Во время компиляции приложения для каждой пользовательской графической компоненты выводится её статический тип. Так, графическая компонента, представленная на листинге 3, имеет следующий статический тип данных:

`VStack<Text, Button<Text>, _ConditionalContent<Image, Text>`. Данный тип не будет изменён во время выполнения программы, что позволяет избежать ресурсоёмкого шага алгоритма отрисовки графического интерфейса — нахождения разницы между деревьями элементов текущего и предыдущего кадров: компилятор имеет информацию о том, какие графические компоненты могут быть изменены во время исполнения программы, а также о их смещении внутри структуры, которой представлен пользовательский интерфейс.

3. Требования к спецификации и компилятору языка разработки мобильных приложений

В данном разделе представлены требования к современному средству разработки мобильных приложений, собранные на основе обзора существующих решений, а также сводная таблица соответствия существующих решений собранным требованиям. Предъявляемые требования были разделены на две группы: функциональные и нефункциональные.

3.1. Функциональные требования

Оптимизация отрисовки графического интерфейса с использованием статической информации о программе

Реактивные обновления пользовательского интерфейса

3.2. Нефункциональные требования

Декларативность описания пользовательского интерфейса мобильного приложения

Предоставление отладочных возможностей

Кроссплатформенная разработка

Поддержка интегрированной средой разработки

3.3. Соответствие существующих решений собранным требованиям

Как видно из таблицы 1, на данный момент наиболее популярные языки разработки мобильных приложений не удовлетворяют всем требованиям, собранным в данной работе.

	<i>VueJS</i>	<i>SwiftUI</i>	<i>Flutter</i>	<i>Flow9</i>
Оптимизация отрисовки графического интерфейса с использованием статической информации о программе	–	+	–	–
Реактивные обновления пользовательского интерфейса	+	+	–	+
Декларативность описания пользовательского интерфейса	+	+	+	+
Кроссплатформенная разработка	+	–	+	+
Предоставление отладочных возможностей	+	+	+	+
Поддержка интегрированной среды разработки	+	+	+	+

Таблица 1: Сводная таблица соответствия существующих решений собранным требованиям

4. Архитектура и особенности реализации

В данном разделе представлена общая архитектура решения и особенности реализации, позволившие решению удовлетворить всем требованиям к современному языку разработки мобильных приложений, указанным в разделе 3.

4.1. Архитектура решения

4.2. Особенности реализации

Встраивание предметного языка разработки мобильных приложений в язык программирования *Accord*

Реализация предметно-ориентированного языка разработки мобильных приложений методом встраивания данного предметного языка в базовый язык, которым в данном случае выступает язык программирования *Accord* сохраняет преимущества и недостатки данного подхода, описанные в разделе 2.2.1. Однако, предметно-специфичные анализ, оптими-

зации и трансформации были учтены на этапе проектирования данного решения, что нивелировало часть недостатков подхода.

Основная функциональность языка *Accord*, использованная для создания предметного языка разработки приложений:

- Легковесные структуры
- Статическая типизация
- Семантический анализ
- Реактивность на уровне языка
- Кроссплатформенность

Разделение дерева компонент на статическую и динамическую части

Благодаря статической типизации, на стадии компиляции возможно разделить пользовательский интерфейс, представленный деревом компонент, на две обособленные в памяти приложения составляющие.

- Статическая часть дерева: отсутствие изменения структуры дерева компонент во время исполнения мобильного приложения позволяет полностью исключить узлы такого дерева из дорогостоящего алгоритма нахождения разницы между деревьями элементов интерфейса.
- Динамическая часть дерева: знание о том, какие элементы интерфейса могут стать точкой начала изменения структуры дерева компонент позволяет компилятору вызывать алгоритм нахождения разницы между деревьями элементов точно, минимизируя количество входных данных этого алгоритма.

Выделение данных состояния приложения в памяти

Декларативность

А. Рекомендации по выбору подхода к созданию предметно-ориентированного языка

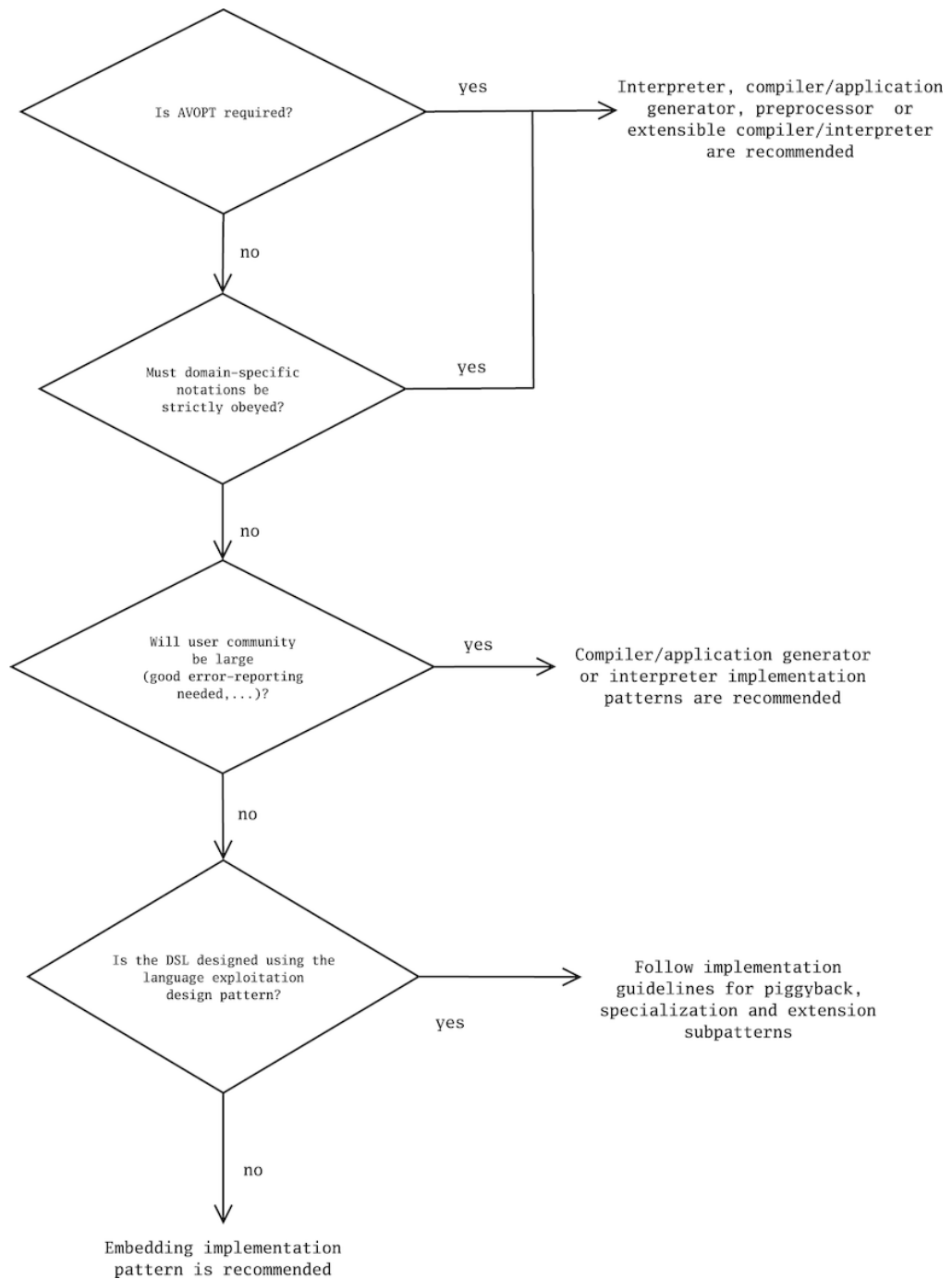


Рис. 2: Алгоритм выбора метода разработки DSL [10]

Список литературы

- [1] ANTLR Homepage. — Access mode: <https://www.antlr.org/> (online; accessed: 18.04.2020).
- [2] Advaita B Mopuru Lahari Gopalakrishnan T. Trends in Processor Architecture of Mobile Phones: A Survey // International Journal of Advanced Science and Technology. — 2020. — May. — Vol. 29, no. 05. — P. 6265 – 6274. — Access mode: <http://sersc.org/journals/index.php/IJAST/article/view/15631>.
- [3] Dart Homepage. — Access mode: <https://dart.dev/> (online; accessed: 10.04.2021).
- [4] Flutter Homepage. — Access mode: <https://flutter.dev/> (online; accessed: 02.12.2020).
- [5] Gray Jeff, Karsai Gabor. An Examination of DSLs for Concisely Representing Model Traversals and Transformations. — 2003. — 01. — P. 325.
- [6] Kelly Steven, Tolvanen Juha-pekka. Domain-Specific Modeling: Enabling Full Code Generation. — 2008. — 04. — ISBN: 978-0-470-03666-2.
- [7] Kolawole Emmanuel Olawale, Lofinmakin Damilola Ayomiposi, Nwido-bie Gabriel. Trends in Mobile Phones Processor Architecture, Academia. — Access mode: https://www.academia.edu/38755927/Trends_in_Mobile_Phones_Processor_Architecture (online; accessed: 02.12.2020).
- [8] Kotlin Homepage. — Access mode: <https://kotlinlang.org/> (online; accessed: 02.12.2020).
- [9] Kramer D., Clark T., Oussena S. MobDSL: A Domain Specific Language for multiple mobile platform deployment // 2010 IEEE Inter-

national Conference on Networked Embedded Systems for Enterprise Applications. — 2010. — P. 1–7.

- [10] Mernik Marjan, Heering Jan, Sloane Anthony. When and How to Develop Domain-Specific Languages // ACM Comput. Surv. — 2005. — 12. — Vol. 37. — P. 316–.
- [11] Mobile Devices Pixel Density Statistics. — Access mode: <https://pixensity.com/list/phone/> (online; accessed: 20.12.2020).
- [12] Mobile Devices Sales Statistics. — Access mode: <https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/> (online; accessed: 02.12.2020).
- [13] React Native Homepage. — Access mode: <https://reactnative.dev/> (online; accessed: 02.12.2020).
- [14] Spinellis Diomidis. Notable design patterns for domain-specific languages // Journal of Systems and Software. — 2001. — Vol. 56, no. 1. — P. 91 – 99. — Access mode: <http://www.sciencedirect.com/science/article/pii/S0164121200000893>.
- [15] Swift HomePage. — Access mode: <https://developer.apple.com/swift/> (online; accessed: 03.04.2021).
- [16] Swift Protocols Documentation. — Access mode: <https://docs.swift.org/swift-book/LanguageGuide/Protocols.html> (online; accessed: 18.04.2021).
- [17] SwiftUI HomePage. — Access mode: <https://developer.apple.com/xcode/swiftui/> (online; accessed: 03.04.2021).
- [18] Vue Native HomePage. — Access mode: <https://vue-native.io/> (online; accessed: 02.12.2020).
- [19] Wile D. S. Supporting the DSL Spectrum // Journal of computing and information technology. — 2001. — Vol. 9, no. 4. — P. 263 – 287.

- [20] A preliminary study on various implementation approaches of domain-specific language / Tomaž Kosar, Pablo E. Martínez López, Pablo A. Barrientos, Marjan Mernik // Information and Software Technology. — 2008. — Vol. 50, no. 5. — P. 390 – 405. — Access mode: <http://www.sciencedirect.com/science/article/pii/S0950584907000419>.