Санкт-Петербургский государственный университет

Поляков Александр Романович

Отчёт по преддипломной научной практике

Декларативный предметно-ориентированный язык разработки мобильных приложений

Уровень образования: магистратура

Направление *09.04.04 «Программная инженерия»*

Основная образовательная программа *ВМ.5666.2019 «Программная инженерия»*

Научный руководитель:
к.ф.-м.н., доцент кафедры системного программирования Д.В. Луцив

Консультант:
к.ф.-м.н., руководитель направления разработки языков программирования
ООО "Техкомпания Хуавей" А.Е. Недоря

Рецензент:
инженер-программист ООО "Техкомпания Хуавей" Д.И. Соломенников

Санкт-Петербург
2021

Оглавление

Введение

Жизнь современного человека 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 [15], также возрастает и разнообразие архитектур процессоров и операционных систем мобильных устройств [2,8]. Это разнообразие, а также необходимость оптимизировать процесс разработки мобильных приложений стали причиной интенсивного развития средств создания мобильных приложений [4, 9, 10, 16, 21, 22].

На сегодняшний день средства разработки мобильных приложений представляют собой многокомпонентные программные и/или программно-аппаратные комплексы, включающие среду разработки на некотором языке программирования, компилятор, отладчик, окружение исполнения для программ, написанных на этом языке, подсистему отрисовки графического интерфейса, отладочные платы целевых устройств и их эмуляторы и так далее.

Несмотря на кажущуюся «обыденность», создание и отрисовка графического интерфейса приложения являются одними из наиболее сложных задач, с которыми сталкиваются разработчики мобильных приложений. Использование декларативных языков для описания интерфейсов и увеличение роли компилятора в процессе отображения пользовательского интерфейса на экране являются тенденциями последних лет.

Accord — 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 Accord написан на языке программирования Go [5], имеет автоматически генерируемый по формальной контекстно-свободной грамматике в расширенной форме Бэкуса-Наура [12] и ручной LL [17] парсеры, оптимизатор, работающий над высокоуровневым промежуточным представлением программы, кодогенерацию в LLVM IR [11] и байткод некоторой виртуальной машины. В будущем спецификация языка Accord и исходный код его компилятора станут открытыми.

Одно из перспективных направлений применения языка Accord — разработка мобильных приложений. На данный момент, универсальный характер языка и реализации его компилятора не позволяют оптимально реализовать необходимую для мобильной разработки функциональность, связанную с программированием интерфейса пользователя. Данная работа фокусируется на внесении изменений в спецификацию языка Accord, позволяющих декларативно описывать пользовательский интерфейс приложений, а также на изменении его компилятора для предоставления возможности оптимизации процесса отображения пользовательского интерфейса с помощью информации о программе, доступной во время компиляции приложения.

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 современному языку разработки мобильных приложений и его компилятору;
- 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: общие сведения; 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. Accord: спецификации

Спецификации в языке Accord — тип данных, являющийся именованным множеством сигнатур методов. Тип данных T называется соответствующим спецификации S тогда и только тогда, когда $SM \subseteq TM$, где SM — множество сигнатур методов спецификации S, TM — множество сигнатур методов типа T. Проверка на соответствие объекта определённого типа той или иной спецификации происходит во время выполнения программы за исключением случаев, когда данная проверка может быть проведена во время компиляции исходного кода с использованием статической информации о программе.

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 [7].

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 программи-

ста. 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 сочетание 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 [13, 23]:

- 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 [6];
- 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 [13, 18, 24].

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

- *Макрокоманда.* Конструкции предметного языка представлены символьными именами, заменяемыми при обработке препроцессором на последовательность программных инструкций базового языка.
- *Транспиляция.* 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-ориенти-

рованными 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.

2.2.3. Отображение пользовательского интерфейса

В данном разделе описан процесс обновления пользовательского интерфейса, ставший де-факто стандартом для популярных современных средств разработки мобильных приложений. Стоит 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 отображения пользовательского интерфейса занимается так называемый движок рендеринга или отрисовки (*rendering engine*) — программное обеспечение, генерирующее изображение по модели. Здесь *модель* — это описание любых объектов или явлений на строго определённом языке или в виде структуры данных.

Процесс отрисовки кадра (рис. 1) состоит из последовательного построения нескольких древовидных структур, каждая из которых является образом графического интерфейса, записанного пользователем на некотором языке.

- *Дерево компонентов* — древовидная структура данных, элемента-

ми которой являются высокоуровневые компоненты интерфейса. Примером такого дерева может являться DOM (от англ. *Document Object Model* — объектная модель документа), использующийся в современных веб-браузерах.

- *Дерево элементов* — древовидная структура данных, изоморфная дереву компонентов, элементы которой могут быть использованы движком отрисовки для сравнения графических компонентов двух соседних кадров.

- *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 дерева 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.

При отрисовке следующего кадра, движок рендеринга перерисовывает только изменившиеся по сравнению с предыдущим кадром части интерфейса. Для этого, между двумя соседними кадрами находится разница, которая и отправляется в движок отрисовки. Нахождение этой разницы является одной из наиболее ресурсоёмких операций во время выполнения приложения, поскольку она требует создания нового дерева, обхода деревьев текущего и предыдущего кадров, а также сравнения соответствующих элементов.

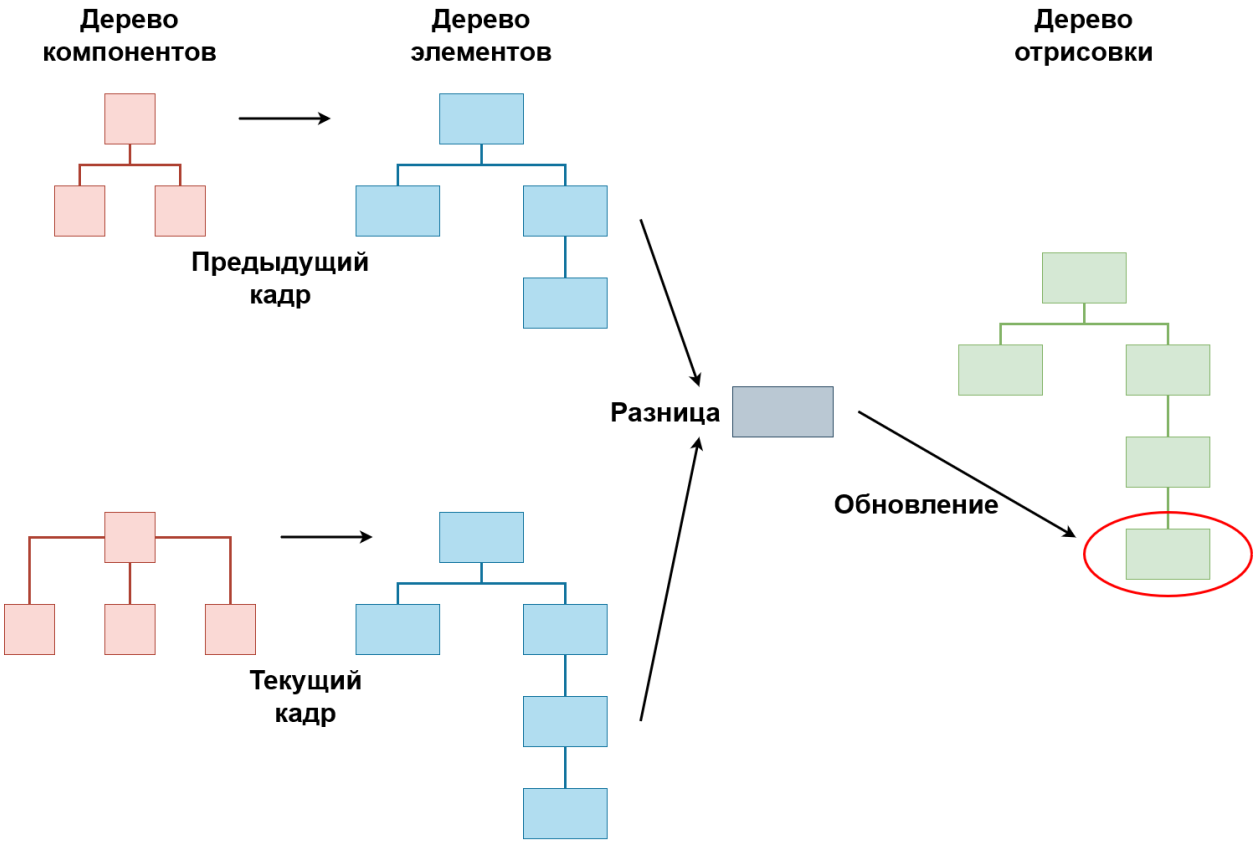

Рис. 1: Процесс отображения пользовательского интерфейса

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.

```
import '...'

class CounterApp extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Counter(someCondition),
        );
    }
}

class Counter extends StatefulWidget {
    final bool condition;
    const Counter(this.condition);

    @override
    _CounterState createState() => _CounterState();
}

class _CounterState extends State<Counter> {
    int _counter = 0;

    void _incrementCounter() {
        setState(
            () { _counter++; }
        );
    }

    @override
    Widget build(BuildContext context) {
        return Column(
            children: <Widget>[
                Text("Current count: $_counter"),
                TextButton(
                    onPressed: _incrementCounter,
                    child: Text("Click on me!")
                ),
                widget.condition ?
                    Image.asset("assets/images/image.png") :
                    Text("Text Component")
            ]
```

```
        )
    }
}
```

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).

Kotlin UI DSL

React Native

Swift/SwiftUI

SwiftUI [21] — 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 [19].

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:

```
struct ContentView : View {
    var condition: bool
    @State var count: Int = 0

    var body: some View {
        VStack {
            Text("Current count: \(count)")
            Button(action: {
                self.count += 1
            }) {
                Text("Click on me!")
            }
            if condition {
                Image("path/to/image")
            } else {
                Text("text component")
            }
        }
    }
}
```

Листинг 3: Счётчик нажатия кнопки на языке SwiftUI

Каждый компонент графического интерфейса, описываемый пользователем, должен явно указывать своё соответствие специальному протоколу [20] 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:

```
protocol View {
    associatedtype Body: View
    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 во время компиляции мобильного приложения

Отрисовка графического интерфейса мобильного приложения является ресурсоёмкой задачей. Так, для современного мобильного устройства, обладающего следующими характеристиками:

- разрешение экрана 1644×3840 точек;
- обновление кадра с частотой 120 Гц;

пропускная способность графического конвейера, при условии хранения цвета одного пикселя в четырёх байтах, должна составлять $1644 * 3840 * 4 * 120 = 3030220800$ байт в секунду, что составляет около 2.8 гигабайт. Такая нагрузка на мобильное устройство недопустима ввиду скромных возможностей охлаждения таких устройств, а также ввиду ограниченного объема заряда аккумуляторной батареи.

Более того, при условии необходимости обновления кадров с частотой 120 Гц, новый кадр должен быть доставлен движку отрисовки не позднее, чем через 8 мс после отрисовки предыдущего. Если флагманские мобильные устройства и способны обеспечить должную производительность классического подхода к обновлению интерфейса, пред-

ставленного в пункте 2.2.3, за счёт увеличенного энергопотребления, то менее мощные устройства на это уже не способны.

Исходя из вышеописанного, становится очевидной важность различных оптимизаций отрисовки интерфейса. В особенности тех, что могут быть применены во время компиляции приложения.

Реактивные обновления пользовательского интерфейса

Реактивное программирование — парадигма программирования, ориентированная на потоки данных и распространение изменений. Это означает существование возможности выражения статических и динамических потоков данных, а также то, что нижележащая модель исполнения должна автоматически распространять изменения благодаря в рамках определённого потока данных. В контексте разработки графического интерфейса, под реактивностью понимается автоматическое обновление пользовательского интерфейса при изменении данных, помеченных как реактивные.

3.2. Нефункциональные требования

Декларативность описания пользовательского интерфейса мобильного приложения

В отличие от императивного программирования интерфейсов, в котором разработчик задаёт как именно необходимо построить интерфейс, декларативное описание интерфейсов позволяет разработчику указать то, что он хочет увидеть на экране, не заботясь о том, каким образом интерфейс будет построен.

Предоставление отладочных возможностей

Отладка программ — неизбежный процесс, возникающий при разработке большого и сложного программного обеспечения. Существуют несколько видов отладки: отладочный вывод, трассировка, использование отладчика. Наиболее совершенным способом отладки является

применение специальных инструментов — отладчиков. Однако, несмотря на это, существуют ситуации, когда использование отладчика невозможно по каким-либо причинам. Поэтому современный язык разработки мобильных приложений должен предоставлять пользователям различные виды отладки для решения разного спектра проблем.

Кроссплатформенная разработка

Возможность разработки и поддержки единой кодовой базы мобильного приложения уже многие годы является потребностью разработчиков. Такая возможность сказывается на стоимости разработки приложения, что, в свою очередь, через стоимость конечного продукта влияет и на пользовательский опыт, получаемый потребителем от взаимодействия с приложением.

Поддержка интегрированной средой разработки

Интегрированная среда разработки стала неотъемлемой частью процесса разработки комплексных программных систем. Мобильный приложения являются примерами таких систем, поскольку зачастую при их разработке требуется не только окружение для работы с исходным кодом, но и для работы с ресурсами приложения, базой данных и так далее.

3.3. Соответствие существующих решений собранным требованиям

В таблице 1 отображено соответствие популярных существующих решений собранным требованиям к языку разработки мобильных приложений и его компилятору. Как видно из таблицы, ни одно из существующих решений не соответствует всем собранным в данной работе требованиям.

	Dart/ Flutter	Kotlin UI DSL	React Native	Swift/ SwiftUI
Оптимизация отрисовки графического интерфейса во время компиляции мобильного приложения	−	−	−	+
Реактивные обновления пользовательского интерфейса	+	+	+	+
Декларативность описания пользовательского интерфейса	+	+	+	+
Кроссплатформенная разработка	+	+	+	−
Предоставление отладочных возможностей	+	+	+	+
Поддержка интегрированной средой 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

Начинать рассказ об архитектуре предлагаемого решения стоит с перечисления основных концепций, лежащих в основе данного решения, с последующим разъяснением каждой из них:

- встраивание предметного языка разработки мобильных приложений в язык программирования общего назначения *Accord*;
- декларативное описание графического интерфейса;
- реактивность — часть языка;
- статическая типизация всех графических компонент.

Встраивание в язык программирования *Accord*

В качестве метода реализации предметного языка разработки мобильных приложений был выбран метод встраивания в базовый язык. Данный выбор был сделан на основе рекомендаций по выбору подхода к созданию предметно-ориентированного языка, представленного в приложении А.

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

Декларативное описание графического интерфейса

Как показано на листинге 5, графический интерфейс мобильного приложения на разработанном языке описывается декларативно.

```
type Counter = struct (View[T]) {
    rx var counter: i32
    var cond: bool

    var body: T = Column(
        Text(''Current count: ${this.counter}''),
        Button(''Click on me!'')
            .onClick(fn() { this.counter += 1 })
            .backgroundColor(Color.Green)
        ConditionalView(
            this.cond,
            Image(''path/to/img''),
            Text(''Condition is false'')
        )
    )
}
```

Листинг 5: Счётчик нажатия кнопки на модифицированном языке Accord

Реактивность — часть языка

В язык программирования *Accord* был добавлен модификатор *rx*, применимый к полям структурных типов и переменным. В контексте отрисовки графического интерфейса, любая модификация данных графической компоненты, помеченных данным модификатором, пометит компоненту как "требующую обновление".

Статическая типизация всех графических компонент

Поле body, определяемое на 5 строке, инициализируется выражением, декларативно описывающим структуру графического интерфейса. С точки зрения выполнения оптимизации отрисовки интерфейса, наиболее важным является тот факт, что поле body имеет статический тип, который по определению данного поля является обобщённым, однако при указании конкретного значения, данный обобщённый тип будет настроен компилятором. Применение обобщённых типов здесь позволяет пользователю не думать о том, какой конкретно тип получит созданная им компонента, в то время как компилятору данная информацию будет доступна после настройки обобщённого типа.

Зная типовую информацию обо всех компонентах графического интерфейса (включая их размеры и взаимное расположение в памяти), мы можем заменить обход дерева элементов интерфейса на точечные вызовы процедур обновления только для тех компонент, которые потенциально могут измениться во время исполнения программы. Например, зная тип компоненты интерфейса *Counter*, мы можем точечно обратиться к каждому из его внутренних подкомпонент при необходимости. Для приведённого примера, компилятор может автоматически сгенерировать оптимизированную процедуру обновления интерфейса 6, в которой изменению и проверке будут подлежать лишь потенциально меняющиеся компоненты:

- Нет необходимости проверять компоненту Column, она как была Column из текста, кнопки и некоторой условной компоненты, так и останется на протяжении всего исполнения программы

- Нет необходимости в проверке кнопки, поскольку она не зависит от потенциально изменяющихся данных

```
fn Counter.rerender() {
    // skip Column
    Text.rerender(fieldAddress(0, 0), args...)
    // skip Button
    ConditionalView.rerender(
        fieldAddress(0, 2), args...,
    )
}
```

Листинг 6: Автоматически сгенерированная процедура обновления интерфейса

5. Результаты

На данный момент в рамках преддипломной научной практики были достигнуты следующие результаты:

- Выполнен обзор предметной
  - Предметно-ориентированные языки
  - Отрисовка графического интерфейса
- Выполнен обзор существующих решений
  - *Dart/Flutter*
  - *Kotlin UI DSL*
  - *React Native*
  - *Swift/SwiftUI*
- Собраны требования к современному языку разработки мобильных приложений и его компилятору
- Предложены изменения спецификации и компилятора языка *Accord*, позволяющие языку *Accord* удовлетворить всем собранным требованиям
- Предложенные изменения реализованы в прототипе компилятора языка *Accord*

Ближайшими планами является проведение апробации полученного решения.

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

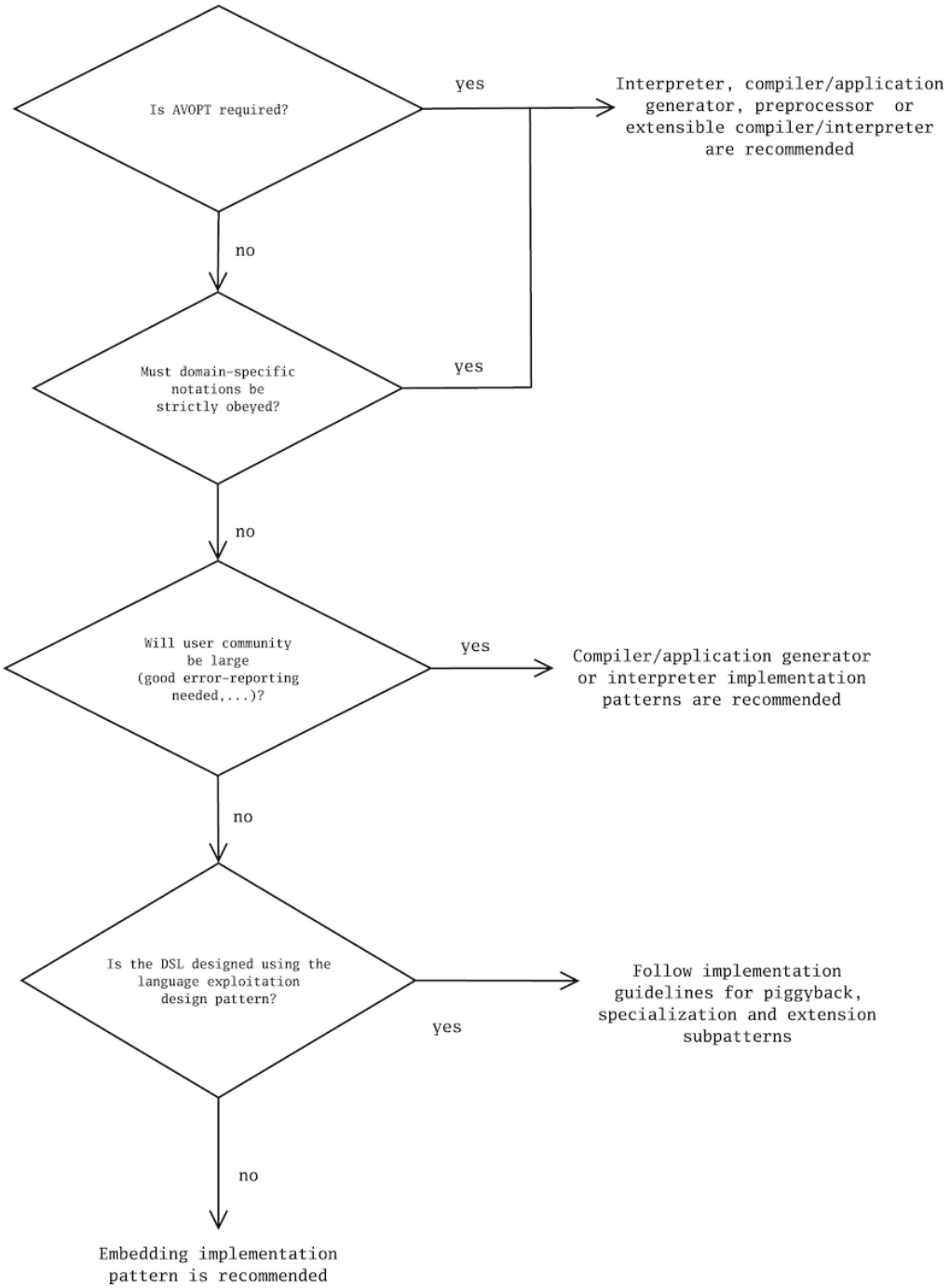

Рис. 2: Фреймворк выбора метода разработки DSL [13]

Список литературы

[1] ANTLR Homepage. — Access mode: https://www.antlr.org/ (online; accessed: 18.04.2021).

[2] Advaitha 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] Go Homepage. — Access mode: https://golang.org/ (online; accessed: 28.04.2021).

[6] Gray Jeff, Karsai Gabor. An Examination of DSLs for Concisely Representing Model Traversals and Transformations. — 2003. — 01. — P. 325.

[7] Kelly Steven, Tolvanen Juha-pekka. Domain-Specific Modeling: Enabling Full Code Generation. — 2008. — 04. — ISBN: 978-0-470-03666-2.

[8] Kolawole Emmanuel Olawale, Lofinmakin Damilola Ayomiposi, Nwidobie 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).

[9] Kotlin Homepage. — Access mode: https://kotlinlang.org/ (online; accessed: 02.12.2020).

[10] Kramer D., Clark T., Oussena S. MobDSL: A Domain Specific Language for multiple mobile platform deployment // 2010 IEEE International Conference on Networked Embedded Systems for Enterprise Applications. — 2010. — P. 1–7.

[11] LLVM IR Language Reference. — Access mode: https://llvm.org/docs/LangRef.html (online; accessed: 28.04.2021).

[12] McCracken Daniel D., Reilly Edwin D. Backus-Naur Form (BNF) // Encyclopedia of Computer Science. — GBR : John Wiley and Sons Ltd., 2003. — P. 129–131. — ISBN: 0470864125.

[13] Mernik Marjan, Heering Jan, Sloane Anthony. When and How to Develop Domain-Specific Languages // ACM Comput. Surv. — 2005. — 12. — Vol. 37. — P. 316–.

[14] Mobile Devices Pixel Density Statistics. — Access mode: https://pixensity.com/list/phone/ (online; accessed: 20.12.2020).

[15] 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).

[16] React Native Homepage. — Access mode: https://reactnative.dev/ (online; accessed: 02.12.2020).

[17] Sippu Seppo, Soisalon-Soininen Eljas. On LL (k) parsing // Information and Control. — 1982. — Vol. 53, no. 3. — P. 141–164.

[18] 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.

[19] Swift HomePage. — Access mode: https://developer.apple.com/swift/ (online; accessed: 03.04.2021).

[20] Swift Protocols Documentation. — Access mode: https://docs.swift.org/swift-book/LanguageGuide/Protocols.html (online; accessed: 18.04.2021).

[21] SwiftUI HomePage. — Access mode: https://developer.apple.com/xcode/swiftui/ (online; accessed: 03.04.2021).

[22] Vue Native HomePage. — Access mode: https://vue-native.io/ (online; accessed: 02.12.2020).

[23] Wile D. S. Supporting the DSL Spectrum // Journal of computing and information technology. — 2001. — Vol. 9, no. 4. — P. 263 – 287.

[24] A preliminary study on various implementation approaches of domain-specific language / Tomaž Kosar, Pablo E. Martı´nez 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.