Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа интеллектуальных систем и суперкомпьютерных технологий

Лабораторная работа

Дисциплина: Проектирование мобильных приложений

Тема: Навигация в приложении

Выполнил студент гр. 3530901/90201 _____Е. В. Бурков
(подпись)

Принял старший преподаватель _____ А. Н. Кузнецов
(подпись)

“_____” _____ 2021 г.

Санкт-Петербург
2021

Содержание

Цели

- Познакомиться с Google Codelabs и научиться его использовать как способ быстрого изучения новых фреймворков и технологий
- Изучить основные возможности навигации внутри приложения: создание новых activity, navigation graph

Задачи

- Познакомится с содержимым курса Jetpack Compose и выполните codelab "Jetpack Compose basics"
- Реализовать навигацию между экранами одного приложения согласно изображению ниже с помощью Activity, Intent и метода startActivityForResult.
- Решить предыдущую задачу с помощью Activity, Intent и флагов Intent либо атрибутов Activity.
- Дополнить граф навигации новым(-и) переходом(-ами) с целью демонстрации какого-нибудь (на свое усмотрение) атрибута Activity или флага Intent, который еще не использовался для решения задачи. Поясните пример и работу флага/атрибута.
- Решить исходную задачу с использованием navigation graph. Все Activity должны быть заменены на Fragment, кроме Activity 'About', которая должна остаться самостоятельной Activity. В отчете сравните все решения.

Инструмент Jetpack Compose

Jetpack Compose это современный набор инструментов для создания нативного пользовательского интерфейса для Андройд. Благодаря возможностям языка Kotlin намного уменьшается количество необходимого кода. При помощи Kotlin составлен проблемно-ориентированный язык (DSL), который решает узконаправленную задачу.

Процесс написания программы состоит из написания “составных” функций (англ. composable). Внутри этих функций вызываются необходимые методы (например Text()) и описываются их параметры. С помощью Jetpack Compose можно также составлять layout’ы, использовать Meterial design (стиль графического дизайна интерфейсов программного обеспечения и приложений, разработанный компанией Google), делать сложные вещи по типу списком и многое другое. Для получения базовых навыков работы с данным инструментом был пройден соответствующий Google Codelab.

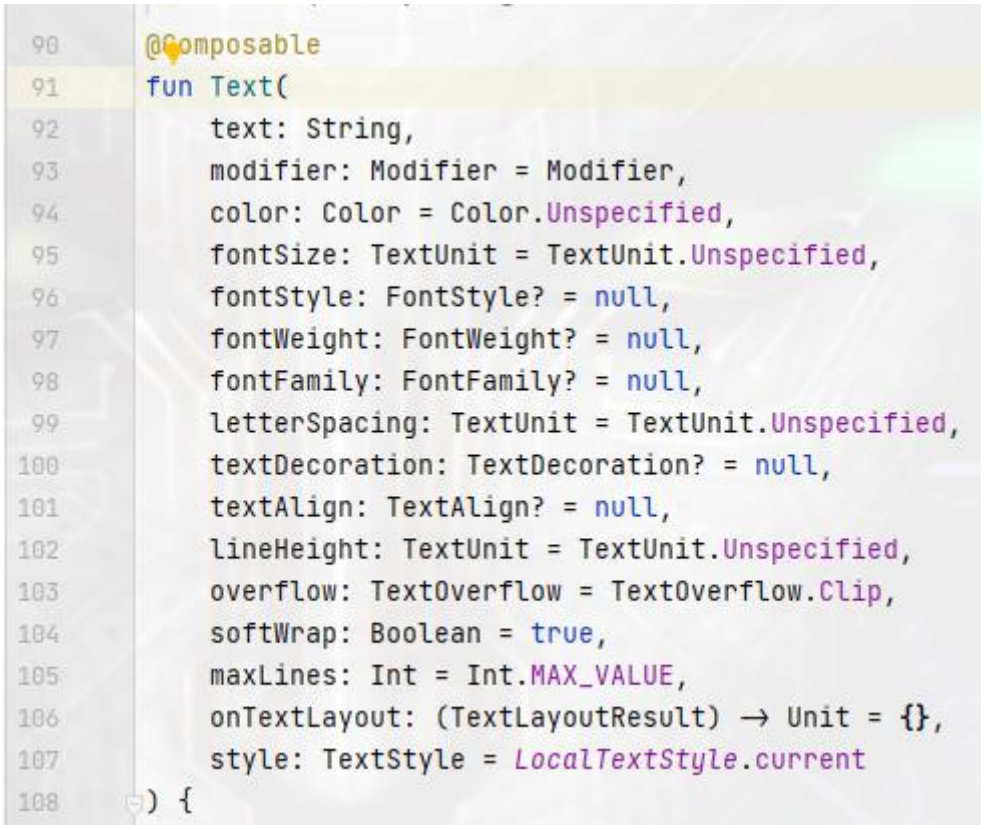
(1б) Выполнение Jetpack Compose Codelab

Для начала необходимо создать проект. Благо для использования Compose в Android Studio уже есть необходимый темплейт. Важно обратить внимание, что использование Compose возможно с 21 версии API (minimumSdkVersion в манифесте). Перед нами открывается файл нашего Activity в котором уже есть несколько функций.

Листинг 1 Функция Greeting

```
@Composable
private fun Greeting(name: String) {
    Text(text = "Hello $name!")
}
```

Composable функции объявляются как классические функции на языке Kotlin и имеют аннотацию “@Composable”. Данная аннотация позволяют функции вызывать другие Composable функции. В данной функции используется библиотечная функция Text(), которая также является Composable.



```
90  @Composable
91  fun Text(
92      text: String,
93      modifier: Modifier = Modifier,
94      color: Color = Color.Unspecified,
95      fontSize: TextUnit = TextUnit.Unspecified,
96      fontStyle: FontStyle? = null,
97      fontWeight: FontWeight? = null,
98      fontFamily: FontFamily? = null,
99      letterSpacing: TextUnit = TextUnit.Unspecified,
100     textDecoration: TextDecoration? = null,
101     textAlign: TextAlign? = null,
102     lineHeight: TextUnit = TextUnit.Unspecified,
103     overflow: TextOverflow = TextOverflow.Clip,
104     softWrap: Boolean = true,
105     maxLines: Int = Int.MAX_VALUE,
106     onTextLayout: (TextLayoutResult) → Unit = {},
107     style: TextStyle = LocalTextStyle.current
108 ) {
```

Рис. 1 Функция Text() в пакете android.compose

Также в файле нас встречает функция DefaultPreview(). Она позволяет выполнить наш Compose код и посмотреть, как он будет выглядеть.

Листинг 2 Функция Preview

```
@Preview(showBackground = true)
@Composable
private fun DefaultPreview() {
    ComposeCodelabTheme {
        Greeting("Android")
    }
}
```

Что касается Activity, вместо привычного нам setContentView нас встречает setContent, который принимает Composable функции.

Далее в работе предлагается проверить как работает функция Surface. Она позволяет устанавливать необходимые плюшки для темы приложения. Также рассказывается про объект Modifier, который может различным образом воздействовать на выход нашей функции.

Для удобства написания приложения возможно использовать пере использование компонентов, для демонстрации этого, наш UI был помещён в отдельную функцию и после эта функция передавалась и в Activity и в DefaultPreview.

При помощи методов Row, Column, Box мы можем работать над взаимными расположениями объектов на интерфейсе. Для сохранения состояния UI используется remember и mutableStateOf. С их помощью было реализовано “открывание” и “закрывание” окон по кнопкам. Далее был добавлен новый экран и реализованы переходы между ними с сохранением состояний UI.

При помощи LazyColumn было реализовано что-то на подобии RecyclerView. Далее были добавлены анимации и выполнено несколько пунктов для улучшения внешнего вида приложения.

Создание новых Activity

Обычно при запуске приложения, для пользователя оно выходит на первый план. Если приложение не было до этого запущено, то создаётся новый Task этого приложения и в него помещается главное Activity. Если приложение уже было запущено, то система работает с уже созданным стэком. Когда одно Activity запускает другое, то новое Activity помещает на верхушке стэка и отображается пользователю. Если пользователь выходит из Activity (back action), то оно убирается из стэка. Для простоты можно сказать, что backstack представляет собой очередь LIFO.

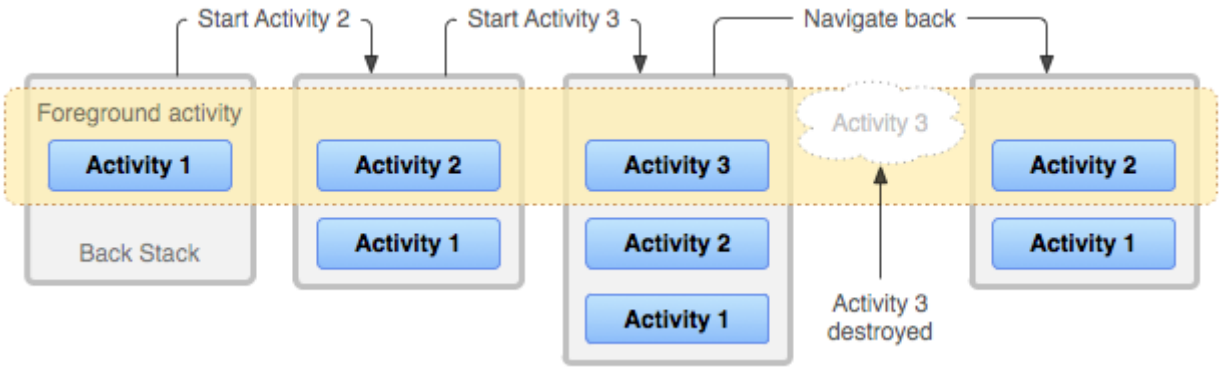


Рис. 2 Backstack

Также может получиться ситуация, когда в стэке находится несколько сущностей одной Activity. Это не всегда является ошибкой, всё зависит от того, какая реализация работы программы нужна программисту. В следующих пунктах рассмотрим способы создания и контролирования Activity.

(2) Решение задачи при помощи метода startActivityForResult

Необходимо при помощи методов startActivity, startActivityForResult, setResult, onActivityResult, finish реализовать навигацию между экранами, представленными на рисунке ниже:

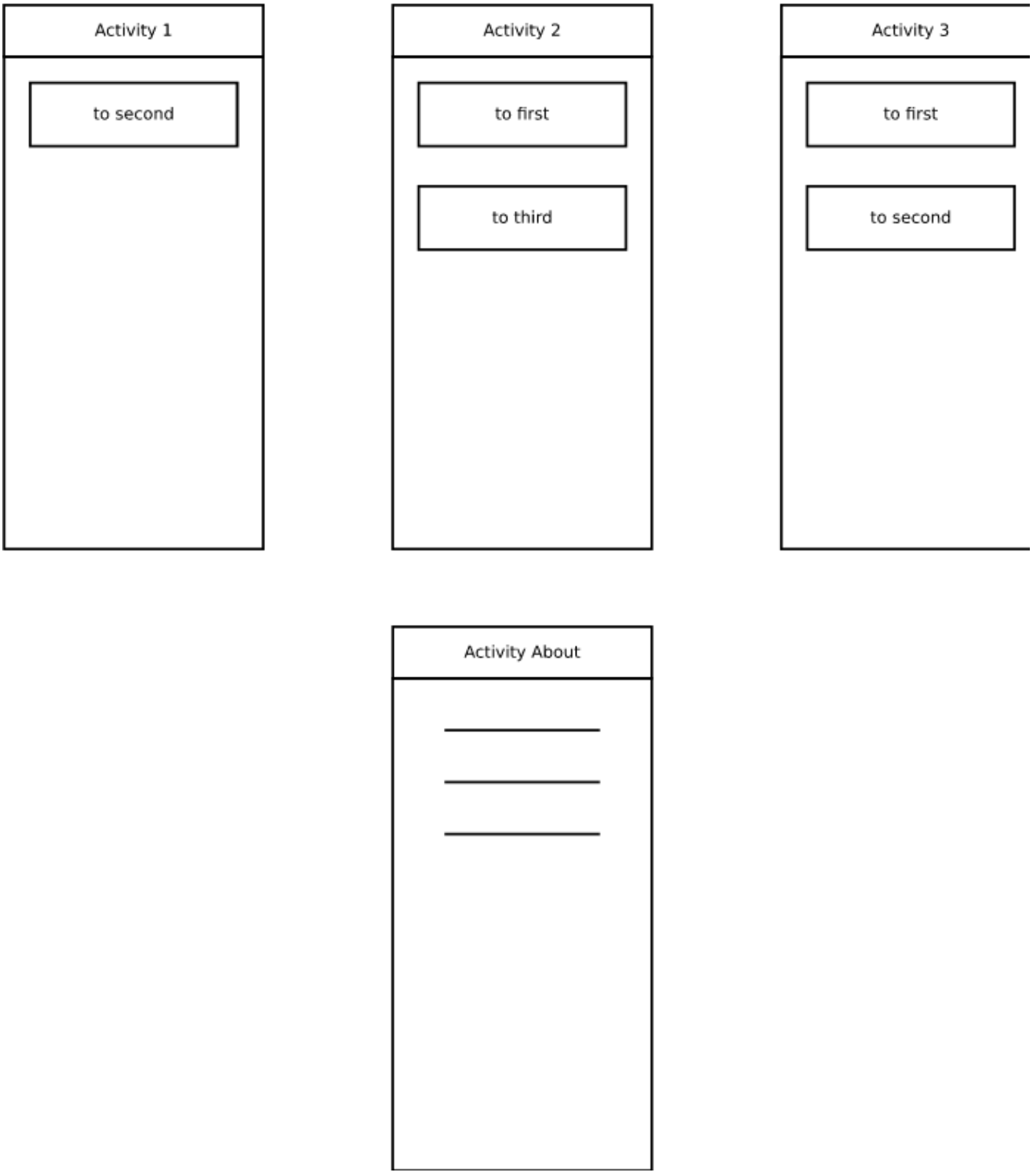


Рис. 3 Activity и возможности перехода между ними

Рассмотрим предложенные нам методы:

- startActivity(Intent) – создаёт новую Activity, которая будет положена на верхушке стэка. Имеет один аргумент Intent, который описывает в какую Activity будет совершён переход. Intent – описание операции, которая должна быть исполнена.
- startActivityForResult(Intent, int) – создаёт новую Activity с ожиданием возврата результата по заданному коду.
- onActivityResult(int, int, Intent) – данный метод обрабатывает результат работы Activity. Параметры будут объяснены в коде.
- setResult(int) – возвращает данные родителю.
- finish – завершение работы Activity.

По заданию необходимо, чтобы в backstack не было разных сущностей одной Activity. Для этого составим небольшую таблицу, в которой определим переходы.

From	To	Backstack before	Backstack after	Переход
First	Second	1	1 2	Запускаем второе Activity из первого
Second	Third	1 2	1 2 3	Запускаем третье Activity из второго
Second	First	1 2	1	Завершаем работу второго Activity
Third	First	1 2 3	1	Завершаем работу третьего и через него же завершаем работу второго
Third	Second	1 2 3	1 2	Завершаем работу третьего Activity

Теперь перейдём к программной реализации. Были созданы 3 Activity и интерфейс для них. Также по заданию дополнительно надо воспользоваться доступ к Activity ‘About’ при помощи Options Menu, что было честно реализовано в программе. Приступим к описанию переходов между экранами. В первой Activity была написана следующая функция для перехода ко второму экрану.

Листинг 3 Переход toSecond

```
private fun toSecond(){
    startActivity(Intent(this,SecondActivity::class.java))
}
```

Для навигации к About Activity был написан следующий код внутри класса Activity.

Листинг 4 Options menu

```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    val inflater: MenuInflater = menuInflater
    inflater.inflate(R.menu.options_menu, menu)
    return true
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return if (item.itemId == R.id.about_item) {
        startActivity(Intent(this,AboutActivity::class.java))
        true
    } else
        super.onOptionsItemSelected(item)
}
```

Далее опишем переходы для второго Activity.

Листинг 5 Переходы из второго Activity

```
private fun toFirst(){
    finish()
}

override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?)
{
    super.onActivityResult(requestCode, resultCode, data)
    if (requestCode == RESULT_CODE && resultCode == Activity.RESULT_OK){
        finish()
    }
}

private fun toThird(){
    startActivityForResult(Intent(this,ThirdActivity::class.java), RESULT_CODE)
}

companion object {
    const val RESULT_CODE = 0
}
```

Для перехода к первому экрану мы просто вызываем finish(), а для перехода к третьему используем startActivityForResult, т. к. из третьего нам нужно будет переходит к первому и для этого разрушать вторую Activity. Если третий экран завершиться с результатом RESULT_OK мы завершаем данную activity.

Листинг 6 Переходы из третьего Activity

```
private fun toFirst() {
    this.setResult(Activity.RESULT_OK)
    finish()
}

private fun toSecond() {
    finish()
}
```

Если переходим ко второму, то просто завершаемся, а если к первому, то проходим через второе Activity и завершаем его.

Проверим нашу реализацию. Для этого используем adb (Android Debug Bridge).

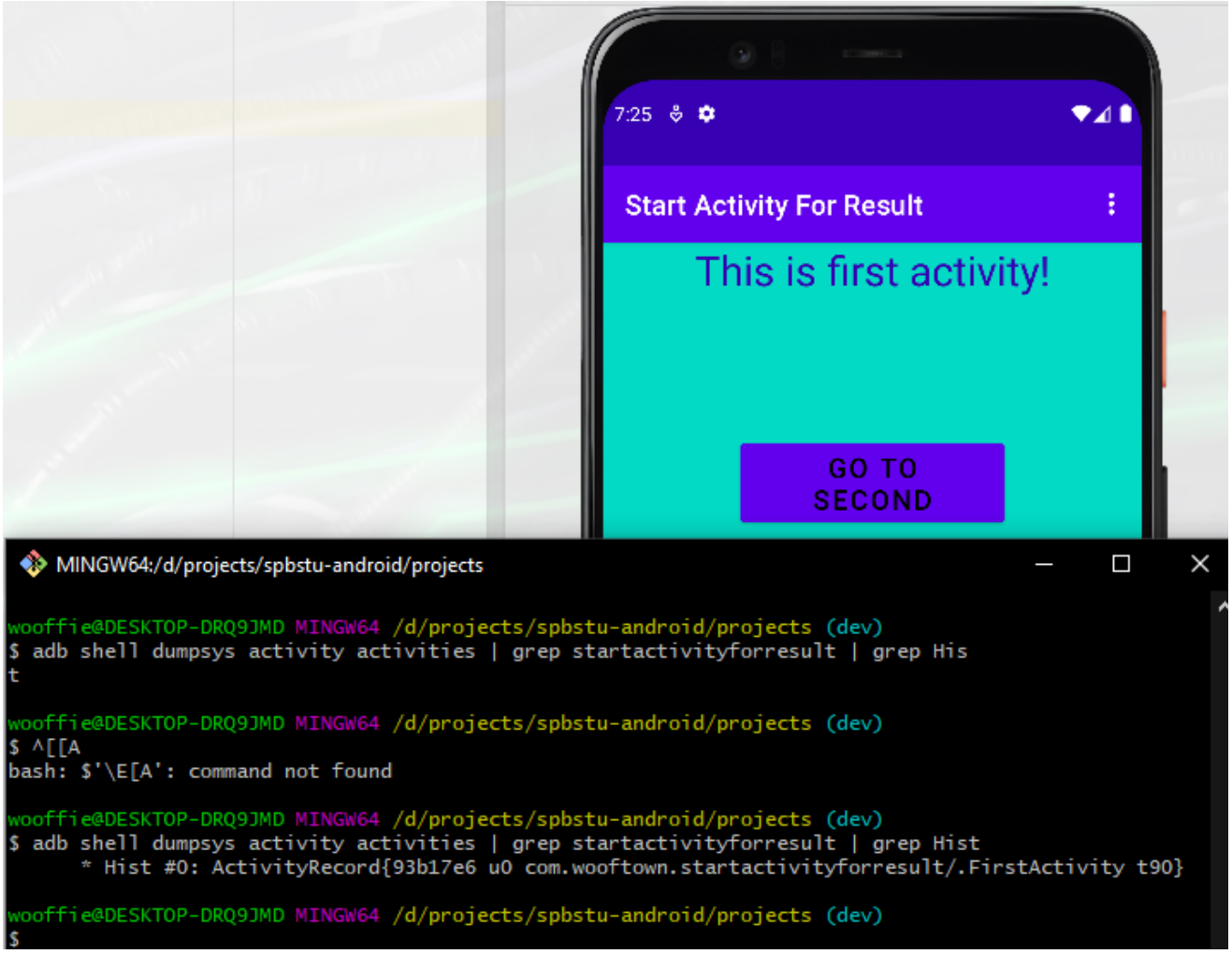


Рис. 4 Просмотр back stack

В ходе проверки программы выяснилось, что разных сущностей одной Activity в back stack не наблюдается.

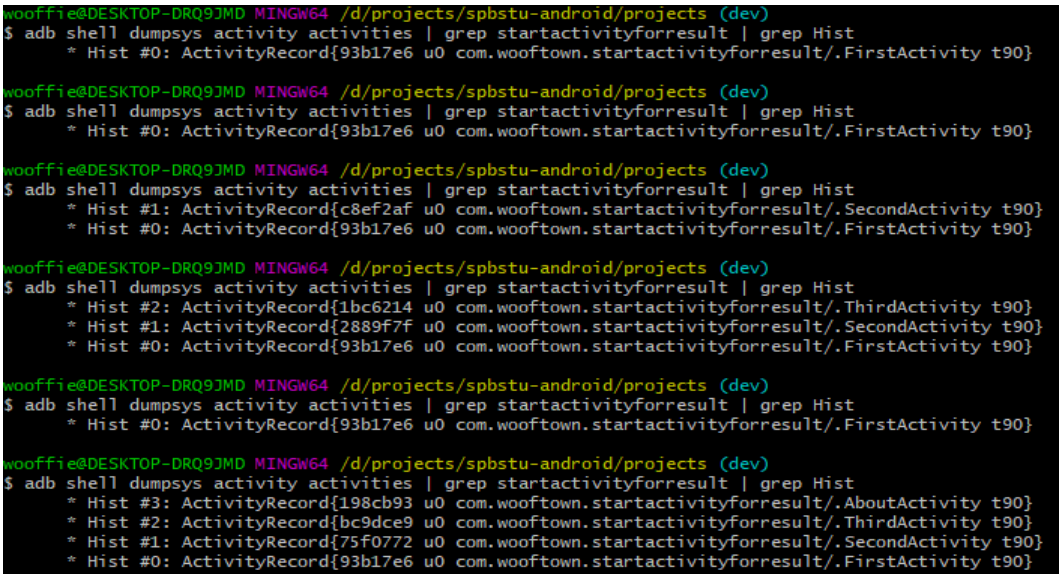


Рис.5 Процесс проверки

Вопрос: что будет если не зарегистрировать Activity в манифесте?

При переходе к данному Activity приложение будет вылетать. Это логично, ведь у ОС должен быть доступ ко всем компонентам в независимости друг от друга. Это является одним из главным принципом работы Android приложения.

(3) Решение задачи при помощи флагов Intent

Решим данную задачу другим способом. Будем использовать startActivity и дополнительные флаги для Intent. Для нас проблемным переходом является из третьего экрана к первому. Зададим флаг FLAG_ACTIVITY_CLEAR_TOP. Он позволяет переходить к Activity, если она уже лежит в стэке.

Листинг 7 Изменённые переходы

Second Activity

```
private fun toFirst() {
    finish()
}

private fun toThird() {
    startActivity(Intent(this, ThirdActivity::class.java))
}
```

Third Activity

```
private fun toFirst() {
    val intent =
Intent(this,FirstActivity::class.java).addFlags(FLAG_ACTIVITY_CLEAR_TOP)
    startActivity(intent)
}

private fun toSecond() {
    finish()
}
```

При тестировании приложение вело себя точно также, как и предыдущее. Только переход от третьего экрана к первому занимал большее количество времени.

Исходя из тестирования, в back stack не хранились дубликаты одной Activity, а когда мы переходили к другому Activity с флагом FLAG_ACTIVITY_CLEAR_TOP, то все Activity выше него по стэку закрывались.

(4) Дополнительный переход

Представим, что программист Вася решил, что при переходе в About Activity приложение будет отображать меню опций. Сделал он это, чтобы потом добавить туда другие пункты и это меню было всегда доступным.

Но так как Вася очень ленивый, то он не позаботился о том, чтобы убрать переход в Actvity, в который сейчас находится пользователь. По итогу имеем в back stack:

```
wooffie@DESKTOP-DRQ9JMD MINGW64 /d/projects/spbstu-android/projects (dev)
$ adb shell dumpsys activity activities | grep intentflagsactivityattributes | grep Hist
      * Hist #5: ActivityRecord{b21b03b u0 com.wooftown.intentflagsactivityattributes/.AboutActivity t94}
      * Hist #4: ActivityRecord{ab8f899 u0 com.wooftown.intentflagsactivityattributes/.AboutActivity t94}
      * Hist #3: ActivityRecord{8b48614 u0 com.wooftown.intentflagsactivityattributes/.AboutActivity t94}
      * Hist #2: ActivityRecord{ca87908 u0 com.wooftown.intentflagsactivityattributes/.AboutActivity t94}
      * Hist #1: ActivityRecord{45b9590 u0 com.wooftown.intentflagsactivityattributes/.AboutActivity t94}
      * Hist #0: ActivityRecord{ceb1ac8 u0 com.wooftown.intentflagsactivityattributes/.FirstActivity t94}
```

Рис. 6 Back stack при переходах из About Activity в About Activity

Вася ушёл в отпуск, а данную вещь никто не исправил. Поэтому начальство приказало Ване исправить это. А так как зарплата Вани обратно пропорционально количеству написанных строк, то он решил исправить это в одну строку, добавив нужный Intent Flag.

- FLAG_ACTIVITY_SINGLE_TOP – Activity не будет запущено, если оно уже находится на верхушке таск стэка.

Результат после добавления флага:

```
wooffie@DESKTOP-DRQ9JMD MINGW64 /d/projects/spbstu-android/projects (dev)
$ adb shell dumpsys activity activities | grep intentflagsactivityattributes | grep Hist
      * Hist #1: ActivityRecord{3d9e1ae u0 com.wooftown.intentflagsactivityattributes/.AboutActivity t95}
      * Hist #0: ActivityRecord{ca33dcb u0 com.wooftown.intentflagsactivityattributes/.FirstActivity t95}

wooffie@DESKTOP-DRQ9JMD MINGW64 /d/projects/spbstu-android/projects (dev)
$ adb shell dumpsys activity activities | grep intentflagsactivityattributes | grep Hist
      * Hist #1: ActivityRecord{3d9e1ae u0 com.wooftown.intentflagsactivityattributes/.AboutActivity t95}
      * Hist #0: ActivityRecord{ca33dcb u0 com.wooftown.intentflagsactivityattributes/.FirstActivity t95}

wooffie@DESKTOP-DRQ9JMD MINGW64 /d/projects/spbstu-android/projects (dev)
$
```

Рис. 7 Back stack для исправленной программы

(5) Решение задачи с помощью Fragments, Navigation Graph

Fragment представляет собой часть UI, которую можно пере использовать. Фрагмент определяет и управляет своим интерфейсом, имеет свой жизненный цикл и может взаимодействовать с другими частями приложения. **Фрагменты не могут жить сами по себе**, они должны быть под крылом Activity или другого фрагмента.

Для использования фрагментов необходимо указать соответствующую зависимость в build.gradle приложения.

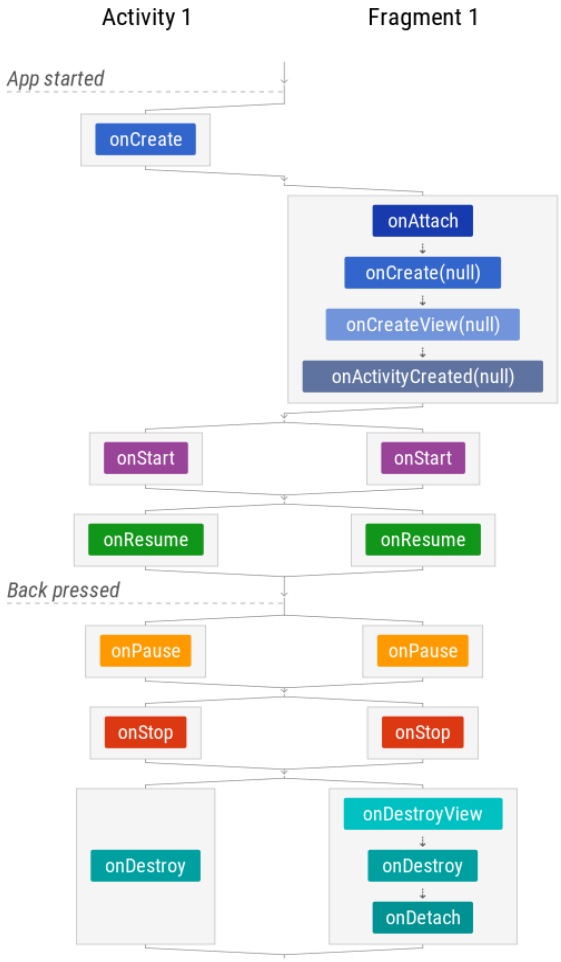


Рис. 8 Жизненный цикл фрагмента

В данном задании нам предлагается использовать Navigation Graph. Навигационный граф – это ресурс, который содержит информацию о направлениях перехода и действиях, которые необходимо выполнять при переходе. Данный граф показывает все пути навигации.

В начале необходимо создать несколько фрагментов. Наследуемся от класса Fragment и для отображения UI указываем нужный ресурс в методе onCreateView (листинги будут позже). Далее необходимо создать ресурс навигационного графа. Внутри него указывается: какие фрагменты или активити буду в ней учавствовать. С помощью перетаскивания мышки от одного фрагмента к другому указываем необходимые переходы.

Далее описывается Activity из которого будут запущены фрагменты. В ресурсе разметки этой активити необходимо добавить FragmentContainerView. FragmentContainerView – это специальный Layout для фрагментов. Наследуется от FrameLayout. После необходимо добавить обработку нажатия кнопок в фрагментах и для перехода воспользоваться Navigation.findNavController(it). navigate(R.id.action_firstFragment_to_secondFragment). Теперь наше приложение функционирует “почти” как надо. Дело в том, что у фрагментов есть свой back stack и они имеют свойства в нём засиживаться, если это не продумано программистом.

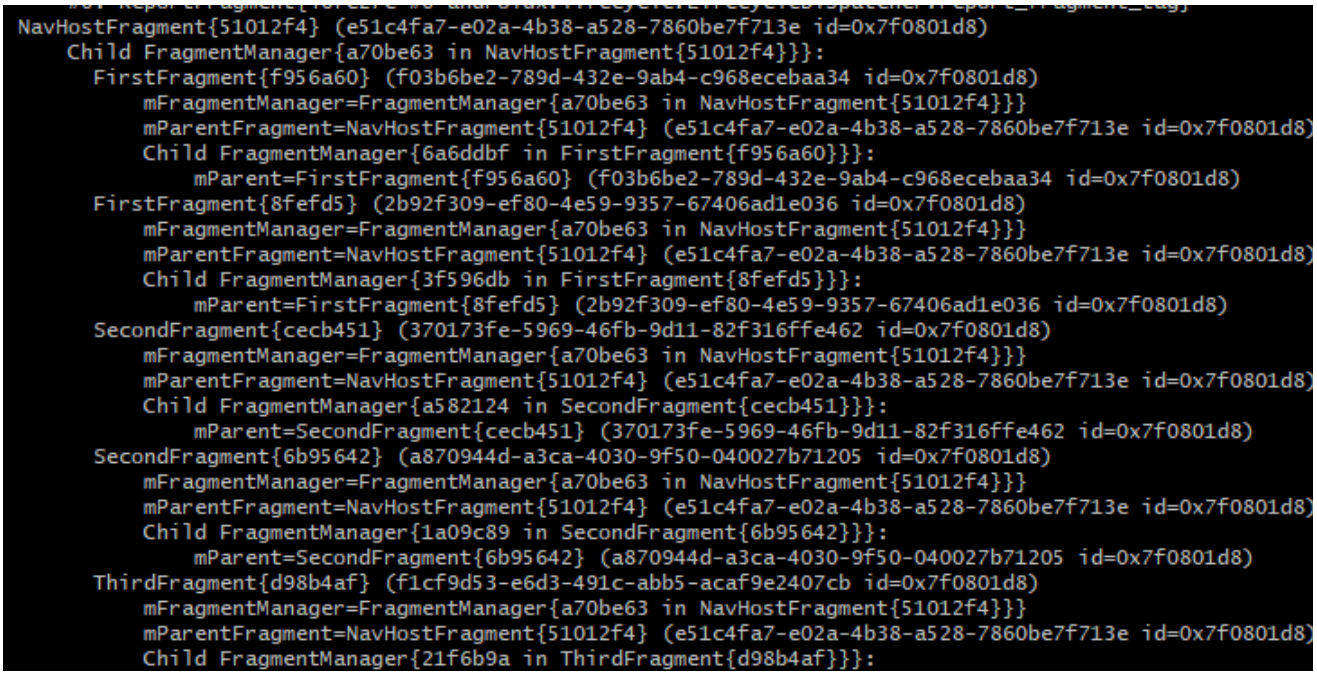


Рис. 9 Back stack фрагментов

Для избегания этого можно использовать FragmentManager. Но и в самом Navigation есть необходимый инструмент. Внутри тега action можно добавить следующие атрибуты:

- app:popUpTo – указать, до какого фрагмента необходимо вытолкнуть другие фрагменты с верхушки стэка.

- app:popUpToInclusive="true" , отмечает что надо исключить дупликаты из back stack. Иначе будет такая картина:

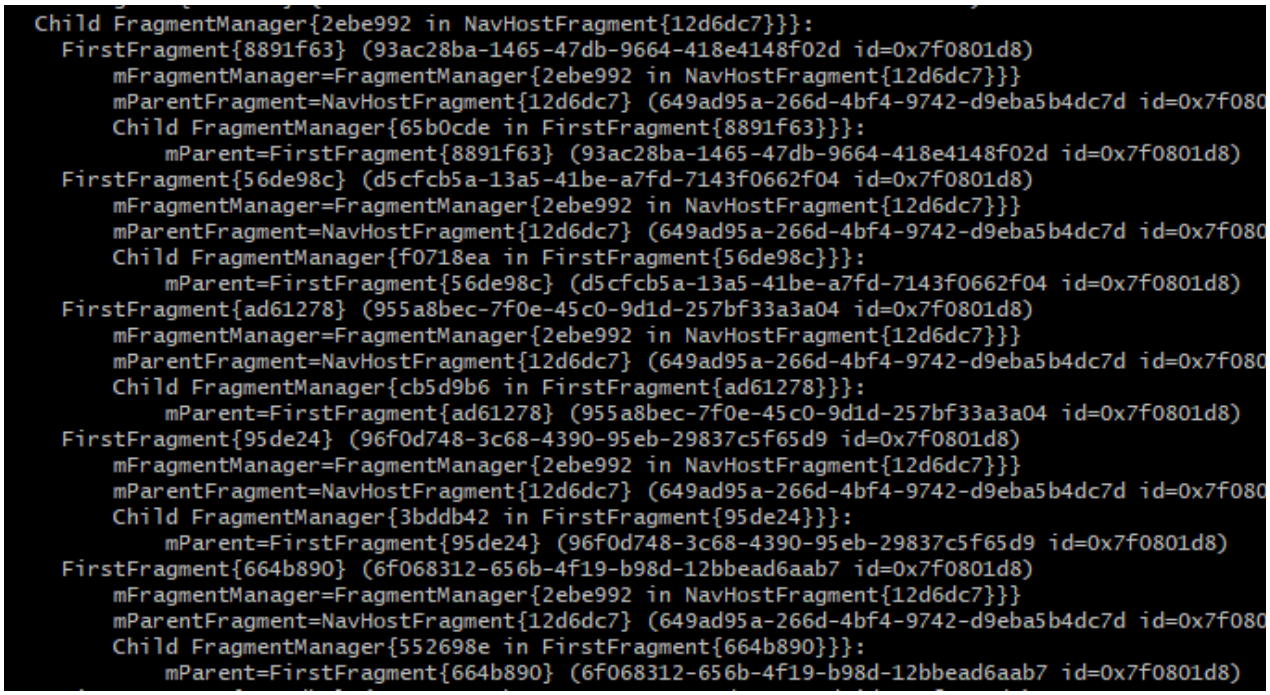


Рис. 10 app:popUpToInclusive="false”

Также навигация может быть глобальной. Она объявляется вне фрагментов и активити в ресурсе навигации, так, например выполнена навигация в About Activity:

Листинг 8 Глобальная навигация

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/firstFragment">

……
    <activity
        android:id="@+id/aboutActivity"
        android:name="com.wooftown.navigation.AboutActivity"
        android:label="AboutActivity" />
</navigation>
```

До этого момента были опущены подробности реализации классов фрагментов, самое время обратить на них внимание.

Так как из всех фрагментов можно перейти в About Activity, то было бы неплохо не копипастить один и тот же код для всех фрагментов, да и навигация глобальная и функция перехода должна быть единой для всех. В связи с этим был написан следующий класс:

Листинг 9 Класс фрагмента

```
package com.wooftown.navigation

import android.os.Bundle
import android.view.*
import android.widget.Button
import androidx.fragment.app.Fragment
import androidx.navigation.Navigation
import androidx.viewbinding.ViewBinding
import com.wooftown.navigation.databinding.FragmentThirdBinding

abstract class OptionedFragment : Fragment() {
    protected var _binding: ViewBinding? = null

    protected val binding get() = _binding!!

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        setHasOptionsMenu(true)
        _binding = FragmentThirdBinding.inflate(inflater, container, false)
        val view = binding.root
        return view
    }


    override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
        inflater.inflate(R.menu.options_menu,menu)

    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        return if (item.itemId == R.id.about_item) {

Navigation.findNavController(binding.root).navigate(R.id.global_about)
            true
        } else
            super.onOptionsItemSelected(item)
    }

}
```

Внутри него уже объявлены переменны для DataBinding и перегружены методы для отрисовки и обработки меню опций. Теперь необходимо лишь наследоваться от

него и для этого фрагмента будет доступно меню и глобальная навигация.

Листинг 10 Наследованный класс

```
package com.wooftown.navigation

import android.content.Intent
import android.os.Bundle
import android.util.Log
import android.view.*
import androidx.fragment.app.Fragment
import android.widget.Button
import androidx.navigation.Navigation
import androidx.navigation.fragment.NavHostFragment
import androidx.viewbinding.ViewBinding
import com.wooftown.navigation.databinding.FragmentFirstBinding

class FirstFragment : OptionedFragment() {

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        setHasOptionsMenu(true)
        _binding = FragmentFirstBinding.inflate(inflater, container, false)
        val view = binding.root
        view.findViewById<Button>(R.id.toSecond).setOnClickListener {

Navigation.findNavController(it).navigate(R.id.action_firstFragment_to_secondFra
gment)
        }
        return view
    }



}
```

Теперь нам необходимо только определить переходы по кнопкам, которые не являются глобальными.

Выводы (ЛР3)

В первой части работы был выполнен Codelab по Jetpack Compose. Данный формат мне понравился, я привык делать что-то маленькими шагами и параллельно узнавать различные нюансы. В ходе выполнения работы удалось сделать приложение со списком (хоть оно у меня и перестало запускаться магическим образом). Jetpack Compose показал себя как отличный инструмент для написания UI. Можно отметить его удобство и конечно же быстроту написания кода.

Во второй части было произведено знакомство с back stack’ом. Он является основой для взаимодействия активити с операционной системой. Были выполнены задания по навигации в приложении и переходам между активити. В сложившейся ситуации, когда стэк содержит дубликаты одной активти, использовался startActivityForResult и флаги Intent. Оба эти инструмента можно использовать вместе при написании более сложной логики работы с Activity (шучу ~~startActityForResult~~ депрекейтед). В последней части работы вместо Activity мы использовали Fragement, который в какой-то мере можно назвать sub-Activity. Для навигации использовался Navigation Graph, он позволяет добавить новый уровень абстракции при описании переходов между фрагментами или активити. В ходе гуглинга были найдены похожие по функционалу инструменты, которые на мой взгляд более удобные и функциональные, чем гугловкий Navigation.

Время на выполнение работы

- 1 – 150 мин (в конце лабы приложение перестало запускаться и пытался пофиксить =()
- 2 – 80 мин
- 3 – 50 мин
- 4 – 20 мин
- 5 – 230 мин
- Отчёт – 30 мин (форматирование)

Тестирование приложений

При разработке любого приложения перед программистами встаёт необходимость в проверке работоспособности каждого элемента. Для этого разрабатываются автоматические тесты для каждого модуля. Примером является фреймворк JUnit. Это очень удобный способ тестировать отдельные методы и классы с использование большого количества различных маленьких тестов.

Но если мы захотим использовать такие тесты в Android приложении, то мы не сможем использовать многие классы, также неудобно тестироваться работу с БД и другими элементами приложения. Как проверить сервис, который мы не можем запустить в вакууме? Поэтому разработчики пришли к тому, чтобы запускать тесты прямо на устройствах.

Инструментальные тесты – это тесты, работающие на реальном устройстве, которые благодаря этому могут использовать все классы и методы системы Android. Они нужны в первую очередь для модульных тестов, когда для тестов требуется использование каких-либо классов Android. Сюда относится тестирование работы с базой данных, с SharedPreferences, с Context и другими классами.

Чтобы начать их использовать необходимо в скрипте сборки указать “запускатор тестов”:

```
android {
    defaultConfig {
  testInstrumentationRunner"android.support.test.runner.AndroidJUnitRunner"
}
}
```

Теперь у нас появилась возможность создавать тестовые классы.

Но приложения также надо испытывать с точки зрения того, чем может пользоваться обычный юзер. Поэтому необходимы UI-тесты, которые проверяют работу элементов через внешние элементы. Наиболее популярным фреймворком является Espresso.

Перечень тестов

Не будем вдаваться в тонкие подробности происходящего и опишем главные детали.

```
import androidx.test.espresso.Espresso.onView
import androidx.test.espresso.action.ViewActions.click
import androidx.test.espresso.assertion.ViewAssertions.matches
import androidx.test.espresso.matcher.ViewMatchers.isDisplayed
import androidx.test.espresso.matcher.ViewMatchers.withId

fun assertViewWithId(viewId: Int) {
    onView(withId(viewId))
        .check(matches(isDisplayed()))
}

fun performClickWithId(viewId: Int) {
    onView(withId(viewId)).perform(click())
}
```

Рис. 11 Вспомогательные функции

Для нахождения объекта на экране необходимо объявить Matcher (witdId) и передать его в onView, этим самым мы нашли нужный элемент. После можно выполнять с ним разные действия. Проверить что он отображается на экране, или нажать на кнопку.

Свои тесты я разделил на несколько типов:

- NavigationTest – проверяет переходы между экранами
- RecreateTest – тестирует как ведёт себя приложения при пересоздании активти
- BackstackTest – проверяет чтобы в стеке не было дубликатов
- AboutTest – проверяет переходы в AboutActivity

Мои тесты прошли 5 из 6 проверок на разных проектах, и я довольный перешёл к интеграции этих тестов в свои проекты.

Первое приложение

По заданию необходимо было просто “вставить” тесты в приложение и с помощью рефакторингов id в статическом классе R сделать так, чтобы они работали корректно. Но проблема в том, что у меня не была реализована кнопка NavigateUp, поэтому тестирование каждого приложения я начинал с добавления кнопки.

В случае с Activity это было сделать очень просто. Необходимо было лишь в методе onCreate разрешить отображения этой самой кнопки, а в манифесте отразить отношения между Activity. Так как задание было довольно простым, то эта задача решалась тривиально. В AboutActivity в методе обработке нажатия этой кнопки вызываем метод finish().

Теперь в тестах можно использовать эту кнопку. **Все тесты выполнились без неудач, следовательно приложение работает корректно с моей точки зрения.**

Второе приложение

Так как приложение никак не отличается, кроме подробностей реализации переходов между Activity, то точно так же, как и в первом добавляем кнопки и тестируем. Все тесты также выполнились без неудач.

Третье приложение

В данном приложении использовались фрагменты, поэтому реализация навигации “вверх” чуть другая.

Была обнаружена ошибка, в реализации меню было определено через фрагменты. Поэтому меню было перенесено в главное Activity. Также, так как используется навигационный граф, то следует выполнять открытие чуть по-другому.

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return if (item.itemId == R.id.about_item) {
        Navigation.findNavController(binding.root).navigate(R.id.global_about)
        true
    } else
        super.onOptionsItemSelected(item)
}
```

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return NavigationUI.onNavDestinationSelected(
        item, findNavController(R.id.fragmentContainerView)
    ) || super.onOptionsItemSelected(item)
}
```

Рис. 12 Было/стало

Для этого необходимо было синхронизировать id пункта меню с id в графе. После для подключения кнопки navigateUp в методе onCreate был написан следующий код:

```
    val navHostFragment = supportFragmentManager
        .findFragmentById(R.id.fragmentContainerView) as NavHostFragment
    navController = navHostFragment.navController

    appBarConfiguration = AppBarConfiguration.Builder(navController.graph).build()

    setupActionBarWithNavController(
        navController,
        appBarConfiguration
    )

}

override fun onSupportNavigateUp(): Boolean {
    return navController.navigateUp(appBarConfiguration) || super.onSupportNavigateUp()
}
```

Рис. 13 Создание навигации “вверх”

Теперь, когда имеется необходимая кнопка можно запускать тесты. И тут нас ждёт **неудача**. Дело в том, чтоб при закрытии AboutActivity через кнопку навигации “вверх” создавалась совершенно новая сущность активити, и мы переходили к первому фрагменту. Для исправления данной ситуации стоит заглянуть в манифест и выбрать launchMode в режиме SingleTop. После данного исправления можно наслаждаться прохождением всех тестов.

Выводы

В этой части работы было произведено знакомство с UI-тестами. Это очень мощный инструментарий при разработке приложений. С помощью его было выполнено тестирование рабочего приложения, а позже эти тесты были применены к неправильным версиям.

Во второй части необходимо было произвести рефакторинг и завезти эти тесты в мои приложения. Для этого было необходимо было сначала добавить кнопку NavigateUp. После исправление ошибки в приложении 3 (которая случилась из-за добавления навигации “вверх”) можно сказать, что все созданные тесты проходятся на моих приложениях. Это отличный способ проверить их, не используя ручные тесты. Некоторые тесты падали из-за долгой анимации переходов, чтобы избежать этого необходимо их выключать.

Время на выполнение работы

- 1 – 120 мин (готовое приложение было в репозитории Espresso, так что почти всё время это изучение справочных материалов)
- 2 – 180 мин
- 3 – 120 мин (время рефакторинга и добавления NavigateUp, дай Бог здоровья Тимофею)
- 4 – 30 мин
- Отчёт – 30 мин (форматирование)

Список источников

- https://developer.android.com
- https://github.com/andrei-kuznetsov/android-lectures
- https://github.com/wooftown/spbstu-android - листинги