GOCE DELCEV UNIVERSITY - STIP
FACULTY OF COMPUTER SCIENCE

The journal is indexed in
EBSCO

ISSN 2545-4803 on line
DOI: 10.46763/BJAMI

BALKAN JOURNAL OF APPLIED MATHEMATICS AND INFORMATICS (BJAMI)

YEAR 2023 VOLUME VI, Number 2

AIMS AND SCOPE:
BJAMI publishes original research articles in the areas of applied mathematics and informatics.

Topics:
1. Computer science;
2. Computer and software engineering;
3. Information technology;
4. Computer security;
5. Electrical engineering;
6. Telecommunication;
7. Mathematics and its applications;
8. Articles of interdisciplinary of computer and information sciences with education, economics, environmental, health, and engineering.

Managing editor
Mirjana Kocaleva Vitanova Ph.D.
Zoran Zlatev Ph.D.

Editor in chief
Biljana Zlatanovska Ph.D.

Lectoure
Snezana Kirova

Technical editor
Biljana Zlatanovska Ph.D.
Mirjana Kocaleva Vitanova Ph.D.

BALKAN JOURNAL
OF APPLIED MATHEMATICS AND INFORMATICS
(BJAMI), Vol 6

ISSN 2545-4803 on line
Vol. 6, No. 2, Year 2023

EDITORIAL BOARD

Adelina Plamenova Aleksieva-Petrova, Technical University – Sofia,
Faculty of Computer Systems and Control, Sofia, Bulgaria
Lyudmila Stoyanova, Technical University - Sofia , Faculty of computer systems and control,
Department – Programming and computer technologies, Bulgaria
Zlatko Georgiev Varbanov, Department of Mathematics and Informatics,
Veliko Tarnovo University, Bulgaria
Snezana Scepanovic, Faculty for Information Technology,
University "Mediterranean", Podgorica, Montenegro
Daniela Veleva Minkovska, Faculty of Computer Systems and Technologies,
Technical University, Sofia, Bulgaria
Stefka Hristova Bouyuklieva, Department of Algebra and Geometry,
Faculty of Mathematics and Informatics, Veliko Tarnovo University, Bulgaria
Vesselin Velichkov, University of Luxembourg, Faculty of Sciences,
Technology and Communication (FSTC), Luxembourg
Isabel Maria Baltazar Simões de Carvalho, Instituto Superior Técnico,
Technical University of Lisbon, Portugal
Predrag S. Stanimirović, University of Niš, Faculty of Sciences and Mathematics,
Department of Mathematics and Informatics, Niš, Serbia
Shcherbacov Victor, Institute of Mathematics and Computer Science,
Academy of Sciences of Moldova, Moldova
Pedro Ricardo Morais Inácio, Department of Computer Science,
Universidade da Beira Interior, Portugal
Georgi Tuparov, Technical University of Sofia Bulgaria
Martin Lukarevski, Faculty of Computer Science, UGD, Republic of North Macedonia
Ivanka Georgieva, South-West University, Blagoevgrad, Bulgaria
Georgi Stojanov, Computer Science, Mathematics, and Environmental Science Department
The American University of Paris, France
Iliya Guerguiev Bouyukliev, Institute of Mathematics and Informatics,
Bulgarian Academy of Sciences, Bulgaria
Riste Škrekovski, FAMNIT, University of Primorska, Koper, Slovenia
Stela Zhelezova, Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, Bulgaria
Katerina Taskova, Computational Biology and Data Mining Group,
Faculty of Biology, Johannes Gutenberg-Universität Mainz (JGU), Mainz, Germany.
Dragana Glušac, Tehnical Faculty "Mihajlo Pupin", Zrenjanin, Serbia
Cveta Martinovska-Bande, Faculty of Computer Science, UGD, Republic of North Macedonia
Blagoj Delipetrov, European Commission Joint Research Centre, Italy
Zoran Zdravev, Faculty of Computer Science, UGD, Republic of North Macedonia
Aleksandra Mileva, Faculty of Computer Science, UGD, Republic of North Macedonia
Igor Stojanovik, Faculty of Computer Science, UGD, Republic of North Macedonia
Saso Koceski, Faculty of Computer Science, UGD, Republic of North Macedonia
Natasa Koceska, Faculty of Computer Science, UGD, Republic of North Macedonia
Aleksandar Krstev, Faculty of Computer Science, UGD, Republic of North Macedonia
Biljana Zlatanovska, Faculty of Computer Science, UGD, Republic of North Macedonia
Natasa Stojkovik, Faculty of Computer Science, UGD, Republic of North Macedonia
Done Stojanov, Faculty of Computer Science, UGD, Republic of North Macedonia
Limonka Koceva Lazarova, Faculty of Computer Science, UGD, Republic of North Macedonia
Tatjana Atanasova Pacemska, Faculty of Computer Science, UGD, Republic of North Macedonia

CONTENT

Sonja Manchevska, Igor Peshevski, Daniel Velinov, Milorad Jovanovski, Marija Maneva, Bojana Nedelkovska
APPLICATION OF GEOSTATISTICS IN THE ANALYSIS AND ADAPTATION OF GEOTECHNICAL PARAMETERS AT COAL DEPOSITS……………………………………….. 7

Darko Bogatinov, Saso Gelev
PROGRAMMING APLC CONTROLLER WITH A LADDER DIAGRAM..…………………… 19

Dalibor Serafimovski, Stojce Recanoski, Aleksandar Krstev, Marija Serafimovska
ANALYSIS OF THE USAGE OF MOBILE DEVICES AS DISTRIBUTED TOOLS FOR PATIENT HEALTH MONITORING AND REMOTE PATIENT DATA ACQUISITION……..... 31

Sasko Dimitrov, Dennis Weiler, Simeon Petrov
RESEARCH ON THE INFLUENCE OF THE VOLUME OF OIL IN FRONT OF THE DIRECT OPERATED PRESSURE RELIEF VALVE ON ITS TRANSIENT PERFORMANCES ……………………………………………………………………………43

Violeta Krcheva, Marija Cekerovska, Mishko Djidrov, Sasko Dimitrov
IMPACT OF CUTTING CONDITIONS ON THE LOAD ON SERVO MOTORSAT A CNC LATHE IN THE PROCESS OF TURNING A CLUTCH HUB……………...…………………... 51

Samoil Malcheski
REICH-TYPE CONTRACTIVE MAPPING INTO A COMPLETE METRIC SPACE AND CONTINUOUS, INJECTIVE AND SUBSEQUENTIALLY CONVERGENT MAPPING…..….. 63

Violeta Krcheva, Mishko Djidrov, Sara Srebrenoska, Dejan Krstev
GANTT CHART AS A PROJECT MANAGEMENT TOOL THAT REPRESENTS A CLUTCH HUB MANUFACTURING PROCESS……………………………………………………….….. 67

Tanja Stefanova, Zoran Zdravev, Aleksandar Velinov
ANALYSIS OF TOP SELLING PRODUCTS USING BUSINESS INTELLIGENCE…… …... 79

Day of Differential Equations
THE APPENDIX…………………………………………………………...……………..….. 91

Slagjana Brsakoska, Aleksa Malcheski
ONE APPROACH TO THE ITERATIONS OF THE VEKUA EQUATION …………………….. 93

Saso Koceski, Natasa Koceska, Limonka Koceva Lazarova, Marija Miteva, Biljana Zlatanovska
CAN CHATGPT BE USED FOR SOLVING ORDINARY DIFFERENTIAL EQUATIONS ... 103

Natasha Stojkovic, Maja Kukuseva Paneva, Aleksandra Stojanova Ilievska, Cveta Martinovska Bande
SEIR+D MODEL OF TUBERCULOSIS …………………………………………………..... 115

Jasmina Veta Buralieva, Maja Kukuseva Paneva
APPLICATION OF THE LAPLACE TRANSFORM IN ELECTRICAL CIRCUITS ...……..... 125

Biljana Zlatanovska, Boro Piperevski
ABOUT A CLASS OF 2D MATRIX OF DIFFERENTIAL EQUATIONS ………....……... 135

ETIMA
THE APPENDIX……………………………………………………………..……………147

Bunjamin Xhaferi, Nusret Xhaferi, Sonja Rogoleva Gjurovska, Gordana J. Atanasovski
BIOTECHNOLOGICAL PEOCEDURE FOR AN AUTOLOGOUS DENTIN GRAFT FOR DENTAL AND MEDICAL PURPOSES………………………………..………………..149

Mladen Mitkovski, Vlatko Chingoski
COMPARATIVE ANALYSIS BETWEEN BIFACIAL AND MONOFACIAL SOLAR PANELS USING PV*SOL SOFTWARE………………………………..………………………… 155

Egzon Milla, Milutin Radonjić
ANALYSIS OF DEVELOPING NATIVE ANDROID APPLICATIONS USING XML AND JETPACK COMPOSE………………………………..…………………………………. 167

Sonja Rogoleva Gjurovska, Sanja Naskova, Verica Toneva Stojmenova, Ljupka Arsovski, Sandra Atanasova
TRANSCUTANEOUS ELECTRICAL NERVE STIMULATION METHOD IN PATIENTS WITH XEROSTOMIA………………………………..…………………………………. 179

Marjan Zafirovski, Dimitar Bogatinov
COMPARATIVE ANALYSIS OF STANDARDS AND METHODOLOGIES FOR MANAGE-MENT OF INFORMATION-SECURITY RISKS OF TECHNICAL AND ELECTRONIC SYS-TEMS OF THE CRITICAL INFRASTRUCTURE ………………………………………….. 187

The Appendix

The Faculty of Electrical Engineering at Goce Delcev University (UGD), has organized the Second International Conference Electrical Engineering, Informatics, Machinery and Automation - Technical Sciences Applied in Economy, Education and Industry-ETIMA on September, 27th-29th 2023.

ETIMA has a goal to gather scientists, professors, experts, and professionals from the field of technical sciences in one place as a forum for exchanging ideas, strengthening multidisciplinary research and cooperation, and promoting the achievements of technology and its impact on every aspect of living. Conference ETIMA was held as an online conference. More than sixty colleagues contributed to this event, from five different countries with more than thirty papers.

The Organizing Committee selected five papers that will be published in this number of the BJAMI.

Balkan Journal of Applied Mathematics and Informatics Online ISSN 2545-4083
Volume VI Number 2 Year 2023 UDC: 004.451:621.395.721.5]:004.439

ANALYSIS OF DEVELOPING NATIVE ANDROID APPLICATIONS USING XML AND JETPACK COMPOSE

EGZON MILLA AND MILUTIN RADONJIĆ

Abstract. Native mobile applications have been the first choice for companies and independent developers to build software that utilizes device hardware and functionality. For Android development, this means using Java/Kotlin for the program logic and XML files for the user interfaces. Recently, Jetpack Compose has gained significant attention in building user interfaces for Android applications. This paper aims to analyze the benefits and limitations of these two technologies. An intensive literature review is conducted and presented to identify the benefits and limitations of XML and Jetpack Compose. In order to compare their development processes and code complexity, we also developed the same application using both technologies. The analysis of the code used in each technology provides an insight into how they uniquely solve the same problem, helping us to identify which one is better suited from a developer's perspective. The results of the study indicate that both XML and Jetpack Compose have their respective strengths and weaknesses. XML provides a structured approach to UI development, fully respecting the separation of concerns between the view (what is being displayed on the screen) and the controller (which data is being sent to the view). On the other hand, Jetpack Compose simplifies the UI development process by offering a declarative approach, which leads to a more readable and maintainable code. This study identified the advantages and disadvantages of using XML and Jetpack Compose in the development of native Android applications and recommended criteria for choosing one of these tools for new users, as well as switching to a new tool for experienced users.

1. Introduction

The Android operating system is a distribution of Linux, which is used in more than 70% of mobile devices around the world, with the first device running it being released in 2008 [9]. Its open-source nature quickly made it one of the most used operating systems. Historically, Java is the most widely used programming language for Android [7], although Kotlin with its less verbose syntax and "code safety" feature, by which the compiler protects the developer from variables that can potentially point to a non-existing (null) object [5], is becoming increasingly more popular.

With Kotlin, there are two ways to build the user interface (UI). The first way is standard and the most popular, through the use of XML files. In [3], it is stated that the key components of this approach are fragments. The authors explain that fragments are comprised of two files: the layout file written in XML, where the developer defines the screen design (width, height, position of components), and what components should be on it (some text, buttons, images, etc.). The other file is a Kotlin file where the developer defines the behavior of the screen and its components. An XML – based project contains the MainActivity.kt file and its corresponding XML layout file. Every application screen is created through the XML files using Fragments [3]. As stated in [6], the UI is built

Keywords. Android, Compose, XML, comparison, analysis, code.

through the hierarchy of layouts and components. The layouts are represented as ViewGroup objects, while the components (widgets) are represented as View objects. The components defined within the layout files are accessed via view binding, which is the recommended method for accessing the components from the fragment class, according to [4].

The second way to build the user interface is by creating composable functions using a UI library named Jetpack Compose [6]. These functions serve as building blocks for screens the user sees in the application. A Jetpack Compose project also contains the MainActivity.kt file but displays application content through the composable functions [6]. Composable functions are annotated with the @Composable annotation above the function name. The authors of [1] state that @Composable annotation is used to let the Compose compiler know that the function intends to convert data into a node of the composable tree. They explain that a node represents a component or a screen. In the Jetpack Compose project, every application screen and component is represented by a composable function in the code.

The primary goal of this paper is to draw a comparison between the two methods and point out the advantages and disadvantages of both, highlighting their respective complexity (or simplicity) by reviewing specific parts of the application and the corresponding code. We performed this by developing the same application twice, using the aforementioned approaches. The motivation for this study is to provide insight into the developer experience (DX) of both methods, as each requires a different way of approaching and reasoning of the problem we are solving.

The paper is organized as follows. In Section 2, the general Android application architecture is presented and applied accordingly to the applications that we develop. Section 3 compares the used methodologies with the following criteria: code readability, complexity, maintainability, and the amount of code required, and it also highlights potential strengths and weaknesses for each specific application functionality. Finally, we provide a general comparison of both methodologies based on our research and developed functionalities, as well as our recommended technology.

2. Material and method

For this study we consulted 9 published books on android technology and development published in the last 10 years that highlight important aspects of the development process, as well as the use of the corresponding components depending on the type of technology used. The key words that were used for this research were the following: Android, Compose, XML, comparison, analysis, code, along with combinations of each. The search included: internet articles on the comparison of the methodologies and books regarding each technology separately. This study was made online by selecting the articles that contained the research key words, then the books that met the needed criteria were selected and analyzed in detail. According to the selected material, we implemented the development of the same application in both XML and Jetpack Compose to gain further insight into the development process and the respective strengths and weaknesses of each technology.

3. The proposed Android application architecture

The application we built for the purpose of this research is named 'Oss'. The idea for the application development was born while trying to solve the problem of journaling the countless techniques learned in the sport of Brazilian Jiu-Jitsu. 'Oss' is a CRUD (Create, Read, Update, Delete) application that has two entities: Position and Technique, with their respective properties. The entities share a One-to-Many relationship, where one position can have many techniques. In both versions of the application, we used the MVVM (Model View View Model) architecture [2]. Figure 1 shows a simplified graph of how the application layers function by using the PositionList screen as an example. As can be noticed from Figure 1, there are three layers in this case:

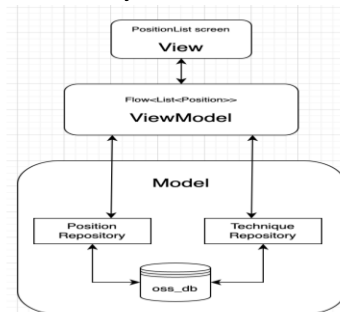

Figure 1. *Application layers for the PositionList screen*

- The data layer (the Model) defines where and how data is stored. In this case, we use the repository pattern to access and modify the data from a database [2].
- The ViewModel is what accesses the data and acts as a link between what appears on the screen and the data. Every CRUD operation is controlled by the ViewModel [2].
- The View is what the user sees [2]. Through the View, the user sends events to the ViewModel. The event can be anything, from clicking some button, or selecting some choice, to performing more complex operations.

4. The Application User Interface

In this section, we will highlight some of the key differences between an application design using XML and an application design using Jetpack Compose. A commonality is that both applications are run inside a single activity, and the screens change accordingly. These screens are fragments in the XML version and composable functions in Jetpack Compose. A fragment is a user interface component that has a Kotlin class that controls its behavior, and a layout in the form of an XML file [4]. The fragments that we create inherit from the Fragment.kt class, which provides our custom fragment with predefined lifecycle methods, user input handlers, and so on. As mentioned in the introduction, in Compose, a composable function is a Kotlin function that represents a component or a screen.

We will present these specific parts of the developed application and their creation in both methodologies:

- The navigation - navigates the screens of the application,
- The bottom navigation bar,

- The PositionList screen - the screen that displays a list of positions, with an image, name, and delete button for each position.

4.1. Developing the navigation

Using XML, we have to create a navigation file that holds the navigation graph (shown in Figure 2). The navigation graph contains fragments, actions, and arguments. The fragments are the destination screens, the screens can receive arguments, and the arrows represent the navigation actions.

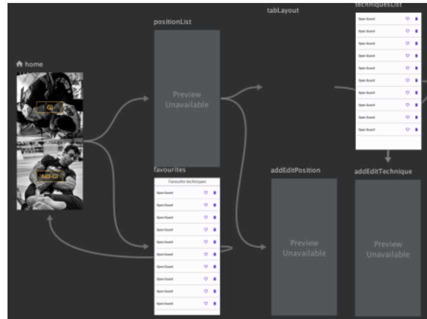

Figure 2. *Navigation graph in design mode*

The default destination is the Home screen (Figure 3), which has an action that navigates the user to the positionList screen named “toPositionListScreen” (shown in Code 1 below).

Code 1. XML for navigation graph

```
<navigation // attributes...
 android:id="@+id/navigation" // id of the navigation graph
app:startDestination="@id/home"> // set the starting destination
  <fragment
    android:id="@+id/home" // id of the destination
    android:name="com.example.oss_xml.ui.home.Home" // auto-generated destination name
    android:label="fragment_home" // label
    tools:layout="@layout/fragment_home"> //used for previewing the destination in Figure 2Figure
1
    <action
      android:id="@+id/toPositionListScreen" // id of the action
      app:destination="@id/positionList"/> //the destination where the action should lead to
  </fragment>
<fragment // positionList fragment attributes... >
  <argument
    android:name="discipline" // received argument name
    app:argType="string" // received argument type
    android:defaultValue="GI" /> // set an optional default value in case no argument is received
```

In activity_main.xml, we define a fragment that hosts our navigation graph (shown in Code 2):

Code 2. Navigation host in activity_main.xml

```
<fragment
    // set width and height here...
   android:id="@+id/fragmentContainerView" // id of the navigation host fragment
   android:name="androidx.navigation.fragment.NavHostFragment"
   app:defaultNavHost="true" // set the default navigation host of the application in case there are more
   app:navGraph="@navigation/navigation" /> // use the navigation graph that we created earlier
```

In [4], it is explained that the navigation host acts as an empty container in which the different destinations can be displayed. Code 3 shows the use of view binding and the navController.

Code 3. View binding and navController in MainActivity.kt

```
class MainActivity : AppCompatActivity() { // below we create binding properties
   private var _binding: ActivityMainBinding? = null
   private val binding get() = _binding!!
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      _binding = ActivityMainBinding.inflate(layoutInflater)   // convert XML into kotlin objects
(inflating)
      setContentView(binding.root) // set the activity content to the root (container of the components)
```

ActivityMainBinding is a generated class that contains references to all the components in activity_main.xml. View binding ensures that we cannot reference a non-existing component and it provides a more efficient way of accessing them [4]. In the Home fragment, we set an onClickListener for navigating to the PositionList fragment, shown in Code 4:

Code 4. Navigation using an onClickListener

```
binding.gi.setOnClickListener{val action = HomeDirections.toPositionListScreen(Discipline.GI)
   Navigation.findNavController(view).navigate(action)}
```

HomeDirections is a class generated by the navigation component that contains the actions defined in the navigation graph. In the action, we set the navigation argument. In the “PositionList” fragment we can access the “discipline” argument in order to list the positions in a specific discipline and display the text (shown in Code 5 and Code 6 respectively).

Code 5. Access navigation arguments in PositionListFragment.kt

```
val navController = rememberNavController()
NavHost(navController = navController, startDestination = Routes.HOME_SCREEN) { //screens }
```

Code 6. Display text based on the received argument

```
val discipline = args.discipline
binding.positionListText.text = "$discipline POSITIONS"
```

In Jetpack Compose, in ActivityMain.kt we create the navController. As stated in [8], the navController handles navigation between screens and manages the navigation stack. In simple terms, the navigation stack represents the memory of the navigation (what screen is the current one, what screen came before it, and so on until the starting (Home) screen of the application). Then we create the navigation host with the NavHost composable function. The NavHost is a component that is added to the user interface of an activity and serves as a placeholder for displaying the different application screens [8]. The required code is shown in Code 7.

Code 7. NavController and NavHost in Jetpack Compose

```
val navController = rememberNavController()
NavHost(navController =  navController, startDestination = Routes.HOME_SCREEN) { //screens }
```

We can define the screens that can be navigated to with a route, shown in Code 8.

Code 8. Routes for navigation in Jetpack Compose

```
const val HOME_SCREEN = "home_screen" // in Routes.kt file
```

Arguments can be passed by adding a query/path parameter to the route and defining each navigation argument with the name, type, and default value, after which we call the composable function [2]. An example of navigating using provided arguments is presented in Code 9.

Code 9. Navigation using arguments in Jetpack Compose

```
composable(route=Routes.POSITION_LIST+"?discipline={discipline}", // value in curly braces
  arguments = listOf( // defined list of arguments
    navArgument(name = "discipline") {type = NavType.StringType, defaultValue = Discipline.GI})){
  PositionListScreen(onPopBackStack = { navController.popBackStack() }, // to previous screen
    onNavigate = { navController.navigate(it.route) })} // navigate using the navController
```

Navigation arguments are available in the corresponding ViewModel via SavedStateHandle, as presented in Code 10.

Code 10. PositionListViewModel.kt

```
var discipline = savedStateHandle.get<String>("discipline")!! // get the “discpline” argument
```

As can be noticed from the presented application code, creating navigation in XML requires a significant number of steps that lead to more files, and more code: creating the navigation graph in Code 1, the navigation host shown in Code 2, defining the actions and arguments for each destination also shown in Code 1, binding the navHost and navigating using onClickListener, shown in Code 3 and Code 4 respectively. A strength of this approach can be the provided design tool for the navigation graph by Android Studio, which makes it easy to visualize how the screens lead to one another.

In Compose, we define the routes for each screen in a Kotlin file as shown in Code 8 and create the navController and the navHost according to Code 7. Inside of the navHost

function's body, we call the composable functions (the screens we created) for navigation (shown in Code 9) and provide the routes and potential navigation arguments. A weakness of this approach can be that the navigation code can become somewhat less readable in the case of many screens. From the provided code, we can conclude that creating navigation is simpler and requires fewer files in Jetpack Compose compared to the XML-based approach, improving maintainability.

4.2. Developing the bottom navigation bar

To create a bottom navigation bar in XML, we first create a menu layout where the menu items are defined (shown in Code 11). In our case, we create a Home and Favorites item.

Code 11. Menu items in XML

```
<menu // menu attributes >
  <item android:id="@+id/home" // item id
    android:icon="@drawable/ic_baseline_home_24" // we add an icon to the menu item
    app:showAsAction="always" // when the item should be visible />
    <item // similarly for the Favourites item />
```

In activity_main.xml, we add a BottomNavigationView and set the menu (shown in Code 12).

Code 12. BottomNavigationView component

```
<com.google.android.material.bottomnavigation.BottomNavigationView
android:id="@+id/bottomNavigationView" // component id
  app:menu="@menu/bottom_app_bar" /> // here we set the menu that we created
```

In MainActivity.kt, we bind the view and set an item listener on it, which handles the navigation between Home and Favourites. The code is shown in Code 13.

Code 13. Bottom bar navigation

```
val bottomNavigationView = binding.bottomNavigationView // get the component by view binding
bottomNavigationView.setOnItemSelectedListener {
  when (it.itemId) {
    R.id.home -> navController.navigate(R.id.home) // when Home is selected, navigate to home screen
    R.id.favourites -> navController.navigate(R.id.favourites) }true}// similarly for Favourites
```

For the bottom navigation bar in Compose, we create a sealed class that defines the properties of a bottom bar item, along with the items themselves (shown in Code 14). The authors of [5] define a sealed class as a class that restricts our class hierarchy to a specific

set of subtypes, where each one can define its own properties and functions. In our case: Home and Favourites.

Code 14. Menu items for the Jetpack Compose approach

```
sealed class BottomBarNav(val route: String, val name: String, val icon: ImageVector) {
object Home : BottomBarNav(route=Routes.HOME_SCREEN, name="Home",
icon=Icons.Default.Home)
// similarly for Favourites }
```

Next, we create a composable function named BottomBar in which the menu items are added, and the navigation is handled. The items are added by creating an extension function [5] which creates the navigationBarItem with the provided parameters. The code is presented in Code 15.

Code 15. BottomBar composable and extension function

```
@Composable
fun BottomBar(navController: NavController) { // the composable function takes navController argument
    val screens = listOf(BottomBarNav.Home, BottomBarNav.Favourites) // a list of possible screens
    val navBackStackEntry by navController.currentBackStackEntryAsState() // backstack functionality
    val currentDestination = navBackStackEntry?.destination // set variable to the latest backstack entry
    BottomAppBar(modifier = Modifier.fillMaxWidth().height(56.dp)) {// call AddItem for each screen...
@Composable
fun RowScope.AddItem(screen: BottomBarNav, currentDestination: NavDestination?, navController:
NavController) {
NavigationBarItem(
// create item, set the icon, current destination, onClick for navigation...)}
```

In Compose, Scaffold provides structuring of commonly used components like a top app bar, bottom app bar, and floating action button. In MainActivity.kt, we call the BottomBar function in the scaffold's bottomBar parameter and set the navController (shown in Code 16).

Code 16. Scaffold with bottomBar parameter

```
Scaffold( bottomBar = {BottomBar(navController = navController)},content = {// navigation
composables})
```

Creating the bottom navigation bar is straightforward. A few extra steps are needed in Compose: a function to add a menu item, and a class that contains the menu attributes (shown in Code 15). In terms of strengths and weaknesses, both approaches provide a maintainable and readable code with a similar size. A potential strength for Jetpack Compose is that Scaffold provides easier structuring in case of multiple items. In the provided code, we found it simpler to create the bottom navigation bar in the XML-based application.

4.3. Developing the PositionList screen

The positionList screen (Figure 4) displays a list of positions that can change dynamically by adding, deleting, or editing a position.

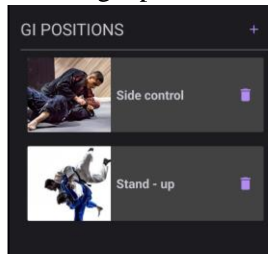

Figure 3. *PositionList screen*

In XML, we create a file for the position with an image, text, and button (shown in Code 17).

Code 17. PositionItem XML layout

```
<ImageView android:id="@+id/image_view_position_image" // image attributes />
<com.google.android.material.textview.MaterialTextView
 android:id="@+id/text_view_position_name" // textView attributes />
<com.google.android.material.button.MaterialButton  android:id="@+id/delete_button" // attributes/>
```

The items are displayed in a recyclerView in the positionList screen along with the discipline text and add button. The relevant code is presented in Code 18.

Code 18. RecyclerView component in PositionListScreen.xml

```
<androidx.recyclerview.widget.RecyclerView android:id="@+id/recycler_view_positions" // id
app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"/> //layout type
```

RecyclerView is used to display a list where each item is represented by an XML layout. In [4], it is mentioned that when an item scrolls off the screen, its layout gets reused by the next item when it scrolls onto the screen, with the corresponding data. To display the items, we create the recyclerView adapter, which acts as a data source for the recyclerView [4]. The adapter (shown in Code 19) instructs the recyclerView how to display a list of items.

Code 19. PositionItemAdapter.kt

```
class PositionItemAdapter(private var positions: List<Position>, val itemClickListener: (Position) ->
Unit,
   val editItemClickListener: (Position) -> Unit, val deleteItemListener: (Position) -> Unit
) : RecyclerView.Adapter<PositionItemAdapter.PositionItemViewHolder>() {
   inner class PositionItemViewHolder(private val binding: LayoutPositionItemBinding) : // item view
      RecyclerView.ViewHolder(binding.root) {
      fun bind(position: Position) = with(binding) { // set item properties and clickListeners...} }
   override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): PositionItemViewHolder {
      val binding = LayoutPositionItemBinding.inflate(LayoutInflater.from(parent.context), parent,false)
      return PositionItemViewHolder(binding)}
   override fun onBindViewHolder(holder: PositionItemViewHolder, position: Int) { // set values and
binding
// setPositions() and getItemCount() functions...
```

In the PositionListFragment file (as shown in Code 20) we initialize the adapter, set the itemClickListeners, set the list of positions to display, and set the adapter of the recyclerView which is created in the positionList.xml file.

Code 20. PositionListFragment.kt

```
adapter = PositionItemAdapter(emptyList(), // initialize the adapter with an empty list
  itemClickListener = { position -> // do something}, // other click listeners here...
lifecycle.coroutineScope.launch { positions.collect { adapter.setPositions(it) } } // set the positions
binding.recyclerViewPositions.adapter = adapter // set the recyclerView adapter
```

For Compose, we create a composable function that represents the position item (shown in Code 21). The position that is passed to the function is used to set the correct data, and onEvent is used to send events to the viewModel according to user input.

Code 21. PositionItem composable

```
@Composable
fun PositionItem(position: Position, onEvent: (PositionListEvent) -> Unit, modifier: Modifier) {
// structure content using rows and columns...(not shown here for sake of brevity)
 AsyncImage(// set image and its attributes)
Text(text = position.name, fontSize = 18.sp, fontWeight = FontWeight.Bold,) //position name
 IconButton(onClick = { onEvent(PositionListEvent.OnDeletePositionClick(position)) }){// delete
button
```

In the PositionListScreen composable, we use LazyColumn with a list of positions. LazyColumn is equivalent to recyclerView, by lazily loading data (loading data when needed).

Code 22. LazyColumn composable in PositionListScreen

```
LazyColumn(modifier = Modifier.fillMaxWidth()) { items(positions.value) { position ->
// for each position
      PositionItem(position = position, // create a PositionItem composable with the required parameters...
        onEvent = positionListViewModel::onEvent,
        modifier = Modifier.fillMaxWidth().combinedClickable( // handle click events...)
```

Creating the positionList screen in XML requires a boilerplate code, as shown in the relevant code sections from Code 17 to Code 20. We need two files for the PositionList fragment (layout and class), one for the position item layout, and one for the adapter. In Compose, we need a composable for the item layout, and a composable for displaying the list within the PositionList screen (shown in Code 21 and Code 22). The main weakness of the XML approach in this case is the number of files and code, increasing complexity. We conclude that it is simpler and faster to create and maintain this kind of list in Compose, along with requiring significantly less code.

5. Results and discussion

Over time, XML-based applications increase in size and complexity. Once components are defined and used in layout files, they must be connected to a fragment class and be inflated (converted into View and ViewGroup objects) in an overridden lifecycle method (onCreateView) [6]. The separation of concerns is the primary principle of this method,

where the screen layouts are defined in XML files and the logic that handles the interactions of the user is defined in the fragment files. This principle is not always beneficial, according to the authors of [6]. In their book, they explain how cohesion (how parts of a given module are related to one another) relates to Android UI. Namely, the problem lies with the different languages used for the layout files (XML) and fragment/activity classes (kotlin), leading to the need for layout inflation in order for these two parts to communicate. This increases complexity and reduces maintainability, having to modify multiple files in case of changes. Jetpack Compose-based UI favors the composition instead of the inheritance tree of XML-based UI. It is declarative, which means that we describe what the program should do, not how to do it [6]. One of the potential drawbacks is the lack of forced separation of concerns, meaning that the logic and the UI can be mixed together. The developer has the freedom to put as much application logic in a UI component as they wish, which may lead to a less readable and poorly structured code. Compose sacrifices this forced structuring of UI building to achieve its simplicity [6]. Writing the UI and the application logic using the same language solves the problem of cohesion that is prevalent in the XML-based approach.

From a developer's perspective, the XML method provides a large array of tools to work with, like the design tools of Android Studio. Along with the many available resources, and being the standard of Android development, it is usually more convenient for solving some issues. Jetpack Compose provides simplicity and less boilerplate code, as can be seen from our examples. However, due to its being a recent technology, there are limited resources to consult when encountering some problems.

Our primary criteria for choosing a technology for a beginner in Android development are simplicity and ease of use. The use of one language (kotlin) for both UI and application logic makes Compose much simpler than the XML method. Additionally, the declarative code makes it easier to create UI by using functions. Therefore, we recommend it for developers who are starting out their Android journey. Android developers using the XML method may consider switching to Compose for the reasons mentioned above. Writing less UI code switches the focus to logical problems. A good example of this could be creating screens that can contain multiple lists. It would be faster to create more composable functions for each list (like PositionList) than to create multiple files for each list as shown in the XML approach.

6. Conclusion

From our research, and by analyzing specific parts of the application in both approaches, we have concluded that the XML-based approach for UI design can be quite complex and time-consuming compared to the Jetpack Compose approach. Even though XML is the standard method of developing Android applications at the moment, we assume that Jetpack Compose will slowly replace the XML-based design.

References

[1] Castillo, J, Shikov, A: "Jetpack Compose internals" (2021, Leanpub)

[2] Dumbravan, A: "Clean Android Architecture - Take a layered approach to writing clean, testable, and decoupled Android applications" (2022, Packt Publishing)

[3] Forrester, A, Boudjnah, E, Dumbravan, A, Tigcal, J: "How to Build Android Apps with Kotlin - A practical guide to developing, testing, and publishing your first Android apps 2" (2023, Packt Publishing)

[4] Griffiths, D, Griffiths, D: "Head First Android Development - A Learner's Guide to Building Android Apps with Kotlin" (2021, O'Reilly Media)

[5] Griffiths, D, Griffiths, D: "Head First Kotlin" (2019, O'Reilly)

[6] Kodeco Team, Buketa, D, Prasad, P: "Jetpack Compose by Tutorials (Second Edition) - Building Beautiful UI With Jetpack Compose" (2023, Kodeco Inc.)

[7] Murphy, M L.: "The Busy Coder's Guide to Android Development" (2017, CommonsWare)

[8] Smyth, N: "Jetpack Compose 1.3 Essentials - Developing Android Apps with Jetpack Compose 1.3, Android Studio, and Kotlin" (2023, eBookFrenzy)

[9] Späth, P: "Pro Android with Kotlin - Developing Modern Mobile Apps with Kotlin and Jetpack" (2023, Apress)

Egzon Milla
University of Montenegro,
Faculty of Electrical Engineering Podgorica, Montenegro
E-mail address: egzonmilla8@gmail.com

Milutin Radonjić
University of Montenegro,
Faculty of Electrical Engineering
Podgorica, Montenegro
E-mail address: mico@ucg.ac.me