Task Core 1 – Spike: Taking Chances

**Link to GitHub repository:** [**https://github.com/SoftDevMobDevJan2024/core1-104177995**](https://github.com/SoftDevMobDevJan2024/core1-104177995)

**Goals:**

* Showcase the individual's proficiency via an individual activity app.
* Recognize the many stages of an activity.
* For testing and debugging, use the IDE and logs
* Create layouts that have numerous constraints.
* Effectively employ listeners as needed.
* Develop an application with multilingual support.

**Tools and Resources Used**

* The course’s modules
* GitHub and Git
* Kotlin programming language and XML files
* Android Studio IDE
* Android Developer: <https://developer.android.com/guide/>

**Knowledge Gaps and Solutions**

# Gap 1: 2 layouts for landscape and portrait screen

By default, the app is oriented in portrait mode. An additional layout resource file must be generated in the activity\_main directory in order to generate a landscape layout.A screenshot of a computer

Description automatically generated

# Gap 2: Constraint Layout deployment

I made my mind to utilize a constraint layout for both the landscape and portrait orientations after observing the demonstration footage and carefully reading the assignment's description, which stated that at least 1 layout had to be a constraint layout.

A screenshot of a computer screen

Description automatically generated A screenshot of a video game

Description automatically generated

# Gap 3: Incoporating localisation

A screenshot of a phone

Description automatically generated A screenshot of a cell phone

Description automatically generated

There are two languages versions of the app: English and Vietnamese. In order to accomplish this, I make an additional resource directory for values, pick the Vietnamese language, and select the locale qualifier. The default language is currently set to English.

Except for the string values between the string tags, the resource files must be exactly the same. The IDs kept in both of the strings.xml files are used by string externalization to provide this functionality. An application's development and usability are improved when the device's language setting is modified because the matching resource file is utilized.

A screen shot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generated

# Gap 4: Employing Logs

At different stages of the activity's life cycle and interactions, log messages are appended in order to inspect the logs in this instance. Log messages are inserted, for example, when the activity is created, restored, or when buttons are touched, to offer comprehensive debugging information about these occurrences. We might inspect these logs by opening Logcat once the program has executed. This is the code's implementation of logging:

roll.setOnClickListener {

dice = diceRoller.roll()

Log.d("MainActivity", "Rolled a dice: $dice")

}

A screenshot of a computer program

Description automatically generated

Logging also helps track the state when the activity's state changes, such as during screen rotations. To achieve this, we override the onSaveInstanceState() function:

override fun onSaveInstanceState(outState: Bundle) { super.onSaveInstanceState(outState) outState.putInt("currentScore", currentScore) Log.d("MainActivity", "Saved currentScore to savedInstanceState: $currentScore")

}x

This setup ensures that the current score is logged and can be restored correctly after state changes, making it easier to debug and maintain the app.

# Gap 5: Utilizing listeners

The listeners were used as the backbone of the program’s logic and functions. I have used the listeners to manage crucial interactions, such as rolling the dice, incrementing/decrementing the score, and resetting the game. For example, the listener for the roll button rolls the dice and updates the UI with the new dice value. Similarly, listeners for the add, sub, and reset buttons handle score modifications and UI updates. The listener for the add button is shown in the code snippet below:

add.setOnClickListener {

currentScore = score.text.toString().toInt()

currentScore += dice

score.text = currentScore.toString()

Log.d("MainActivity", "Added dice to currentScore: $currentScore")

}

A screenshot of a phone

Description automatically generated A screenshot of a cell phone

Description automatically generated

# Gap 6: Preserving the state after rotating applications

For this part, the onSaveInstanceState method is utilized to save the current score, ensuring its preservation upon screen rotation. This is demonstrated in the code snippet below. The score TextView is the element that requires saving upon screen rotation.

# override fun onSaveInstanceState(outState: Bundle) {

# super.onSaveInstanceState(outState)

# val score = findViewById<TextView>(R.id.score)

# // Save the currentScore and text color into the outState Bundle

# outState.putInt("currentScore", currentScore)

# outState.putInt("textColor", score.currentTextColor)

# Log.d("MainActivity", "Saved currentScore to savedInstanceState: $currentScore")

# }

# A screenshot of a phone Description automatically generated A screenshot of a computer Description automatically generated

# Gap 7: IDE application

I found and fixed bugs during the working process by using the aid of the IDE's run, debug, test, and log capabilities.

A screen shot of a computer

Description automatically generated

**Open Issues and Recommendations**

During the development of the app, I encountered an issue during the testing phase. Despite the app functioning correctly during manual testing and not introducing any new errors, the automated tests failed again and again. Upon investigation, I discovered that the randomness of the dice rolls, which produce values between 1 and 6, made it impossible to consistently produce the exact results needed for comparison in some tests. To solve this, I set the dice roller to use a predefined list of fixed dice rolls. The code I used is shown below:

* Added code:

class MockDiceRoller(private val fixedRolls: List<Int>) : DiceRoller {

private var index = 0

override fun roll(): Int {

val result = fixedRolls[index % fixedRolls.size]

index++

return result

}

}

Furthermore, I still need to modify the rule in the class MainActivityTest to replace the random dice roller with a mock dice roller that returns predefined values.

private fun setMockDiceRoller(fixedRolls: List<Int>) {

mActivityScenarioRule.scenario.onActivity { activity ->

activity.setDiceRoller(MockDiceRoller(fixedRolls))

}

}

With the adjusted code, the test ensures that every call to roll the dice will return values from the fixedRolls list in order, allowing for precise control over the dice roll outcomes during testing. The test returned passed as the screenshot below.

A screen shot of a cell phone

Description automatically generated