Task C – Spike: Core 1 – Taking Chances

GitHub repo: https://github.com/SoftDevMobDevJan2024/core1-104222196

Goals:

This section is an overview highlighting what the task is aiming to teach or upskill.

Build a one-activity game that can handle button clicks, save its state on an orientation change, and support two languages and two layouts. Practice navigating the Android Studio IDE and using logs for debugging.

The following list outlines the goal broken down into more specific knowledge gaps involved in the goal.

- Handling button clicks
- Displaying the game state
  - Using enums
  - Using data classes
  - Changing displayed text
  - Enabling and disabling buttons
- Saving state
  - Using ViewModel
  - Using LiveData and associating an Observer
- Supporting two languages
  - Using the String resource type
- Supporting two layouts
  - Using ConstraintLayout
- Debugging with logs

Tools and Resources Used

This section lists related software, tools, libraries, API's, and other resources used for this knowledge gap.

- Android Studio
- View.setOnClickListener function: https://developer.android.com/reference/kotlin/android/view/View#setonclicklistener
- findViewById function: https://developer.android.com/reference/android/view/View#findViewById(int)
- TextView.setText function: https://developer.android.com/reference/kotlin/android/widget/TextView#settext
- TextView.setTextColor function: https://developer.android.com/reference/kotlin/android/widget/TextView#settextcolor
- View.setEnabled: https://developer.android.com/reference/android/view/View#setEnabled(boolean)
- ViewModel overview: https://developer.android.com/topic/libraries/architecture/viewmodel
- LiveData overview: https://developer.android.com/topic/libraries/architecture/livedata
- MutableLiveData documentation: https://developer.android.com/reference/androidx/lifecycle/MutableLiveData?hl=en
- String resource usage: https://developer.android.com/guide/topics/resources/string-resource
- Log class documentation: https://developer.android.com/reference/kotlin/android/util/Log

Knowledge Gaps and Solutions

This section presents the listed knowledge gaps and their solutions with supporting images, screenshots and captions where appropriate/required.

Gap 1: Handling button clicks

1. After adding the necessary buttons into the layouts and giving them the appropriate ID, reference them in the `onCreate` function of the Activity with `findViewById<Button>(R.id.buttonId)` where `buttonId` is the actual ID of the button:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    setContentView(R.layout.layout)

    val rollButton = findViewById<Button>(R.id.roll)
    val addButton = findViewById<Button>(R.id.add)
    val subtractButton = findViewById<Button>(R.id.subtract)
    val resetButton = findViewById<Button>(R.id.reset)
}
```

2. After obtaining the references to the button, associate them with the appropriate event handler with the `setOnClickListener` function in `onCreate`:

```
rollButton.setOnClickListener {
    model.roll()
}

addButton.setOnClickListener {
    model.add()
}

subtractButton.setOnClickListener {
    model.subtract()
}

resetButton.setOnClickListener {
    model.reset()
}
```

The event handling code will be implemented below.

Gap 2: Displaying the game state

1. The application state at any given moment contains three values: score, dice number, and game state. The game state must be one of three values: ROLLING (dice can be rolled, cannot add/subtract), ROLLED (dice cannot be rolled, must add/subtract), or WON (score is 20). This is captured in the following enumeration:

```
enum class GameState {
    ROLLING, ROLLED, WON
}
```

2. The application state can be captured in the following data class with default values:

```
data class DiceRollUIState (
    val score: Int = 0,
    val diceValue: Int = 0,
    val state: GameState = GameState.ROLLING
)
```

3. Create the text views necessary to display the dice value and the score in the layout file. Afterward, reference them in `onCreate` in the same way as referencing the buttons:

```
val diceView = findViewById<TextView>(R.id.dice)
val scoreView = findViewById<TextView>(R.id.score)
```

4. The UI can be updated with the following code. Note that setting a button's `isEnabled` field to `true` or `false` will enable or disable it. The text views' contents and colors are changed with the `text` property and `setTextColor` function respectively:

```
val dices = arrayOf("\u2610", "\u2680", "\u2681", "\u2682", "\u2683", "\u2684",
"\u2685")
```

```
when (state.state) {
    GameState.ROLLED -> {
        rollButton.isEnabled = false
        addButton.isEnabled = true
        subtractButton.isEnabled = true
    }
    GameState.ROLLING -> {
        rollButton.isEnabled = true
        addButton.isEnabled = false
        subtractButton.isEnabled = false
    }
    GameState.WON -> {
        rollButton.isEnabled = false
        addButton.isEnabled = false
        subtractButton.isEnabled = false
    }
}

diceView.text = dices[state.diceValue]

if (state.score < 20) {
    scoreView.setTextColor(Color.BLACK)
} else if (state.score > 20) {
    scoreView.setTextColor(Color.RED)
} else {
    scoreView.setTextColor(Color.GREEN)
}

scoreView.text = state.score.toString()
```

`state` is an object of `DiceRollUIState` which contains three fields: `score`, `diceValue`, and `state`. The code above is not directly inside `onCreate`, but rather inside an `Observer` which will be discussed below.

Gap 3: Saving state

To save the application's state, we will use `ViewModel` in conjunction with `LiveData`. `ViewModel` is a business logic state holder that persists state across configuration changes. `LiveData` is a lifecycle-aware data holder that alerts active `Observers` when the data it holds changes.

1. First, add the following implementation in the `build.gradle` file to be able to instantiate our `DiceRollViewModel` class:

```
dependencies {
   ...
    implementation "androidx.activity:activity-ktx:1.7.2"
}
```

2. Next, create the class `DiceRollViewModel`, which inherits from `ViewModel`:

```
import androidx.lifecycle.ViewModel
import androidx.lifecycle.MutableLiveData
import kotlin.math.max
import kotlin.random.Random

class DiceRollViewModel : ViewModel() {
    val uiState = MutableLiveData(DiceRollUIState())
    private val random = Random(1)
    fun roll() {
        uiState.value = uiState.value?.copy(
            diceValue = random.nextInt(1, 7),
            state = GameState.ROLLED
        )
    }

    fun add() {
        val score = uiState.value?.score
        val diceValue = uiState.value?.diceValue

        if (score != null && diceValue != null) {
            val newScore = score + diceValue

            uiState.value = uiState.value?.copy(
                score = newScore,
                state = if (newScore == 20) GameState.WON else GameState.ROLLING
            )
        }
    }

    fun subtract() {
        val score = uiState.value?.score
        val diceValue = uiState.value?.diceValue

        if (score != null && diceValue != null) {
            val newScore = max(score - diceValue, 0)

            uiState.value = uiState.value?.copy(
                score = newScore,
                state = if (newScore == 20) GameState.WON else GameState.ROLLING
            )
        }
    }

    fun reset() {
        uiState.value = DiceRollUIState()
    }
}
```

Inside `onCreate`, we instantiate our class with:

```
val model: DiceRollViewModel by viewModels()
```

Our class keeps the application state in a `MutableLiveData` object, which is a mutable version of `LiveData`. We initialize the application state by supplying `MutableLiveData`'s constructor with a new `DiceRollUIState` object. Note that our class also contains the functions to update the application's state (which can be viewed as business logic.) The UI buttons call these functions when clicked, changing the value inside `MutableLiveData`, which in turn triggers an associated `Observer`. This `Observer` can be implemented and attached to `MutableLiveData` as follows (inside `onCreate`):

```
val stateObserver = Observer<DiceRollUIState> { state ->
    when (state.state) {
        GameState.ROLLED -> {
            rollButton.isEnabled = false
            addButton.isEnabled = true
            subtractButton.isEnabled = true
        }
        GameState.ROLLING -> {
            rollButton.isEnabled = true
```

```
                addButton.isEnabled = false
                subtractButton.isEnabled = false
            }
            GameState.WON -> {
                rollButton.isEnabled = false
                addButton.isEnabled = false
                subtractButton.isEnabled = false
            }
        }

        diceView.text = dices[state.diceValue]

        if (state.score < 20) {
            scoreView.setTextColor(Color.BLACK)
        } else if (state.score > 20) {
            scoreView.setTextColor(Color.RED)
        } else {
            scoreView.setTextColor(Color.GREEN)
        }

        scoreView.text = state.score.toString()
}

model.uiState.observe(this, stateObserver)
```

Now every time the application state changes, the code inside the `Observer` will run. Note that this code is the same as the one used to control the content and appearance of the UI written in the previous section.

Gap 4: Supporting two languages

Instead of hard-coding string values into the UI components, we can specify them in a String resource file located under `res/values` and reference them inside the layout.xml file.

1. Create the String resource file with the necessary strings (`strings.xml`):

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Taking Chances</string>
    <string name="roll">Roll!</string>
    <string name="add">Add</string>
    <string name="subtract">Subtract</string>
    <string name="reset">Reset</string>
</resources>
```

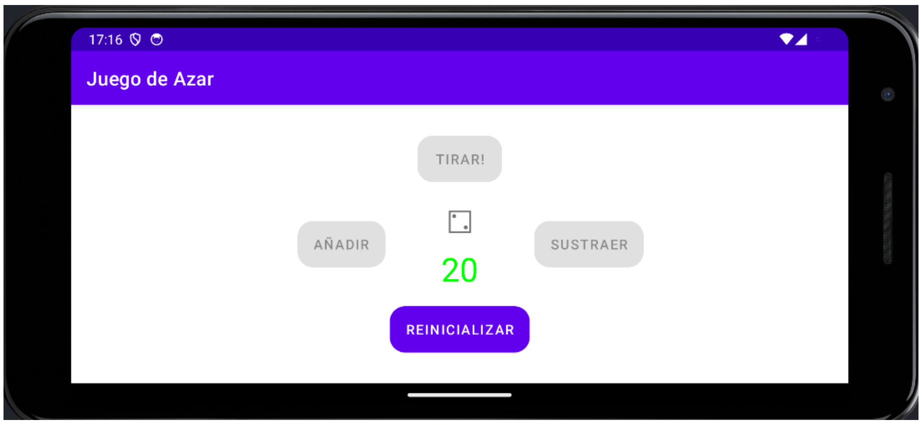
2. Inside `layout.xml`, reference the name of these strings instead (do for all components):

```
<Button
    ...
    android:text="@string/roll" />
```

3. To support another language, create another String resource file. Its name will be that of the original string file plus a hyphen and the first two letters of the desired language (for example, `strings-es.xml`, `strings-fr.xml`, `strings-vi.xml`, …)

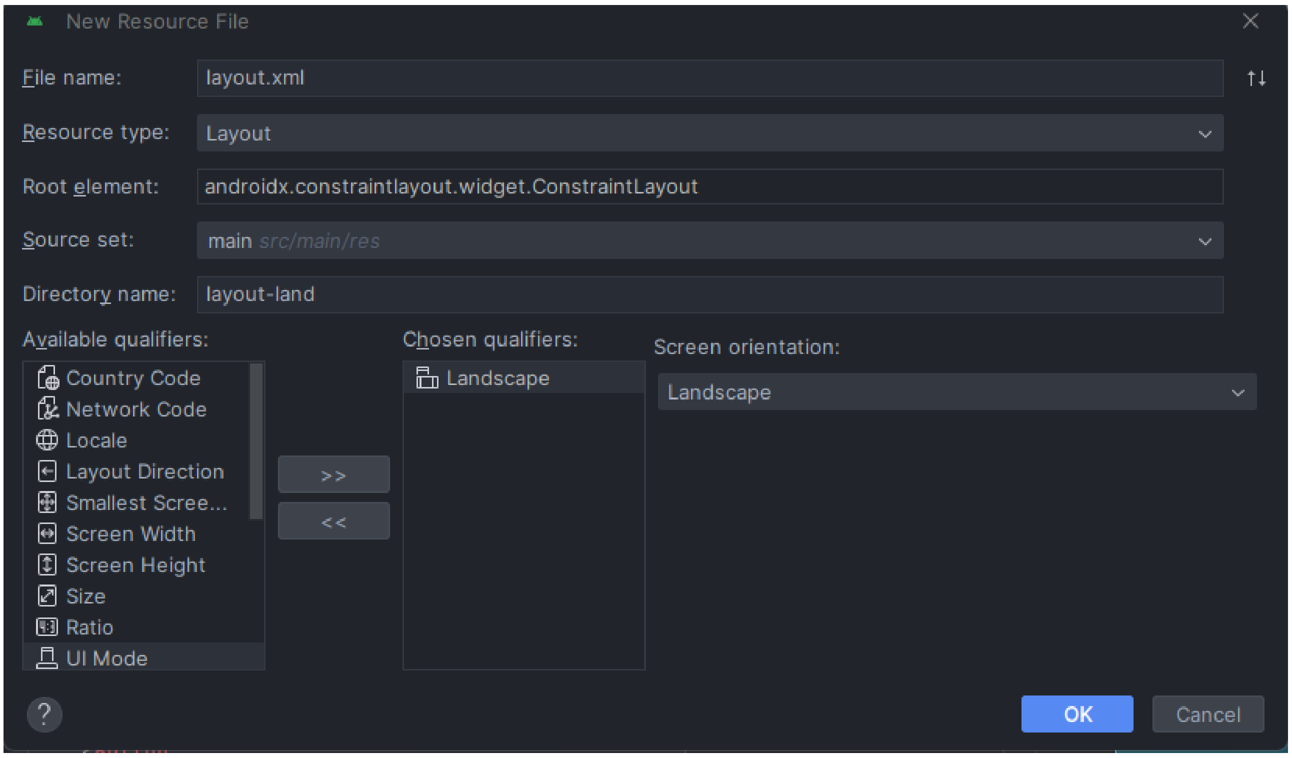
```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Juego de Azar</string>
    <string name="roll">Tirar!</string>
    <string name="add">Añadir</string>
    <string name="subtract">Sustraer</string>
    <string name="reset">Reinicializar</string>
</resources>
```

The application is now available in another language (Spanish in the image below):



Gap 5: Landscape layout with ConstraintLayout

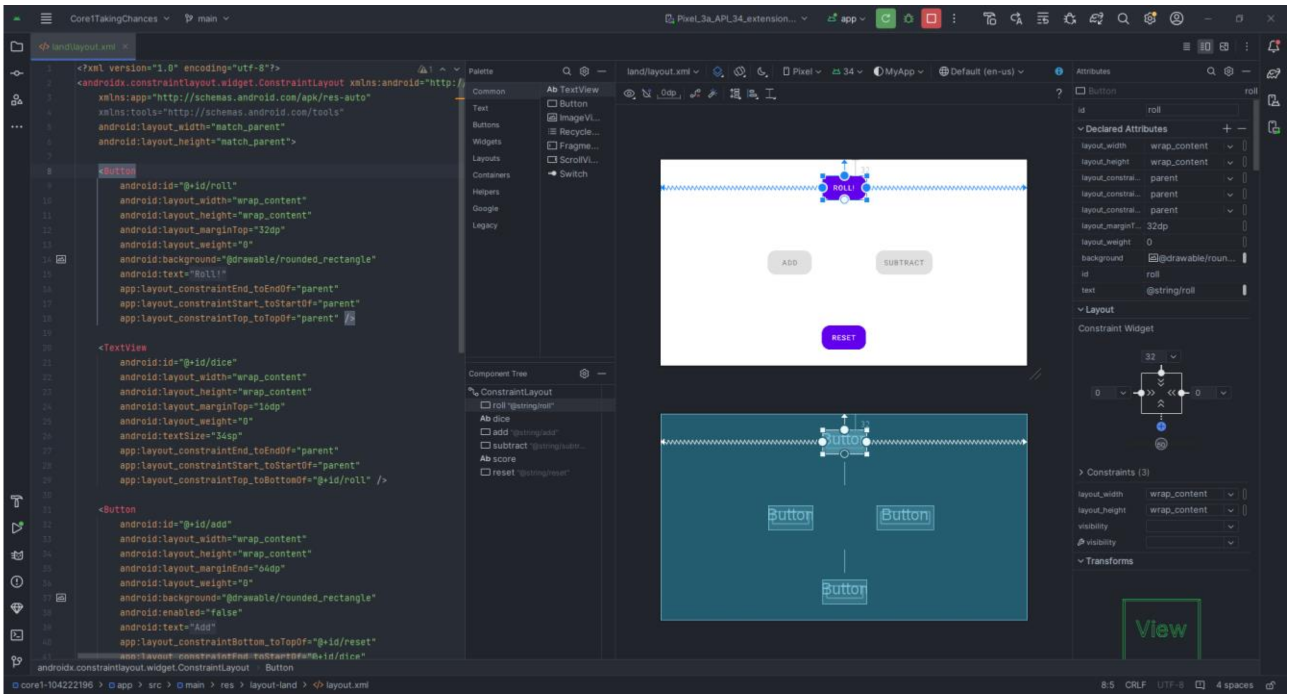
1. To create another layout for landscape mode using the `ConstraintLayout`, right-click the `res` folder, choose New – Android Resource File, and specify the settings as follows:



New Resource File

File name: layout.xml
Resource type: Layout
Root element: androidx.constraintlayout.widget.ConstraintLayout
Source set: main src/main/res
Directory name: layout-land
Available qualifiers: Country Code, Network Code, Locale, Layout Direction, Smallest Scree..., Screen Width, Screen Height, Size, Ratio, UI Mode
Chosen qualifiers: Landscape
Screen orientation: Landscape

2. After adding the layout file, drag and drop the necessary components into the UI builder. To change the position of a component, click on it and drag the circle on one of its 4 sides to one side of the screen or another component. Alternatively, edit the XML code:



```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http:/
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
        android:id="@+id/roll"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="32dp"
        android:layout_weight="0"
        android:background="@drawable/rounded_rectangle"
        android:text="Roll!"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/dice"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:layout_weight="0"
        android:textSize="34sp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/roll" />

    <Button
        android:id="@+id/add"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginEnd="64dp"
        android:layout_weight="0"
        android:background="@drawable/rounded_rectangle"
        android:enabled="false"
        android:text="Add"
        app:layout_constraintBottom_toTopOf="@+id/reset"
        app:layout_constraintEnd_toStartOf="@+id/dice"
```

androidx.constraintlayout.widget.ConstraintLayout › Button

core1-104222196 › app › src › main › res › layout-land › layout.xml

Gap 6: Using logs

1. To start using logs, first import the `Log` class in the source files in which they are called:

```kotlin
import android.util.Log
```

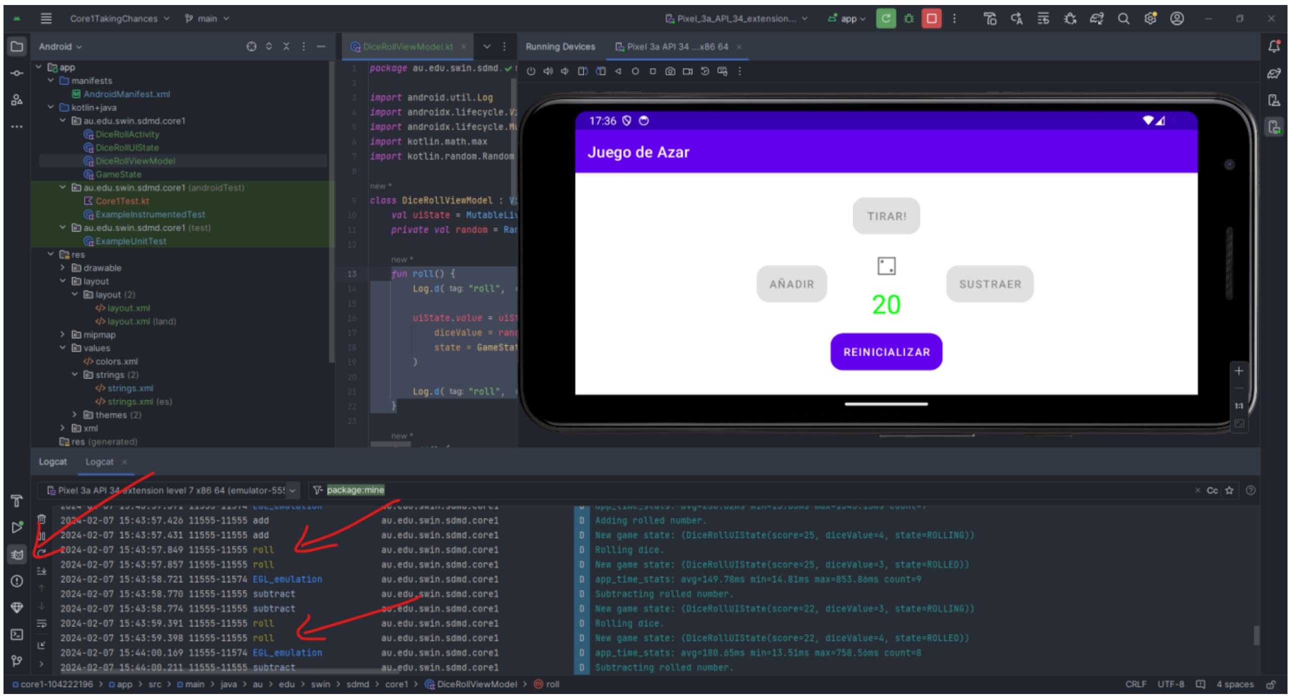
2. There are many types of logs: debug, error, verbose, etc. Call the suitable one, passing in the `tag` and `message`:

```kotlin
fun roll() {
    Log.d("roll", "Rolling dice.")

    uiState.value = uiState.value?.copy(
        diceValue = random.nextInt(1, 7),
        state = GameState.ROLLED
    )

    Log.d("roll", "New game state: (${uiState.value?.toString()})")
}
```

3. Run the application and view the logs in the `Logcat` window:



```
package au.edu.swin.sdmd.

import android.util.Log
import androidx.lifecycle.Vi
import androidx.lifecycle.Mu
import kotlin.math.max
import kotlin.random.Random

class DiceRollViewModel : Vi
    val uiState = MutableLiv
    private val random = Ran

    fun roll() {
        Log.d( tag: "roll",

        uiState.value = uiS
            diceValue = rand
            state = GameStat
        )

        Log.d( tag: "roll",
    }
```

```
2024-02-07 15:43:57.426 11555-11555 add            au.edu.swin.sdmd.core1   D  Adding rolled number.
2024-02-07 15:43:57.431 11555-11555 add            au.edu.swin.sdmd.core1   D  New game state: (DiceRollUIState(score=25, diceValue=4, state=ROLLING))
2024-02-07 15:43:57.849 11555-11555 roll           au.edu.swin.sdmd.core1   D  Rolling dice.
2024-02-07 15:43:57.857 11555-11555 roll           au.edu.swin.sdmd.core1   D  New game state: (DiceRollUIState(score=25, diceValue=3, state=ROLLED))
2024-02-07 15:43:58.721 11555-11574 EGL_emulation  au.edu.swin.sdmd.core1   D  app_time_stats: avg=149.78ms min=14.81ms max=853.86ms count=9
2024-02-07 15:43:58.770 11555-11555 subtract       au.edu.swin.sdmd.core1   D  Subtracting rolled number.
2024-02-07 15:43:58.774 11555-11555 subtract       au.edu.swin.sdmd.core1   D  New game state: (DiceRollUIState(score=22, diceValue=3, state=ROLLING))
2024-02-07 15:43:59.391 11555-11555 roll           au.edu.swin.sdmd.core1   D  Rolling dice.
2024-02-07 15:43:59.398 11555-11555 roll           au.edu.swin.sdmd.core1   D  New game state: (DiceRollUIState(score=22, diceValue=4, state=ROLLED))
2024-02-07 15:44:00.169 11555-11574 EGL_emulation  au.edu.swin.sdmd.core1   D  app_time_stats: avg=180.65ms min=13.51ms max=758.56ms count=8
2024-02-07 15:44:00.211 11555-11555 subtract       au.edu.swin.sdmd.core1   D  Subtracting rolled number.
```

Open Issues and Recommendations

This section outlines any open issues, risks, and/or bugs, and highlights potential approaches for trying to address them in the future.

Issue: `MutableLiveData` is public

One of the motivations for using `ViewModel` is to separate the business logic from the UI code. The parts that contain the business logic should only expose functions that the UI can call to update the data and prevent data from being changed directly. The current implementation of the `DiceRollViewModel` class leaves the `MutableLiveData` field public, allowing outside code to directly change it by calling `setValue`. A safer implementation is to keep the `MutableLiveData` private and convert it into an immutable public `LiveData` field.