Data Detective – Audit 3

Entwicklungsprojekt WS 25/26

von Florian Roß und Karim Khemiri

Inhaltsverzeichnis

Moderne App Entwicklung mit Kotlin

Klassiche Kotlin-Android Umsetzung:

- App Launch über MainActivity
- Jetpack Compose als moderne Grundlage für Android Apps
- Navigations Controller zum Wechsel von Screens
- ViewModel zur Nutzung des App internen States

```kotlin
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            DataDetectiveTheme {
                val navController = rememberNavController()
                val viewModel: GameViewModel = viewModel()

                NavHost(navController = navController, startDestination = "home") {

                    composable( route = "home") {
                        HomeScreen(
                            viewModel = viewModel,
                            onStartGame = { navController.navigate( route = "game") },
                            onNews = {navController.navigate( route = "news")},
                            onAchievements = {navController.navigate( route = "achievements")},
                        )
                    }
```

Grafikbibliothek Wechsel:

Compose UI statt Kotlin Notebooks mit Kandy und Lets Plot

-> Etwas weniger Funktionsvielfalt aber für unsere Zwecke des Rapid Protype mehr als genug (PoC 102)

```kotlin
class GameViewModel : ViewModel() {
    5 Usages
    var userProfile by mutableStateOf( value = UserProfile( name = "Karim"))
        private set
    5 Usages
    var currentTask by mutableStateOf<Task?>( value = null)
        private set
    5 Usages
    var selectedAnswerIndex by mutableStateOf<Int?>( value = null)

    init {
        nextQuestion()
    }
    2 Usages
    fun nextQuestion() {
        val base = sampleDataSet.random()

        currentTask = TaskGenerator.generate(base)
        selectedAnswerIndex = null
    }
```

Überarbeitete PoCs

PoC 102 Darstellung von Diagrammen ✅/❌
Verschiedene Diagrammtypen sollen korrekt dargestellt und erzeugt werden können (mit externen Statistik Libs)

PoC 107 Stilistische Manipulationen ❌
Es soll geprüft werden, ob Diagramme über rein visuelle Stilmittel (Fehlende Beschriftungen,Style(Farben,strichstärken, hervorhebungen etc.), Overlaps / Occlusion) manipuliert werden können

PoC 108 Antwortlogik und Eindeutigkeit der richtigen Lösung ✅
Es soll geprüft werden, ob für jede automatische generierte Aufgabe:
- genau eine eindeutig korrekte Antwort existiert
- falsche Antworten plausibel aber klar unterscheidbar sind

PoC 109 Reale Anwendungsbeispiele zu Statistik-Manipulationen ❌
Es soll geprüft werden, ob nach Lösungsabgabe passende hochwertige Realbeispiele bereitgestellt werden

PoC 110 Gamification ✅/❌
Es soll geprüft werden, ob die App genügend Spielspaß und Erfolgserlebnisse liefert, zur Gewährleistung der Nutzermotivation.
geplante Gamification Methoden:Experience System, XP Rewards (Titel, Customization), Achievements, Daily Challenges/Streaks, User Statistik, Schwierigkeitsgrade

PoC 1: Dynamische Aufgabengenerierung

PoC 101 Dynamische Generierung von Statistikaufgaben
Beschreibung des Vorhabens:
Statistikaufgaben sollen automatisch und konsistent aus Basis-Datensätzen erzeugt werden.
(...)

Umsetzung:
TaskGenerator.genrate(...)
Zufällige Auswahl:
- Datensatz
- Kompatibler Diagrammtyp
- Kompatibler Manipulationstyp
- Stärke der Manipulatonsparameter(sinnvoll begrenzt)
- Antwortmöglichkeiten(mit genau einer richtgen)

Ergebnis:
Vollständige Aufgabe die an ManipulatedChart übergeben werden kann

```
object TaskGenerator {
    2 Usages
    fun generate(base: ChartData): Task {
        val chartType = base.supportedChartTypes.random()     //Zufälliger Diagrammtyp aus erlaubten Typen
        val allowedManipulations = chartType.allowedManipulations         //Erlaubte Manipulationen für den Diagrammtyp
        val manipulationType = allowedManipulations.random()         //Zufällige Manipulation aus den erlaubten
        val random = Random( seed = base.id)
        //Erzeugt ein Manipulations-Objekt, das beschreibt, wie das Diagramm manipuliert wird
        val manipulation = Manipulation(type = manipulationType, //ManipulationsTyp
            intensity = ManipulationParameters.manipulationIntensity(manipulationType, random), //zufällige Stärke der Manipulation
            categoryRange = if (manipulationType == ManipulationType.TRUNCATED_CATEGORY_AXIS)
                ManipulationParameters.categoryRange(base.xData.size, random)//erzeugt bei Kategorie-Achse Manipulation einen zufälligen Bereich von Kategorien
            else null)
        val correctAnswer = AnswerPool.correctAnswerFor(manipulationType) //Richtige Antwort basierend auf Manipulationstyp
        val (options, correctIndex) = generateAnswerOptions(correctAnswer,  pool = AnswerPool.allAnswers)//Erzeugt Antwortmöglichkeiten
        return Task(
```

PoC 3: Automatisch manipulierte Diagramme

PoC 103 Generierung von manipulierten Diagrammen

Beschreibung des Vorhabens

Es soll geprüft werden, ob manipulierte Diagramme automatisch aus unveränderten Basisdatensätzen erzeugt werden können
Das System soll

- ein unverändertes Basisdiagramm als Ausgangspunkt verwenden
- abhängig vom Manipulationstyp und Manipulationsstärkeparameter Diagrammmerkmale dynamisch verändern
- die Manipulation rein über Darstellungslogik umsetzen, nicht durch veränderung der Rohdaten (...)

Umsetzung:
ManipulatedChart() bekommt eine Task übergeben und berechnet daraus die manipulierten Chart-Werte.

Aus diesen Werten wird dann das jeweilige diagramm gerendert.

```
//Rendering je nach ChartTyp
when (task.chartType) {
    ChartType.BAR -> {
        BarChart(...) }
    ChartType.LINE -> {
        LineChart(...) }
    ChartType.HORIZONTAL_BAR -> {...}
}
```

```
@Composable
fun ManipulatedChart(task: Task, showCorrect: Boolean, modifier: Modifier = Modifier
    .fillMaxWidth().height( height = 260.dp)) {
    val ydata = task.yValues
    val xData = task.xData
    val unit = task.unit
    val manipulation = task.manipulation
    //unmanipulierte y-Werte
    val rawMinY = 0f
    val rawMaxY = ydata.maxOrNull() ?: 1f
    //Zielwerte für Werte-Achse
    val targetMinY = if (showCorrect) rawMinY
                     else manipulation.type.manipulateMinValue(ydata, manipulation)

    val targetMaxY = if (showCorrect) rawMaxY
                     else manipulation.type.manipulateMaxValue(ydata, manipulation)
    //Gesamtanzahl der Kategorien
    val totalCount = xData.size
    //Startpunkt des sichbaren Kategoriebereich (wird nur verändert bei TRUNCATED_CATEGORY_AXIS Manipulation)
    val targetCategoryStart = if (showCorrect || manipulation?.categoryRange == null) { 0f }
                              else { manipulation.categoryRange.first.toFloat() }
    //Endpunkt des sichbaren Kategoriebereich (wird nur verändert bei TRUNCATED_CATEGORY_AXIS Manipulation)
    val targetCategoryEnd = if (showCorrect || manipulation?.categoryRange == null) { (totalCount - 1).toFloat() }
                            else { manipulation.categoryRange.last.toFloat() }
```

PoC 4: Animierte Auflösung

PoC 104 Animierte Auflösung von Manipulationen
Beschreibung des Vorhabens:
Es soll geprüft werden, ob statistische Manipulationen animiert aufgelöst werden können. Dabei soll zwischen manipulierten und korrekten Zustand gewechselt werden, ohne das Diagramm neu zu laden (...)

Umsetzung:
In ManipulatedChart kann durch showCorrect(boolean) zwischen manipuliertem und korrektem Zustand gewechselt werden.

Über animateFloatAsState wird dieser Wechsel animiert.

```
//Animationen für showCorrect wechsel
val animatedMinY by animateFloatAsState(
    targetValue = targetMinY,
    animationSpec = tween( durationMillis = 1200, easing = FastOutSlowInEasing))

val animatedMaxY by animateFloatAsState(
    targetValue = targetMaxY,
    animationSpec = tween( durationMillis = 1200, easing = FastOutSlowInEasing))

val animatedCategoryStart by animateFloatAsState(
    targetValue = targetCategoryStart,
    animationSpec = tween( durationMillis = 1200, easing = FastOutSlowInEasing))

val animatedCategoryEnd by animateFloatAsState(
    targetValue = targetCategoryEnd,
    animationSpec = tween( durationMillis = 1200, easing = FastOutSlowInEasing))
```

PoC 5/6: Manipulationen

PoC 105 Werteachsen Manipulationen :

Bei Werteachsen Manipulation wird die Werteachse angehoben (also z.B. Startpunkt nicht bei 0)

PoC 106 Kategorieachsen Manipulationen :

Bei Kategorieachsen Manipulation wird der Kategoriebereich verringert(also z.B. weniger Balken)

```
enum class ManipulationType {
    5 Usages
    TRUNCATED_VALUE_AXIS { //Verkürzte Werte-Achse (Y bei Bar/Line, X bei HorizontalBar) damit Achse nicht bei 0 beginnt
        override fun manipulateMinValue(data: List<Float>, manipulation: Manipulation): Float {
            val min = data.minOrNull() ?: 0f
            return min * manipulation.intensity } //Minimalwert wird auf einen Wert über 0 gesetzt
    },
    5 Usages
    TRUNCATED_CATEGORY_AXIS { //Verkürzte Kategorie-Achse(X bei Bar, Y bei HorizontalBar)
        override fun categoryRange(size: Int, manipulation: Manipulation): IntRange? =
            manipulation.categoryRange //berechnet eingegrenzten Kategorie Bereich
    },
```

Über manipulation.intensity wird gesichert, das die Manipulation immer unterschiedlich stark ist.

```
fun manipulationIntensity(type: ManipulationType, random: Random): Float =
    when (type) {
        ManipulationType.TRUNCATED_VALUE_AXIS -> random.nextFloat() * 0.13f + 0.80f // Achse beginnt bei 80-98% des Minimalwerts
```

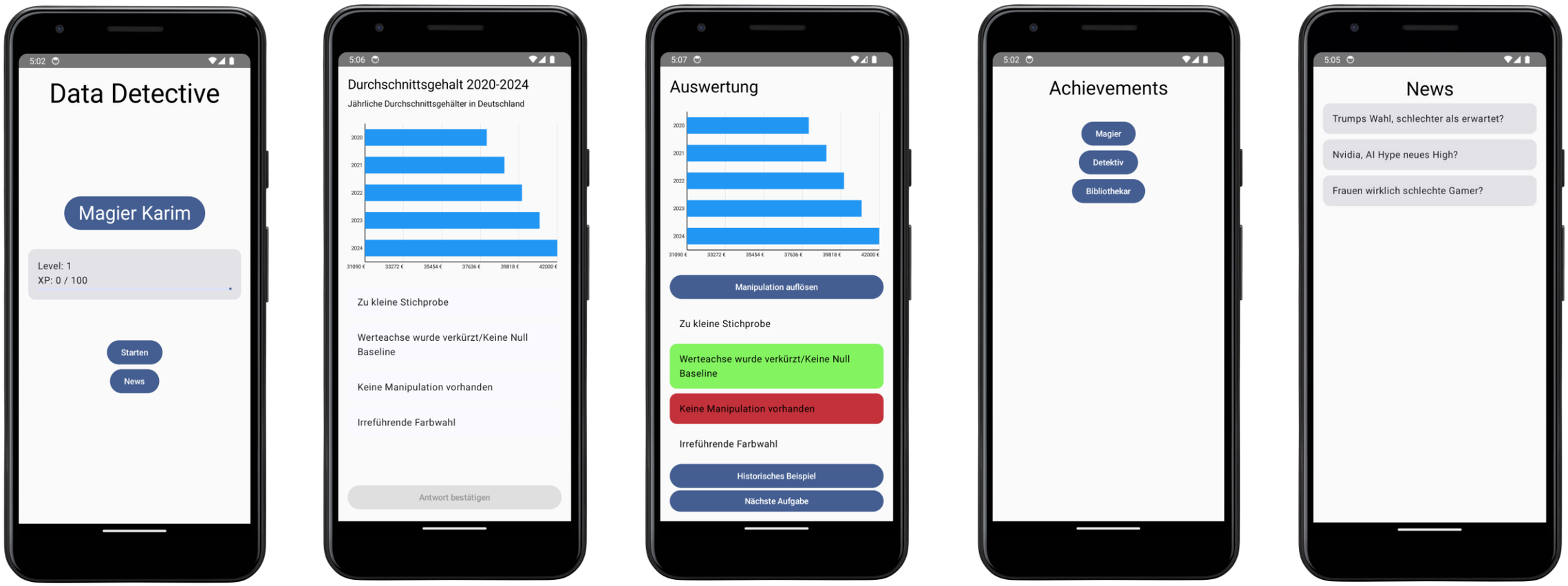
Über manipulation.categoryRange wird gesichert, dass die Größe des Kategoriebereichs immer zufällig ist.

```
fun categoryRange(size: Int, random: Random): IntRange {
    //Wert für Sichtbaren Ausschnitt für 40-70 % der Kategorien
    val windowSize = (size * (random.nextFloat() * 0.3f + 0.4f)).toInt().coerceAtLeast( minimumValue = 2)
    // Zufälliger Startpunkt
    val start = random.nextInt( from = 0,  until = size - windowSize + 1)

    return start ≤ until < (start + windowSize)
}
```

Data Detective Rapid Protype

(Placeholder für den Foliensatz)



Main | **Aufgabe** | **Auflösung** | **Achievements (Titel)** | **News**

Weitere Planung (Audit 4)

Gamification:
- Ausbau des XP/Level Systems
- Erste Achievements und dessen Titel
- Playercard/Skill-Graph (zusätzlicher Screen) zur Einsicht des Spielverhaltens
- Vervollständigungsgraphen

- Expertentest und Evaluation mit anschließender Iteration der Gamification abhängig vom Feedback

Anwendungslogik:
- Algorithmische Kategorisierung
- Algorithmische Erstellung der Aufgaben
- Implementation der wichtigsten Statistik-Methoden mit dessen Auflösung

- Persistente Speicherung der Nutzerdaten