Eggventure

Technical White Paper

Kerem Gürbüz: 587049
Angel Mechkarov: 586684

10.07.2025

GitHub:
https://github.com/angelmech/eggventure
Präsentation:
https://mediathek.htw-berlin.de/video/mobile-betriebssysteme-und-netzwerke-eggventure-app/4401e692c0c721d650a2e804243079a1

Inhaltsverzeichnis

1. Projektbeschreibung

Die Applikation kombiniert einen klassischen Lauftracker mit Videospiel-Elementen, um Bewegung im Alltag durch Gamification zu fördern. Je mehr Nutzer:innen laufen, desto mehr unterschiedliche Kreaturen schalten sie frei. Diese Kreaturen werden in einer Sammlung dargestellt.

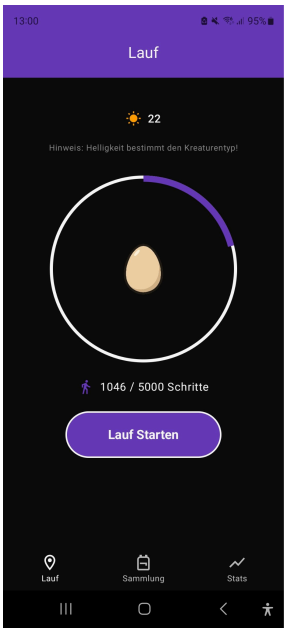
2. Benutzeroberfläche

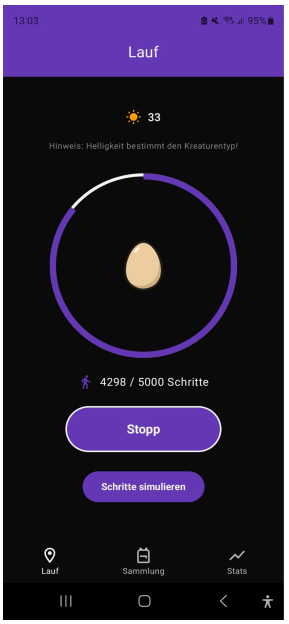
Die Benutzeroberfläche ist in drei Tabs unterteilt.

2.1 Lauf

Der Lauf-Tab dient dazu, Läufe per Knopfdruck zu starten und zu beenden. Der Fortschritt bis zur nächsten Kreatur wird in einem kreisförmigen Fortschrittsbalken dargestellt.

Nutzer:innen können ebenfalls sehen, welche Helligkeit die Applikation misst in Lux. Der Nutzen dieser Messungen ist in den Use-Cases beschrieben.

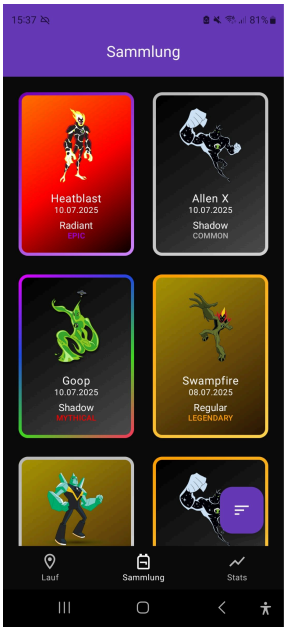


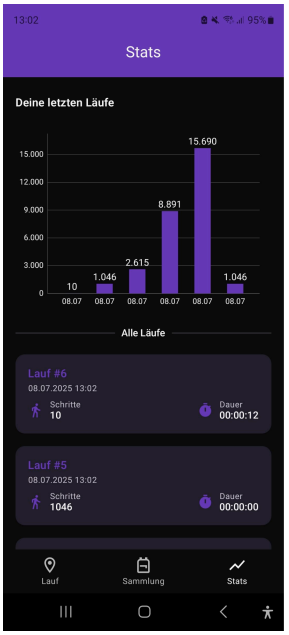


2.2 Sammlung

Die Sammlung zeigt alle vom Nutzer gesammelten Kreaturen an, gemeinsam mit relevanten Daten, wie der Seltenheit, dem Sammeldatum, und dem Typen einer Kreatur.

Die Kreaturen können ebenfalls nach ihrem Sammeldatum, ihrer Seltenheit, ihrem Namen und ihrem Typ sortiert werden.





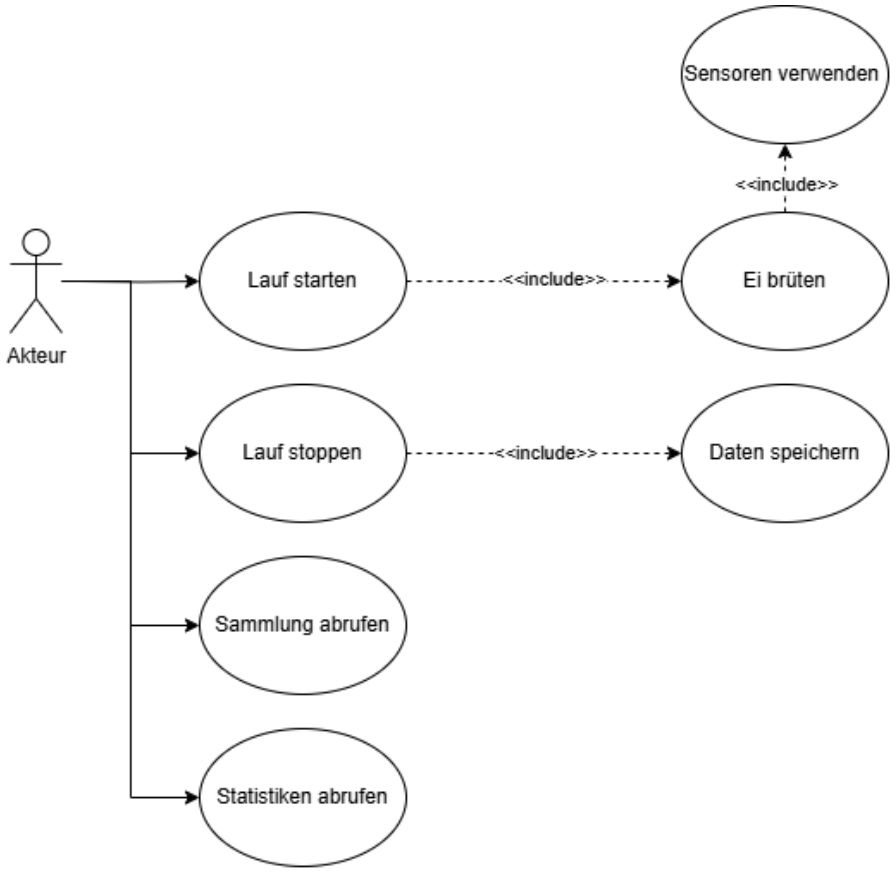
2.3 Stats

Die Stats bieten eine Ansicht zu vergangenen Läufen, inklusive ihrem Datum, ihrer Dauer und der gemessenen Schrittzahl.

Zudem beinhalten sie eine graphische Darstellung der letzten 7 Läufe, welche mit der Bibliothek MPAndroidChart realisiert wurde.

3. Verwendung der Applikation

3.1 Use-Case Diagramm



3.2 Use-Cases

Lauf Starten und Daten speichern

Nutzer:innen starten den Schrittzähler mit einem Buttonklick. Damit wird die Sensorfunktion initiiert, welche die gelaufenen Schritte zählt. Während des Laufes werden Schritte und Laufdauer protokolliert.

Ei brüten

Nachdem Nutzer:innen eine vorgegebene Distanz erreichen, wird eine neue Kreatur freigeschaltet und in einer Datenbank gespeichert. Diese Kreaturen können unterschiedliche Seltenheiten als Eigenschaft haben.

Sensoren verwenden

Beim Ausbrüten eines Eis werden die Daten aus dem Helligkeitssensor gelesen und beeinflussen, welche Kreatur schlüpft. Kreaturen sind einem der drei Typen Shadow, Regular und Radiant zugewiesen und können nur schlüpfen, wenn sie in der jeweiligen Helligkeit ausgebrütet werden.

Lauf Stoppen und Daten speichern

Nutzer:innen beenden schließlich den Lauf durch einen Buttonklick. Hiermit ist der Lauf beendet und die erfassten Daten werden in einer Datenbank gespeichert.

Sammlung abrufen
Nutzer:innen können ihre gesammelten Kreaturen in einer Sammlung betrachten. Kreaturen haben Eigenschaften wie Name, Seltenheit, Typ und Datum.

Statistiken abrufen
Nutzer:innen können vergangene Läufe in einer Liste einsehen. Dargestellt sind ebenfalls Details zu dem Lauf wie die Anzahl der Schritte, Laufdauer, Datum und Uhrzeit.

4. Technologie

Sensoren
Zur Erfassung der Aktivitäten wird in erster Linie der Schrittzähler des Geräts genutzt (Sensor-Typ: `TYPE_STEP_COUNTER`). Die Berechtigung zur Verwendung des Sensors wird bei der ersten Betätigung des Lauf Starten Buttons eingeholt.

Zudem wird der Lichtsensor (`TYPE_LIGHT`) des Geräts verwendet, um den Typ der ausgebrüteten Kreatur zu beeinflussen. Hierbei wird permanent während der Nutzung der Applikation die Helligkeit gemessen, um den Sensor zu kalibrieren. Eine einmalige Messung, etwa nur beim Brüten, führt sonst zu ungenauen Werten. Auf diese Weise wird die reale Umgebung von Nutzer:innen auf spielerische Weise in die Entwicklung der Kreatur integriert.

Datenbanken
Zur persistenten Speicherung der Fortschritte (z.B. Kreaturen, Schritte, Läufe) wird eine relationale Datenbank genutzt. Erfasste Daten werden hierbei lokal auf dem Gerät gespeichert. Die Datenspeicherung wird mit Hilfe der von Android bereitgestellten Room-Persistenzbibliothek realisiert, die als Abstraktionsschicht über SQLite fungiert.

Technisch gesehen besteht Room aus drei Hauptkomponenten:

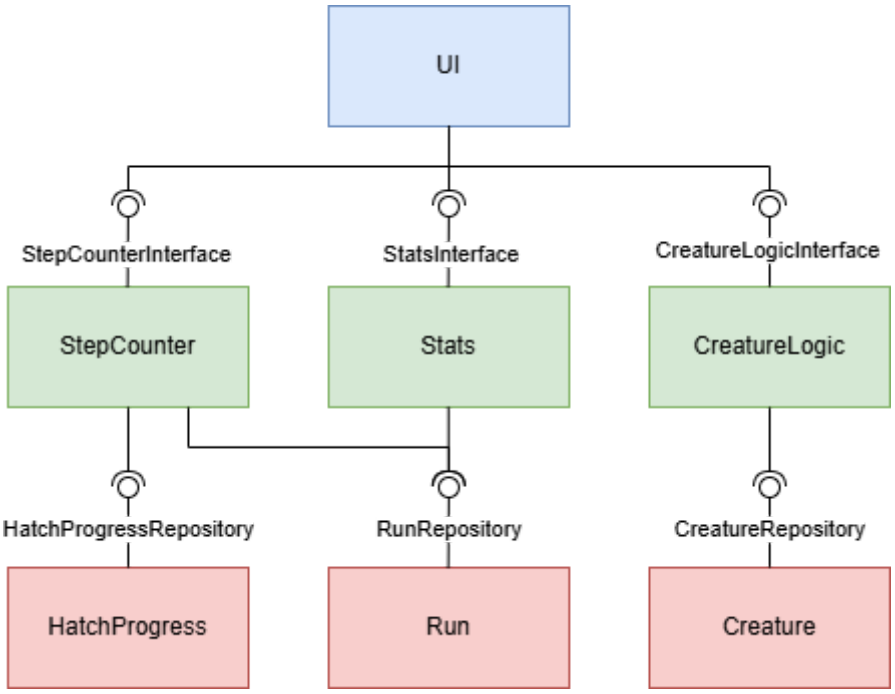
- Entity-Klassen, welche die Tabellen in der Datenbank repräsentieren. Jede Instanz einer Entity-Klasse entspricht hierbei einer Zeile in einer Tabelle.
- Data Access Objects (DAOs), die die Methoden für den Datenbankzugriff enthalten (z.B. `@Insert`, `@Update`, `@Delete` und `@Query` für benutzerdefinierte SQL-Abfragen).
- Die Database Klasse, die als Hauptzugangspunkt zur Datenbank dient (mit `@Database` annotiert). Die Bereitstellung einer einzigen Instanz dieser Datenbank für die gesamte Anwendung (als Singleton) wird dabei über ein `companion object` in `AppDatabase` gewährleistet.

Room bietet eine Compile-Time-Überprüfung von SQL-Abfragen, was Fehler reduziert. Zusätzlich werden alle Datenbankoperationen asynchron in einem Hintergrund-Thread ausgeführt, um die Main-UI-Thread nicht zu blockieren.

5. Komponenten

Die Applikation ist nach dem MVVM-Architekturstil konzipiert und nutzt Jetpack Compose für den Aufbau der Benutzeroberfläche. Dieser Ansatz fördert eine klare Trennung der Verantwortlichkeiten der Komponenten, was die Wartbarkeit, Testbarkeit und Skalierbarkeit der Anwendung erheblich verbessert. Die MVVM-Architektur unterteilt die Anwendung in drei Hauptkomponenten: Model, View und ViewModel, welche in dem Komponentendiagramm farblich gekennzeichnet sind.

5.1 Komponentendiagramm



5.2 Komponentenbeschreibung

1. View (UI)

Die UI-Komponente stellt das User Interface der Anwendung dar und ist mit Jetpack Compose implementiert. Sie ist für die Anzeige der Daten und die Verarbeitung von Benutzerinteraktionen verantwortlich. Als View im MVVM-Muster beobachtet die UI-Komponente die Daten, die von den ViewModels bereitgestellt werden, und aktualisiert sich, wenn sich diese Daten ändern.

2. Model

Die Model-Komponenten sind für die Bereitstellung und Persistenz von Daten verantwortlich. Die Datenpersistenz wird hierfür durch die Room Bibliothek realisiert.

Die HachtProgress-Komponente repräsentiert den aktuellen Fortschritt eines Ausbrütungsprozesses. Sie bietet das HatchProgressRepository-Interface an.

Die Run-Komponente speichert die Daten zu einzelnen Läufen, wie die Schrittzahl, die Laufdauer, das Datum und die gelaufene Distanz. Sie stellt das RunRepository-Interface bereit.

Die Creature-Komponente stellt die Sammlung bereits ausgebrüteter Kreaturen dar. Hierzu werden Kreatureigenschaften wie der Name, das Freischaltungsdatum, der Typ und die Seltenheit der einzelnen Kreaturen gespeichert.

3. ViewModel

Die Viewmodel-Komponenten agieren als Vermittler zwischen der UI und dem Model. Sie stellen den Status der Benutzeroberfläche dar und beinhalten die dazu notwendige Geschäftslogik. Zudem stellen sie Factories bereit, über die sie aufgerufen werden.

Die StepCounter-Komponente nutzt Gerätesensoren zur Erfassung gelaufener Schritte und deren Verarbeitung im Kontext des Brütfortschritts. Bei Erreichen eines definierten Schrittziels löst sie ein Hatch-Ereignis aus, setzt den Fortschritt zurück und übergibt den aktuellen Umgebungslichtwert zur Weiterverarbeitung an die CreatureLogic-Komponente. Ferner stellt die Komponente der Benutzeroberfläche (UI) relevante Zustandsdaten wie die aktuelle Schrittzahl und den Brütfortschritt zur visuellen Repräsentation bereit.

Die Stats-Komponente ist für die Sammlung und Aufbereitung der Statistikdaten zuständig und stellt zudem Methoden zur Formatierung dieser Daten für die Darstellung auf der Statistiken Seite bereit.

Die CreatureLogic-Komponente verwaltet die Sammlung der Kreaturen, die Nutzer:innen während der Verwendung der Applikationen ausbrüten. Sie implementiert die Ausbrütungslogik, welche Seltenheit und Typ basierend auf probabilistischen Modellen und Umweltparametern wie Licht festlegt. Zudem ist sie für die Sortierungsalgorithmen der Kreaturensammlung zuständig.

6. Qualitätssicherung

6.1 UI-Tests

Ziel

UI-Tests prüfen, ob die App aus Sicht der Benutzeroberfläche korrekt funktioniert. Dabei geht es um den realistischen Ablauf von Interaktionen während der Nutzung der Applikation, wie die korrekte Reaktion der UI auf Buttonklicks.

Testmethodik

Ein Großteil der Funktionalität wurde zunächst manuell durch visuelle Kontrolle während der Entwicklung überprüft. Dabei wurden gezielt häufige Nutzungsszenarien, wie das Schlüpfen eines Eis, das Hinzufügen neuer Kreaturen zur Sammlung oder die Navigation zwischen den Hauptbildschirmen, mehrfach getestet.
Ergänzend kommen automatisierte UI-Tests mit dem Jetpack Compose UI Testing-Framework zum Einsatz. Diese simulieren gezielte Benutzerinteraktionen und validieren die korrekte Reaktion der Benutzeroberfläche. Da die App vollständig mit Jetpack Compose entwickelt wurde, sind diese Tests für das Projekt optimal.

UI-Szenarien und Testfälle

Fortschrittsanzeige & Schritte: Die UI zeigt während eines Laufs die aktuelle Schrittzahl und eine Fortschrittsleiste an. Getestet wird, ob sich beides bei simulierten und echten Schritten korrekt aktualisiert. Sobald ein Ei schlüpft, muss der Fortschritt zurückgesetzt und eine neue Kreatur in der Sammlung eingeblendet werden.

Buttons und Klickverhalten: UI-Tests stellen sicher, dass Buttons wie „Lauf Starten“ oder „Schritte simulieren“ zuverlässig reagieren. Ein Klick startet/beendet Läufe oder simuliert Schritte. Es wird auch geprüft, ob der Button-Zustand (aktiv/inaktiv) zur App-Logik passt.

Kreaturen-Sammlung: Nach einem erfolgreichen Hatch muss die neue Kreatur korrekt in der Sammlung angezeigt werden. Die Liste soll vollständig und aktuell bleiben. Auch Sortieroptionen (z. B. nach Datum oder Seltenheit) werden getestet, sowohl funktional (siehe Unit-Tests) als auch visuell.

Statistiken: Die Stats Seite zeigt die vergangenen Läufe. UI-Tests prüfen, ob diese Daten korrekt aktualisiert und dargestellt werden – auch nach mehreren Hatch-Events oder App-Neustarts.

Navigation: Die Navigation per Bottom Bar muss zuverlässig zwischen den Ansichten (Lauf, Sammlung, Stats) funktionieren. Tests prüfen, ob UI-Zustände beim Wechsel erhalten bleiben.

Kompatibilität testen

Es wird geprüft, ob die Anwendung auf unterschiedlichen Bildschirmgrößen und Geräteklassen konsistent funktioniert. Hierbei werden verschiedene physische Geräte sowie der Android Emulator genutzt.

Tests mit Compose UI Testing APIs

Testname	Beschreibung	Implementierung
`startButton_toggleTracking_showsCorrectText`	Testet, ob sich der Buttontext korrekt zwischen "Lauf Starten" und "Stopp" ändert, wenn man auf ihn klickt.	- UI-Test mit Compose - Aufruf: .performClick() auf Button - Assertion: .onNodeWithText("Stopp").assertIsDisplayed()
`stepProgress_displaysCorrectSteps`	Überprüft, ob der Fortschritts-Text wie z.B. "100 / 5000 Schritten" korrekt angezeigt wird.	- UI-Test mit Compose - Assertion: .onNodeWithText("523 / 5000 Schritten", substring = true).assertIsDisplayed()
`fakeStepButton_visibilityDependsOnTrackingState`	Testet Sichtbarkeit und Aktivierungsstatus des "Schritte simulieren"-Buttons abhängig vom Tracking-Zustand.	- Initial: .assertIsNotEnabled() - Klick: .onNodeWithText("Lauf Starten").performClick() - Danach: .assertIsEnabled()
`navigationBar_allScreensNavigable`	Testet Navigation über die BottomBar zu den drei Hauptscreens ("Lauf", "Sammlung", "Stats").	- Mehrere .performClick() auf onNodeWithText() - Danach .assertIsDisplayed() auf jeweilige Views
`progressBar_updatesOnStepIncrease`	Überprüft, ob die Schrittanzeige nach Klick auf „Schritte simulieren“ korrekt aktualisiert wird.	- Klick: .performClick() auf "Lauf Starten" - Dann auf "Schritte simulieren" - Assertion: .onNodeWithText("523 / 5000 Schritten").assertIsDisplayed()

6.2 Integrationstests

Ziel

Integrationstests prüfen, ob die verschiedenen Teile der App zusammen richtig funktionieren. In Eggventure bedeutet das vor allem, dass das Zusammenspiel zwischen dem StepCounter ViewModel und der Room-Datenbank überprüft wird, um sicherzustellen, dass Daten richtig erfasst, gespeichert und angezeigt werden.

Prüfung der Schrittzählung und Datenpersistenz

Ein wichtiger Punkt ist, dass die Schritte korrekt gezählt und an das Repository weitergegeben werden. Wenn genug Schritte gesammelt wurden, soll das Ei schlüpfen. Die Tests überprüfen, ob dann der Fortschritt zurückgesetzt wird und die Datenbank entsprechend aktualisiert wird.

LiveData-Aktualisierung prüfen
Die Integrationstests kontrollieren, ob das ViewModel bei Datenänderungen die LiveData-Objekte richtig aktualisiert. Dadurch wird sichergestellt, dass die UI bei Änderungen, wie etwa dem Schlüpfen eines Eis, automatisch mit aktuellen Daten versorgt wird.

Umgang mit Fehlern
Es wird auch getestet, wie die App reagiert, wenn es Probleme gibt, etwa wenn die Datenbank nicht erreichbar ist oder Daten fehlen. In solchen Fällen soll die App stabil bleiben und zum Beispiel sinnvolle Default-Werte verwenden, statt abzustürzen.

Beispiele Integrationstests

Testname	Beschreibung	Implementierung / Tool
`stepIncreasesAndTriggersHatchEvent`	Prüft, ob Schritte korrekt gezählt werden und LiveData-Beobachter benachrichtigt werden.	- Integrationstest - ViewModel: StepCounter - LiveData + Mock-Observer - Assertion: observer.onChanged(2)
`hatchResetsProgressOnSuccess`	Prüft, ob bei erfolgreichem Hatch das Hatch-Progress zurückgesetzt wird (stepCount = 0).	- Integrationstest - ViewModel: StepCounter - Mock: hatchCreature() - Assertion: stepCount == 0
`hatchCreatureReturnsTrueWhenSuccess`	Prüft, ob CreatureLogic.hatchCreature() true zurückgibt, wenn ein CreatureEntity erfolgreich erstellt wird.	- Integrationstest - Klasse: CreatureLogic - Mock: EggHatchEvent.processHatchEvent() - Assertion: Assert.assertTrue(result)
`hatchCreatureReturnsFalseWhenNoCreature`	Prüft, ob CreatureLogic.hatchCreature() false zurückgibt, wenn kein Creature-Objekt erzeugt wird (null).	- Integrationstest - Klasse: CreatureLogic - Mock: EggHatchEvent.processHatchEvent() mit null-Rückgabe - Assertion: Assert.assertFalse(result)

6.3 Instrumented Tests

Instrumentierte Tests werden auf einem echten Android-Gerät oder Emulator ausgeführt. Ihr Fokus liegt darauf, Android-spezifische Systemfunktionen und Framework-Komponenten unter realen Bedingungen zu überprüfen. Dazu zählen beispielsweise der Zugriff auf Sensoren, der Umgang mit dem Android-Context, Lifecycle-Callbacks, LiveData-Verhalten oder die Persistenz über Room.

Ziele von Instrumented Tests

- **Test von Android-spezifischen APIs und Framework-Komponenten:**
  Validierung des Zugriffs auf Sensoren, Context-Objekte, Lifecycle-Methoden, LiveData, Datenbankoperationen (Room) etc. unter realen Android-Bedingungen.

- **Validierung hardware-nahen Verhaltens:**
  Simulation und Verarbeitung von Sensorereignissen (z. B. Schrittzähler, Lichtsensor) sowie Überprüfung der korrekten Reaktion der App darauf.

- **Sicherstellung von UI- und OS-Interaktionen:**
  Überprüfung, ob LiveData korrekt beobachtet wird, BroadcastReceiver ordnungsgemäß empfangen werden oder Services richtig gestartet und gestoppt werden.

- **Test unter echten Geräte- und Emulatorbedingungen:**
  Ausgleich der Unterschiede zwischen der JVM (Unit-/Integrationstests) und dem realen Android-System (Context, Coroutines, Ressourcen, Persistenz).

Typische Anwendungsfälle

Sensorverhalten (Schrittzähler, Lichtsensor): Überprüfung, ob die App korrekt auf echte Sensorsignale reagiert. Dies erfolgt sowohl automatisiert in instrumentierten Tests als auch manuell mittels Log-Ausgaben auf realen Geräten und Emulatoren.

Lifecycle- und Context-bezogene Funktionen: Sicherstellung, dass Lifecycle-Callbacks korrekt ausgelöst werden und Kontext-abhängige Operationen, wie z. B. Datenbankzugriffe oder Service-Interaktionen, unter realen Bedingungen funktionieren.

Persistenzprüfung mit Room: Automatisierte sowie manuelle (mit Android Studio App Inspection) Überprüfung, ob Daten unter realen Bedingungen korrekt gespeichert, geladen oder zurückgesetzt werden.

UI-Tests und Integrationstests mit Android-Komponenten: Viele instrumentierte Tests überschneiden sich mit UI- und Integrationstests, da sie ebenfalls auf realen Geräten laufen und häufig das Zusammenspiel von UI und Systemkomponenten prüfen. So werden etwa Nutzerflüsse getestet, die Sensorinput, UI-Interaktionen und Datenpersistenz gleichzeitig betreffen.

Beispiele Instrumented Tests

Testname	Beschreibung	Implementierung
`testStepCountingAndHatchingOnDevice`	Prüft, ob nach 5000 Schritten ein Hatch-Event ausgelöst wird und der Schrittzähler zurückgesetzt wird.	- Verwendung von LiveData-Beobachtung unter Android, - echte ViewModel-Logik - Ausführung mit Android-Context
`testStepCountIncreasesCorrectly`	Validiert, ob der Schrittzähler korrekt auf Schrittzugaben reagiert.	- Beobachtung von LiveData in Android-Umgebung - Ausführung mit echtem Lifecycle/Looper über InstantTaskExecutorRule
`simulateStepSensorTrigger_updatesStepCount`	Simuliert Sensorereignisse und prüft, ob diese den Schrittzähler beeinflussen.	- Indirekte Nutzung des Sensorsystems durch abstrahierte StepSensorManager, LiveData, Android-Kontext
`simulateLightSensorInput_affectsHatch`	Überprüft, ob Lichtwerte aus dem Lichtsensor Einfluss auf das Hatch-Verhalten haben.	- Nutzung eines gefälschten EnvironmentSensorManager, der Flow liefert - Test läuft unter echtem Android-Context

6.4 End-to-end Tests

End-to-End Tests überprüfen den vollständigen Ablauf einer Nutzerinteraktion vom UI bis zur Datenpersistenz. Sie stellen sicher, dass alle Komponenten der Anwendung zusammen korrekt funktionieren. Diese Abläufe werden manuell getestet.

In der App Eggventure könnte ein typischer End-to-End-Test folgendermaßen aussehen:

- Die App wird geöffnet und ein Lauf wird gestartet.
- Es werden Schritte simuliert (oder tatsächlich gegangen).
- Während des Laufs wird auf den Statistik- oder Sammlungsbildschirm gewechselt.
- Zwischendurch wird ein Ei ausgebrütet (Art der Kreatur wurde durch Lichteinfluss vom Lichtsensor bestimmt)
- Die geschlüpfte Kreatur wird korrekt in der Sammlung angezeigt.
- Anschließend wird der Lauf gestoppt.
- Der Statistikbildschirm zeigt die aufgezeichneten Daten korrekt an.

Durch diesen Ablauf wird überprüft, ob UI, Business-Logik, Sensorverarbeitung und Datenspeicherung korrekt funktionieren und miteinander kommunizieren, unter realistischen Bedingungen.

6.5 Funktionalitätstests (Unit-Tests)

Die Einzelfunktionen der App-Logik werden isoliert überprüft, insbesondere die Datenverarbeitung, Statusverwaltung und das Event Handling innerhalb des ViewModels.
Hierzu wird vorrangig JUnit zum Testen verwendet. Des Weiteren wurde auch AndroidX Test (Sammlung von Jetpack-Bibliotheken) benutzt.
In Kotlin ist es üblich, Unit-Tests beschreibende und ausdrucksstarke Namen zu geben, die das zu testende Verhalten klar wiedergeben. Zudem werden Android-spezifische Logs, die zum Debuggen während der Entwicklung genutzt wurden, aus dem Code entfernt, da JUnit-Tests auf der JVM laufen und sonst Fehler auftreten beim Testen.

Komponente: `StepCounter` (ViewModel)
Ziel: Die Tests prüfen die zentrale Logik zur Schrittzählung, Fortschrittsverfolgung und Hatch-Erkennung im StepCounter-ViewModel. Dabei werden sowohl reale als auch simulierte Schritte berücksichtigt, Hatch-Events verarbeitet, der Tracking-Zustand verwaltet und Lichtwerte für Kreaturentypen einbezogen. Zusätzlich wird sichergestellt, dass Fortschritte korrekt gespeichert und zurückgesetzt sowie unerwünschte Zustände (z. B. negative Schritte, doppelte Listener) vermieden werden. Die Tests decken typische Abläufe ebenso wie Grenzfälle ab.

Testname	**Ziel**	**Testimplementierung (Mocks and Asserts)**
`initProgress creates new progress if none exists`	Prüft, ob ein neuer Fortschritt erstellt wird, wenn keiner existiert	- Mock: coEvery { hatchProgressRepository.getLastHatchProgress() } returns null - Mock: coEvery { hatchProgressRepository.insertProgress(any()) } just Runs - Aufruf: viewModel.initProgress() - Assert: Assert.assertEquals(0, viewModel.stepCount.value)
`initProgress uses last saved progress if available`	Prüft, ob gespeicherter Fortschritt korrekt geladen wird	- Mock: coEvery { hatchProgressRepository.getLastHatchProgress() } returns entity - Aufruf: viewModel.initProgress() - Assert: Assert.assertEquals(100, viewModel.stepCount.value)
`startTracking registers listener and resets tracking`	Prüft, ob der Listener registriert und Tracking korrekt gestartet wird	- Aufruf: viewModel.startTracking() - Assert: Assert.assertTrue(viewModel.isTracking.value == true) - Verify: verify { stepSensorManager.registerListener(any()) }
`stopTracking unregisters listener and saves run`	Prüft, ob der Listener entfernt und der Lauf gespeichert wird	- Aufruf: viewModel.stopTracking() - Assert: Assert.assertFalse(viewModel.isTracking.value!!) - Verify: verify { stepSensorManager.unregisterListener() }

		- coVerify: coVerify { runPersistence.saveRun(any(), any(), any()) }
`addFakeStep below hatchGoal updates stepCount`	Prüft, ob Schritte korrekt addiert und gespeichert werden	- Mock: coEvery { hatchProgressRepository.getLastHatchProgress() } returns entity - Mock: coEvery { hatchProgressRepository.updateHatchProgress(any(), any()) } just Runs - Aufruf: viewModel.addFakeStep(50) - Assert: Assert.assertEquals(50, viewModel.stepCount.value)
`addFakeStep triggers hatch when reaching goal`	Prüft, ob bei Erreichen des Ziels ein Hatch ausgelöst wird	- Mock: coEvery { hatchProgressRepository.getLastHatchProgress() } returns entity - Mock: coEvery { creatureLogic.hatchCreature(...) } returns true - Aufruf: viewModel.addFakeStep(1) - Assert: Assert.assertEquals(0, viewModel.stepCount.value) - Assert: Assert.assertTrue(viewModel.eggHatched.value == true)
`onSensorChanged updates step count and triggers hatch when goal is reached`	Prüft, ob Schritte korrekt gezählt und ein Hatch bei Ziel erreicht wird	- Mock: coEvery { creatureLogic.hatchCreature(...) } returns true - Aufruf: viewModel.onStepSensorDataChanged(StepSensorData(totalSteps = 5001)) - Assert: Assert.assertEquals(0, viewModel.stepCount.value) - Assert: Assert.assertTrue(viewModel.eggHatched.value == true)
`startTracking does not register listener again if already tracking`	Prüft, ob Mehrfach-Registrierungen vermieden werden	- Aufrufe: viewModel.startTracking() zweimal - Verify: verify(exactly = 1) { stepSensorManager.registerListener(any()) }
`addFakeStep does not exceed hatch goal`	Prüft, ob Schritte nicht über das Ziel hinausgehen	- Mock: coEvery { hatchProgressRepository.getLastHatchProgress() } returns entity (Fortschritt nahe Ziel) - Aufruf: viewModel.addFakeStep(200) - Assert: Assert.assertEquals(5000, viewModel.stepCount.value)
`onSensorChanged ignores non-step-counter events`	Prüft, ob irrelevante Sensorwerte ignoriert werden	- Aufruf: viewModel.onStepSensorDataChanged(StepSensorData(totalSteps = 0)) - Assert: Assert.assertEquals(false,

		viewModel.eggHatched.value)
`onCleared unregisters sensor listener`	Prüft, ob der Listener beim Aufräumen entfernt wird	- Aufruf: viewModel.onCleared() - Verify: verify { stepSensorManager.unregisterListener() }
`light level falls back to 0 if null`	Prüft, ob bei fehlendem Lichtsensorwert korrekt 0 verwendet wird	- Mock: every { environmentSensorManager.observeLight() } returns flowOf(0f) - Mock: coEvery { creatureLogic.hatchCreature(..., 0f) } returns true - Aufruf: viewModel.addFakeStep(1) - Assert: Assert.assertTrue(viewModel.eggHatched.value == true)
`initProgress handles repository error gracefully`	Prüft, ob Fehler beim Initialisieren abgefangen werden	- Mock: coEvery { hatchProgressRepository.getLastHatchProgress() } throws Exception() - Aufruf: viewModel.initProgress() - Assert: Assert.assertEquals(0, viewModel.stepCount.value)
`light sensor emits correct value`	Prüft, ob der Lichtwert korrekt gelesen wird	- Assert: Assert.assertEquals(120f, viewModel.currentLightLevel.value)
`hatch resets progress after successful hatch`	Prüft, ob Fortschritt nach einem erfolgreichen Hatch zurückgesetzt wird	- Mock: coEvery { creatureLogic.hatchCreature(...) } returns true - Aufruf: viewModel.addFakeStep(1) - Assert: stepCount.value == 0 - Assert: eggHatched.value == true

Komponente: `CreatureLogic` (ViewModel)
Ziel: Die Tests überprüfen umfassend die Kernlogik rund um das Schlüpfen von Kreaturen sowie den Umgang mit Sensorwerten. Dabei wird sichergestellt, dass Kreaturen nur dann schlüpfen, wenn die erforderliche Schrittzahl erreicht wurde, dass der Fortschritt korrekt gespeichert und bei Erfolg zurückgesetzt wird. Ebenso prüfen die Tests die korrekte Bestimmung des Kreaturentyps anhand von Lichtwerten, inklusive der Handhabung von Grenzfällen und fehlenden Werten. Zudem wird die Kreaturen Liste auf Sortierfunktionen für die Anzeige der Kreaturen in der Sammlung getestet.

Testname	**Ziel**	**Testimplementierung (Mocks and Asserts)**
`hatchCreature calls processHatchEvent and returns true when hatched`	Prüft, ob processHatchEvent aufgerufen wird und ein Wesen erfolgreich geschlüpft wird	- processHatchEvent(...) gibt ein CreatureEntity zurück - Aufruf: creatureLogic.hatchCreature(...) - Assert: Assert.assertTrue(result) - Verify: coVerify { eggHatchEventMock.processHatchEvent(...) }
`hatchCreature returns false when no creature hatched`	Prüft, ob bei nicht erreichter Schrittanzahl false zurückgegeben wird	- processHatchEvent(...) gibt null zurück - Aufruf: creatureLogic.hatchCreature(...) - Assert: Assert.assertEquals(false, result)
`lastLightLevel is updated correctly`	Prüft, ob der Lichtwert intern korrekt aktualisiert wird	- Zugriff auf privates Field _lastLightLevel - Setze Wert: stateFlow.value = 42.0f - Assert: Assert.assertEquals(42.0f, creatureLogic.lastLightLevel.value)
`processHatchEvent updates progress and returns null if currentSteps below goal`	Prüft, dass Fortschritt gespeichert und kein Wesen geschlüpft wird, wenn Ziel nicht erreicht	- updateHatchProgress(1, 3000) wird aufgerufen - Assert: Assert.assertNull(result) - Verify: coVerify { hatchProgressRepository.updateHatchProgress(...) }
`determineType returns SHADOW for light below 700`	Prüft, ob Lichtwert kleiner als 700 zum Typ SHADOW führt	- Reflection-Aufruf der Methode - Assert: Assert.assertEquals(SHADOW, type)
`determineType returns REGULAR for light between 700 and 10000`	Prüft, ob Lichtwert zwischen 700 und 10000 zum Typ REGULAR führt	Assert: Assert.assertEquals(REGULAR, type)
`determineType returns RADIANT for light above 10000`	Prüft, ob Lichtwert größer als 10000 zum Typ RADIANT führt	- Assert: Assert.assertEquals(RADIANT, type)

`pickCreatureBasedOnRarityAndType falls back to any creature if none match type`	Prüft, ob ein Fallback-Wesen geliefert wird, wenn kein Typ-Match vorhanden ist	- DataInterface liefert nur 1 Creature ohne Typ-Match - Assert: Assert.assertEquals("Fallback", result.name)
`rarityRoll always returns valid Rarity`	Prüft, ob rarityRoll() immer eine gültige Rarity liefert	- 100x Invocation - Assert: Rarity.values().contains(result)
`processHatchEvent works with null lightLevel`	Prüft, ob auch ohne Lichtwert ein Wesen geschlüpft werden kann	- Assert: Assert.assertNotNull(result)
`default sort mode is DEFAULT and preserves order`	Prüft, dass der Standard-Sortiermodus DEFAULT ist und die Sortierung die ursprüngliche Reihenfolge beibehält	- Assert: sortMode.value == DEFAULT - Assert: sort(creatures) == creatures
`toggleRaritySort toggles between BY_RARITY and DEFAULT`	Prüft, dass toggleRaritySort() zwischen BY_RARITY und DEFAULT wechselt	- Nach erstem Aufruf: sortMode.value == BY_RARITY - Nach zweitem Aufruf: sortMode.value == DEFAULT
`sort sorts by rarity when in BY_RARITY mode`	Prüft, dass die Sortierung nach Seltenheit funktioniert, wenn Modus BY_RARITY ist	- sortMode wird auf BY_RARITY gesetzt - Assert: Ergebnis ist Liste sortiert nach rarity
`resetSort sets mode to DEFAULT`	Prüft, dass resetSort() den Modus zurück auf DEFAULT setzt	- Nach Aufruf: Assert: sortMode.value == DEFAULT

Komponente: `Stats` (ViewModel)
Das StatsViewModel ist für das Bereitstellen und Formatieren von Laufstatistiken verantwortlich. Dazu zählt das Auslesen vergangener Läufe aus dem RunRepository sowie die Aufbereitung von Dauer- und Zeitstempeln für die Anzeige in der Benutzeroberfläche. Die folgenden Unit-Tests prüfen zentrale Funktionen des ViewModels, insbesondere die Datenformatierung und die korrekte Initialisierung der StateFlows bei Verwendung von gemockten Repository-Daten.

Testname	**Ziel**	**Testimplementierung (Mocks and Asserts)**
`formatDuration_returnsCorrectFormat`	Prüft, ob formatDuration eine Zeit korrekt in das Format hh:mm:ss umwandelt.	- Input: 3605000 (1h, 0m, 5s) - Assert: formatDuration(3605000) == "01:00:05"
`formatDate_returnsCorrectDateString`	Prüft, ob ein Timestamp korrekt formatiert wird.	- Input: 0L - Assert: formatDate(0L) enthält "1970" oder "01.01.1970" je nach Locale
`getAllRuns_emitsCorrectRuns`	Validiert, ob allRuns korrekt mit Werten aus runRepository.getAllRuns() befüllt wird.	- Mock: runRepository.getAllRuns() gibt flowOf(testRuns) zurück - Assert: allRuns.value == testRuns
`getLastRun_emitsCorrectLastRun`	Prüft, ob lastRun korrekt initialisiert wird.	- Mock: runRepository.getLastRun() gibt testRuns.last() - Assert: lastRun.value == testRuns.last()
`getLast7Runs_emitsCorrectSubset`	Validiert, ob die letzten 7 Läufe korrekt gesetzt werden.	- Mock: runRepository.getLast7Runs() gibt testRuns - Assert: last7Runs.value == testRuns

7. Schnittstellen

7.1 Model

```
/**
* Repository interface for managing hatch progress data.
*/
interface HatchProgressRepository {
    /**
     * Inserts a new hatch progress entry into the data source.
     *
     * @param progress The [HatchProgressEntity] to be inserted.
     */
    suspend fun insertProgress(progress: HatchProgressEntity)

    /**
     * Retrieves the current accumulated steps for the ongoing hatch
progress.
     *
     * @return The current number of steps, or null if no progress is found.
     */
    suspend fun getHatchProgressSteps(): Int?

    /**
     * Updates an existing hatch progress entry with new accumulated steps.
     *
     * @param id The ID of the hatch progress entry to update.
     * @param stepsAccumulated The new total accumulated steps.
     */
    suspend fun updateHatchProgress(id: Int, stepsAccumulated: Int)
}


/**
* Repository interface for managing creature data.
*/
interface CreatureRepository {
    /**
     * Inserts a new creature into the data source.
     *
     * @param creatures The [CreatureEntity] to be inserted.
     */
    suspend fun insertCreature(creatures: CreatureEntity)

    /**
     * Retrieves all creatures from the data source.
     *
     * @return A list of all [CreatureEntity] objects.
     */
    suspend fun getAllCreatures(): List<CreatureEntity>
}
```

```kotlin
/**
* Repository interface for managing run data.
*/
interface RunRepository {
    /**
     * Inserts a new run into the data source.
     *
     * @param run The [RunEntity] to be inserted.
     */
    suspend fun insertRun(run: RunEntity)

    /**
     * Retrieves all runs from the data source as a Flow.
     * The Flow will emit a new list of runs whenever the underlying data
changes.
     *
     * @return A Flow emitting a list of [RunEntity] objects.
     */
    fun getAllRuns(): Flow<List<RunEntity>>

    /**
     * Retrieves the most recent run from the data source.
     *
     * @return The last [RunEntity], or null if no runs are found.
     */
    suspend fun getLastRun(): RunEntity?

    /**
     * Retrieves the last 7 runs from the data source as a Flow.
     *
     * @return A Flow emitting a list of the last 7 [RunEntity] objects.
     */
    fun getLast7Runs(): Flow<List<RunEntity>>
}
```

7.2 ViewModel

```
/**
* Interface for managing step counting and tracking egg hatching progress.
*/
interface StepCounterInterface {
    /**
     * The current step count.
     * @return A LiveData object that holds the current step count as an
Integer.
     */
    val stepCount: LiveData<Int>

    /**
     * Indicates whether the step tracking is currently active.
     * @return A LiveData object that holds a Boolean value; true if
tracking is active, false otherwise.
     */
    val isTracking: LiveData<Boolean>

    /**
     * Indicates whether the egg has hatched.
     * @return A LiveData object that holds a Boolean value; true if the egg
has hatched, false otherwise.
     */
    val eggHatched: LiveData<Boolean>

    /**
     * Initializes the progress, loading any previously saved state.
     */
    fun initProgress()

    /**
     * Starts the step tracking service.
     */
    fun startTracking()

    /**
     * Stops the step tracking service.
     */
    fun stopTracking()

    /**
     * ONLY FOR PROGRESS TESTING PURPOSES
     * Adds a specified number of fake steps to the current step count.
     * @param fakeSteps The number of fake steps to add.
     */
    fun addFakeStep(fakeSteps: Int)
}
```

```kotlin
/**
* Interface for managing and retrieving statistics related to runs.
*/
interface StatsInterface {
    /**
     * Retrieves all recorded runs.
     * @return A StateFlow emitting a list of RunEntity objects.
     */
    val allRuns: StateFlow<List<RunEntity>>

    /**
     * Retrieves the most recent run.
     * @return A StateFlow emitting the last RunEntity object, or null if no
runs have been recorded.
     */
    val lastRun: StateFlow<RunEntity?>

    /**
     * Retrieves the last 7 recorded runs.
     * @return A StateFlow emitting a list of the last 7 RunEntity objects.
     */
    val last7Runs: StateFlow<List<RunEntity>>

    /**
     * Formats the duration from milliseconds into a human-readable format
(hh:mm:ss).
     * @param durationMillis The duration in milliseconds.
     * @return A formatted string representing the duration.
     */
    fun formatDuration(durationMillis: Long): String

    /**
     * Formats the timestamp into a readable date string.
     * @param timestamp The timestamp in milliseconds.
     * @param dateOnly Whether to only include the date (default: false).
     */
    fun formatDate(timestamp: Long, dateOnly: Boolean = false): String
}
```

```
/**
* Interface for managing the logic related to creatures in the application.
*/
interface CreatureLogicInterface {


    /**
     * sorts list of owned creatures sorted by rarity.
     */
    fun toggleSortByRarity()

    /**
     * Resets the sorting of creatures to the default order.
     */
    fun resetSort()


    /**
     * Initiates the hatching process for a creature.
     *
     * @param creatureId The ID of the creature to hatch.
     * @param hatchProgressSteps The number of steps required to complete
the hatching process.
     * @param hatchGoal The goal for the hatching process, typically the
number of steps needed to hatch the creature.
     * @param totalSteps The total number of steps taken by the user, which
may be used to track progress.
     * @param light The current light level, which may influence the
hatching process or the type of creature hatched.
     *
     * @return True if the hatching process was successfully initiated,
false otherwise.
     */
    suspend fun hatchCreature(
        creatureId: Int,
        totalSteps: Int,
        hatchProgressSteps: Int,
        hatchGoal: Int,
        light: Float?): Boolean
}
```

7.3 Utils

```
/**
* Interface for managing step sensor events.
*/
interface StepSensorManager {
    /**
     * Registers a listener for step sensor events.
     *
     * @param listener The listener to register.
     */
    fun registerListener(listener: SensorEventListener)

    /**
     * Unregisters the previously registered listener.
     */
    fun unregisterListener()
}



/* * Interface for managing environment sensors like light.
* This interface defines methods to start and stop reading sensor
values.
*/
interface EnvironmentSensorManager {
    /**
     * Stops reading sensor values.
     * This method should be called when the sensor readings are no
longer needed to free up resources.
     */
    fun stopReading()


    /**
     * Observes the light sensor values as a Flow.
     * This method allows continuous observation of light sensor
values.
     *
     * @return A Flow emitting light sensor values.
     */
    fun observeLight(): Flow<Float>
}
```