POLITECHNIKA RZESZOWSKA
KATEDRA INFORMATYKI I AUTOMATYKI
DR INŻ. MATEUSZ POMIANEK

ĆWICZENIE LABORATORYJNE – Testy CI/CL

GitHub · Testowanie · CI/CD z GitHub Actions

Programowanie Aplikacji Mobilnych (Android / Kotlin / Jetpack Compose)

Spis treści

1. Git i GitHub: podstawy profesjonalnego workflow
2. Tworzenie i konfiguracja publicznego repozytorium
3. Struktura projektu Android / Kotlin i konwencje kodu
4. Testy jednostkowe z JUnit 5 i MockK
5. Testowanie coroutines i Flow
6. Testy integracyjne: Room, Retrofit i Hilt
7. Testy interfejsu użytkownika z Espresso / Compose UI Test
8. Analiza statyczna: Detekt, ktlint i Android Lint
9. Pokrycie kodu: JaCoCo
10. GitHub Actions: kompletny potok CI/CD
11. Badges, szablony PR i ochrona gałęzi
12. Zadania do wykonania
13. Kryteria oceniania
14. Najczęstsze błędy i ich rozwiązania

1. Git i GitHub: podstawy profesjonalnego workflow

Git to rozproszony system kontroli wersji. GitHub to platforma hostująca repozytoria Git oraz dodająca narzędzia pracy zespołowej: pull requesty, recenzje kodu, issues, wiki i GitHub Actions. Rozróżnienie jest istotne: Git działa lokalnie bez GitHuba, natomiast GitHub jest usługą opartą na repozytoriach Git.

Analogia

Git jest dziennikiem pokładowym statku: zapisuje każdą zmianę kursu, autora i czas. GitHub jest portem macierzystym: przechowuje kopię dziennika, umożliwia wspólną pracę oraz organizuje odprawę jakościową przed wejściem kodu do gałęzi main.

1.1. Model gałęzi: GitHub Flow

W projektach mobilnych skuteczny jest prosty model: main zawsze gotowa do wydania, krótkotrwałe gałęzie funkcjonalne i obowiązkowy pull request przed scaleniem.

Gałąź	Cel	Kto tworzy / scala	Zasada
main	Kod stabilny. Kompiluje się i przechodzi testy.	Scalanie przez PR	Nigdy nie wykonuj bezpośredniego push do main.
feature/nazwa	Pojedyncza funkcja lub poprawka.	Deweloper	Nazwy: feature/weather-search, fix/crash-on-startup.
hotfix/nazwa	Krytyczna poprawka produkcyjna.	Senior / maintainer	Tylko dla błędów blokujących użytkownika.
develop	Opcjonalna gałąź integracyjna.	Zespół	Używana częściej w Gitflow niż w GitHub Flow.

1.2. Conventional Commits

Conventional Commits porządkują historię zmian i umożliwiają automatyczne generowanie CHANGELOG oraz wersjonowanie semantyczne.

```
# Format: <typ>(<zakres>): <opis>
# Typy: feat, fix, docs, style, refactor, test, chore, perf, ci

git commit -m "feat(weather): dodaj wyszukiwanie miast po nazwie"
git commit -m "fix(api): napraw parsowanie temperatury ujemnej"
git commit -m "test(viewmodel): dodaj testy jednostkowe HomeViewModel"
git commit -m "ci: skonfiguruj GitHub Actions z JUnit i JaCoCo"

git commit -m "feat(api)!: zmień format odpowiedzi WeatherApi"
# Wykrzyknik oznacza breaking change.
```

Automatyczna weryfikacja

Workflow może odrzucać PR z niepoprawnym tytułem albo commitami typu „fix2”, „poprawka”, „aaa”. Dzięki temu historia repozytorium pozostaje czytelna przez cały semestr.

2. Tworzenie i konfiguracja publicznego repozytorium

2.1. Inicjalizacja repozytorium

```
# GitHub > New repository
# Nazwa: WeatherApp-Android
# Widoczność: Public
# .gitignore: Android
# Licencja: Apache 2.0

git clone https://github.com/TWOJ_LOGIN/WeatherApp-Android.git
cd WeatherApp-Android

git config --global user.name "Jan Kowalski"
git config --global user.email "jan.kowalski@edu.pl"
```

2.2. .gitignore dla Androida

Poprawny .gitignore zapobiega dodaniu plików generowanych, lokalnych ścieżek SDK i sekretów. Szczególnie istotny jest local.properties oraz pliki keystore.

```
*.iml
.gradle/
/local.properties
```

```
/.idea/
/*/build/

*.jks
*.keystore
key.properties
google-services.json
local.defaults.properties

*.apk
*.aab
*.ap_
```

2.3. README.md

README powinien pozwolić nowej osobie uruchomić projekt i testy w krótkim czasie. Minimalny zakres to: opis, wymagania, uruchomienie, testy, architektura i status CI.

```
# WeatherApp

[![CI](https://github.com/LOGIN/WeatherApp-Android/actions/workflows/ci.yml/badge.svg)](...)
[![Coverage](https://codecov.io/gh/LOGIN/WeatherApp-Android/branch/main/graph/badge.svg)](...)

Aplikacja pogodowa pobierająca dane z Open-Meteo API.

## Wymagania
- Android Studio
- JDK 17
- Android SDK API 34+

## Testy
```bash
./gradlew test
./gradlew connectedAndroidTest
./gradlew detekt
./gradlew jacocoTestReport
```
```

2.4. Szablony issues i pull requestów

```
mkdir -p .github/ISSUE_TEMPLATE

# .github/ISSUE_TEMPLATE/bug_report.md
---
name: Zgłoszenie błędu
about: Poinformuj o problemie, aby pomóc nam go naprawić
labels: bug
---

## Opis błędu
## Kroki do odtworzenia
## Oczekiwane zachowanie
## Aktualne zachowanie
## Środowisko

# .github/pull_request_template.md
## Opis zmian
## Powiązany issue
Closes #

## Checklist
- [ ] Kod kompiluje się bez ostrzeżeń
- [ ] Dodano lub zaktualizowano testy
- [ ] ./gradlew test przechodzi lokalnie
- [ ] Detekt i ktlint nie zgłaszają nowych problemów
```

3. Struktura projektu na Androida i konwencje kodu

Projekt WeatherApp stosuje MVVM z elementami Clean Architecture. Kierunek zależności jest jednokierunkowy: UI zna Domain, Domain nie zna UI ani Data, Data implementuje interfejsy zdefiniowane w Domain. Taki podział upraszcza testowanie i wymianę implementacji.

```
app/
├── src/main/kotlin/pl/edu/pam/weather/
│   ├── ui/
│   │   ├── screens/home/HomeScreen.kt
│   │   ├── screens/home/HomeViewModel.kt
│   │   └── components/WeatherCard.kt
│   ├── domain/
│   │   ├── model/WeatherData.kt
│   │   ├── repository/WeatherRepository.kt
│   │   └── usecase/GetWeatherUseCase.kt
│   ├── data/
│   │   ├── remote/WeatherApi.kt
│   │   ├── remote/dto/WeatherDto.kt
│   │   ├── local/WeatherDatabase.kt
│   │   └── WeatherRepositoryImpl.kt
│   └── di/AppModule.kt
├── src/test/kotlin/...          # testy JVM
└── src/androidTest/kotlin/...   # testy instrumentowane
```

3.1. Zależności testowe

```
dependencies {
    testImplementation("org.junit.jupiter:junit-jupiter:5.10.2")
    testImplementation("io.mockk:mockk:1.13.10")
    testImplementation("org.jetbrains.kotlinx:kotlinx-coroutines-test:1.8.0")
    testImplementation("app.cash.turbine:turbine:1.1.0")
    testImplementation("com.google.truth:truth:1.4.2")

    androidTestImplementation("androidx.test.ext:junit:1.1.5")
    androidTestImplementation("androidx.test.espresso:espresso-core:3.5.1")
    androidTestImplementation("androidx.compose.ui:ui-test-junit4")
    androidTestImplementation("com.google.dagger:hilt-android-testing:2.51")
    androidTestImplementation("androidx.room:room-testing:2.6.1")
}

tasks.withType<Test> {
    useJUnitPlatform()
}
```

4. Testy jednostkowe z JUnit 5 i MockK

Test jednostkowy sprawdza pojedynczą klasę lub funkcję w izolacji od zależności. W projekcie Android typowo testuje się use case, ViewModel, mapery DTO oraz walidację danych.

> **Analogia**
> Test jednostkowy przypomina test części samochodu poza samochodem. Silnik, hamulce i skrzynia biegów można sprawdzić osobno, zanim złożymy cały system w testach integracyjnych.

Element	Znaczenie
@ExtendWith(MockKExtension::class)	Integracja MockK z JUnit 5.
@MockK	Mock wymagający jawnego zdefiniowania zachowania.
@RelaxedMockK	Mock zwracający wartości domyślne.
coEvery { ... } returns ...	Definicja zachowania funkcji suspend.

coVerify { ... }	Weryfikacja wywołania funkcji suspend.
assertThat(...).isEqualTo(...)	Czytelne asercje Google Truth.

4.1. Przykład: GetWeatherUseCaseTest

```
@ExtendWith(MockKExtension::class)
class GetWeatherUseCaseTest {

    @MockK
    private lateinit var repository: WeatherRepository

    private lateinit var useCase: GetWeatherUseCase

    @BeforeEach
    fun setUp() {
        useCase = GetWeatherUseCase(repository)
    }

    @Test
    fun `returns weather data for valid city`() = runTest {
        val expected = WeatherData(
            city = "Warszawa",
            temperature = 22.5,
            humidity = 65,
            windSpeed = 12.0,
            weatherCode = 1,
        )
        coEvery { repository.getWeather("Warszawa") } returns Result.success(expected)

        val result = useCase("Warszawa")

        assertThat(result.isSuccess).isTrue()
        assertThat(result.getOrNull()).isEqualTo(expected)
        coVerify(exactly = 1) { repository.getWeather("Warszawa") }
    }

    @Test
    fun `returns failure when repository throws exception`() = runTest {
        coEvery { repository.getWeather(any()) } throws IOException("Brak sieci")

        val result = useCase("Gdańsk")

        assertThat(result.isFailure).isTrue()
        assertThat(result.exceptionOrNull()).isInstanceOf(IOException::class.java)
    }
}
```

4.2. Test mapowania DTO

```
@Test
fun `maps dto to domain model correctly`() {
    val dto = WeatherDto(
        currentWeather = CurrentWeatherDto(
            temperature = 22.5,
            windspeed = 12.0,
            weathercode = 1,
        )
    )

    val result = dto.toDomain(cityName = "Warszawa")

    assertThat(result.city).isEqualTo("Warszawa")
    assertThat(result.temperature).isWithin(0.001).of(22.5)
    assertThat(result.windSpeed).isWithin(0.001).of(12.0)
    assertThat(result.weatherCode).isEqualTo(1)
}
```

5. Testowanie coroutines i Flow

Kod asynchroniczny wymaga kontroli wirtualnego czasu oraz obserwacji emisji. kotlinx-coroutines-test dostarcza runTest, StandardTestDispatcher i advanceUntilIdle, a Turbine upraszcza testowanie Flow.

```
@Test
fun `uiState shows loading then success after search`() = runTest {
    val weatherData = WeatherData(city = "Poznań", temperature = 18.0)
    coEvery { getWeather("Poznań") } returns Result.success(weatherData)

    viewModel.searchCity("Poznań")

    assertThat(viewModel.uiState.value).isInstanceOf(UiState.Loading::class.java)
    advanceUntilIdle()

    val state = viewModel.uiState.value
    assertThat(state).isInstanceOf(UiState.Success::class.java)
    assertThat((state as UiState.Success).data.city).isEqualTo("Poznań")
}
```

5.1. Turbine

```
@Test
fun `favorites flow emits updated list after add`() = runTest {
    val repository = FakeWeatherRepository()

    repository.getFavorites().test {
        assertThat(awaitItem()).isEmpty()

        repository.addFavorite("Łódź")

        val updated = awaitItem()
        assertThat(updated).hasSize(1)
        assertThat(updated.first().city).isEqualTo("Łódź")

        cancelAndIgnoreRemainingEvents()
    }
}
```

Obiekt testowy	Opis	Kiedy stosować
Mock	Obiekt generowany przez bibliotekę, konfigurowany przez every/coEvery.	Gdy weryfikujesz interakcje.
Stub	Zależność zwracająca proste, z góry ustalone odpowiedzi.	Gdy nie interesuje Cię liczba wywołań.
Fake	Ręczna, uproszczona implementacja interfejsu.	Gdy potrzebny jest stan i bardziej realistyczne zachowanie.
Spy	Prawdziwy obiekt częściowo śledzony lub podmieniany.	Rzadko; zwykle przy testach kodu legacy.

6. Testy integracyjne: Room, Retrofit i Hilt

Testy integracyjne sprawdzają współpracę komponentów: DAO z bazą w pamięci, Retrofit z lokalnym serwerem HTTP oraz Hilt z testowymi modułami zależności.

6.1. Room DAO

```
class WeatherDaoTest {
    private lateinit var database: WeatherDatabase
    private lateinit var dao: WeatherDao
```

```
    @BeforeEach
    fun setUp() {
        val context = ApplicationProvider.getApplicationContext<Context>()
        database = Room.inMemoryDatabaseBuilder(context, WeatherDatabase::class.java)
            .allowMainThreadQueries()
            .build()
        dao = database.weatherDao()
    }

    @AfterEach
    fun tearDown() {
        database.close()
    }

    @Test
    fun `insert and retrieve weather entity`() = runTest {
        val entity = WeatherEntity(city = "Białystok", temperature = 15.2)

        dao.insertWeather(entity)

        val result = dao.getWeatherForCity("Białystok")
        assertThat(result).isNotNull()
        assertThat(result!!.temperature).isWithin(0.001).of(15.2)
    }
}
```

6.2. Retrofit + MockWebServer

```
@Test
fun `parses successful weather response correctly`() = runTest {
    mockWebServer.enqueue(
        MockResponse()
            .setResponseCode(200)
            .setBody("""
                { "current": {
                    "temperature_2m": 22.5,
                    "relative_humidity_2m": 65,
                    "wind_speed_10m": 12.0,
                    "weather_code": 1
                } }
            """.trimIndent())
    )

    val result = api.getCurrentWeather(52.23, 21.01, "temperature_2m")

    assertThat(result.current.temperature2m).isWithin(0.001).of(22.5)
    assertThat(result.current.relativeHumidity2m).isEqualTo(65)
}
```

7. Testy interfejsu użytkownika z Espresso / Compose UI Test

Testy UI uruchamiają aplikację na emulatorze lub urządzeniu. Są najdroższe w utrzymaniu, dlatego powinny sprawdzać krytyczne przepływy użytkownika, a nie każdą drobną gałąź logiki.

```
@HiltAndroidTest
@RunWith(AndroidJUnit4::class)
class HomeScreenTest {

    @get:Rule(order = 0)
    val hiltRule = HiltAndroidRule(this)

    @get:Rule(order = 1)
    val composeRule = createAndroidComposeRule<MainActivity>()

    @Before
    fun setUp() {
        hiltRule.inject()
```

```
    }

    @Test
    fun searchCity_displaysWeatherCard() {
        with(composeRule) {
            onNodeWithText("Wpisz nazwę miasta").performClick()
            onNodeWithText("Wpisz nazwę miasta").performTextInput("Rzeszów")
            onNodeWithContentDescription("Szukaj").performClick()

            onNodeWithText("Rzeszów").assertIsDisplayed()
            onNodeWithText("20.5°C").assertIsDisplayed()
        }
    }
}
./gradlew connectedAndroidTest
./gradlew connectedAndroidTest \
  -Pandroid.testInstrumentationRunnerArguments.class=pl.edu.pam.weather.ui.HomeScreenTest
```

8. Analiza statyczna: Detekt, ktlint i Android Lint

Analiza statyczna sprawdza kod bez jego uruchamiania. Detekt wykrywa problemy jakościowe, ktlint pilnuje formatowania, a Android Lint sprawdza problemy specyficzne dla platformy Android.

Narzędzie	Co sprawdza	Kiedy uruchamiać
Detekt	Złożoność, długie metody, duplikację, magiczne liczby, martwy kod.	Przed commitem i w CI.
ktlint	Formatowanie, importy, wcięcia, styl Kotlin.	Lokalnie przez ktlintFormat; w CI przez ktlintCheck.
Android Lint	Uprawnienia, zasoby, przestarzałe API, problemy manifestu.	W Android Studio i w CI.

```
detekt {
    config.setFrom("$rootDir/config/detekt/detekt.yml")
    buildUponDefaultConfig = true
    autoCorrect = false
}

# config/detekt/detekt.yml
complexity:
  LongMethod:
    threshold: 60
  CyclomaticComplexMethod:
    threshold: 15
style:
  MaxLineLength:
    maxLineLength: 120
```

8.1. Pre-commit hook

```
#!/bin/sh
set -e

echo "Uruchamianie Detekt i ktlintCheck..."
./gradlew detektMain
./gradlew ktlintCheck

echo "Weryfikacja przeszła pomyślnie."
```

9. Pokrycie kodu: JaCoCo

JaCoCo mierzy, które instrukcje, linie, metody i gałęzie zostały wykonane przez testy. Pokrycie jest wskaźnikiem pomocniczym: wysokie pokrycie nie gwarantuje jakości testów, ale niskie pokrycie wskazuje obszary ryzyka.

```
tasks.register<JacocoReport>("jacocoTestReport") {
    dependsOn("testDebugUnitTest")

    reports {
        xml.required.set(true)
        html.required.set(true)
        csv.required.set(false)
    }

    val excludes = listOf(
        "**/di/**",
        "**/*_Factory*",
        "**/*_HiltModules*",
        "**/R.class",
        "**/R$*.class",
        "**/BuildConfig.*",
    )

    sourceDirectories.setFrom(files("src/main/kotlin", "src/main/java"))
    executionData.setFrom(fileTree(layout.buildDirectory) {
        include("**/*.exec", "**/*.ec")
    })
}
```

Metryka	Co mierzy	Cel
Instruction Coverage	Instrukcje bajtkodu wykonane przez testy.	> 70% dla domain/usecase.
Branch Coverage	Gałęzie if/else/when wykonane przez testy.	> 60% dla logiki warunkowej.
Line Coverage	Linie kodu uruchomione w testach.	> 70% dla klas domenowych.
Method Coverage	Metody wywołane przez testy.	> 80% dla logiki biznesowej.

10. GitHub Actions: kompletny potok CI/CD

GitHub Actions automatyzuje kontrolę jakości: analizę statyczną, testy jednostkowe, raport pokrycia oraz testy instrumentowane. Workflow powinien działać dla push i pull_request.

```
name: CI - Testy i analiza statyczna

on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]
  workflow_dispatch:

concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true

env:
  JAVA_VERSION: '17'

jobs:
  static-analysis:
    name: Detekt + ktlint + Android Lint
```

```
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-java@v4
        with:
          java-version: ${{ env.JAVA_VERSION }}
          distribution: temurin
          cache: gradle
      - run: chmod +x gradlew
      - run: ./gradlew detektMain detektTest --continue
      - run: ./gradlew ktlintCheck
      - run: ./gradlew lint

  unit-tests:
    name: Testy jednostkowe + JaCoCo
    runs-on: ubuntu-latest
    needs: static-analysis
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-java@v4
        with:
          java-version: ${{ env.JAVA_VERSION }}
          distribution: temurin
          cache: gradle
      - run: chmod +x gradlew
      - run: ./gradlew testDebugUnitTest --continue
      - run: ./gradlew jacocoTestReport
      - uses: codecov/codecov-action@v4
        with:
          token: ${{ secrets.CODECOV_TOKEN }}
          files: app/build/reports/jacoco/jacocoTestReport/jacocoTestReport.xml
          fail_ci_if_error: true

  instrumented-tests:
    name: Testy UI na emulatorze
    runs-on: ubuntu-latest
    needs: unit-tests
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-java@v4
        with:
          java-version: ${{ env.JAVA_VERSION }}
          distribution: temurin
          cache: gradle
      - run: chmod +x gradlew
      - uses: reactivecircus/android-emulator-runner@v2
        with:
          api-level: 34
          arch: x86_64
          disable-animations: true
          script: ./gradlew connectedDebugAndroidTest --stacktrace
```

10.1. Walidacja pull requesta

```
name: Walidacja Pull Requesta

on:
  pull_request:
    types: [opened, edited, synchronize]

jobs:
  validate-pr-title:
    runs-on: ubuntu-latest
    steps:
      - uses: amannn/action-semantic-pull-request@v5
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        with:
          types: |
            feat
            fix
            docs
```

```
            style
            refactor
            test
            chore
            ci
            perf
          requireScope: false
```

11. Badges, szablony PR i ochrona gałęzi

Badges są dynamicznymi elementami README pokazującymi stan projektu. Ochrona gałęzi main wymusza proces code review i przejście testów przed scaleniem.

```
[![CI](https://github.com/USER/REPO/actions/workflows/ci.yml/badge.svg?branch=main)](...)
[![Coverage](https://codecov.io/gh/USER/REPO/graph/badge.svg)](...)
[![License: Apache 2.0](https://img.shields.io/badge/License-Apache%202.0-blue.svg)](...)
[![Kotlin](https://img.shields.io/badge/Kotlin-2.0-7F52FF.svg?logo=kotlin)](...)
[![Min SDK](https://img.shields.io/badge/Android-API%2026%2B-brightgreen?logo=android)](…)
```

Ustawienie	Co robi
Require a pull request before merging	Blokuje bezpośredni push do main.
Require approvals: 1	Wymaga co najmniej jednej akceptacji.
Require status checks to pass	Wymaga zielonych jobów CI.
Require branches to be up to date	Wymaga aktualizacji gałęzi PR z main.
Require conversation resolution	Wymaga rozwiązania komentarzy w PR.

```
# .github/CODEOWNERS
app/src/main/kotlin/  @prowadzacy-login
.github/workflows/    @prowadzacy-login @student-login
*                     @prowadzacy-login
```

12. Zadania do wykonania

Zadania budują kompletne repozytorium projektu WeatherApp w Kotlinie z testami oraz automatyzacją CI/CD. Realizuj je w podanej kolejności.

Weryfikacja

Przedstaw repozytorium, wynik komend Gradle, screenshot zielonego workflow oraz raport pokrycia.

Zadanie 1: Repozytorium i konwencje

1. Załóż publiczne repozytorium WeatherApp-Android z licencją Apache 2.0 i szablonem .gitignore dla Androida.
2. Skonfiguruj .gitignore tak, aby ignorował local.properties, pliki keystore, build/ i artefakty APK/AAB.
3. Przygotuj README.md z opisem, wymaganiami, instrukcją uruchomienia, testami i miejscem na badges.
4. Dodaj szablony issue i pull requesta.
5. Włącz Branch Protection dla main: PR, jedna akceptacja i status checks.

Zadanie 2: Testy na wszystkich poziomach

6. Dodaj zależności testowe: JUnit 5, MockK, coroutines-test, Turbine i Truth.
7. Napisz minimum 8 testów jednostkowych dla use case, mapperów i logiki ulubionych miast.
8. Napisz minimum 3 testy coroutines / Flow z runTest i Turbine.
9. Napisz minimum 3 testy integracyjne Room DAO.
10. Napisz minimum 4 testy UI na emulatorze z testowymi zależnościami Hilt.

Zadanie 3: GitHub Actions CI/CD

11. Utwórz ci.yml z jobami static-analysis, unit-tests i instrumented-tests.
12. Skonfiguruj Detekt, ktlint i Android Lint.
13. Skonfiguruj JaCoCo i wysyłkę raportu XML do Codecov.
14. Dodaj workflow walidacji PR z Conventional Commits.
15. Zaktualizuj README o faktyczne badges oraz liczbę testów.

Zadanie 4: Jakość kodu i dokumentacja

16. Skonfiguruj pre-commit hook uruchamiający Detekt i ktlint.
17. Dodaj CHANGELOG.md zgodny z Keep a Changelog.
18. Skonfiguruj Dependabot dla Gradle i GitHub Actions.
19. Na podstawie raportu JaCoCo popraw pokrycie co najmniej jednej klasy poniżej 50%.

13. Kryteria oceniania

Zadanie	Punkty	Co weryfikuje prowadzący
Repozytorium	20	Publiczne repo, README, szablony, Branch Protection, pierwszy PR.
Testy	30	Minimum 14 testów JVM i 4 testy UI. Wszystkie przechodzą.
CI/CD	30	Trzy zielone joby, raport JaCoCo, walidacja PR, badges.
Jakość i dokumentacja	20	Pre-commit hook, CHANGELOG, Dependabot i poprawa pokrycia.
Razem	100	Ocena końcowa zgodnie z progami.

Ocena	Punkty / wymagania
5.0	90–100 pkt; wszystkie zadania, pokrycie > 75%, min. 25 testów.
4.5	80–89 pkt; zadania 1–3 kompletne, pokrycie > 65%, min. 20 testów.
4.0	70–79 pkt; repozytorium, testy i podstawowy CI.
3.5	60–69 pkt; zadania 1–2 i częściowy CI.
3.0	50–59 pkt; repozytorium, minimum 6 testów JVM i podstawowy workflow.
2.0	0–49 pkt; brak repozytorium, zbyt mało testów lub CI nie uruchamia się.

14. Najczęstsze błędy i ich rozwiązania

Błąd / objaw	Przyczyna i rozwiązanie
local.properties trafił do repozytorium	Usuń z indeksu: git rm --cached local.properties, dodaj wpis do .gitignore i wy-

	konaj commit.
No tests found for given includes	Brakuje useJUnitPlatform() albo importujesz @Test z JUnit 4 zamiast JUnit 5.
coEvery unresolved reference	Brakuje importu io.mockk.coEvery albo funkcja nie jest suspend.
Android Gradle plugin requires Java 17	Ustaw java-version: 17 w setup-java i sprawdź lokalnie ./gradlew --version.
Raport JaCoCo ma 0%	Sprawdź dependsOn(testDebugUnitTest), executionData i katalogi z klasami Kotlin.
Detekt blokuje MagicNumber	Wyodrębnij stałe albo dopisz uzasadnione wartości do ignoreNumbers.
Test UI nie widzi elementu	Sprawdź wstrzykiwanie Hilt, stabilność identyfikatorów semantics i oczekiwanie na stan UI.
Permission denied przy ./gradlew	Dodaj chmod +x gradlew w workflow albo zatwierdź uprawnienie wykonania w repozytorium.
Codecov badge pokazuje unknown	Sprawdź CODECOV_TOKEN i ścieżkę do raportu XML.
Turbine nie otrzymuje elementu	Flow nie emituje wartości; zweryfikuj kolejność akcji i użycie runTest.

Przydatne komendy Gradle

```
./gradlew test
./gradlew testDebugUnitTest
./gradlew test --tests 'pl.edu.pam.weather.usecase.GetWeatherUseCaseTest'
./gradlew connectedAndroidTest
./gradlew detektMain
./gradlew ktlintCheck
./gradlew ktlintFormat
./gradlew lint
./gradlew jacocoTestReport
./gradlew clean
```

Podsumowanie: ćwiczenie koncentruje się na inżynierii oprogramowania: poprawnej historii repozytorium, testowalnej architekturze, automatyzacji kontroli jakości i przejrzystej dokumentacji.