Module 7 Critical Thinking Assignment: Unit Converter Testing App

Alexander Ricciardi

Colorado State University Global

CSC475: Platform-Based Development

Professor Herbert Pensado

March 30, 2025

Module 7 Critical Thinking Assignment: Unit Converter Testing App

This documentation is part of the Module 7 Critical Thinking Assignment from CSC475: Platform-Based Development at Colorado State University Global. The documentation provides an overview of the Android application's functionality and testing scenarios, including the application's Kotlin code source overview and output screenshots. It also reflects on the obstacles faced during the application's development and the skills acquired. The application is coded in Kotlin 2.0.21 and is named "Unit Converter Testing App."

The Assignment Direction:

Option #1: "Unit Converter Testing"
Challenge: Create a unit conversion app that converts between different units (e.g., temperature, length, weight). Write and execute unit tests using the Android Testing Framework to ensure the accuracy of the conversion calculations.
Please ensure that your submission includes the following components:

- Source code file(s) containing the program implementation.
- A 1-page paper explaining the program's purpose, the obstacles faced during its development, and the skills acquired. The paper should also include screenshots showcasing the successful execution of the program.
- Compile and submit your pseudocode, source code, and screenshots of the application executing the application, the results and GIT repository in a single document.

Program Description:

The program is a small Android app that allows a user to convert

- Temperatures from Celsius to Fahrenheit, and vice versa.
- Length from meters to feet and from kilometers to miles, and vice versa.
- Weight from kilograms to pounds, and vice versa.

The app code also provides unit tests using the Android Testing Framework to ensure the accuracy of the conversion calculations. Note that the main purpose of this project is to demonstrate the implementation of unit testing in Android Studio using JUnit and Hamcrest.

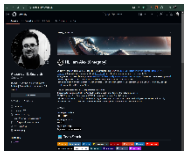
⚠️ My notes:

The application is developed using Kotlin 2.0.21 and the following:

- Jetpack Compose (composeBom = "2024.09.00"): UI
- JUnit (4): Unit Tests
- Hamcrest (1.3): Unit Tests Hamcrest assertions

Git Repository:

This is a picture of my GitHub page:



I use GitHub as my Distributed Version Control System (DVCS), the following is a link to my GitHub, Omegapy.
My GitHub repository that is used to store this assignment is named My-Academics-Portfolio, and the link to this specific assignment is: https://github.com/Omegapy/My-Academics-Portfolio/tree/main/Platform-Based-Dev-Android-CSC475/Module-7-Critical-Thinking

Project Map:

- Module-7-CTA-UnitConvertTesting.docx (this file, App documentation)
- testDebugUnitTest folder contains the index.html (Unit tests report)

The project files from the app, file structure:

```
UnitConverterTesting/
├── app/                          # Application module
│   ├── src/                      # Source code
│   │   ├── main/                 # Main source code
│   │   │   ├── java/             # Java/Kotlin source code
│   │   │   │   └── com/example/unitconvertertesting/
│   │   │   │       ├── MainActivity.kt         # Main app activity with UI
│   │   │   │       ├── UnitConverter.kt        # Conversion logic
│   │   │   │       └── ui/theme/               # UI theme
│   │   │   │           ├── Color.kt            # Color
│   │   │   │           ├── Theme.kt            # Theme
│   │   │   │           └── Type.kt             # Typography
│   │   │   ├── res/              # Android resources
│   │   │   │   ├── drawable/     # Graphics/drawable resources
│   │   │   │   ├── mipmap-*/     # App icon resources (various densities)
│   │   │   │   ├── values/       # String, style and other resources
│   │   │   │   └── xml/          # XML configuration files
│   │   │   └── AndroidManifest.xml  # App manifest
│   │   │
│   │   ├── test/                 # Unit tests
│   │   │   └── java/com/example/unitconvertertesting/
│   │   │       └── UnitConverterTest.kt       # Tests for conversion logic
```

Reflection

Like my prior critical thinking assignments, this assignment was challenging, but in a different way. To fully understand how to implement unit tests within the Android environment, I had to do quite a bit of research. The class material, with the YouTube tutorial series starting with “Why Do We Test Our Code? - Android Testing - Part 1” (Lacker, 2020), was very helpful in providing a good initial understanding of what unit tests are. Additionally, the tutorials, such as “Build local unit tests” (Android Developers, n.d.), “Kotlin JUnit: Essential Testing Techniques for Developers” (Trivedi, 2024), and “JUnit4: Getting started” (Junit Org., 2020), were useful for learning how to code them.

Although unit tests do not require an Android emulator or device, as they are executed on the developer's PC or workstation using Java Virtual Machine (JVM), they test the business logic of the app (BrowserStack, 2025). I also decided to implement a basic user interface (UI) to test the calculations manually.

This app was created based on the “Empty Activity” template from Android Studio. In the app/build.gradle file, the JUnit and Hamcrest dependencies were added, see Code Snippet 1.

Code Snippet 1

JUnit and Hamcrest Dependencies

```
dependencies {

    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.lifecycle.runtime.ktx)
    implementation(libs.androidx.activity.compose)
    implementation(platform(libs.androidx.compose.bom))
    implementation(libs.androidx.ui)
    implementation(libs.androidx.ui.graphics)
    implementation(libs.androidx.ui.tooling.preview)
    implementation(libs.androidx.material3)
    testImplementation(libs.junit)
    testImplementation("org.hamcrest:hamcrest-all:1.3")
    androidTestImplementation(libs.androidx.junit)
    androidTestImplementation(libs.androidx.espresso.core)
    androidTestImplementation(platform(libs.androidx.compose.bom))
    androidTestImplementation(libs.androidx.ui.test.junit4)
    debugImplementation(libs.androidx.ui.tooling)
    debugImplementation(libs.androidx.ui.test.manifest)
}
```

Note: Android Studio can add these dependencies automatically within the app/build.gradle.kts file.

JUnit is an open-source unit testing framework for Java and, by consequence, for Kotlin, as Kotlin is fully compatible with Java. Note that Junit5 is available; however, I implemented Junit4 as my unit tests are basic and the JUnit4 is easier to integrate and has less overhead than JUnit5; as JUnit4 needs only a single monolithic JAR file, but JUnit needs three components: JUnit Platform, JUnit Jupiter, and JUnit Vintage (Symflower, 2023). Nonetheless, JUnit5 can be used in this project as its JUnit Vintage module allows running JUnit3 and JUnit4 tests on the JUnit5 implementations.

Hamcrest is a library that assists in writing software tests in Java. It allows the use of a different set of assertions (Hamcrest assertions) than the one used by JUnit; this assertion improves how the tested objects are matched to the wanted outcome, giving a more specific description of failure if it happens. The Hamcrest team describes the utility of the library as follows:

> *When writing tests it is sometimes difficult to get the balance right between over specifying the test (and making it brittle to changes), and not specifying enough (making the test less valuable since it continues to pass even when the thing being tested is broken). Having a tool that allows you to pick out precisely the aspect under test and describe the values it should have, to a controlled level of precision, helps greatly in writing tests that are “just right”. Such tests fail when the behaviour of the aspect under test deviates from the expected behaviour, yet continue to pass when minor, unrelated changes to the behaviour are made.*

In my code, I implemented both assertions (Hamcrest and JUnit) for educational purposes and to demonstrate the differences between the two. Note that the local unit test file is in the `java\com\example\unitconvertertesting\test` directory, called `UnitConverterTest.kt`.

The Business Logic

The business is found in `the java\com\example\unitconvertertesting` directory in the file named `UnitCoverter.kt` within the singleton object.

Temperatures:

- `celsiusToFahrenheit(celsius: Double): Double` function converts from Celsius to Fahrenheit using the following formula: $Fahrenheit = \left(Celsius * \frac{9}{5}\right) - 32$
- `fahrenheitToCelsius(fahrenheit: Double): Double` function converts from Fahrenheit to Celsius using the following formula: $Celsius = (Fahrenheit - 32) * \frac{9}{5}$

Lengths:

- `metersToFeet(meters: Double): Double` function converts meters to feet using the following formula: $feet = 3.28084 * meters$
- `feetToMeters(feet: Double): Double` function converts feet to meters using the following formula: $metters = feet \ / \ 3.28084$

Distances:

- `kilometersToMiles(kilometers: Double): Double` function converts kilometers to miles using the following formula: $miles = 0.621371 * kilometers$
- `milesToKilometers(miles: Double): Double` function converts miles to kilometers using the following formula: $kilometers = miles \ / \ 0.621371$

Weights:

- `kilogramsToPounds(kilograms: Double): Double` function converts kilograms to pounds using the following formula: $pounds = 2.20462 * kilograms$
- `poundsToKilograms(pounds: Double): Double` function converts pounds to kilograms using the following formula: $pounds = kilograms \ / \ 2.2046$

Note that the function `convert(value: Double, sourceUnit: String, targetUnit: String): Double` routes the conversion of the `value: Double argument`based on the argument `sourceUnit : String` and `targetUnit: String.`
One of the challenges I encounter when converting values

Unit Tests

The unit tests can be found in the `java\com\example\unitconvertertesting\test\UnitConverterTest.kt` Kotlin file. As mentioned previously, I implemented both Hamcrest and JUnit assertions for educational purposes and to demonstrate the differences between the two.
One of the challenges I encountered when converting values within the testing process was calculation results comparison problems due to floating-point precision issues. For example, when converting kilograms to pounds, the floating-point computation might produce a result like 32.80839999999999 instead of the mathematically precise 32.8084, making the comparison fail. To fix this issue, the delta tests a integrated within the code; this allows the test to pass if the actual result is within the delta value of the expected result, for example, see Code Snippet 2.

Code Snippet 2

Delta Implementation

```
// Delta value for floating-point precision
private val delta = 0.01

/**
 * Test: 1 kilogram should return approximately 2.20462 pounds.
 */
@Test
fun kilogramsToPounds_oneKilogram_returns2Point20Pounds() {
    val result = UnitConverter.kilogramsToPounds(1.0)
    // JUnit assertion
    // assertEquals(2.20462, result, delta)
    // Hamcrest matcher
    assertThat(result, `is`(closeTo(2.20462, delta)))
}
```

Note: The code snippet illustrates how the delta tests are implemented within the JUnit and Hamcrest assertions functions by using a delta value of 0.01.

Another issue is handling user input validation, like non-numeric values and empty strings. This was important to handle as it could cause the application to crash when processing invalid inputs (like "abc" or an empty string) into a numerical type (like Double). It was handled by implementing catch statements within the composable function `UnitConverterApp()` in the `MainActivity.kt` file. See Code Snippet 4 for the implementation example

Code Snippet 3

Input Error Handling

```
// Convert button
Button(
    onClick = {
        try {
            val inputValueDouble = inputValue.toDouble()
            val result = UnitConverter.convert(
                inputValueDouble,
                selectedFromUnit,
                selectedToUnit
            )
            val df = DecimalFormat("#.####")
            resultValue = df.format(result)
        } catch (e: NumberFormatException) {
            resultValue = "Invalid input"
        } catch (e: IllegalArgumentException) {
            resultValue = e.message ?: "Conversion not supported"
        }
    },
    modifier = Modifier.fillMaxWidth()
) {
    Text("Convert")
}

Spacer(modifier = Modifier.height(24.dp))

// Result
if (resultValue.isNotEmpty()) {
    Text(
```

```
        text = "Result: $resultValue ${selectedToUnit}",
        style = MaterialTheme.typography.titleLarge
    )
}
```

Note: the code snippet illustrates how input errors are handled by the *Convert Button* section of the Composable function `UnitConverterApp()` in the `MainActivity.kt` file.

The Unit Tests

The unit tests are contained in the class `UnitConverterTest` within the `UnitConverterTest.kt` file. The following lists the different methods of the class representing different unit tests:

Temperature Conversion Tests:

- `celsiusToFahrenheit_zeroCelsius_returnsThirtyTwoFahrenheit()` - Tests that 0°C converts to 32°F
- `celsiusToFahrenheit_hundredCelsius_returns212Fahrenheit()` - Tests that 100°C converts to 212°F
- `fahrenheitToCelsius_thirtyTwoFahrenheit_returnsZeroCelsius()` - Tests that 32°F converts to 0°C
- `fahrenheitToCelsius_212Fahrenheit_returns100Celsius()` - Tests that 212°F converts to 100°C
- `celsiusToFahrenheit_negativeForty_returnsNegativeForty()` - Tests that -40°C converts to -40°F
- `fahrenheitToCelsius_negativeForty_returnsNegativeForty()` - Tests that -40°F converts to -40°C

Length Conversion Tests:

- `metersToFeet_oneMeter_returns3Point28Feet()` - Tests that 1m converts to 3.28084ft
- `feetToMeters_3Point28Feet_returnsOneMeter()` - Tests that 3.28084ft converts to 1m
- `kilometersToMiles_oneKilometer_returns0Point62Miles()` - Tests that 1km converts to 0.621371 miles
- `milesToKilometers_oneMile_returns1Point61Kilometers()` - Tests that 1 mile converts to 1.609344km
- `metersToFeet_zeroMeters_returnsZeroFeet()` - Tests that 0m converts to 0ft
- `metersToFeet_largeValue_returnsCorrectValue()` - Tests that 1000m converts to 3280.84ft

Weight Conversion Tests:

- `kilogramsToPounds_oneKilogram_returns2Point20Pounds()` - Tests that 1kg converts to 2.20462lbs
- `poundsToKilograms_onePound_returns0Point45Kilograms()` - Tests that 1lb converts to 0.453592kg
- `kilogramsToPounds_decimalValue_returnsCorrectValue()` - Tests that 0.5kg converts to 1.10231lbs
- `poundsToKilograms_decimalValue_returnsCorrectValue()` - Tests that 0.5lbs converts to 0.226796kg

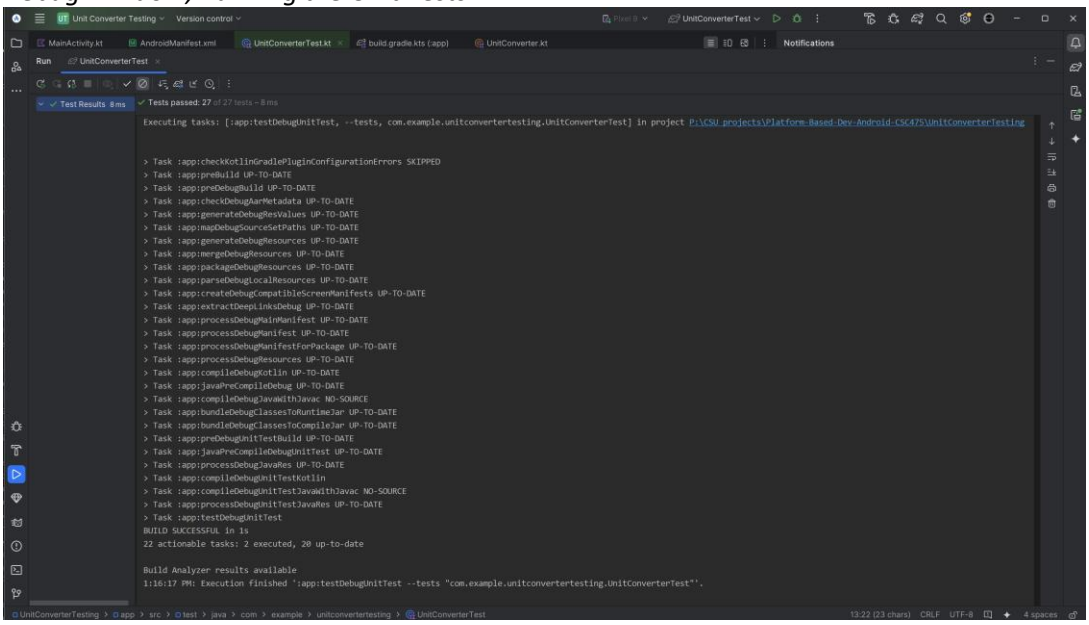
General Conversion Function Tests:

- `convert_celsiusToFahrenheit_returnsCorrectValue()` - Tests generic convert function for temperature
- `convert_metersToFeet_returnsCorrectValue()` - Tests generic convert function for length
- `convert_kilogramsToPounds_returnsCorrectValue()` - Tests generic convert function for weight
- `convert_sameUnits_returnsOriginalValue()` - Tests that conversion between same units returns original value
- `convert_sameTemperatureUnits_returnsOriginalValue()` - Tests conversion between same temperature units
- `convert_sameWeightUnits_returnsOriginalValue()` - Tests conversion between same weight units
- `convert_bidirectionalTemperatureConversion_returnsOriginalValue()` - Tests C → F → C equals original
- `convert_bidirectionalLengthConversion_returnsOriginalValue()` - Tests m → ft → m equals original
- `convert_bidirectionalWeightConversion_returnsOriginalValue()` - Tests kg → lb → kg equals original
- `convert_incompatibleUnits_throwsException()` - Tests error handling for incompatible units
- `convert_invalidUnit_throwsException()` - Tests error handling for invalid unit names

The unit tests include various scenarios to ensure that the conversion logic is functioning as intended across a range of inputs. The following screenshots illustrate the outputs of the tests and the functionalities.

Figure 1

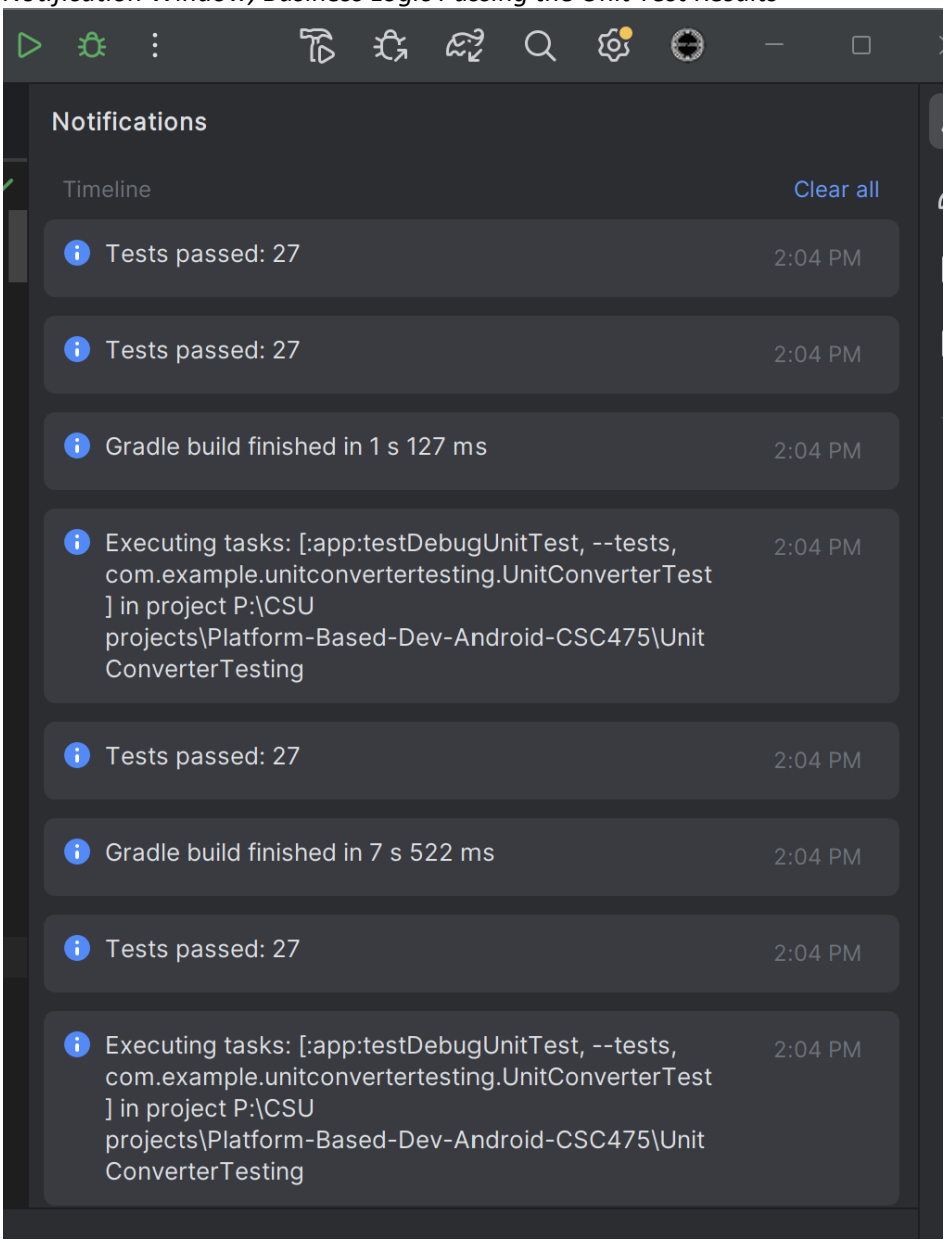
Debug Window, Running the Unit Tests



Note: The figure illustrates that the test units themselves run without error. Here, the code logic passed the unit tests

Figure 2

Notification Window, Business Logic Passing the Unit Test Results

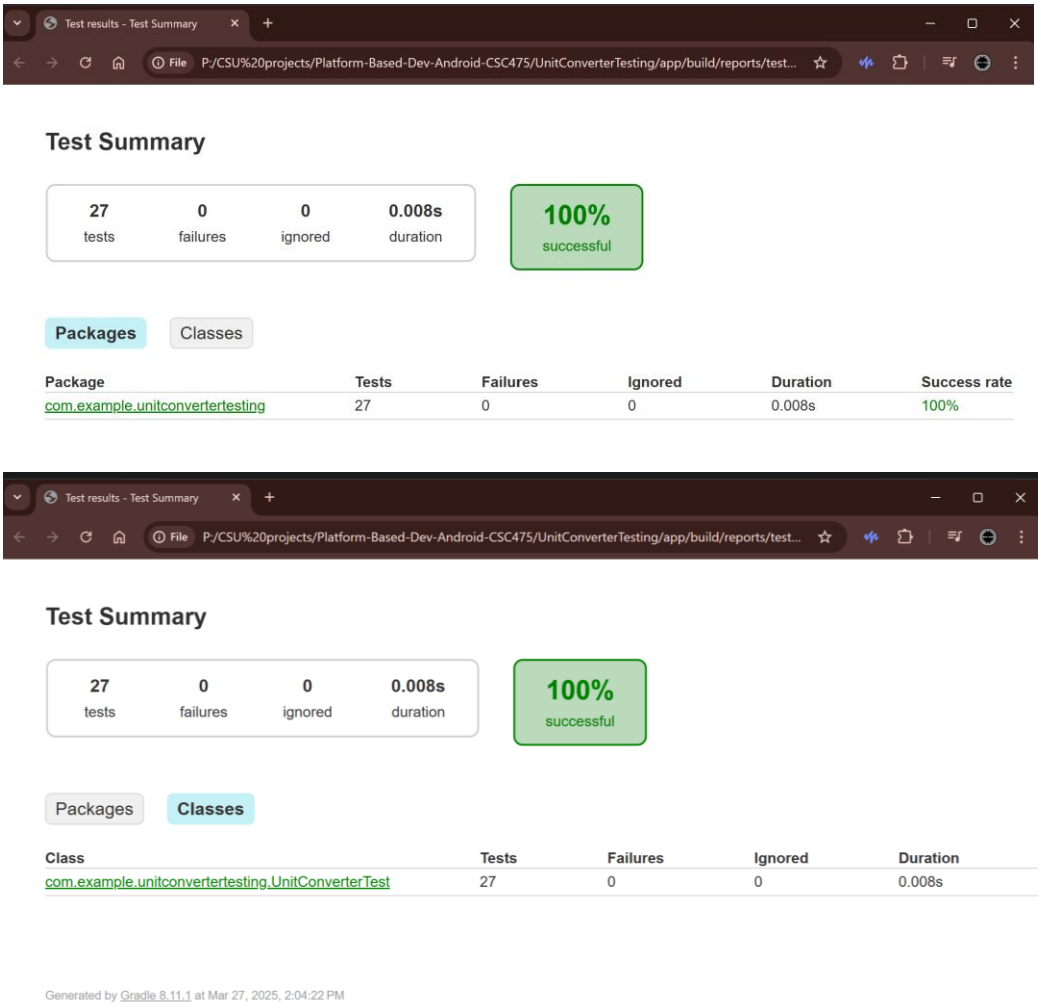


Note: The figure illustrates the notification displaying the code logic passing the unit tests.

Please see the next page.

Figure 3

Test Report Index File Part-1, Business Logic Passing the Unit Test Results



Note: The figure illustrates the `index.html` file, which displays the code logic that passed all the unit tests contain in the package and class. The `index.html` is automatically generated by Gradle's built-in test reporting mechanism. See Code Snippet 3

Code Snippet 4

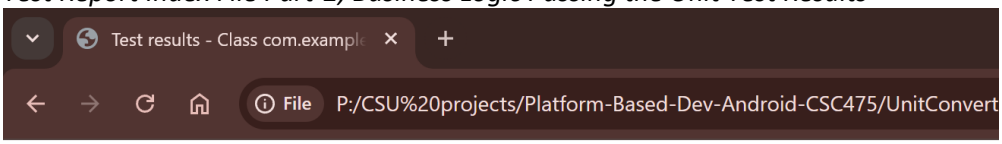
Build Text Report

```
testOptions {
    unitTests.all {
        it.reports {
            html.required.set(true)
            junitXml.required.set(true)
        }
    }
}
```

Note: Android Studio is automatically generated through the Gradle file. It generates an `app\build\reports\tests` directory that contains the directory `testDebugUnitTest`, which contains the `index.html` file

Figure 4

Test Report Index File Part-2, Business Logic Passing the Unit Test Results



Class com.example.unitconvertertesting.UnitConverterTest

all > com.example.unitconvertertesting > UnitConverterTest

27	0	0	0.008s
tests	failures	ignored	duration

100% successful

Tests

Test	Duration	Result
celsiusToFahrenheit_hundredCelsius_returns212Fahrenheit	0s	passed
celsiusToFahrenheit_negativeForty_returnsNegativeForty	0s	passed
celsiusToFahrenheit_zeroCelsius_returnsThirtyTwoFahrenheit	0s	passed
convert_bidirectionalLengthConversion_returnsOriginalValue	0.003s	passed
convert_bidirectionalTemperatureConversion_returnsOriginalValue	0s	passed
convert_bidirectionalWeightConversion_returnsOriginalValue	0s	passed
convert_celsiusToFahrenheit_returnsCorrectValue	0s	passed
convert_incompatibleUnits_throwsException	0s	passed
convert_invalidUnit_throwsException	0.002s	passed
convert_kilogramsToPounds_returnsCorrectValue	0s	passed
convert_metersToFeet_returnsCorrectValue	0s	passed
convert_sameTemperatureUnits_returnsOriginalValue	0s	passed
convert_sameUnits_returnsOriginalValue	0s	passed
convert_sameWeightUnits_returnsOriginalValue	0s	passed
fahrenheitToCelsius_212Fahrenheit_returns100Celsius	0s	passed
fahrenheitToCelsius_negativeForty_returnsNegativeForty	0s	passed
fahrenheitToCelsius_thirtyTwoFahrenheit_returnsZeroCelsius	0.001s	passed
feetToMeters_3Point28Feet_returnsOneMeter	0.001s	passed
kilogramsToPounds_decimalValue_returnsCorrectValue	0s	passed
kilogramsToPounds_oneKilogram_returns2Point20Pounds	0s	passed
kilometersToMiles_oneKilometer_returns0Point62Miles	0s	passed
metersToFeet_largeValue_returnsCorrectValue	0s	passed
metersToFeet_oneMeter_returns3Point28Feet	0s	passed
metersToFeet_zeroMeters_returnsZeroFeet	0.001s	passed
milesToKilometers_oneMile_returns1Point61Kilometers	0s	passed
poundsToKilograms_decimalValue_returnsCorrectValue	0s	passed
poundsToKilograms_onePound_returns0Point45Kilograms	0s	passed

Generated by Gradle 8.11.1 at Mar 27, 2025, 2:04:22 PM

Note: The figure illustrates the `index.html` file, which displays all the unit tests. Here, the code logic passed all the unit tests.

The following screenshots illustrate a simulation where the code logic is not passing a specific unit test. This is done to demonstrate the functionalities of the unit tests. The code of the unit test was modified for this purpose. In the conversion test `celsiusToFahrenheit_zeroCelsius_returnsThirtyTwoFahrenheit()`, the result value was modified by entering the wrong Celsius temperature to fail the test. See Code Snippet 5.

Code Snippet 5

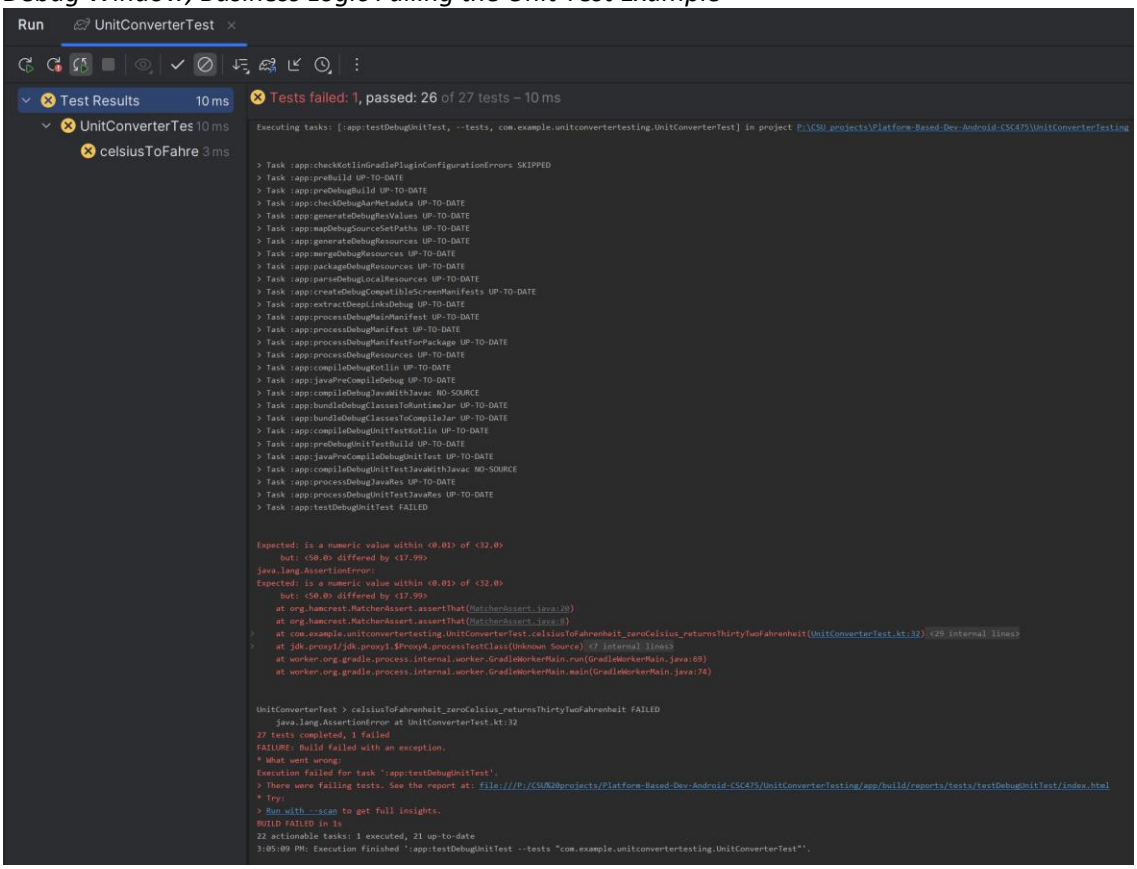
Modified Celsius to Fahrenheit Test Example

```
@Test
fun celsiusToFahrenheit_zeroCelsius_returnsThirtyTwoFahrenheit() {
    // val result = UnitConverter.celsiusToFahrenheit(0.0)
    val result = UnitConverter.celsiusToFahrenheit(10.0) // Modified code
    // JUnit assertion: Checks that the result equals 32.0 within the delta value
    // assertEquals(32.0, result, delta)

    // Hamcrest matcher: more readable way to verify that result is close to 32.0
    // The closeTo matcher checks that the actual value is within the range of delta
    // of the expected value.
    assertThat(result, `is`(closeTo(32.0, delta)))
}
```

Note: In this test, the value of Celsius temperature was modified from 0.0 to 10.0.

Figure 5

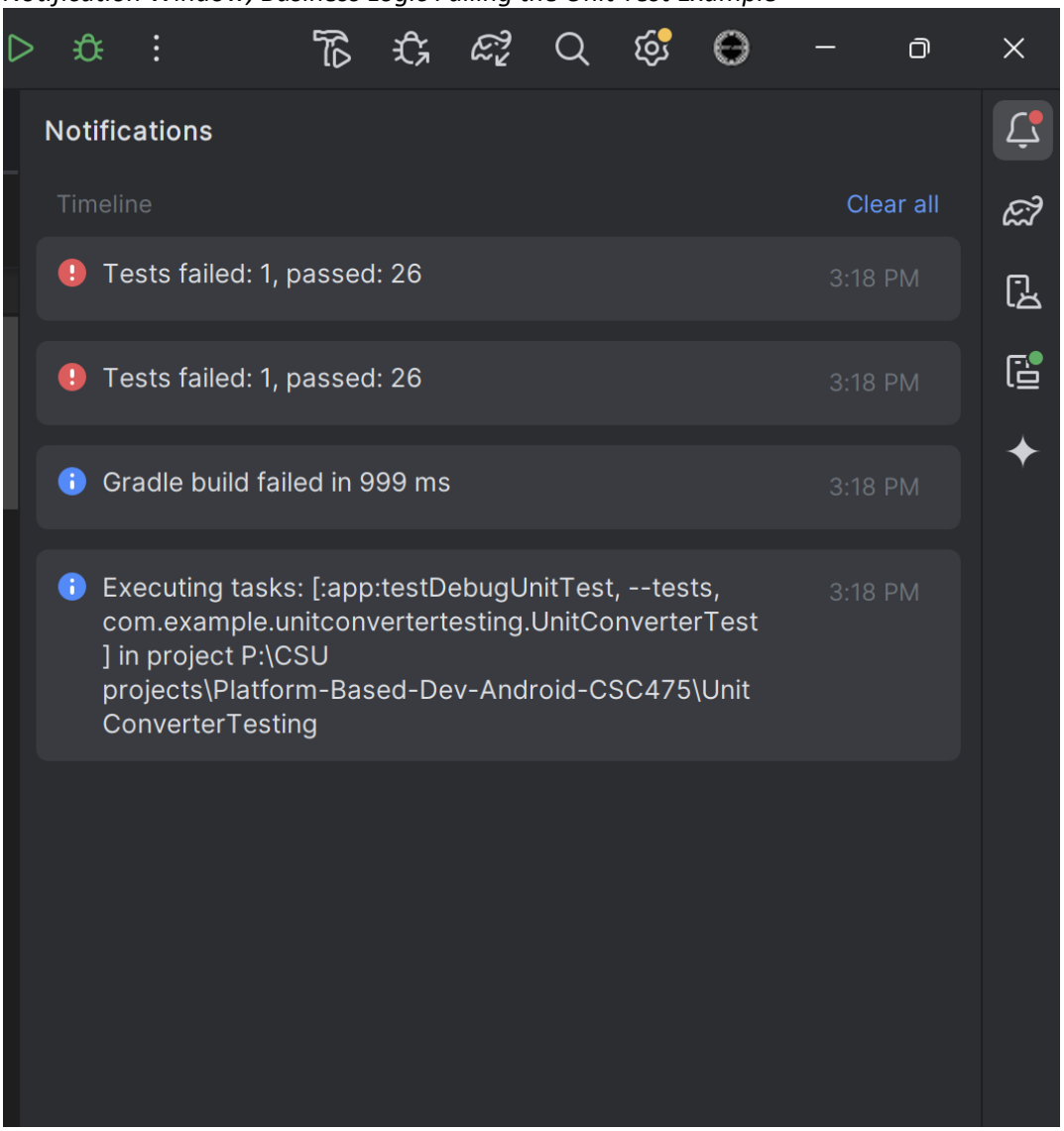
Debug Window, Business Logic Failing the Unit Test Example



Note: The figure illustrates the debug window showing that the code logic failed the unit test `celsiusToFahrenheit_zeroCelsius_returnsThirtyTwoFahrenheit(UnitConverterTest.kt:32.`

Figure 6

Notification Window, Business Logic Failing the Unit Test Example

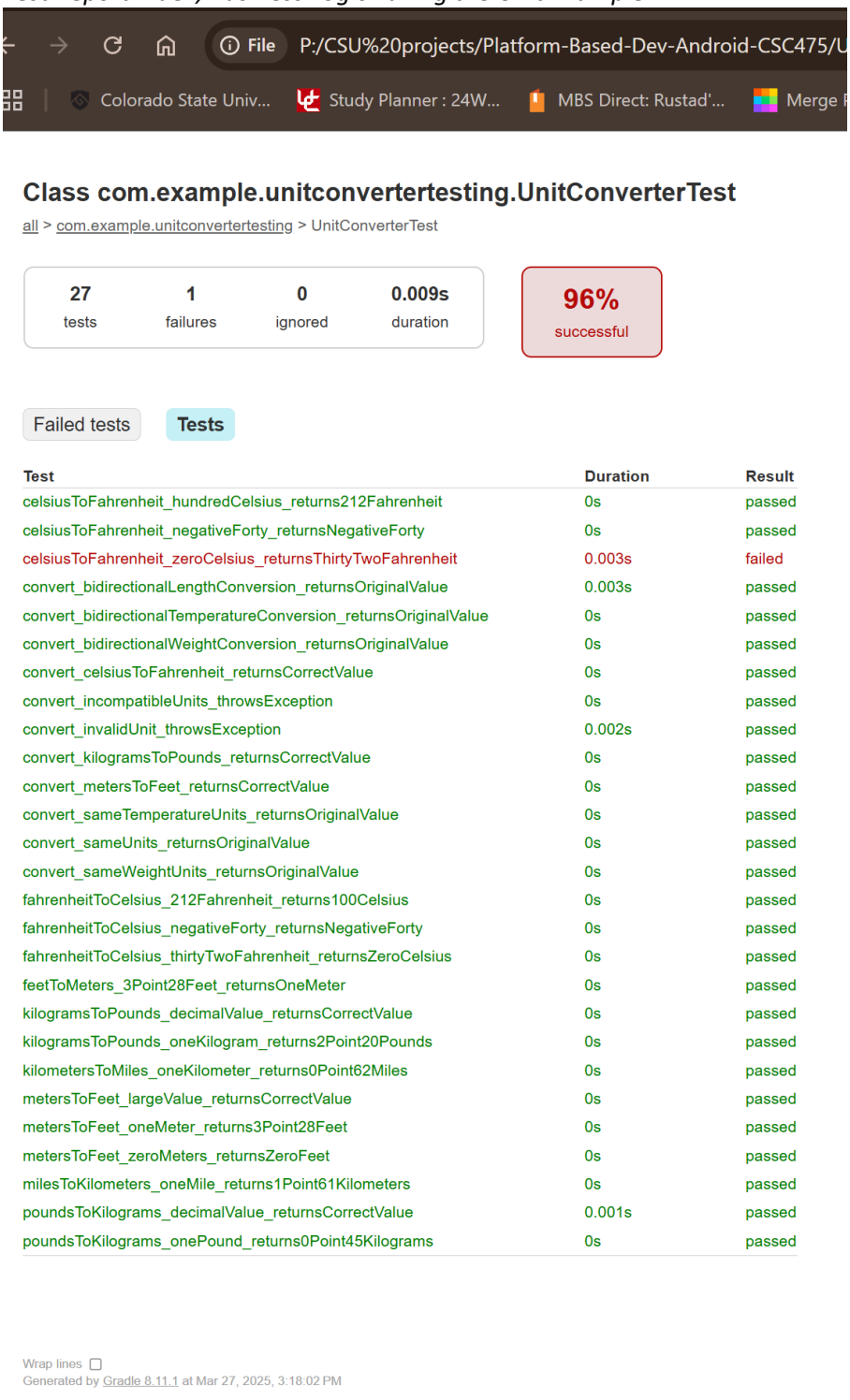


Note: The figure illustrates the notification window showing that the code logic failed the unit test.

Please *see the next page.*

Figure 7

Test Report Index, Business Logic Failing the Unit Example



Note: The figure illustrates the `index.html` file, which displays all the unit tests. Here, the code logic failed the `celsiusToFahrenheit_zeroCelsius_returnsThirtyTwoFahrenheit` unit test.

The following screenshots showcase the difference between using JUnit and Hamcrest asserts. The `celsiusToFahrenheit_zeroCelsius_returnsThirtyTwoFahrenheit()` unit test was modified a little more for that purpose, see Code Snippet 6.

Code Snippet 6

Using JUnit Assertion Instead of Hamcrest Assertion

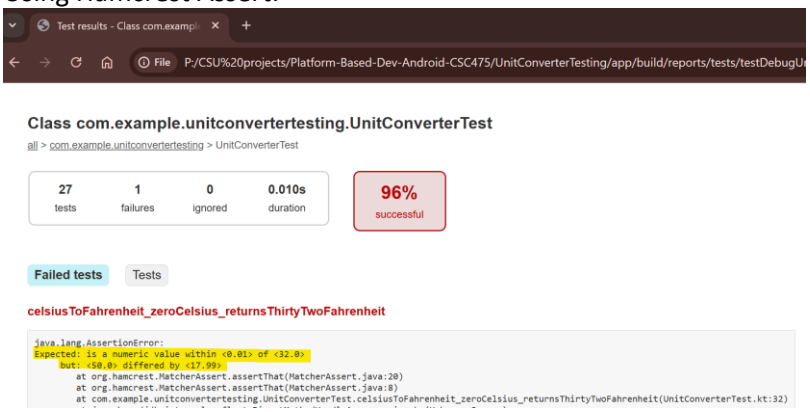
```
@Test
fun celsiusToFahrenheit_zeroCelsius_returnsThirtyTwoFahrenheit() {
    // val result = UnitConverter.celsiusToFahrenheit(0.0)
    val result = UnitConverter.celsiusToFahrenheit(10.0) // Modified code
    // JUnit assertion: Checks that the result equals 32.0 within the delta value
    assertEquals(32.0, result, delta) // Modified code

    // Hamcrest assertion - matcher: more readable way to verify that result is close to 32.0
    // The closeTo matcher checks that the actual value is within the range of delta
    // of the expected value.
    // assertThat(result, `is`(closeTo(32.0, delta)))
}
```

Note: The code snippet illustrates the use JUnit assertions by commenting out the `assertEquals(32.0, result, delta)` code line and commenting the `assertThat(result, `is`(closeTo(32.0, delta)))` code line.

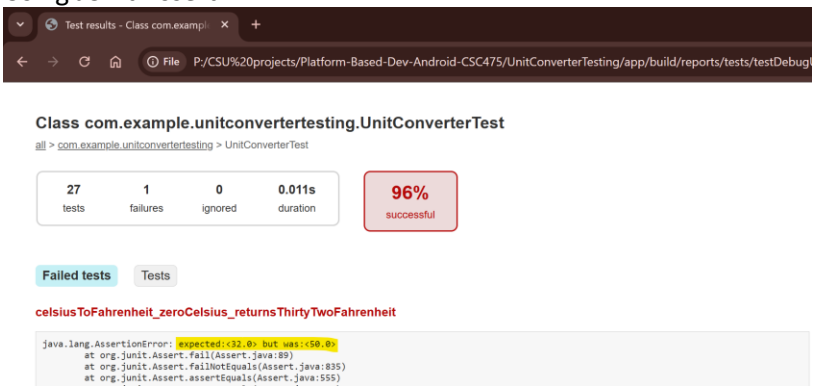
Figure 8

Test Report Index – Differences between JUnit and Hamcrest Asserts Example

Using Hamcrest Assert:



Using JUnit Assert:



Note: The figure illustrates the differences between JUnit and Hamcrest asserts within the `index.html` file. This demonstrates that the Hamcrest assert outputs significantly more information about what caused the test to fail.

To summarize, throughout the process of implementing JUnit4 and Hamcrest asserts and unit tests for this app, I learned how to utilize JUnit and Hamcrest for Android unit testing and improved my coding and problem-solving skills. Additionally, I practice setting up dependencies, designing the business logic for various conversions (temperature, length, distance, weight), and creating unit tests. Furthermore, I addressed challenges like floating-point precision and user input validation

Screenshots

This section illustrates the functionality of the app by utilizing screenshots

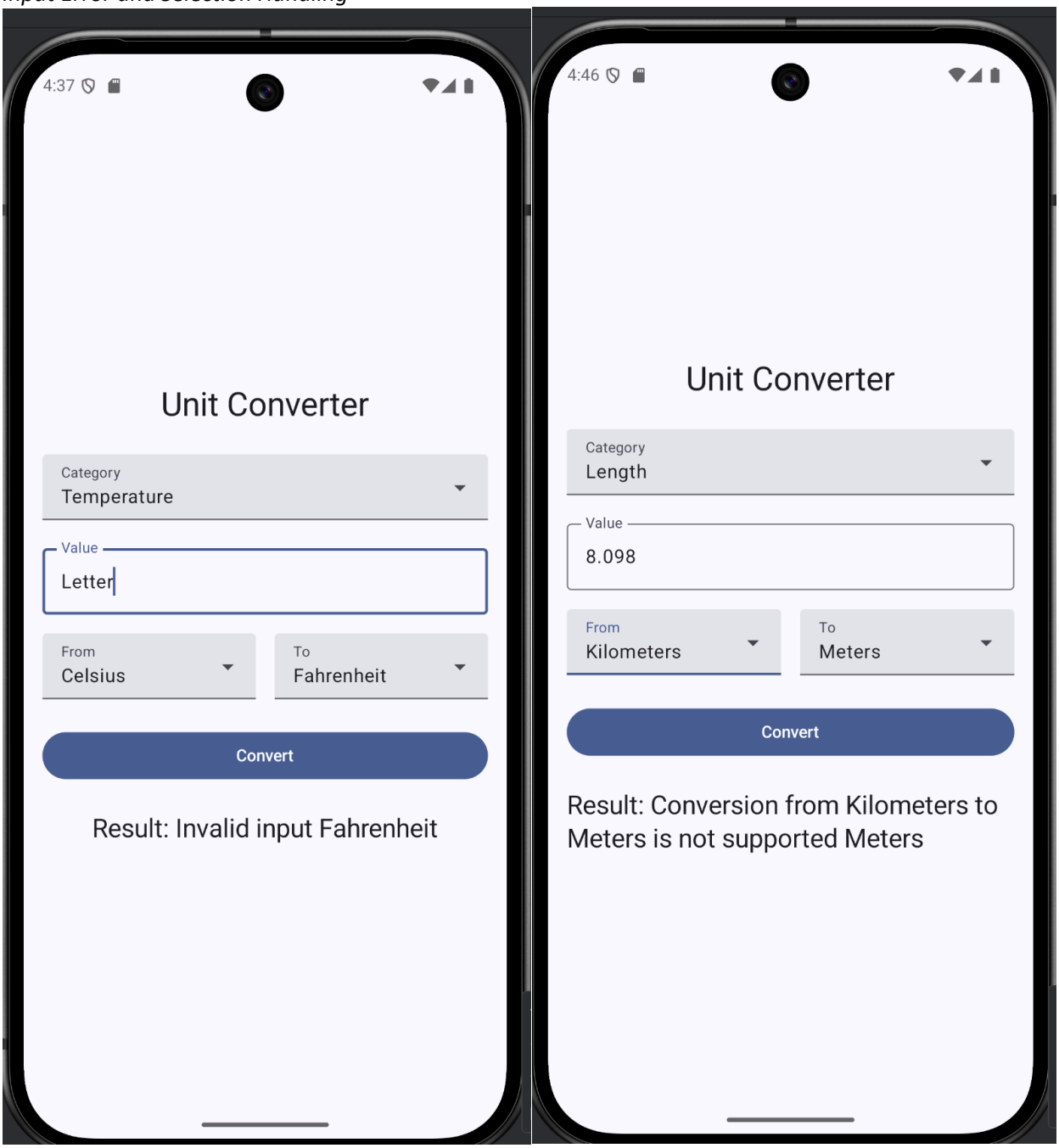
Figure 9

Home Screen

Note: The figure illustrates the home screen and the different dropdown menus.

Figure 10

Input Error and Selection Handling

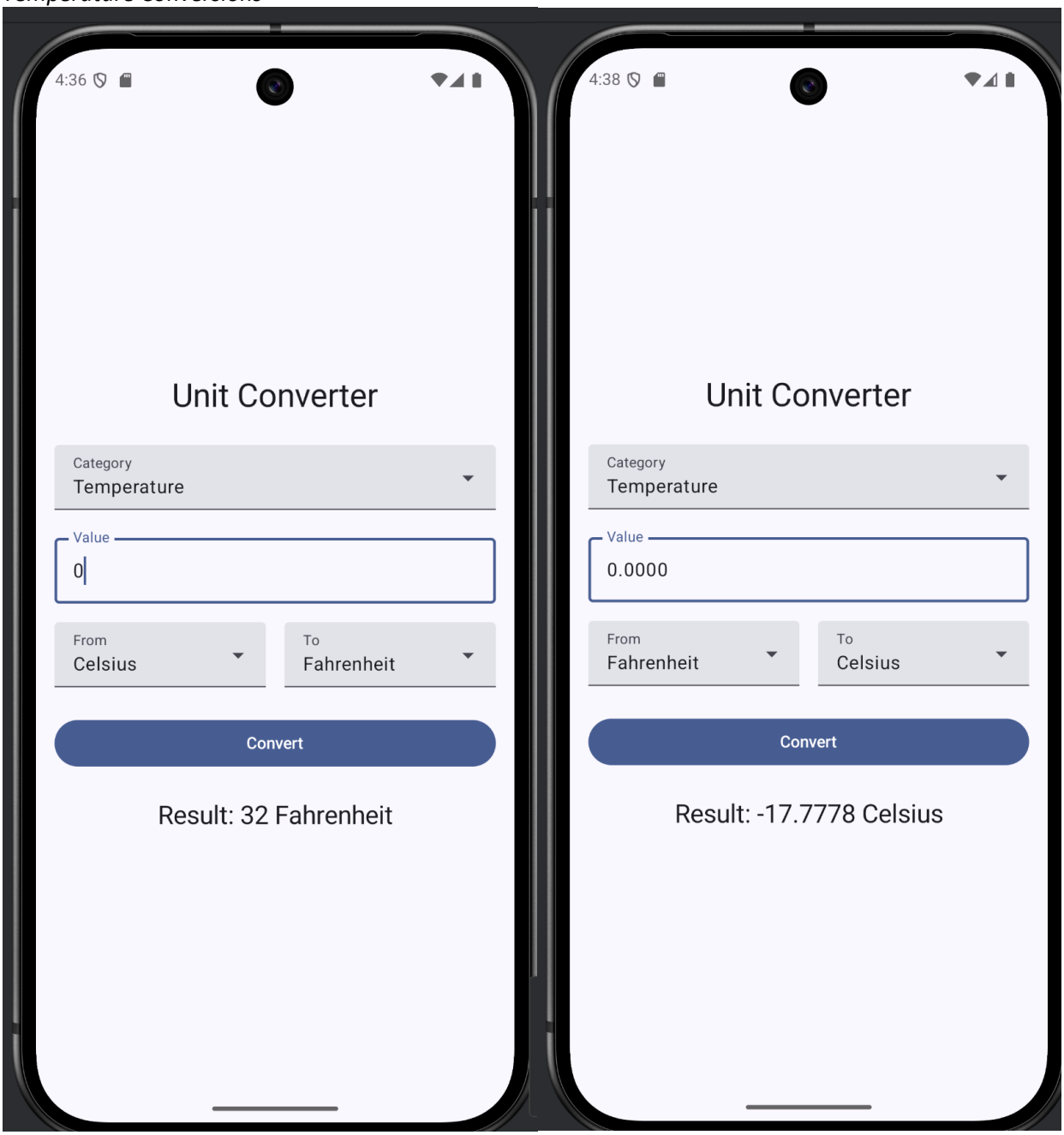


Note: The figure illustrates the app's input and selection error handling.

Please see the next page.

Figure 11

Temperature Conversions

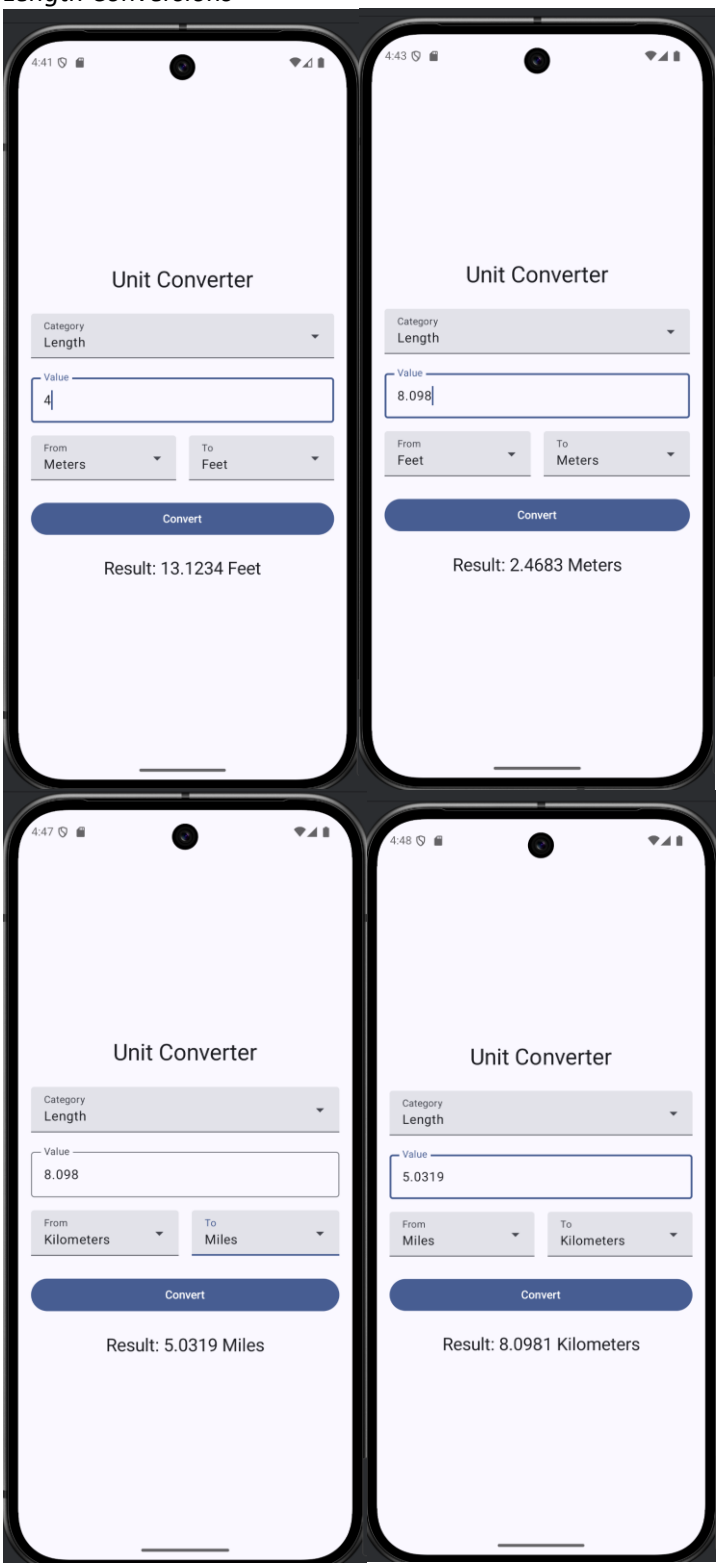


Note: The figure illustrates the app temperature conversions.

Please see the next page.

Figure 12

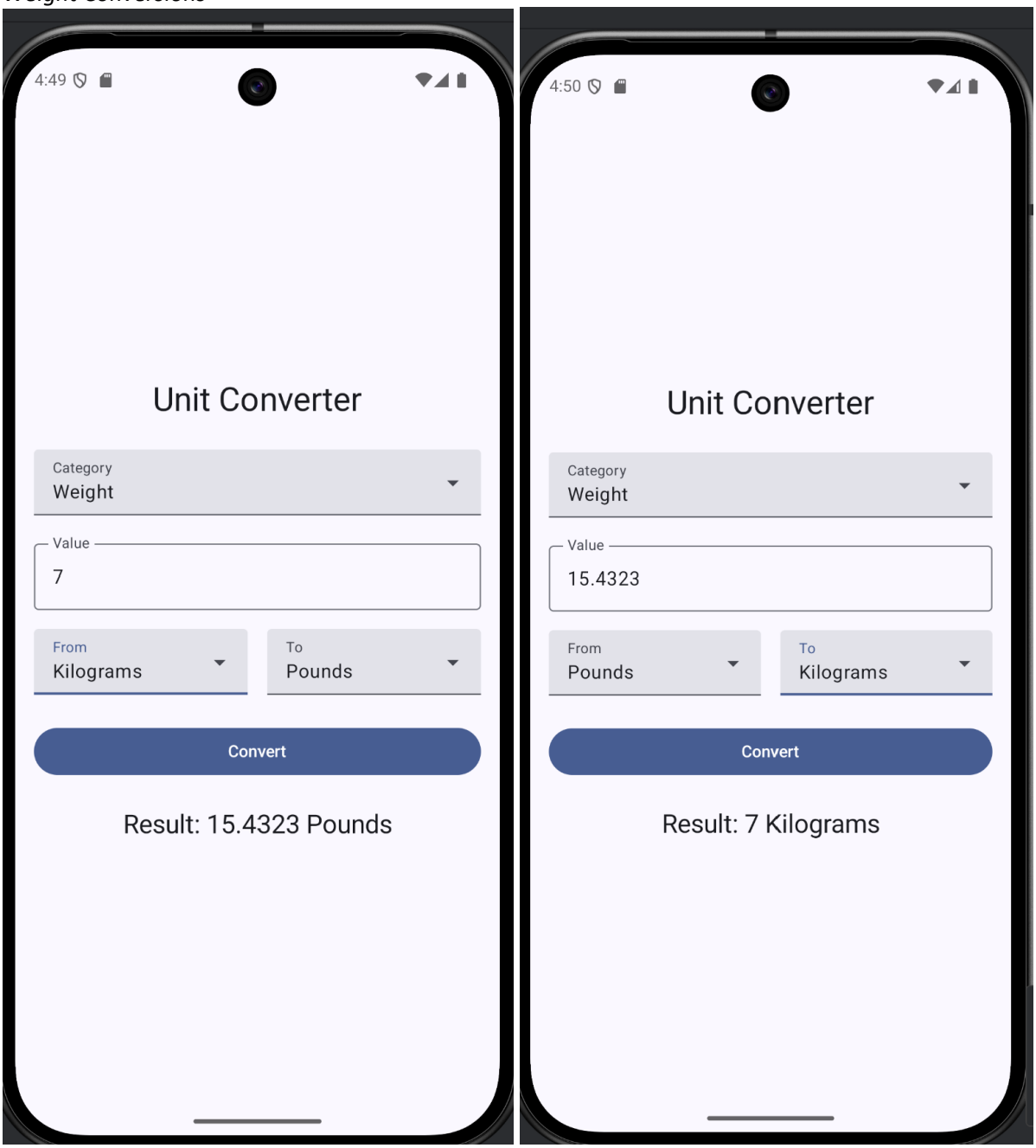
Length Conversions



Note: The figure illustrates the app length conversions.

Figure 13

Weight Conversions



Note: The figure illustrates the app weight conversions.

As shown in figures 1 to 13, the app works as intended.

References:

Android Developers (n.d.). Build local unit tests. Android. https://developer.android.com/training/testing/local-tests

BowserStack (2025, March 18). *What is Android unit testing?* BowserStack. https://www.browserstack.com/guide/android-unit-testing

JUnit Org. (2020, June 24). Junit4 - *Getting started* [GitHub Wiki]. https://github.com/junit-team/junit4/wiki/Getting-started

Lackner, P. (2020, August 12). *Why do we test our code? - Android testing - Part 1* [Video]. YouTube. https://www.youtube.com/watch?v=EkfVL5vCDmo

Symflower (2023, July 19). How to migrate from JUnit 4 to JUnit 5: a step-by-step guide. Symflower. https://symflower.com/en/company/blog/2023/migrating-from-junit-4-to-5/

Trived, T. (2024, November 12). Kotlin JUnit: Essential testing techniques for developers. DhiWise. https://www.dhiwise.com/post/kotlin-junit-essential-testing-techniques-for-developers