International Institute of Information Technology, Bangalore

CS 731: Software Testing

Submitted by,
MT2022093 Rupenkumar Rakholiya
MT2022142 Gautam Rizwani

Under the Guidance of
Prof. Meenakshi D Souza

Table of Contents

Introduction

- Purpose: This project aims to outline the implementation of data flow-based testing, specifically focusing on the utilization of du-paths for designing test cases in software projects.

Overview

Project Scope:

- The main goal of this project is to ensure that we thoroughly understand and test how information or data moves within the code. We're using a specific technique called "du-paths" to create tests that cover every possible path where data is defined (set) in the code and then used elsewhere. Even when there are loops (repetitive sections) in the code, we're making sure to include all these paths to fully test how data flows through these loops. This approach helps us create tests that are really accurate and reliable because they cover all the different ways data moves, ensuring that our code works as expected in various scenarios.

Getting to know the importance of data flow-based testing:

- Ensuring Code Robustness: Data flow-based testing plays a critical role in enhancing the reliability and robustness of software code.
- Identifying Data Flow Anomalies: By analyzing data flow within the code, this methodology helps detect and address potential anomalies, such as variable misuse, uninitialized variables, and potential data corruption issues.

Objectives:

- Coverage Enhancement: The primary objective is to achieve comprehensive coverage, particularly focusing on all-du-paths and all-defs criteria within the codebase.
- Defect Identification: Identify potential defects or vulnerabilities associated with data flow within the code.
- Test Case Design: Develop effective test cases based on du-paths to ensure comprehensive coverage and effective validation of code functionalities.

Significance of Du-Paths:

- Precision in Testing: Du-paths offer a granular approach to testing, ensuring that every path from a variable's definition to its usage is covered.
- Loop Handling: Addressing complexities related to loops within the code, ensuring thorough coverage even in loop-involved scenarios.

Outcome Expectations:

- Improved Code Quality: The project aims to contribute to enhanced code quality by addressing potential data flow issues, leading to more reliable and efficient software.
- Reduced Risks: Mitigating risks associated with data flow-related defects, leading to minimizing chances of unexpected behavior or vulnerabilities in the codebase.

Code Implementation:

Project Aim:

- Our project facilitates the transformation of data or values from one format, system, or unit to another.

Use Cases:

- Currency Conversion: Facilitates quick and up-to-date currency conversions based on real-time exchange rates.
- Measurement Conversions: Allows users to convert between different units of measurements, such as miles to kilometers, pounds to kilograms, etc.
- Temperature and Time Zone Conversion: Helps users convert temperatures between Celsius and Fahrenheit or adjust time zones effortlessly.

Unit Converter:

Pseudo code:

```
switch (unit1) {
                case "milli":
                    kilo = input1 / 1000000;
                    break;

                case "centi":
                    kilo = input1 / 100000;
                    break;

                case "gram":
                    kilo = input1 / 1000;
                    break;
```

```
                case "kilo":
                    kilo = input1;
                    break;

                case "matrictonnes":
                    kilo = input1 * 1000;
                    break;

                case "pounds":
                    kilo = input1 / 2.20462;
                    break;

                case "ounces":
                    kilo = input1 / 35.274;
                    break;

                default:
                    return new ResponseEntity(HttpStatus.BAD_REQUEST);
            }

            switch (unit2) {
                case "milli":
                    res = kilo * 1000000;
                    break;

                case "centi":
                    res = kilo * 100000;
                    break;

                case "gram":
                    res = kilo * 1000;
                    break;

                case "kilo":
                    res = kilo;
                    break;

                case "matrictonnes":
                    res = kilo / 1000;
                    break;

                case "pounds":
                    res = kilo * 2.20462;
```

```
                    break;

                case "ounces":
                    res = kilo / 35.274;
                    break;

                default:
                    return new ResponseEntity(HttpStatus.BAD_REQUEST);
            }
```

Test Case Description:

DU Path 1:

Definitions:

- unit1
- unit2
- res
- input1

Uses:

- payload.get("unit1")
- payload.get("unit2")
- res
- Double.parseDouble((String)payload.get("input1"))

Path:
payload.get("unit1") ==> unit1 ==> payload.get("unit2") ==> unit2 ==>
Double.parseDouble((String) payload.get("input1")) ==> input1 ==> res

DU Path 2:

Definitions:

- setWeight
- unit1
- unit2
- res
- input1
- kilo

Uses:

- setWeight

- unit1
- unit2
- res
- input1
- kilo

Path:
setWeight ==> unit1 ==> setWeight ==> unit2 ==> input1 ==> kilo ==> res

DU Path 3:

Definitions:
- setWeight
- unit1
- unit2
- res
- input1
- kilo

Uses:
- setWeight
- unit1
- unit2
- res
- input1
- kilo

Path:
setWeight ==> unit1 ==> setWeight ==> unit2 ==> input1 ==> kilo ==> res

Time Zone Converter:

- It allows us to convert times between different time zones around the world, making it easier to coordinate meetings or events across different regions.

DU Path 1:

Definitions:
- sourceTime
- sourceTimeZone
- targetTimeZone
- convertedTime

Uses:
- timeZoneRequest.getSourceTime()
- timeZoneRequest.getSourceTimeZone()
- timeZoneRequest.getTargetTimeZone()
- sourceTime
- sourceTimeZone
- convertedTime

Path:
timeZoneRequest.getSourceTime() ==> sourceTime ==>
timeZoneRequest.getSourceTimeZone() ==> sourceTimeZone ==>
timeZoneRequest.getTargetTimeZone() ==> targetTimeZone ==> sourceTime ==>
convertedTime

DU Path 2:

Definitions:
- dateFormat

Uses:
- dateFormat
- timeZoneRequest.getSourceTimeZone()
- sourceTimeZone
- timeZoneRequest.getTargetTimeZone()

Path:

dateFormat ==> timeZoneRequest.getSourceTimeZone() ==> sourceTimeZone ==>
dateFormat ==> timeZoneRequest.getTargetTimeZone() ==> targetTimeZone

DU Path 3:

Definitions:

- dateFormat
- sourceTime

Uses:

- dateFormat
- timeZoneRequest.getSourceTimeZone()
- sourceTimeZone
- dateFormat.parse(timeZoneRequest.getSourceTime())

Path:
dateFormat ==> timeZoneRequest.getSourceTimeZone() ==> sourceTimeZone ==>
dateFormat.parse(timeZoneRequest.getSourceTime()) ==> sourceTime

DU Path 4:

Definitions:

- convertedTime

Uses:

- dateFormat.format(sourceTime)
- convertedTime

Path:
dateFormat.format(sourceTime) ==> convertedTime

Hexadecimal/Decimal/Binary Converter:

- This type of converter helps in converting numbers between different numeral systems. For example, converting decimal numbers to binary or hexadecimal and vice versa.

Pseudo code:

```
switch (request.getConversionType()) {
                case "decimalToBinary":
                    int decimalValue = Integer.parseInt(input);
                    output = Integer.toBinaryString(decimalValue);
                    break;

                case "binaryToDecimal":
                    int binaryValue = Integer.parseInt(input, 2);
                    output = String.valueOf(binaryValue);
                    break;

                case "decimalToHexadecimal":
                    int decimalValueForHex = Integer.parseInt(input);
                    output =
Integer.toHexString(decimalValueForHex).toUpperCase();
                    break;

                case "hexadecimalToDecimal":
                    int decimalValueForHex = Integer.parseInt(input, 16);
                    output = String.valueOf(decimalValueForHex);
                    break;
```

DU Path 1:

Definitions:

- input
- conversionType
- output

Uses:

- request.getInput()

- request.getConversionType()
- Integer.parseInt(input)
- Integer.toBinaryString(decimalValue)
- Integer.parseInt(input, 2)
- String.valueOf(binaryValue)
- Integer.parseInt(input)
- Integer.toHexString(decimalValueForHex).toUpperCase()
- Integer.parseInt(input, 16)
- String.valueOf(decimalValueForHex)

Path: "decimalToBinary" case

DU Path 2:

Definitions:
- input
- conversionType
- output

Uses:
- request.getInput()
- request.getConversionType()
- Integer.parseInt(input)
- Integer.toBinaryString(decimalValue)
- Integer.parseInt(input, 2)
- String.valueOf(binaryValue)
- Integer.parseInt(input)
- Integer.toHexString(decimalValueForHex).toUpperCase()
- Integer.parseInt(input, 16)
- String.valueOf(decimalValueForHex)

Path: "binaryToDecimal" case

DU Path 3:

Definitions:
- input
- conversionType
- output

Uses:

- request.getInput()
- request.getConversionType()
- Integer.parseInt(input)
- Integer.toBinaryString(decimalValue)
- Integer.parseInt(input, 2)
- String.valueOf(binaryValue)
- Integer.parseInt(input)
- Integer.toHexString(decimalValueForHex).toUpperCase()
- Integer.parseInt(input, 16)
- String.valueOf(decimalValueForHex)

Path: "decimalToHexadecimal" case

DU Path 4:

Definitions:

- input
- conversionType
- output

Uses:

- request.getInput()
- request.getConversionType()
- Integer.parseInt(input)
- Integer.toBinaryString(decimalValue)
- Integer.parseInt(input, 2)
- String.valueOf(binaryValue)
- Integer.parseInt(input)
- Integer.toHexString(decimalValueForHex).toUpperCase()
- Integer.parseInt(input, 16)
- String.valueOf(decimalValueForHex)

Path: "hexadecimalToDecimal" case

DU Path 5:

Definitions:

- input
- conversionType

- output

Uses:
- request.getInput()
- request.getConversionType()
- NumberFormatException e

Path: Exception block for NumberFormatException

Temperature Converter:

- Temperature Converter swiftly and accurately transforms temperatures between Celsius, Fahrenheit, and Kelvin through an intuitive user interface.

DU Path 1:

Definitions:

- temperature
- sourceUnit
- targetUnit
- convertedTemperature

Uses:

- request.getTemperature()
- request.getSourceUnit()
- request.getTargetUnit()
- isValidUnit(sourceUnit)
- isValidUnit(targetUnit)
- convertFromCelsius(temperature, targetUnit)
- convertFromFahrenheit(temperature, targetUnit)
- convertFromKelvin(temperature, targetUnit)
- BigDecimal("9/5")
- BigDecimal("32")
- new BigDecimal("273.15")

DU Path 2:

Definitions:

- temperature
- sourceUnit
- targetUnit
- convertedTemperature

Uses:

- request.getTemperature()
- request.getSourceUnit()
- request.getTargetUnit()
- isValidUnit(sourceUnit)
- !isValidUnit(targetUnit)

DU Path 3:

Definitions:

- temperature
- sourceUnit
- targetUnit
- convertedTemperature

Uses:

- request.getTemperature()
- request.getSourceUnit()
- request.getTargetUnit()
- isValidUnit(sourceUnit)
- isValidUnit(targetUnit)
- convertFromCelsius(temperature, targetUnit)
- Exception

DU Path 4:

Definitions:

- temperature
- sourceUnit
- targetUnit
- convertedTemperature

Uses:

- request.getTemperature()
- request.getSourceUnit()
- request.getTargetUnit()
- !isValidUnit(sourceUnit)

Path for testConvertFromCelsiusToFahrenheit:

1. convertFromCelsius(new BigDecimal("25"), "fahrenheit")
2. Result: 77
3. convertFromCelsius(new BigDecimal("25"), "invalidUnit")
4. Result: IllegalArgumentException

Path for testConvertFromCelsiusToKelvin:

1. convertFromCelsius(new BigDecimal("25"), "kelvin")
2. Result: 298.15
3. convertFromCelsius(new BigDecimal("25"), "invalidUnit")
4. Result: IllegalArgumentException

Path for testConvertFromFahrenheitToCelsius:

1. convertFromFahrenheit(new BigDecimal("77"), "celsius")
2. Result: 25
3. convertFromFahrenheit(new BigDecimal("77"), "invalidUnit")
4. Result: IllegalArgumentException

Currency Converter:

Currency Converter swiftly and accurately converts between global currencies, leveraging real-time exchange rates for precise calculations.

DU Path 1:

Definitions:

- sourceCurrency
- targetCurrency
- amount
- sourceToUSD
- targetAmount

Uses:

- request.getSourceCurrency()
- request.getTargetCurrency()
- request.getAmount()
- exchangeRates.get(sourceCurrency)
- amount.divide(exchangeRates.get(sourceCurrency), 4, BigDecimal.ROUND_HALF_UP)
- exchangeRates.get(targetCurrency), sourceToUSD.multiply(exchangeRates.get(targetCurrency))

DU Path 2:

Definitions:

- sourceCurrency
- targetCurrency
- amount
- sourceToUSD
- targetAmount

Uses:

- request.getSourceCurrency()
- request.getTargetCurrency()
- request.getAmount()
- exchangeRates.containsKey(sourceCurrency)
- exchangeRates.containsKey(targetCurrency)

DU Path 3:

Definitions:

- sourceCurrency
- targetCurrency
- amount
- sourceToUSD
- targetAmount

Uses:

- request.getSourceCurrency()
- request.getTargetCurrency()
- request.getAmount()
- exchangeRates.get(sourceCurrency)
- amount.divide(exchangeRates.get(sourceCurrency), 4, BigDecimal.ROUND_HALF_UP)

DU Path 4:

Definitions:

- sourceCurrency
- targetCurrency
- amount
- sourceToUSD
- targetAmount

Uses:

- request.getSourceCurrency()
- request.getTargetCurrency()
- request.getAmount()
- exchangeRates.containsKey(sourceCurrency)

Validation and Results:

- **Testing Execution:** We carefully ran tests that check how data moves in the code, using a specific method we planned.
- **Test Case Execution:** We made sure to run the tests we designed, using tools that help us do this accurately.
- **Coverage Measurement:** We kept track of how much of the code our tests actually covered, especially the paths data takes.
- **Coverage Results:** Our tests covered a lot of different ways data moves in the code, which was really good.
- **Defect Identification:** We found and wrote down problems we discovered, showing how our method helped find these issues.
- **Impact on Code Quality:** Testing the way we did made the code better and more dependable.
- **Validation of Design:** Our tests made sure the code behaved as it should in different situations.

Conclusion:

- The conclusion of this project highlights the significance of employing du-path-based testing in comprehensively validating data flow within code. By embracing this method, the project ensures robust test coverage, particularly focusing on how data moves from its creation to its utilization, even within complex loop structures. This meticulous testing approach enhances code reliability, minimizing the likelihood of unforeseen issues arising from data movement within the software. Ultimately, the project demonstrates the effectiveness of du-path testing in ensuring thorough and accurate validation of data flow, contributing to overall code quality and dependability.

Contribution

MT2022093 Rupenkumar Rakholiya

- Testing and code part of Unit Converter, Time Zone Converter, and Hexadecimal / Decimal / Binary Converter.
- Around 30% of report making.

MT2022142 Gautam Rizwani

- Testing and coding part of Temperature Converter, Currency Converter.
- Around 70% of report making.