
White Box Testing

Validation and Verification of Software

Cristian Gustavo Castro Xum
Cristina Martín Bris

Iteration 1

	Details
<i>ID</i>	1
<i>Name</i>	testIRRResultOK
<i>Scenario</i>	Test Internal Rate formula with valid arguments to return a successful result
<i>Input</i>	IR 1 2 10 1 2
<i>Expected output</i>	11.0
<i>Actual output</i>	11.0

	Details
<i>ID</i>	2
<i>Name</i>	testIRRNotNumericArgs
<i>Scenario</i>	Test Future Value formula with not numeric rate value
<i>Input</i>	IRR m m -1 1
<i>Expected output</i>	"The value m is not numeric"
<i>Actual output</i>	"The value m is not numeric"

	Details
<i>ID</i>	3
<i>Name</i>	testPMTSResultOK
<i>Scenario</i>	Test PMTS formula with valid argument
<i>Input</i>	PMTS 1 5 30

<i>Expected output</i>	0,97
<i>Actual output</i>	0,97

	Details
<i>ID</i>	4
<i>Name</i>	testPMTSNotNumericArgs
<i>Scenario</i>	Test PMTS formula with not numeric rate value
<i>Input</i>	PMTS r m x
<i>Expected output</i>	The value r s is not numeric
<i>Actual output</i>	The value r is not numeric

	Details
<i>ID</i>	5
<i>Name</i>	testPMTSPeriodsLessZero
<i>Scenario</i>	Test PMTS with number of periods equal to -1
<i>Input</i>	PMTS 1 -1 1 1
<i>Expected output</i>	The value -1 should be greater than 0
<i>Actual output</i>	The value -1 should be greater than 0

	Details
ID	6
Name	testPMTSInvalidNumberPaymentValues
Scenario	Test PMTS with invalid number of arguments
Input	PMTS 1 3 2 1 1
Expected output	The size of arguments is not right
Actual output	The size of arguments is not right

	Details
ID	7
Name	testPMTCTestResultOK
Scenario	Test Future Value formula with valid arguments
Input	PMTCT 1 1 1 1
Expected output	30.0
Actual output	3.0

	Details
ID	8
Name	testPMTCTestNotNumericArgs
Scenario	Test NPV formula with not numeric rate value
Input	PMTCT p m x
Expected output	The value p is not numeric
Actual output	The value p is not numeric

	Details
<i>ID</i>	9
<i>Name</i>	testPMTCPeriodsLessZero
<i>Scenario</i>	Test NPV formula with not numeric rate value
<i>Input</i>	PMT C 1 -1 1
<i>Expected output</i>	The values should be greater than 0
<i>Actual output</i>	The values should be greater than 0

	Details
<i>ID</i>	10
<i>Name</i>	testPMTCInvalidNumberPaymentValues
<i>Scenario</i>	Test PMTC with invalid number of arguments
<i>Input</i>	PMT C 1 3 2 1 1 5
<i>Expected output</i>	The size of arguments is not right
<i>Actual output</i>	The size of arguments is not right

	Details
<i>ID</i>	11
<i>Name</i>	testPVResultOK
<i>Scenario</i>	Test Present Value formula with valid arguments
<i>Input</i>	PV 1 2 1 2
<i>Expected output</i>	1.0
<i>Actual output</i>	1.0

	Details
<i>ID</i>	12
<i>Name</i>	testPVNotNumericPeriod
<i>Scenario</i>	Test Present Value formula with not numeric rate value
<i>Input</i>	PV x 1 1 1
<i>Expected output</i>	The value x is not numeric
<i>Actual output</i>	The value x is not numeric

	Details
<i>ID</i>	13
<i>Name</i>	testPVNotNumericVn
<i>Scenario</i>	testPV formula with not numeric rate value
<i>Input</i>	PV 1 1 1 x
<i>Expected output</i>	Invalid numbers

<i>Actual output</i>	Invalid numbers
----------------------	-----------------

	Details
<i>ID</i>	14
<i>Name</i>	testPVInvalidNumberPaymentValues
<i>Scenario</i>	Test PresentValue with number of periods less than number of payment values provided
<i>Input</i>	PV 5 3 2 1
<i>Expected output</i>	Number of periods does not match with number of payment values
<i>Actual output</i>	Number of periods does not match with number of payment values

	Details
<i>ID</i>	15
<i>Name</i>	testPVInvalidNumberPaymentValues
<i>Scenario</i>	Test PresentValue with number of periods less than number of payment values provided
<i>Input</i>	PV 5 3 2 1
<i>Expected output</i>	Number of periods does not match with number of payment values
<i>Actual output</i>	Number of periods does not match with number of payment values

	Details
ID	16
Name	testFVResultOK
Scenario	Test Future Value formula with valid arguments
Input	FV 1 1 1 1
Expected output	3.0
Actual output	3.0

	Details
ID	17
Name	testFVNotNumericRate
Scenario	Test Future Value formula with not numeric rate value
Input	FV a 1 1 1
Expected output	The value a is not numeric
Actual output	The value a is not numeric

	Details
ID	18
Name	testFVNotNumericVn
Scenario	Test Future Value formula with not numeric rate value
Input	FV 1 1 1 a
Expected output	Invalid numbers
Actual output	Invalid numbers

	Details
<i>ID</i>	19
<i>Name</i>	testFVPeriodsLessZero
<i>Scenario</i>	Test with number of periods equal to -1
<i>Input</i>	FV 1 0 1 1
<i>Expected output</i>	The value -1 should be greater than 0
<i>Actual output</i>	The value -1 should be greater than 0

	Details
<i>ID</i>	20
<i>Name</i>	testFVInvalidNumberPaymentValues
<i>Scenario</i>	Test FutureValue with number of periods less than number of payment values provided
<i>Input</i>	FV 1 3 2 1
<i>Expected output</i>	Number of periods does not match with number of payment values
<i>Actual output</i>	Number of periods does not match with number of payment values

	Details
<i>ID</i>	21
<i>Name</i>	testUnknownOperation
<i>Scenario</i>	Test with an unknown operation code
<i>Input</i>	AX 1 1 1 1
<i>Expected output</i>	Invalid operation

<i>Actual output</i>	Invalid operation
----------------------	-------------------

	Details
<i>ID</i>	22
<i>Name</i>	testNPVResultOK
<i>Scenario</i>	Test NPV formula with valid arguments
<i>Input</i>	NPV 1 2 10 1 2
<i>Expected output</i>	11.0
<i>Actual output</i>	11.0

	Details
<i>ID</i>	23
<i>Name</i>	testNPVNotNumericRate
<i>Scenario</i>	Test NPV formula with not numeric rate value
<i>Input</i>	NPV m m x x
<i>Expected output</i>	The value m is not numeric
<i>Actual output</i>	The value m is not numeric

	Details
<i>ID</i>	24
<i>Name</i>	testNPVNotNumericVn
<i>Scenario</i>	Test NPV formula with not numeric payment value
<i>Input</i>	NPV 1 2 10 1 a
<i>Expected output</i>	Invalid numbers

<i>Actual output</i>	Invalid numbers
----------------------	-----------------

	Details
<i>ID</i>	25
<i>Name</i>	testPVInvalidNumberPaymentValues
<i>Scenario</i>	Test NPV with number of periods less than number of payment values provided
<i>Input</i>	NPV 5 3 2
<i>Expected output</i>	Number of periods does not match with number of payment values
<i>Actual output</i>	Number of periods does not match with number of payment values

Running the test cases

ID	Expected output	Observed output	Failure
1	11.0	11.0	None
2	The value m is not numeric	The value m is not numeric	None
3	0,97	0,97	None
4	The value r s is not numeric	The value r s is not numeric	None
5	The value -1 should be greater than 0	The value -1 should be greater than 0	None
6	The size of arguments is not right	The size of arguments is not right	None

7	30.0	3.0	Yes
8	The value p is not numeric	The value p is not numeric	None
9	The values should be greater than 0	The values should be greater than 0	None
10	The size of arguments is not right	The size of arguments is not right	None
11	1.0	1.0	None
12	The value x is not numeric	The value x is not numeric	None
13	Invalid numbers	Invalid numbers	None
14	Number of periods does not match with number of payment values	Number of periods does not match with number of payment values	None
15	Number of periods does not match with number of payment values	Number of periods does not match with number of payment values	None
16	3.0	3.0	None
17	The value a is not numeric	The value a is not numeric	None
17	Invalid numbers	Invalid numbers	None
18	The value -1 should be greater than 0	The value -1 should be greater than 0	None
20	Number of periods does not match with number of payment values	Number of periods does not match with number of payment values	None
21	Invalid operation	Invalid operation	None
22	11.0	11.0	None
23	The value m is not numeric	The value m is not numeric	None
24	Invalid numbers	Invalid numbers	None
25	Number of periods	Number of periods	None

	does not match with number of payment values.	does not match with number of payment values	
--	---	--	--

Notes:

- Failure on test 7 because the rate factor was missing on the PMTC formula

EclEmma Code Coverage: 93,1%

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
▼ financialcalc	84.0 %	1,185	225	1,410
▶ src/test/java	67.5 %	336	162	498
▼ src/main/java	93.1 %	849	63	912
▼ com.financial.calc.util	93.8 %	510	34	544
▶ Cleaner.java	86.5 %	166	26	192
▶ Operation.java	98.3 %	297	5	302
▶ ExceptionMessages.java	0.0 %	0	3	3
▶ CalculatorException.java	100.0 %	9	0	9
▶ Operands.java	100.0 %	38	0	38
▼ com.financial.calc	79.7 %	114	29	143
▶ Client.java	78.2 %	104	29	133
▶ FinancialCalculator.java	100.0 %	10	0	10
▼ com.financial.calc.formulas	100.0 %	225	0	225
▶ FinancialFormula.java	100.0 %	11	0	11
▶ FutureValue.java	100.0 %	40	0	40
▶ InternalRate.java	100.0 %	38	0	38
▶ NetPresentValue.java	100.0 %	38	0	38
▶ PaymentCapitalRecovery.java	100.0 %	30	0	30
▶ PaymentSinkingFund.java	100.0 %	30	0	30
▶ PresentValue.java	100.0 %	38	0	38

Iteration 2

	Details
<i>ID</i>	26
<i>Name</i>	testFVMoreThanTwoDecimal
<i>Scenario</i>	Test FV with a number in the args with more than two decimals
<i>Input</i>	FV 1 1 2 2.22222
<i>Expected output</i>	The values should not have more than two decimals
<i>Actual output</i>	6.22

	Details
<i>ID</i>	27
<i>Name</i>	testNegativeValues
<i>Scenario</i>	Test FV with a negative number in the input
<i>Input</i>	FV -1 1.1 1.1 1.1
<i>Expected output</i>	The values should be greater than 0
<i>Actual output</i>	The values should be greater than 0

	Details
<i>ID</i>	28
<i>Name</i>	testInvalidSizeArgument
<i>Scenario</i>	Test FV with less arguments
<i>Input</i>	FV 1
<i>Expected output</i>	The size of arguments is not right
<i>Actual output</i>	The size of arguments is not right

	Details
<i>ID</i>	29
<i>Name</i>	testMainFVOK
<i>Scenario</i>	Test main function with the call FV with right inputs.
<i>Input</i>	FV 1 1 1 1
<i>Expected output</i>	3.0
<i>Actual output</i>	3.0

	Details
<i>ID</i>	30
<i>Name</i>	testMainFVNoOK
<i>Scenario</i>	Test main function with the call FV with a string value in the input
<i>Input</i>	FV 1 1 1 m
<i>Expected output</i>	Invalid numbers
<i>Actual output</i>	Invalid numbers

	Details
ID	31
Name	testMainPVOK
Scenario	Test main function with the call PV with a righth input
Input	PV 1 2 1 2
Expected output	1.0
Actual output	1.0

ID	Expected output	Observed output	Failure
26	The values should not have more than two decimals	6.22	Yes
27	The values should be greater than 0	The values should be greater than 0	None
28	The size of arguments is not right	The size of arguments is not right	None
29	3.0	3.0	None
30	Invalid numbers	Invalid numbers	None
31	1.0	1.0	None

Notes

- Failure on test 26. In the specification it was written that the maximum number of decimals that a number could have was two. We implemented the program without taking this into account and thanks to this test we were able to realize this.

EclEmma Code Coverage:98,6%

Element	Coverage	Covered Instructio...	Missed Instru...
▼ fc	85,7 %	1.328	
> src/test/java	67,1 %	425	
▼ src/main/java	98,6 %	903	
> com.financial.calc.util	97,6 %	532	
> com.financial.calc	100,0 %	142	
> com.financial.calc.formulas	100,0 %	229	

Iteration 3

	Details
ID	32
Name	OperationTestIRR
Scenario	Testing IRR with a invalid number of payment values
Input	IRR 1 2 10 1
Expected output	Number of periods does not match with number of payment values

<i>Actual output</i>	Number of periods does not match with number of payment values
----------------------	--

	Details
<i>ID</i>	33
<i>Name</i>	testPMTSResultOK2
<i>Scenario</i>	Testing PMTS function without optional argument in the input
<i>Input</i>	PMTS 1 5
<i>Expected output</i>	0.0
<i>Actual output</i>	0.0







	Details
<i>ID</i>	34
<i>Name</i>	testFVResultOK2
<i>Scenario</i>	Testing FV with exactly two decimal paymnet value
<i>Input</i>	PV 1 2 1 2.22
<i>Expected output</i>	1.06
<i>Actual output</i>	1.06

	Details
ID	35
Name	testFVPeriod
Scenario	Testing FV with a no numeric period
Input	PV 1 m 1 2.3
Expected output	The value m is not numeric
Actual output	The value m is not numeric

Running the test cases

ID	Expected output	Observed output	Failure
32	Number of periods does not match with number of payment values	Number of periods does not match with number of payment values	None
33	0.0	0.0	None
34	1.06	1.06	None
35	The value m is not numeric	The value m is not numeric	None

EclEmma Code Coverage:100%

Element	Coverage	Covered Instructio...	Missed Instruct
Financial_Ca	 84,4 %	1.450	
> src/test/java	 64,6 %	491	
src/main/java	 100,0 %	959	
> com.financial.calc	 100,0 %	148	
> com.financial.calc.formulas	 100,0 %	229	
> com.financial.calc.util	 100,0 %	582	

ANNEX- SOURCE CODE

```

1  package com.financial.calc;
//AAAAAAAAAAAAAAAAAAAAAAAAAAAAH
import com.financial.calc.formulas.FutureValue;
import com.financial.calc.formulas.InternalRate;
import com.financial.calc.formulas.NetPresentValue;
import com.financial.calc.formulas.PaymentCapitalRecovery;
import com.financial.calc.formulas.PaymentSinkingFund;
import com.financial.calc.formulas.PresentValue;
import com.financial.calc.util.CalculatorException;
import com.financial.calc.util.Operands;
import com.financial.calc.util.Operation;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

/**
 * Client class which receives from shell parameters
 * and use the FinancialCalculator based on the
 * required operation .
 * @author cristina.martin
 * @author cristian.castro
 */
public class Client {

    public static boolean termino=false;

    /**
     * Method to choose a formula based on the kind of operation
     * provided by the user .
     *
     * @param operation the enum operation to be done .
     * @param arguments an string args which contains the args received from shell.
     * @return Formula to compute the result .
     * **/
    private Formula chooseFormula(Operation operation, String[] arguments)
        throws CalculatorException {

        Formula formula;
        Operands operands = operation.buildOperands(arguments);

        switch (operation) {
            case PRESENT_VALUE:
                formula = new PresentValue(operands);
                break;
            case FUTURE_VALUE:
                formula = new FutureValue(operands);
                break;
            case PAYMENT_CAPITAL_RECOVERY:
                formula = new PaymentCapitalRecovery(operands);
                break;
            case PAYMENT_SINKING_FUND:
                formula = new PaymentSinkingFund(operands);
                break;
            case NET_PRESENT_VALUE:
                formula = new NetPresentValue(operands);
                break;
            case INTERNAL_RATE_OF_RETURN:
                formula = new InternalRate(operands);
                break;
            default:
                throw new CalculatorException(FinancialCalculator.INVALID_OPERATION);
        }

        return formula;
    }

    public Double getResult(String line) throws IOException, CalculatorException {

        Double result = 0.0;
        double scale = Math.pow(10, 2);

        // for (String line = reader.readLine(); line != null; line = reader.readLine()) {
        try {
            String[] arguments = line.split("\\s+");
            Operation operation = Operation.fromOperationString(arguments[0].toUpperCase());
            Formula formula = chooseFormula(operation, arguments);
            FinancialCalculator financialCalculator = new FinancialCalculator(formula);
            result = financialCalculator.compute();
        } catch (CalculatorException exception) {
            throw new CalculatorException(exception.getMessage(), exception);
        }
    }
}

```

```

        return Math.round(result * scale) / scale;
    }

    /**
     * Main method which receives the arguments from shell and manages
     * the flow of the input and output of financial operations .
     * @throws CalculatorException
     */
    public static void main (String[] args) throws IOException {

        try {
            Client client = new Client();
            BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
            String line = reader.readLine();
            Double result = client.getResult(line);
            System.out.println(result);
        } catch (CalculatorException e) {
            System.out.println(e.getMessage());
        }

        termino = true;
    }
}

```

```

1  package com.financial.calc.util;

import static com.financial.calc.util.Cleaner.getOperands;
import static com.financial.calc.util.Cleaner.getOperandsPMTTC;
import static com.financial.calc.util.Cleaner.getOperandsPMTS;

import com.financial.calc.FinancialCalculator;

/**
 * Operation Enum class to build operands based on the
 * arguments provided .
 * @author cristina.martin
 * @author cristian.castro
 */
public enum Operation {

    PRESENT_VALUE("PV") {

        public Operands buildOperands(String[] arguments) throws CalculatorException {

            Operands operands;
            try {
                operands = getOperands(arguments);
                // Check that the # periods + 1 = # payments
                if (operands.getPeriods() != operands.getPaymentValues().size()) {
                    throw new
                        CalculatorException(FinancialCalculator.INVALID_NUMBER_PAYMENT_VALUES);
                }
            } catch (CalculatorException e){
                throw new CalculatorException(e.getMessage(), e);
            }

            return operands;
        }
    },

    FUTURE_VALUE("FV") {

        public Operands buildOperands(String[] arguments)
            throws CalculatorException {

            Operands operands;
            try {
                operands = getOperands(arguments);
                // Check that the # periods + 1 = # payments
                if (operands.getPeriods() + 1 != operands.getPaymentValues().size()) {
                    throw new
                        CalculatorException(FinancialCalculator.INVALID_NUMBER_PAYMENT_VALUES);
                }
            } catch (CalculatorException e){
                throw new CalculatorException(e.getMessage(), e);
            }

            return operands;
        }
    },

    PAYMENT_CAPITAL_RECOVERY("PMTTC"){

        public Operands buildOperands(String[] arguments) throws CalculatorException {
            Operands operands;
            try {
                operands = getOperandsPMTTC(arguments);
            } catch (CalculatorException e){
                throw new CalculatorException(e.getMessage(), e);
            }

            return operands;
        }
    },

    PAYMENT_SINKING_FUND("PMTS"){

        public Operands buildOperands(String[] arguments) throws CalculatorException {
            Operands operands;
            try {
                operands = getOperandsPMTS(arguments);
            } catch (CalculatorException e){
                throw new CalculatorException(e.getMessage(), e);
            }

            return operands;
        }
    },

    NET_PRESENT_VALUE("NPV") {

```

```

    public Operands buildOperands(String[] arguments) throws CalculatorException {
        Operands operands;
        try {
            operands = getOperands(arguments);
            if (operands.getPeriods() + 1 != operands.getPaymentValues().size()) {
                throw new
                    CalculatorException(FinancialCalculator.INVALID_NUMBER_PAYMENT_VALUES);
            }
        } catch (CalculatorException e) {
            throw new CalculatorException(e.getMessage(), e);
        }

        return operands;
    }
},
INTERNAL_RATE_OF_RETURN("IRR"){
    public Operands buildOperands(String[] arguments) throws CalculatorException {

        Operands operands;
        try {
            Cleaner cl = new Cleaner();
            operands = cl.getOperands(arguments);
            if (operands.getPeriods() + 1 != operands.getPaymentValues().size()) {
                throw new
                    CalculatorException(FinancialCalculator.INVALID_NUMBER_PAYMENT_VALUES);
            }
        } catch (CalculatorException e) {
            throw new CalculatorException(e.getMessage(), e);
        }
        return operands;
    }
},
UNKNOWN("UNKNOWN"){
    public Operands buildOperands(String[] arguments) throws CalculatorException {

        return null;
    }
};

private String operation;

Operation(String operation) {
    this.operation = operation;
}

public String getOperation(){
    return this.operation;
}

public abstract Operands buildOperands(String[] arguments) throws CalculatorException;

/**
 * Method for get a valid Operation enum from a string.
 * If the operation is not found then return a UNKNOWN operation .
 * @param operationString the name of operation .
 * @return Operation the type of operation to perform .
 */
public static Operation fromOperationString(String operationString)
{
    for(Operation operation : Operation.values()) {
        if(operation.getOperation().equals(operationString)) {
            return operation;
        }
    }

    return UNKNOWN; //not found
}
}

```



```

1  package com.financial.calc.util;

import java.util.List;

/**
 * Operands class to compute the formula using the params .
 *
 * @author cristina.martin
 * @author cristian.castro
 */
public class Operands {
    /**
     * Represent the interest rate per period
     */
    private Double rate;
    /**
     * Specifies number of payments in annuity
     */
    private Integer periods;
    /**
     * Specifies present cash flow
     */
    private Double presentCash;
    /**
     * Specifies future cash flow
     */
    private Double futureCash;
    /**
     * List of payments including initial cost
     */
    private List<Double> paymentValues;

    public Double getRate() {
        return rate;
    }

    public void setRate(Double rate) {
        this.rate = rate;
    }

    public Integer getPeriods() {
        return periods;
    }

    public void setPeriods(Integer periods) {
        this.periods = periods;
    }

    public Double getPresentCash() {
        return presentCash;
    }

    public void setPresentCash(Double presentCash) {
        this.presentCash = presentCash;
    }

    public Double getFutureCash() {
        return futureCash;
    }

    public void setFutureCash(Double futureCash) {
        this.futureCash = futureCash;
    }

    public List<Double> getPaymentValues() {
        return paymentValues;
    }

    public void setPaymentValues(List<Double> paymentValues) {
        this.paymentValues = paymentValues;
    }
}

```

```

1  package com.financial.calc.util;

import java.math.BigDecimal;
import java.util.Arrays;
import java.util.stream.Collectors;

import com.financial.calc.FinancialCalculator;

/**
 * Utility cleaner class to verify that arguments are correct .
 * @author cristina.martin
 * @author cristian.castro
 */
public class Cleaner {

    public Cleaner() {

    }

    /**
     * Check that a value is Integer and return its value
     * otherwise return null .
     * @param value the string value .
     * @return the parsed value .
     */
    public static Integer checkIntValue(String value)
        throws CalculatorException {
        Integer intValue;

        try {
            intValue = Integer.valueOf(value);
            if (intValue <= 0) {
                throw new
                    CalculatorException(
                        FinancialCalculator.NUMBER_LESS_OR_EQUAL_ZERO);
            }
        } catch (NumberFormatException e) {
            throw new
                CalculatorException(
                    String.format(FinancialCalculator.NOT_NUMERIC_VALUE, value));
        }

        return intValue;
    }

    /**
     * Check that a value is Double and return its value
     * otherwise return null .
     * @param value the string value .
     * @return the parsed value .
     */
    public static Double checkDoubleValue(String value)
        throws CalculatorException {
        Double doubleValue;

        try {
            doubleValue = Double.valueOf(value);

            if (doubleValue <= 0) {
                throw new
                    CalculatorException(
                        FinancialCalculator.NUMBER_LESS_OR_EQUAL_ZERO);
            } else if (BigDecimal.valueOf(doubleValue).scale() > 2) {
                throw new
                    CalculatorException(
                        FinancialCalculator.MORE_THAN_TWO_DECIMALS);
            }
        } catch (NumberFormatException e) {
            throw new
                CalculatorException(
                    String.format(FinancialCalculator.NOT_NUMERIC_VALUE, value));
        }

        return doubleValue;
    }

    /** Method to verify that input values from shell are right for
     * FutureValue, PresentValue, IRR, NPV formulas.
     * Check if:
     *     Contains valid rate (double value greater than 0)

```

```

*         Contains valid periods (integer value greater than 0)
*         Contains a set of values for payment (double values greater than 0)
* @param args the arguments provided by the user.
* @return true if the arguments are correct .
**/
public static Operands getOperands(String[] args)
    throws CalculatorException {

    int size = args.length;
    Operands operands = new Operands();

    if (size >= 3) {
        try {
            Double rate = checkDoubleValue(args[1]);
            Integer periods = checkIntValue(args[2]);

            operands.setRate(rate);
            operands.setPeriods(periods);
            operands.setPaymentValues(
                Arrays.stream(args)
                    .skip(3)
                    .map(Double::valueOf)
                    // .filter(value -> !(BigDecimal.valueOf(value).scale() > 2))
                    .collect(Collectors.toList()));

        } catch (NumberFormatException e) {
            throw new
                CalculatorException(FinancialCalculator.INVALID_NUMBERS);
        }
    } else {
        throw new
            CalculatorException(FinancialCalculator.INVALID_SIZE_OF_ARGUMENTS);
    }

    return operands;
}

/** Method to verify that input values from shell are right for
 * PTMC formula.
 * Check if:
 *     Contains valid rate (double value greater than 0)
 *     Contains valid periods (integer value greater than 0)
 *     Contains a set of values for payment (double values greater than 0)
 * @param args the arguments provided by the user.
 * @return true if the arguments are correct .
 **/
public static Operands getOperandsPMTC(String[] args)
    throws CalculatorException {

    int size = args.length;
    Operands operands = new Operands();

    if (size == 4) {
        Double rate = checkDoubleValue(args[1]);
        Integer periods = checkIntValue(args[2]);
        Double payment = checkDoubleValue(args[3]);

        operands.setRate(rate);
        operands.setPeriods(periods);
        operands.setPresentCash(payment);

    } else {
        throw new
            CalculatorException(FinancialCalculator.INVALID_SIZE_OF_ARGUMENTS);
    }

    return operands;
}

/** Method to verify that input values from shell are right for
 * PTMS formula.
 * Check if:
 *     Contains valid rate (double value greater than 0)
 *     Contains valid periods (integer value greater than 0)
 *     Contains a set of values for payment (double values greater than 0)
 * @param args the arguments provided by the user.
 * @return true if the arguments are correct .
 **/
public static Operands getOperandsPMTS(String[] args)
    throws CalculatorException {

    int size = args.length;
    Operands operands = new Operands();

```

```
if (size == 4) {
    Double rate = checkDoubleValue(args[1]);
    Integer periods = checkIntValue(args[2]);
    Double payment = checkDoubleValue(args[3]);

    operands.setRate(rate);
    operands.setPeriods(periods);
    operands.setFutureCash(payment);

}else if (size==3) {
    Double rate = checkDoubleValue(args[1]);
    Integer periods = checkIntValue(args[2]);
    Double payment = 0.0;
    operands.setRate(rate);
    operands.setPeriods(periods);
    operands.setFutureCash(payment);
}
else {
    throw new
        CalculatorException(FinancialCalculator.INVALID_SIZE_OF_ARGUMENTS);
}

return operands;
}
```

CalculatorException.java

```
1  package com.financial.calc.util;

/**
 * Custom calculator exception to retrieve .
 *
 * @author cristina.martin
 * @author cristian.castro
 */
public class CalculatorException extends Exception {
    public CalculatorException(String message) { super(message); }
    public CalculatorException(String message, Throwable cause) { super(message, cause); }
}
```

Formula.java

```
1  package com.financial.calc;

/**
 * Interface with a single method to apply Command pattern.
 * @author cristina.martin
 * @author cristian.castro
 */
public interface Formula {

    Double compute();

}
```

```
1  package com.financial.calc.formulas;

import com.financial.calc.Formula;
import com.financial.calc.util.Operands;

/**
 * FinancialFormula which implements Formula to compute the result
 * based on the arguments provided .
 * @author cristina.martin
 * @author cristian.castro
 */
public abstract class FinancialFormula implements Formula {

    protected Double rate;
    protected Integer periods;

    public FinancialFormula(Operands operands){
        this.rate = operands.getRate();
        this.periods = operands.getPeriods();
    }

    public abstract Double compute();
}
```

```

1  package com.financial.calc;

import static junit.framework.TestCase.fail;
import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThat;

import com.financial.calc.util.CalculatorException;
import java.io.IOException;
import org.junit.Before;
import org.junit.Test;

public class PaymentSinkingFund {

    Client client;

    @Before
    public void init() {
        client = new Client();
    }

    /**
     * Scenario: Test PMTS formula with valid arguments .
     * Input: FV 10 5 3.
     * Expected: 0.97 .
     */
    @Test
    public void testPMTSResultOK() throws IOException {
        String args = "PMTS 1 5 30";

        try {
            Double result =client.getResult(args);
            assertEquals(Double.valueOf(0.97), result);
        } catch (CalculatorException e) {
            fail("No exception should be thrown");
        }
    }

    @Test
    public void testPMTSResultOK2() throws IOException {
        String args = "PMTS 1 5";

        try {
            Double result =client.getResult(args);
            assertEquals(Double.valueOf(0.00), result);
        } catch (CalculatorException e) {
            fail("No exception should be thrown");
        }
    }

    /**
     * Scenario: Test PMTS formula with not numeric rate value .
     * Input: IR m m x x .
     * Expected: A message showing that The value m is not numeric .
     */
    @Test
    public void testPMTSNotNumericArgs() throws IOException {
        String args = "PMTS r m x";

        try {
            client.getResult(args);
            fail("An exception should be thrown");
        } catch (CalculatorException e) {
            assertThat(e.getMessage(), is(String.format(FinancialCalculator.NOT_NUMERIC_VALUE, "r")));
        }
    }

    /**
     * Scenario: Test PMTS with number of periods equal to -1 .
     * Input: PMTS 1 -1 1 1 .
     * Expected: "The value -1 should be greater than 0".
     */
    @Test
    public void testPMTSPeriodsLessZero() throws IOException {
        String args = "PMTS 1 -1 1";

        try {
            client.getResult(args);
            fail("An exception should be thrown");
        } catch (CalculatorException e) {
    
```



```

        assertThat(e.getMessage(), is(FinancialCalculator.NUMBER_LESS_OR_EQUAL_ZERO));
    }
}

/**
 * Scenario: Test PMTS with invalid number of arguments .
 * Input: PMTS 1 3 2 1 1 .
 * Expected: "The size of arguments is not right".
 */
@Test
public void testPMTSInvalidNumberPaymentValues() throws IOException {
    String args = "PMTS 1 3 2 1 1";

    try {
        client.getResult(args);
        fail("An exception should be thrown");
    } catch (CalculatorException e) {
        assertThat(e.getMessage(), is(FinancialCalculator.INVALID_SIZE_OF_ARGUMENTS));
    }
}
}

```

```

1  package com.financial.calc.formulas;

import com.financial.calc.util.Operands;
import java.util.List;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * Class for computation of PresentValue which extends from FinancialFormula
 *
 *
 * @author cristina.martin
 * @author cristian.castro
 */
public class PresentValue extends FinancialFormula {

    private List<Double> paymentValues;

    public PresentValue(Operands operands) {
        super(operands);
        this.paymentValues = operands.getPaymentValues();
    }

    /**
     * Implement compute method for a PresentValue formula .
     * @return the result of computation .
     */
    public Double compute() {

        AtomicInteger counter = new AtomicInteger(1);
        double localRate = 1 + rate;

        return paymentValues
            .stream()
            .mapToDouble(value -> value / Math.pow(localRate, counter.getAndIncrement()))
            .sum();
    }
}

```

```

1  package com.financial.calc;

import static junit.framework.TestCase.fail;
import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThat;

import com.financial.calc.util.CalculatorException;
import java.io.IOException;
import org.junit.Before;
import org.junit.Test;

public class InternalRateTest {

    Client client;

    @Before
    public void init() {
        client = new Client();
    }

    /**
     * Scenario: Test Internal Rate formula with valid arguments .
     * Input: IR 1 2 10 1 2 .
     * Expected: 11.0 .
     */
    @Test
    public void testIRRResultOK() throws IOException {
        String args = "IRR 1 2 10 1 2";

        try {
            Double result = client.getResult(args);
            assertEquals(Double.valueOf(11.0), result);
        } catch (CalculatorException e) {
            fail("No exception should be thrown");
        }
    }

    /**
     * Scenario: Test Future Value formula with not numeric rate value .
     * Input: IR m m -1 1 .
     * Expected: A message showing that The value m is not numeric .
     */
    @Test
    public void testIRRNotNumericArgs() throws IOException {
        String args = "IRR m m -1 1";

        try {
            client.getResult(args);
            fail("An exception should be thrown");
        } catch (CalculatorException e) {
            assertThat(e.getMessage(), is(String.format(FinancialCalculator.NOT_NUMERIC_VALUE, "m")));
        }
    }

    /**
     * Scenario: Test without values
     * Input: IRR
     * Expected: A message showing that the size of the arguments is not righth.
     */
    @Test
    public void OperationTestIRR() throws IOException {
        String args = "IRR 1 2 10 1";
        try {
            client.getResult(args);
        } catch (CalculatorException e) {
            assertThat(e.getMessage(), is(String.format(FinancialCalculator.INVALID_NUMBER_PAYMENT_VALUES, "?")));
        }
    }
}

```

FutureValue.java

```
1  package com.financial.calc.formulas;

import com.financial.calc.util.Operands;
import java.util.List;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * Class for computation of FutureValue which extends from FinancialFormula
 *
 *
 * @author cristina.martin
 * @author cristian.castro
 */
public class FutureValue extends FinancialFormula {

    private List<Double> paymentValues;

    public FutureValue(Operands operands) {
        super(operands);
        this.paymentValues = operands.getPaymentValues();
    }

    /**
     * Implement compute method for a FutureValue formula .
     * @return the result of computation .
     *
     */
    public Double compute() {

        AtomicInteger counter = new AtomicInteger(periods );
        double localRate = 1 + rate;

        return paymentValues
            .stream()
            .mapToDouble(value -> value * Math.pow(localRate, counter.getAndDecrement()))
            .sum();

    }

}
```

PaymentCapitalRecovery.java

```
1  package com.financial.calc.formulas;

import com.financial.calc.util.Operands;

/**
 * Class for computation of PaymentCapitalRecovery which extends from FinancialFormula
 *
 * @author cristina.martin
 * @author cristian.castro
 */
public class PaymentCapitalRecovery extends FinancialFormula {

    private Double presentCash;

    public PaymentCapitalRecovery(Operands operands) {
        super(operands);
        this.presentCash = operands.getPresentCash();
    }

    /**
     * Implement compute method for a PaymentCapitalRecovery formula .
     * @return the result of computation .
     */
    public Double compute() {

        double power_rate = Math.pow(1 + rate, periods);
        return presentCash * rate* power_rate / (power_rate - 1);
    }
}
```

```

1  package com.financial.calc;

import static junit.framework.TestCase.fail;
import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThat;

import com.financial.calc.util.CalculatorException;
import java.io.IOException;
import org.junit.Before;
import org.junit.Test;

public class InternalRateTest {

    Client client;

    @Before
    public void init() {
        client = new Client();
    }

    /**
     * Scenario: Test Internal Rate formula with valid arguments .
     * Input: IR 1 2 10 1 2 .
     * Expected: 11.0 .
     */
    @Test
    public void testIRRResultOK() throws IOException {
        String args = "IRR 1 2 10 1 2";

        try {
            Double result = client.getResult(args);
            assertEquals(Double.valueOf(11.0), result);
        } catch (CalculatorException e) {
            fail("No exception should be thrown");
        }
    }

    /**
     * Scenario: Test Future Value formula with not numeric rate value .
     * Input: IR m m -1 1 .
     * Expected: A message showing that The value m is not numeric .
     */
    @Test
    public void testIRRNotNumericArgs() throws IOException {
        String args = "IRR m m -1 1";

        try {
            client.getResult(args);
            fail("An exception should be thrown");
        } catch (CalculatorException e) {
            assertThat(e.getMessage(), is(String.format(FinancialCalculator.NOT_NUMERIC_VALUE, "m")));
        }
    }

    /**
     * Scenario: Test without values
     * Input: IRR
     * Expected: A message showing that the size of the arguments is not righth.
     */
    @Test
    public void OperationTestIRR() throws IOException {
        String args = "IRR 1 2 10 1";
        try {
            client.getResult(args);
        } catch (CalculatorException e) {
            assertThat(e.getMessage(), is(String.format(FinancialCalculator.INVALID_NUMBER_PAYMENT_VALUES, "?")));
        }
    }
}

```

```

1  package com.financial.calc;

import static junit.framework.TestCase.fail;
import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThat;

import com.financial.calc.util.CalculatorException;
import java.io.IOException;
import org.junit.Before;
import org.junit.Test;

public class NetPresentValue {

    Client client;

    @Before
    public void init() {
        client = new Client();
    }

    /**
     * Scenario: Test NPV formula with valid arguments .
     * Input: IR 1 2 10 1 2 .
     * Expected: 11.0 .
     */
    @Test
    public void testNPVResultOK() throws IOException {
        String args = "NPV 1 2 10 1 2";

        try {
            Double result = client.getResult(args);
            assertEquals(Double.valueOf(11.0), result);
        } catch (CalculatorException e) {
            fail("No exception should be thrown");
        }
    }

    /**
     * Scenario: Test NPV formula with not numeric rate value .
     * Input: IR m m x x .
     * Expected: A message showing that The value m is not numeric .
     */
    @Test
    public void testNPVNotNumericRate() throws IOException {
        String args = "NPV m m x x";

        try {
            client.getResult(args);
            fail("An exception should be thrown");
        } catch (CalculatorException e) {
            assertThat(e.getMessage(), is(String.format(FinancialCalculator.NOT_NUMERIC_VALUE, "m")));
        }
    }

    /**
     * Scenario: Test NPV formula with not numeric rate value .
     * Input: "NPV 1 2 10 1 a" .
     * Expected: A message showing that The value m is not numeric .
     */
    @Test
    public void testNPVNotNumericVn() throws IOException {
        String args = "NPV 1 2 10 1 a";

        try {
            client.getResult(args);
            fail("An exception should be thrown");
        } catch (CalculatorException e) {
            assertThat(e.getMessage(), is(FinancialCalculator.INVALID_NUMBERS));
        }
    }

    /**
     * Scenario: Test NPV with number of periods less than number
     *           of payment values provided .
     * Input: NPV 5 3 2 .
     * Expected: "Number of periods does not match with number of payment values".
     */

```

```
@Test
public void testPVInvalidNumberPaymentValues() throws IOException {
    String args = "NPV 5 3 2";

    try {
        client.getResult(args);
        fail("An exception should be thrown");
    } catch (CalculatorException e) {
        assertThat(e.getMessage(), is(FinancialCalculator.INVALID_NUMBER_PAYMENT_VALUES));
    }
}
}
```



```

1  package com.financial.calc;

import static junit.framework.TestCase.fail;
import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThat;

import com.financial.calc.util.CalculatorException;
import java.io.IOException;
import org.junit.Before;
import org.junit.Test;

public class PaymentCapitalRecoveryTest {

    Client client;

    @Before
    public void init() {
        client = new Client();
    }

    /**
     * Scenario: Test Future Value formula with valid arguments .
     * Input: PMTC 1 1 1 1 .
     * Expected: 3.0 .
     */
    @Test
    public void testPMTCTestResultOK() throws IOException {
        String args = "PMTTC 10 5 3";

        try {
            Double result = client.getResult(args);
            assertEquals(Double.valueOf(30.0), result);
        } catch (CalculatorException e) {
            fail("No exception should be thrown");
        }
    }

    /**
     * Scenario: Test NPV formula with not numeric rate value .
     * Input: IR m m x x .
     * Expected: A message showing that The value m is not numeric .
     */
    @Test
    public void testPMTCTestNotNumericArgs() throws IOException {
        String args = "PMTTC p m x";

        try {
            client.getResult(args);
            fail("An exception should be thrown");
        } catch (CalculatorException e) {
            assertThat(e.getMessage(), is(String.format(FinancialCalculator.NOT_NUMERIC_VALUE, "p")));
        }
    }

    /**
     * Scenario: Test with number of periods equal to -1 .
     * Input: PMTC 1 -1 1 1 .
     * Expected: "The value -1 should be greater than 0".
     */
    @Test
    public void testPMTCTestPeriodsLessZero() throws IOException {
        String args = "PMTTC 1 -1 1";

        try {
            client.getResult(args);
            fail("An exception should be thrown");
        } catch (CalculatorException e) {
            assertThat(e.getMessage(), is(FinancialCalculator.NUMBER_LESS_OR_EQUAL_ZERO));
        }
    }

    /**
     * Scenario: Test PMTC with invalid number of arguments .
     * Input: PMTC 1 3 2 1 1 .
     * Expected: "The size of arguments is not right".
     */
    @Test
    public void testPMTCTestInvalidNumberPaymentValues() throws IOException {
        String args = "PMTTC 1 3 2 1 1 5";
    }
}

```

```
try {
    client.getResult(args);
    fail("An exception should be thrown");
} catch (CalculatorException e) {
    assertThat(e.getMessage(), is(FinancialCalculator.INVALID_SIZE_OF_ARGUMENTS));
}
}
```

```

1  package com.financial.calc;

import static junit.framework.TestCase.fail;
import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThat;

import com.financial.calc.util.CalculatorException;
import java.io.IOException;
import org.junit.Before;
import org.junit.Test;

public class PaymentSinkingFund {

    Client client;

    @Before
    public void init() {
        client = new Client();
    }

    /**
     * Scenario: Test PMTS formula with valid arguments .
     * Input: FV 10 5 3.
     * Expected: 0.97 .
     */
    @Test
    public void testPMTSResultOK() throws IOException {
        String args = "PMTS 1 5 30";

        try {
            Double result =client.getResult(args);
            assertEquals(Double.valueOf(0.97), result);

        } catch (CalculatorException e) {
            fail("No exception should be thrown");
        }
    }

    @Test
    public void testPMTSResultOK2() throws IOException {
        String args = "PMTS 1 5";

        try {
            Double result =client.getResult(args);
            assertEquals(Double.valueOf(0.00), result);

        } catch (CalculatorException e) {
            fail("No exception should be thrown");
        }
    }

    /**
     * Scenario: Test PMTS formula with not numeric rate value .
     * Input: IR m m x x .
     * Expected: A message showing that The value m is not numeric .
     */
    @Test
    public void testPMTSNotNumericArgs() throws IOException {
        String args = "PMTS r m x";

        try {
            client.getResult(args);
            fail("An exception should be thrown");

        } catch (CalculatorException e) {
            assertThat(e.getMessage(), is(String.format(FinancialCalculator.NOT_NUMERIC_VALUE, "r")));
        }
    }

    /**
     * Scenario: Test PMTS with number of periods equal to -1 .
     * Input: PMTS 1 -1 1 1 .
     * Expected: "The value -1 should be greater than 0".
     */
    @Test
    public void testPMTSPeriodsLessZero() throws IOException {
        String args = "PMTS 1 -1 1";

        try {
            client.getResult(args);
            fail("An exception should be thrown");

        } catch (CalculatorException e) {

```

```

        assertThat(e.getMessage(), is(FinancialCalculator.NUMBER_LESS_OR_EQUAL_ZERO));
    }
}

/**
 * Scenario: Test PMTS with invalid number of arguments .
 * Input: PMTS 1 3 2 1 1 .
 * Expected: "The size of arguments is not right".
 */
@Test
public void testPMTSInvalidNumberPaymentValues() throws IOException {
    String args = "PMTS 1 3 2 1 1";

    try {
        client.getResult(args);
        fail("An exception should be thrown");
    } catch (CalculatorException e) {
        assertThat(e.getMessage(), is(FinancialCalculator.INVALID_SIZE_OF_ARGUMENTS));
    }
}
}

```

PresentValueTest.java

```

1  package com.financial.calc;

import static junit.framework.TestCase.fail;
import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThat;

import com.financial.calc.util.CalculatorException;
import java.io.IOException;
import org.junit.Before;
import org.junit.Test;

public class PresentValueTest {

    Client client;

    @Before
    public void init() {
        client = new Client();
    }

    /**
     * Scenario: Test Present Value formula with valid arguments .
     * Input: PV 1 2 1 2 .
     * Expected: 1.0 .
     */
    @Test
    public void testFVResultOK() throws IOException {
        String args = "PV 1 2 1 2";

        try {
            Double result = client.getResult(args);
            assertEquals(Double.valueOf(1.0), result);
        } catch (CalculatorException e) {
            fail("No exception should be thrown");
        }
    }

    @Test
    public void testFVResultOK2() throws IOException {
        String args = "PV 1 2 1 2.22";

        try {
            Double result = client.getResult(args);
            assertEquals(Double.valueOf(1.06), result);
        } catch (CalculatorException e) {
            fail("No exception should be thrown");
        }
    }

    public void testFVMoreThan2Decimal() throws IOException {
        String args = "PV 1 1 1 2.2233";

        try {
            Double result = client.getResult(args);
            fail("throw exception");
        } catch (CalculatorException e) {
            assertThat(e.getMessage(), is(String.format(FinancialCalculator.MORE_THAN_TWO_DECIMALS)));
        }
    }

    /**
     * Scenario: Test Present Value formula with not numeric rate value .
     * Input: PV x 1 1 1 .
     * Expected: A message showing that The value x is not numeric .
     */
    @Test
    public void testPVNotNumericPeriod() throws IOException {
        String args = "PV x 1 1 1";

        try {
            client.getResult(args);
            fail("An exception should be thrown");
        } catch (CalculatorException e) {
            assertThat(e.getMessage(), is(String.format(FinancialCalculator.NOT_NUMERIC_VALUE, "x")));
        }
    }
}

```

```

/**
 * Scenario: Test FV formula with not numeric rate value .
 * Input: "PV 1 1 1 x" .
 * Expected: A message showing that The value m is not numeric .
 */
@Test
public void testFVNotNumericVn() throws IOException {
    String args = "FV 1 1 1 x";

    try {
        client.getResult(args);
        fail("An exception should be thrown");
    } catch (CalculatorException e) {
        assertThat(e.getMessage(), is(FinancialCalculator.INVALID_NUMBERS));
    }
}

/**
 * Scenario: Test PresentValue with number of periods less than number
 *           of payment values provided .
 * Input: FV 1 3 2 1 1 .
 * Expected: "Number of periods does not match with payment values".
 */
@Test
public void testPVInvalidNumberPaymentValues() throws IOException {
    String args = "PV 5 3 2 1";

    try {
        client.getResult(args);
        fail("An exception should be thrown");
    } catch (CalculatorException e) {
        assertThat(e.getMessage(), is(FinancialCalculator.INVALID_NUMBER_PAYMENT_VALUES));
    }
}

@Test
public void testFVPeriod() throws IOException {
    String args = "PV 1 m 1 2.3";

    try {
        System.out.println("ee:" + client.getResult(args));
        fail("throw exception");
    } catch (CalculatorException e) {
        System.out.println(e.getMessage());
        assertThat(e.getMessage(), is(String.format(FinancialCalculator.NOT_NUMERIC_VALUE, "m")));
    }
}
}

```

```

1  package com.financial.calc;

import com.financial.calc.util.CalculatorException;

import java.io.IOException;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;
import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThat;
import static org.junit.Assert.assertTrue;
import static junit.framework.TestCase.fail;

public class FutureValueTest {

    Client client;

    @Before
    public void init() {
        client = new Client();
    }

    /**
     * Scenario: Test Future Value formula with valid arguments .
     * Input: FV 1 1 1 1 .
     * Expected: 3.0 .
     */
    @Test
    public void testFVResultOK() throws IOException {
        String args = "FV 1 1 1 1";

        try {
            Double result = client.getResult(args);
            assertEquals(Double.valueOf(3.0), result);
        } catch (CalculatorException e) {
            fail("No exception should be thrown");
        }
    }

    /**
     * Scenario: Test Future Value formula with not numeric rate value .
     * Input: FV a 1 1 1 .
     * Expected: A message showing that The value a is not numeric .
     */
    @Test
    public void testFVNotNumericRate() throws IOException {
        String args = "FV a 1 1 1";

        try {
            client.getResult(args);
            fail("An exception should be thrown");
        } catch (CalculatorException e) {
            assertThat(e.getMessage(), is(String.format(FinancialCalculator.NOT_NUMERIC_VALUE, "a")));
        }
    }

    /**
     * Scenario: Test FV formula with not numeric rate value .
     * Input: "NPV 1 2 10 1 a" .
     * Expected: A message showing that The value a is not numeric .
     */
    @Test
    public void testFVNotNumericVn() throws IOException {
        String args = "FV 1 1 1 a";

        try {
            client.getResult(args);
            fail("An exception should be thrown");
        } catch (CalculatorException e) {
            assertThat(e.getMessage(), is(FinancialCalculator.INVALID_NUMBERS));
        }
    }
}

```

```

/**
 * Scenario: Test with number of periods equal to -1 .
 * Input: FV 1 -1 1 1 .
 * Expected: "The value -1 should be greater than 0".
 */
@Test
public void testFVPeriodsLessZero() throws IOException {
    String args = "FV 1 0 1 1";

    try {
        client.getResult(args);
        fail("An exception should be thrown");
    } catch (CalculatorException e) {
        assertEquals(e.getMessage(), is(FinancialCalculator.NUMBER_LESS_OR_EQUAL_ZERO));
    }
}

/**
 * Scenario: Test FutureValue with number of periods less than number
 * of payment values provided .
 * Input: FV 1 3 2 1 1 .
 * Expected: "Number of periods does not match with payment values".
 */
@Test
public void testFVInvalidNumberPaymentValues() throws IOException {
    String args = "FV 1 3 2 1";

    try {
        client.getResult(args);
        fail("An exception should be thrown");
    } catch (CalculatorException e) {
        assertEquals(e.getMessage(), is(FinancialCalculator.INVALID_NUMBER_PAYMENT_VALUES));
    }
}

/**
 * Scenario: Test with an unknown operation code
 * Input: AX 1 1 1 1 .
 * Expected: A message showing that Invalid operation.
 */
@Test
public void testUnknownOperation() throws IOException {
    String args = "AX 1 1 1 1";

    try {
        client.getResult(args);
        fail("An exception should be thrown");
    } catch (CalculatorException e) {
        assertEquals(e.getMessage(), is(FinancialCalculator.INVALID_OPERATION));
    }
}

@Test
public void testNegativeValuesDouble() throws IOException {
    String args = "FV -1.1 1.1 1.1 1.1";

    try {
        client.getResult(args);
        fail("An exception should be thrown");
    } catch (CalculatorException e) {
        assertEquals(e.getMessage(), is(FinancialCalculator.NUMBER_LESS_OR_EQUAL_ZERO));
    }
}

@Test
public void testInvalidSizeArgument() throws IOException {
    String args = "FV 1";

    try {
        client.getResult(args);
        fail("An exception should be thrown");
    } catch (CalculatorException e) {
        assertEquals(e.getMessage(), is(FinancialCalculator.INVALID_SIZE_OF_ARGUMENTS));
    }
}

@Test
public void testRateWithMoreThanTwoDecimals() throws IOException {
    String args = "FV 1.222 2 1 2 3";

```



```

    try {
        client.getResult(args);
        fail("An exception should be thrown");

    } catch (CalculatorException e) {
        assertThat(e.getMessage(), is(FinancialCalculator.MORE_THAN_TWO_DECIMALS));
    }
}

@Test
public void testRateWithTwoDecimals() throws IOException {
    String args = "FV 1.22 2 1 2 3";
    try {
        Double result = client.getResult(args);
        assertEquals(Double.valueOf(12.37), result);
    } catch (CalculatorException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

}

}

```

MainTest.java

```

1  package com.financial.calc;

import com.financial.calc.util.CalculatorException;

import java.io.IOException;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;
import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThat;
import static org.junit.Assert.assertTrue;
import static junit.framework.TestCase.fail;

public class MainTest {

    Client client;

    @Before
    public void init() {
        client = new Client();
    }

    /**
     * Scenario: Test Main finished .
     * Input: FV 1 1 1 1 .
     * Expected: boolean termino = true .
     * @return
     * @throws CalculatorException
     */
    @Test
    public void testMainFVOK() throws CalculatorException {
        String [] args = {"FV 1 1 1 1"};
        try {
            client.main(args);
            assertTrue(Client.termino==true);
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    /**
     * Scenario: Test Main not finished .
     * Input: FV 1 1 1 .
     * Expected: boolean termino = false .
     * @return
     */
    @Test
    public void testMainFVNoOK() {
        String [] args = {"FV 1 1 1 m"};
        try {
            client.main(args);
        } catch (IOException e) {

            //assertThat(e.getMessage(), is(String.format(FinancialCalculator.INVALID_NUMBERS, "m")));
            assertTrue(Client.termino==false);
        }
    }

    @Test
    public void testMainPVOK() throws CalculatorException {
        String [] args = {"PV 1 2 1 2"};
        try {
            client.main(args);
            assertTrue(Client.termino==true);
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

```

CleanerTest.java

```
1  package com.financial.calc;

import com.financial.calc.util.CalculatorException;
import com.financial.calc.util.Cleaner;

import java.io.IOException;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;
import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThat;
import static junit.framework.TestCase.fail;

public class CleanerTest {

    Cleaner cl;

    //Test
    public void testCheckIntValue() throws IOException {
        String value = "m";
        try {
            cl.checkIntValue(value);
        } catch (CalculatorException e) {
            // TODO Auto-generated catch block
            assertThat(e.getMessage(), is(String.format(FinancialCalculator.NOT_NUMERIC_VALUE, "m")));
        }
    }

}
```