UQAC
UNIVERSITÉ DU QUÉBEC
À CHICOUTIMI

Norme ISO 29110 – Architecture et tests unitaires

Travail #3

6GEI264 – Vérification et validation des logiciels

Hiver 2018

Département des Sciences Appliquées

Module d'ingénierie

Présenté à M. Jean-Luc Cyr

Par

Pier-Olivier Vermette
VERP07029605

Jean-Sébastien St-Pierre
STPJ15018206

Date de remise: 22 février 2018

1. Architecture du logiciel Calculatrice

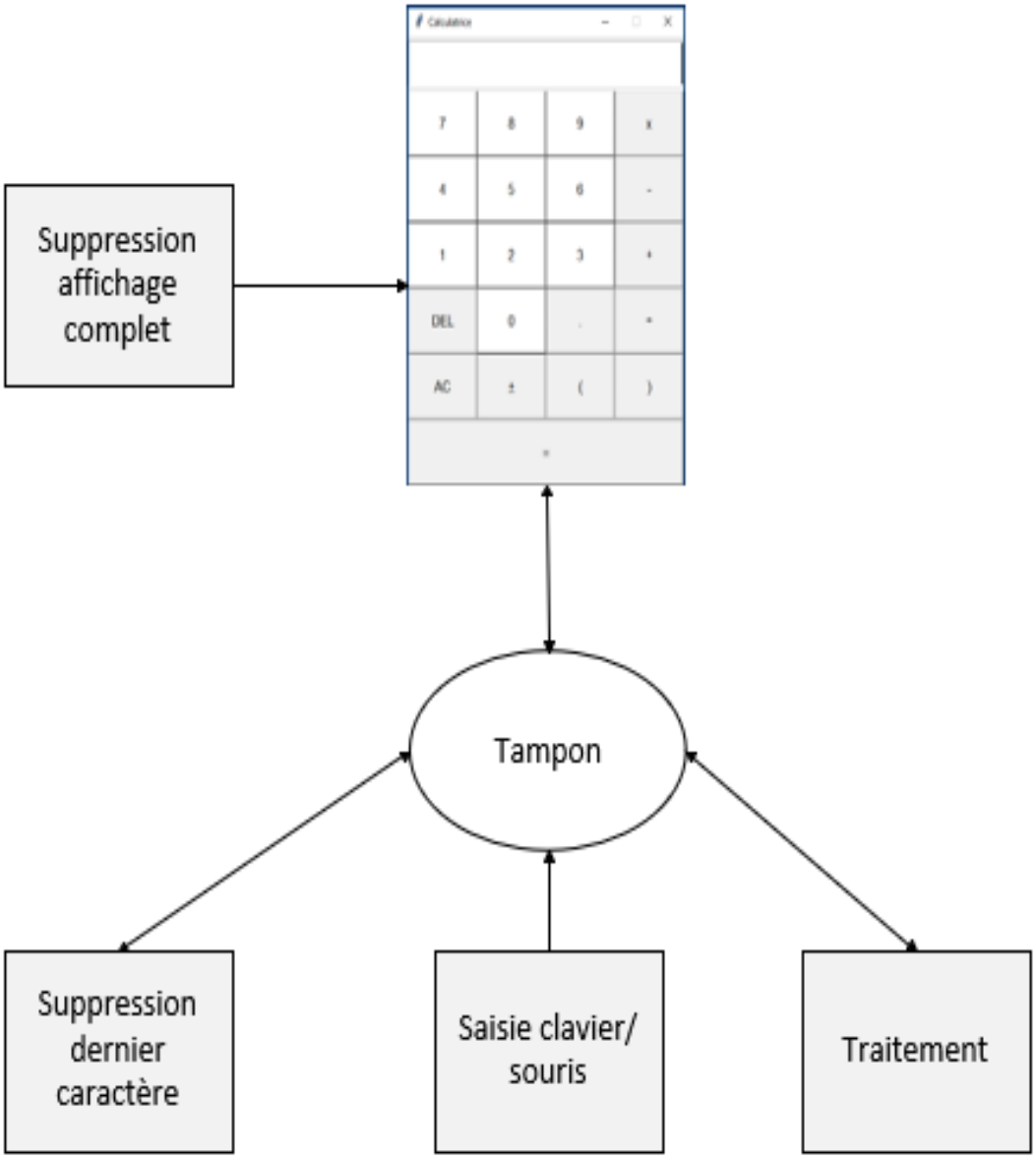

Figure 1 - Diagramme d'architecture du logiciel Calculatrice

À l'exception de la fonction de Suppression de l'affichage complet, les modules du logiciel interagissent avec l'interface utilisateur via une mémoire temporaire (tampon) servant à relayer les données entre lesdits modules.

2. Tests unitaires et tests d'interface utilisateur

Afin de garantir la fiabilité et la robustesse du logiciel, une série de tests unitaires est effectuée en rapport avec les exigences fonctionnelles établies dans le document de spécifications. Cette section présente l'ensemble de ces tests réalisés avec le *framework* « unittest » et figurant dans les fichiers Unittest_Calculatrice.py et GUITests.py. Les liens avec les exigences concernées sont spécifiés en caractères blancs. Afin de respecter les standards internationaux, les descriptifs figurant dans le code sont rédigés en anglais.

```
Addition :
Exigences CALC-FSR-002 et CALC-FSR-005

    '''Adding 3 and 6 together should be 9.00'''
    def test_TwoPosInt(self):
        x = calc.evaluateExpression(self,'3+6')
        self.assertEqual(x, 9.0)

    '''Adding -4 and 23.4 should be 19.40'''
    def test_NegIntPosfloat(self):
        x = calc.evaluateExpression(self,'-4 + 23.4')
        self.assertEqual(x, 19.4)

    '''Adding -23423 and 467.324 should be 23890.32'''
    def test_IntAndFloat(self):
        x = calc.evaluateExpression(self, '23423 + 467.324')
        self.assertEqual(x, 23890.32)

    '''Adding 467.324 and -23423 should be 23890.32'''
    def test_FloatAndInt(self):
        x = calc.evaluateExpression(self, '467.324 + 23423')
        self.assertEqual(x, 23890.32)

    '''Adding 3.5 and 7.4333 should be 10.93'''
    def test_FloatAndFloat(self):
        x = calc.evaluateExpression(self, ' 3.5 + 7.4333')
        self.assertEqual(x, 10.93)

    '''Adding -4 and 0 should be -4.00'''
    def test_NegtAndZero(self):
        x = calc.evaluateExpression(self, ' -4 + 0')
        self.assertEqual(x, -4.0)

Addition:
Exigences CALC-FSR-002, CALC-FSR-005 et CALC-FSR-008

    '''Adding  99999980000001 and 19999998 should be
99999999999999.0'''
    def test_TwoBigInts(self):
        x = calc.evaluateExpression(self, ' 99999980000001 + 19999998')
```

```
        self.assertEqual(x, 99999999999999.0)

    '''Adding  99999980000001.001 and 19999998.999 should be 00
1000000000000.0'''
    def test_TwoBigFloats(self):
        x = calc.evaluateExpression(self, ' 99999980000001.001 +
19999998.999')
        self.assertEqual(x, 100000000000000.0)

Soustraction:
Exigences CALC-FSR-002 et CALC-FSR-005


    '''Subtracting 4 from 2 should be -2'''
    def test_PosIntAndPosInt(self):
        x = calc.evaluateExpression(self,'2 - 4')
        self.assertEqual(x, -2)

    '''Subtracting 0 from -6 should be -6'''
    def test_NegIntAndZero(self):
        x = calc.evaluateExpression(self, '-6 - 0')
        self.assertEqual(x, -6)

    '''Subtracting 0 from 42 should be 42'''
    def test_PosIntAndZero(self):
        x = calc.evaluateExpression(self, '42 - 0')
        self.assertEqual(x, 42)

    '''Subtracting 2.25 from -2 should be -4.25'''
    def test_NegIntAndPosFloat(self):
        x = calc.evaluateExpression(self, '-2 - 2.25')
        self.assertEqual(x, -4.25)

    '''Subtracting 9 from 14.56 should be 5.56'''
    def test_PosFloatAndPosInt(self):
        x = calc.evaluateExpression(self, '14.56 - 9')
        self.assertEqual(x, 5.56)

    '''Subtracting 1.35 from 9 should be 7.65'''
    def test_PosIntAndPosFloat(self):
        x = calc.evaluateExpression(self, '9 - 1.35')
        self.assertEqual(x, 7.65)

    '''Subtracting 1.35 from 0.29 should be -1.06'''
    def test_PosIntAndPosFloat(self):
        x = calc.evaluateExpression(self, '0.29 - 1.35')
        self.assertEqual(x, -1.06)

    '''Adding a substraction of 10.99 to 1000 should be 989.01'''
    def test_AddSignWithMinusSign(self):
        x = calc.evaluateExpression(self, '1000 + - 10.99')
        self.assertEqual(x, 989.01)
```

```
    '''Subtracting 23.555555555 from 1.00001293423 should be -
22.55554262077'''
    def test_TwoLargeDecimals(self):
        x = calc.evaluateExpression(self, '1.00001293423 -
23.555555555')
        self.assertEqual(x, -22.56)

    '''Subtracting 134514353425 from 64353673457867 should be
64219159104442'''
    def test_TwoLargeInt(self):
        x = calc.evaluateExpression(self, '64353673457867 -
134514353425')
        self.assertEqual(x, 64219159104442)


Multiplication:
Exigences CALC-FSR-002 et CALC-FSR-005

    '''Multiplying 3 and 6 together should be 18.00'''
    def test_TwoPosInt(self):
        x = calc.evaluateExpression(self,'3*6')
        self.assertEqual(x, 18.00)

    '''Multiplying -60 and 13.32 together should be -799.20'''
    def test_NegIntPosfloat(self):
        x = calc.evaluateExpression(self,'-60 * 13.32')
        self.assertEqual(x, -799.20)

    '''Multiplying 182 and -75.987 together should be 18'''
    def test_PosIntAndNegFloat(self):
        x = calc.evaluateExpression(self, '182 * -75.987')
        self.assertEqual(x, -13829.63)

    '''Multiplying 467.324 and 0 should be 0.00'''
    def test_FloatAndZero(self):
        x = calc.evaluateExpression(self, '467.324 * 0')
        self.assertEqual(x, 0.00)

    '''Multiplying 0 and 999999 should be 0.00'''
    def test_ZeroAndInt(self):
        x = calc.evaluateExpression(self, ' 0 * 999999')
        self.assertEqual(x, 0.00)


Multiplication:
Exigences CALC-FSR-002, CALC-FSR-005 et CALC-FSR-008

    '''Multiplying 99999999 and 99999999 should be 99999980000001
        Reaches display limits'''
    def test_TwoBigInts(self):
        x = calc.evaluateExpression(self, ' 9999999 * 9999999')
        self.assertEqual(x, 99999980000001.0)
```

```
    '''Multiplying 9999999.999 and 9999999.002 should be
99999990010000.00'''
    def test_TwoBigFloats(self):
        x = calc.evaluateExpression(self, ' 9999999.999 * 9999999.002')
        self.assertEqual(x, 99999990010000.0)


Division:
Exigences CALC-FSR-002 et CALC-FSR-005
    '''Divizing 18 by 6 should be 3.0'''
    def test_TwoPosInt(self):
        x = calc.evaluateExpression(self,'18 / 6')
        self.assertEqual(x, 3.0)

    '''Divizing -60 by 13.32 should be -4.51'''
    def test_NegIntPosfloat(self):
        x = calc.evaluateExpression(self,'-60 / 13.32')
        self.assertEqual(x, -4.50)

    '''Divizing 182 by -75.987 should be -2.40'''
    def test_PosIntAndNegFloat(self):
        x = calc.evaluateExpression(self, '182 / -75.987')
        self.assertEqual(x, -2.40)

Division:
Exigences CALC-FSR-002, CALC-FSR-005 et CALC-FSR-007

    '''Divizing 467.324 and 0 should generate exception'''
    def test_FloatAndZero(self):
        try:
            x = calc.evaluateExpression(self, '467.324 / 0')
        except ZeroDivisionError:
          print("")


Priorités et chaînes d'opérations:
Exigences CALC-FSR-002, CALC-FSR-003, CALC-FSR-004 et CALC-FSR-005

    '''Should consider operation priority and result to -586.74'''
    def test_AllFourOperatorsWithFloatAndInt(self):
        x = calc.evaluateExpression(self,'104 + 67 - 64.58 / 3 * 35.2')
        self.assertEqual(x, -586.74)

    '''Should handle parenthesis and result to 1325.79'''
    def test_Parenthesis(self):
        x = calc.evaluateExpression(self, '(104 + (67 - (64.58 / 2)) *
35.2)')
        self.assertEqual(x, 1325.79)
```

```
Gestion des entrées invalides:
Exigence CALC-FSR-007

    '''Should not accept chars inside the operation'''
    def test_Chars(self):
        try:
            x = calc.evaluateExpression(self,'Chars should not be
accepted 3 + 4')
        except:
            self.assertTrue(True)
        else:
            self.assertTrue(False)

    '''Should not accept a division by zero'''
    def test_DivisonByZero(self):
        try:
            x = calc.evaluateExpression(self, '3 / 0')
        except:
            self.assertTrue(True)
        else:
            self.assertTrue(False)

    '''Should not accept many zeros before a number'''
    def test_ManyZeros(self):
        try:
            x = calc.evaluateExpression(self, '000005 + 2')
        except:
            self.assertTrue(True)
        else:
            self.assertTrue(False)

    '''Should accept a comma followed by a number'''
    def test_NoZeroAndComma(self):
            x = calc.evaluateExpression(self, '.45 - .32')
            self.assertEqual(x, 0.13)
```

Interface graphique :

```
Exigence CALC-FSR-001

# List of possible number buttons
    numberButtonArray = [B8, B9, B10, B11, B12, B13, B14, B15, B16,
B17]
    # List of mathematical operator buttons
    operatorButtonArray = [B4, B5, B6, B7]


    for i in range(1,32):
```

```
        cpt = 0
        generatedByButtons = ""
        loopCondition = True

        while(loopCondition or cpt < 2):

            if(cpt > 0):
                # Add an operator
                rand = self.randomOperatorGen()
                operatorButtonArray[rand].invoke()
                generatedByButtons = generatedByButtons +
operatorButtonArray[rand]['text']
            # Add atleast one number at first
            rand = self.randomNumberGen()
            numberButtonArray[rand].invoke()
            generatedByButtons = generatedByButtons +
numberButtonArray[rand]['text']
            # While generator returns True, add numbers to the integer
part of the expression
            while (self.randomBoolGen()):
                rand = self.randomNumberGen()
                numberButtonArray[rand].invoke()
                generatedByButtons = generatedByButtons +
numberButtonArray[rand]['text']
            # Randomly decide if there will be a decimal value
            if(self.randomBoolGen()):
                B18.invoke()
                generatedByButtons = generatedByButtons + B18['text']
                rand = self.randomNumberGen()
                numberButtonArray[rand].invoke()
                generatedByButtons = generatedByButtons +
numberButtonArray[rand]['text']
                # While generator returns True, add numbers to the
decimal part of the expression
                while (self.randomBoolGen()):
                    rand=self.randomNumberGen()
                    numberButtonArray[rand].invoke()
                    generatedByButtons = generatedByButtons +
numberButtonArray[rand]['text']
            cpt += 1
            # Start deciding if we should keep going once we have
atleast 2 numbers and 1 operator inbetween
            if(cpt>2):
                loopCondition = self.randomBoolGen()

        displayedOp = self.e.get()
        # Compare what is displayed on the GUI and what the events made
internally

        print('Activated buttons : ' + generatedByButtons)
        print('Display is :        ' + displayedOp)
        if (generatedByButtons == displayedOp):
```

```
            print('----------> OK : Display matches buttons sequence <-
---------')
        else:
            print('********** ERROR : Display does not match buttons
sequence! **********')
        try:
            if (eval(displayedOp) == eval(generatedByButtons)):
                print('----------> OK : Mathematic expressions provided
same results <----------.')
            else:
                print('********** Warning : Mathematic expressions
provided different results **********')
        except:
            print('----------> OK : Invalid input ash been caught <----
------.')

        self.clearall()
    #print('Trying button AC')
    B2.invoke()
    displayedOp = self.e.get()
    if (displayedOp == ""):
        print('----------> OK : Display has been cleared <----------')
    else:
        print('********** ERROR : Display has not been cleared!
**********')
    B19.invoke()
    B20.invoke()
    B1.invoke()
    displayedOp = self.e.get()
    if (displayedOp == "()-"):
        print('----------> OK : Display matches buttons sequence <-----
-----')
    else:
        print('********** ERROR : Display does not match buttons
sequence! **********')
    B3.invoke()
    B3.invoke()
    displayedOp = self.e.get()
    if (displayedOp == "("):
        print('----------> OK : Display matches buttons sequence <-----
-----')

    else:
        print('********** ERROR : Display does not match buttons
sequence! **********')
    self.clearall()

    #Testing "=" with mock
    self.egale = MagicMock(return_value = '----------> OK : The "="
button has been verified with mock <----------')
    print(B0.invoke())

    print('********** RUNNED ' + str(i+4)  +' GUI TESTS **********')
```

```
def randomNumberGen(self):
    rand = random.randint(0, 9)
    return rand
def randomOperatorGen(self):
    rand = random.randint(0, 3)
    return rand
def randomBoolGen(self):
    rand = random.randint(0,1)
    if(rand == 0):
        return False
    else:
        return True
```

3. Procédure de lancement des tests unitaires

La procédure de lancements des tests unitaires est disponible dans le fichier Readme.txt accompagnant l'application. La procédure comprend 35 tests d'interface utilisateur ainsi que 34 tests vérifiant les fonctions internes du logiciel. Si des problèmes devaient survenir lors de l'exécution des tests, veuillez contacter le support technique.