My style of software development

—Explain with a real project by Pengfei Gao

Preface

- In this presentation, I will explain my style of software development with my recent project bi3.0.

- To be honest, I didn't take any time to learn any style of software development before, until I was told to do this presentation regarding Test Driven Development. And after some research from Intent, I found my style is extremely similar to Test Driven Development (almost identical)

- Before I start, I want to emphasis, there are a lot of things we need to consider before jumping into the development, e.g. analysis the requirement from both functional and technical perspective, design, do some simple simulation, even mathematic proof especially you figure out some greedy algorithm to apply (e.g. Dijkstra shortest path algorithm only works for positive weighted edge, if you apply this on negative weighted edge, you'll get trouble.) … The content of this presentation assume all these prerequisite have been done already.

Add a test

- To write a test, developer must clearly understand the feature’s specification and requirements.

- The code on the right side is the test case written by me when I develop the tokenizer for the compiler.

```
const assert = require('assert');
const Tokenizer = require('../../src/compiler/expTokenizer');

describe('expTokenizer', function () {
    describe('', function () {

        it('test1', async () => {
            const expStr = '(10*$$.colA+sum(余额表.{(账期>=issue(
, 10, $$.colD, sum(1,2)))/-(sum($10.colC: $20.colC))'
            const tokens = Tokenizer.tokenize(expStr)
            const expected =
[
  [ '(', 't_symbols' ],
  [ '10', 't_integer_constant' ],
  [ '*', 't_ops' ],
  [ '$$.colA', 't_reference' ],
  [ '+', 't_ops' ],
  [ 'sum', 't_str_constant' ],
  [ '(', 't_symbols' ],
  [ '余额表', 't_str_constant' ],
  [ '.', 't_symbols' ],
  [ '{', 't_symbols' ],
  [ '(', 't_symbols' ],
  [ '账期', 't_str_constant' ],
  [ '>=', 't_compare_ops' ],
  [ 'issue', 't_str_constant' ],
  [ '(', 't_symbols' ],
  [ '2019-01', 't_str_constant' ],
  [ ',', 't_symbols' ],
  [ 'sum', 't_str_constant' ],
  [ '(', 't_symbols' ],
  [ '1', 't_integer_constant' ],
  [ ',', 't_symbols' ],
  [ '3', 't_integer_constant' ],
  [ ')', 't_symbols' ],
  [ ')', 't_symbols' ],
  [ 'and', 't_boolean_ops' ],
  [ '账期', 't_str_constant' ],
  [ '<=', 't_compare_ops' ],
  [ '选择账期', 't_str_constant' ],
  [ ')', 't_symbols' ],
  [ 'and', 't_boolean_ops' ],
  [ '(', 't_symbols' ],
  [ '(', 't_symbols' ],
  [ '科目编码', 't_str_constant' ],
  [ '=', 't_compare_ops' ],
  [ '$$.colB', 't_reference' ],
  [ 'or', 't_boolean_ops' ],
  [ '科目编码', 't_str_constant' ],
  [ '=', 't_compare_ops' ],
```

Run all tests and see if new tests fails

- I do this every time before write any code for two reasons
- Firstly, I need to verify the test environment and all other tests are working correctly
- Secondly, the new tests should be fail for the expected reason
- The code on the right side is a test case to test reserved variable type (which starts with %) for a token, and the reserved variable is the new type to be developed

```
});

it('test4', async () => {
    const expStr = 'tableA.{issue=%rv1}.colA'
    const tokens = Tokenizer.tokenize(expStr)
    const expected =
        [ [ 'tableA', 't_str_constant' ],
        [ '.', 't_symbols' ],
        [ '{', 't_symbols' ],
        [ 'issue', 't_str_constant' ],
        [ '=', 't_compare_ops' ],
        [ 'rv1', 't_reserved_variable' ],
        [ '}', 't_symbols' ],
        [ '.', 't_symbols' ],
        [ 'colA', 't_str_constant' ] ]
    assert.deepEqual(tokens, expected)
});
```

Write the code

- The only purpose of the written code at this point is to pass the test, developer should avoid write code which has not been covered by test cases

```
    }

    execCell(row,col){
        const rowsOfValue = this.rowsOfValue
        const rowsOfExpCell = this.rowsOfExpCell
        if (col in rowsOfValue[row]) return rowsOfValue[row][col]
        const adj = this.GR.adj
        //const reverseOrder = ['$'+(row+1)+'.'+col]
        const reverseOrder = [Util.rowColToRef(row+1,col)]
        let i = 0
        while ( i < reverseOrder.length) {
            const v = reverseOrder[i]
            for (const w of adj[v]){
                reverseOrder.push(w)
            }
            i++
        }
        //console.info(reverseOrder)
        while (reverseOrder.length > 0){
            const v = reverseOrder.pop()
            let [row,col] = Util.refToRowCol(v)
            row -= 1
            if (col in rowsOfValue[row]) continue
            else {
                const cell = rowsOfExpCell[row][col]
                const expExecutor = new ExpExecutor(cell.engine,rows
                const res = expExecutor.exec()
            }
        }
        return rowsOfValue[row][col]
    }

```

Run tests

- If all test case pass, developer can be confident that the code meets the test requirements, and do not have any regression

```
        it('test3', async () => {
            const rowsNameVersion =
                    [
                        {
                            colA: '$2.colC',
                            colB: '123',
                            colC: '321',
                            _id: '1',
                        },
                        {
                            colA: '111',
                            colB: '222',
                            colC: '$$.colA+$$.colB',
                            _id: '2',
                        },
                        {
                            colA: '$4.colC',
                            colB: '123',
                            colC: '321',
                            _id: '3',
                        },
                        {
                            colA: '333',
                            colB: '444',
                            colC: '$$.colA+$$.colB',
                            _id: '4',
                        },
                    ];
            const tableName = 'table1'
            const table = new Table(rowsNameVersion, tableName);
            table.execCell(0,'colA')
            const actual = table.rowsOfValue
            const expected = [ { colA: 333 }, { colB: 222, colA: 111, colC: 333 }, {}, {} ]
            assert.deepEqual(expected, actual)

            table.execCell(2,'colA')
            const actual2 = table.rowsOfValue
            const expected2 =
                [ { colA: 333 },
                { colB: 222, colA: 111, colC: 333 },
                { colA: 777 },
                { colB: 444, colA: 333, colC: 777 } ]
            assert.deepEqual(expected2, actual2)
        });
```

Refactor code

- The 3 things I consider most when refactor code
- Code performance (Can be improved by logical, data structure, algorithm …)
- Readability (Naming convention, design pattern …)
- Duplicate code

```
/**
Created by Pengfei Gao on 2019-12-04
*/

const {
        KEYWORDS_ALL,
        TYPE_OPS,
        TYPE_COMPARE_OPS,
        TYPE_SYMBOLS,
        TYPE_BOOLEAN_OPS,
        TYPE_UNARY_BOOLEAN_OPS,
        TYPE_KEYWORD_CONSTANT,
        TYPE_STR_CONSTANT,
        TYPE_INTEGER_CONSTANT,
        TYPE_FLOAT_CONSTANT,
        TYPE_REFERENCE,
        TYPE_UNARY_OPS,
        TYPE_RESERVED_VARIABLE,
        CATEGORY_ROOT,
        CATEGORY_EXPRESSION,
        CATEGORY_TERM,
        CATEGORY_FUNCTIONCALL,
        CATEGORY_PARAMLIST,
        CATEGORY_PARAMETER,
        CATEGORY_SELECTORCOMBO,
        CATEGORY_TABLENAME,
        CATEGORY_COLUMNNAME,
        CATEGORY_SELECTORS,
        CATEGORY_SELECTORSTERM,
        CATEGORY_SELECTOR,
        CATEGORY_FUNCTIONNAME,
        CATEGORY_REFRANGE
} = require("./lexical")


class TreeNode {

    constructor(parent,content,level,category,type='',note=''){
        this.parent = parent
        this.content = content
        this.note = note
        this.level = level
        this.category = category
        this.type = type
        this.children = []
        this.value = null
    }

    traversalLeafs(arr){
        if (this.children.length == 0) arr.push(this)
```

Repeat

- Starting with another new test, the cycle is then repeated to push forward the functionality

- The test code on the right shows that, to develop a executor of a compiler, first, I develop a executor which can execute arithmetic expression e.g. 3*(2+1), then I develop a executor which can execute arithmetic and function expression e.g. sum(3,2+1,2), last, I develop a executor which can execute arithmetic, function, and selectorCombo (which is very similar to a SQL statement) e.g. tableA.{(colA = str(aa,2)) and (colB = 1+2) and (colC > 3)}.colC

```
 });
it('test7', async () => {
     const expStr = 'sum(1+sum(3+2*3,sum(3+4,2.1)),1)'
     const tokens = Tokenizer.tokenize(expStr)
     const engine = new Engine(tokens)
     const executor = new Executor(engine)
     const actual = executor.exec()
     assert.equal(actual,20.1)
 });
it('test8', async () => {
     const tableA = [
         {
             colA:'aa1',
             colB:'bb1',
             colC:'1',
         },
         {
             colA:'aa2',
             colB:'bb2',
             colC:'2',
         },
         {
             colA:'aa2',
             colB:'3',
             colC:'3',
         },
         {
             colA:'aa2',
             colB:'3',
             colC:'4',
         },
     ]
     const tableB = [{colA:'aa1'}]
     const globalCache = {
         tables: {
             tableA,
             tableB,
         }
     }
     const expStr = 'tableA.{(colA = str(aa,2)) and (colB = 1+2) and (colC > 3)}.colC'
     const tokens = Tokenizer.tokenize(expStr)
     const engine = new Engine(tokens)
     const executor = new Executor(engine,[],-1,null,globalCache)
     const actual = executor.exec()
     assert.deepEqual(actual,['4'])
 });
```