

Qualitätssicherungskonzept
Nachrichtenkommunikation für das THW

na17b

Inhaltsverzeichnis

1	Dokumentationskonzept	1
1.1	Interne Dokumentation	1
1.2	Quelltextnahe strukturierte Dokumentation	1
1.3	Entwurfsbeschreibung	1
2	Organisatorische Festlegungen/Coding Standard	1
2.1	Code Layout	1
2.2	Leerzeichen	2
2.3	Kommentare	3
2.4	Namenskonventionen	3
3	Testkonzept	3
3.1	Komponententests	3
3.2	GUI Tests	6
4	Continuous Integration	7

1 Dokumentationskonzept

1.1 Interne Dokumentation

Die interne Dokumentation bezieht sich ausschließlich auf möglichst kurze funktionsinterne Erläuterungen. Diese Kommentare sind über der zu erläuternden Zeile auf Englisch zu verfassen.

1.2 Quelltextnahe strukturierte Dokumentation

Die quelltextnahe strukturierte Dokumentation dient zur Erläuterung der implementierten Klassen oder Funktionen. Somit soll es bei der späteren Bearbeitung oder Erweiterung des Programmes durch projektunabhängige Programmierer möglich sein, problemlos und schnell zu sehen was der Zweck der Klassen beziehungsweise Funktionen ist. Zu der Dokumentation von Python wird pydoc genutzt. Für die Dokumentation von javascript, worauf das genutzte Framework vue.js basiert, wird jsdoc genutzt. Auch diese erfolgt in Englisch. Somit können etablierte Begriffe genutzt werden, ohne sich auf eine Mischung aus Deutsch und Englisch oder das Eindeutschen von Begriffen verlassen zu müssen. Dadurch sollen Missverständnisse und dadurch mögliche Fehler vermieden werden.

1.3 Entwurfsbeschreibung

Die Entwurfsbeschreibung ist ein Dokument zur Dokumentation von Modellierungs-, Struktur- und Designentscheidungen. Mit dieser externen Dokumentation sollen die genannten Entscheidungen begründet werden. Dadurch soll außerdem externen Programmierern erläutert werden, worauf bei dem Umgang der Software geachtet werden muss. Die Entwurfsbeschreibung ist eine Dokumentation, die im Verlauf des Praktikums entsteht, da sämtliche Entscheidungen erst noch eindeutig gemacht und dementsprechend dokumentiert werden müssen.

2 Organisatorische Festlegungen/Coding Standard

Ein einheitlicher Coding-Standard ist sinnvoll, da er eine gute Lesbarkeit des Codes garantiert. Das ist essentiell nicht nur für die Zusammenarbeit im Team, nur wenn man den Code der anderen Mitglieder lesen kann und versteht, kann auch sinnvolle Kommunikation über eventuelle Fehler stattfinden. Dies gilt auch für Dritte, die sich mit dem Projekt befassen und es unter Umständen auch fortführen wollen.

2.1 Code Layout

- Einrückungen
 - Generell werden für Einrückungen Tabs mit der Länge von vier Zeichen empfohlen. Dabei ist gerade bei Python zu beachten, dass Tabs und Leerzeichen nicht gemischt werden dürfen, bei Javascript ist es jedem freigestellt was er benutzen möchte.
 - Folgezeilen sollten klar erkennbar sein, wobei aber die “vier Leerzeichen einrücken“-Regel nicht gilt. z.B:

(1)

```
foo = long_function_name(var_one, var_two,
                          var_three, var_four)
```

(2)

```
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

- Zeilenlänge
 - Die maximale Zeilenlänge ist auf 80 Zeichen beschränkt.
- Zeilenumbrüche bei binären Operatoren
 - Zeilenumbrüche sollten immer vor binären Operatoren stehen, da so einfacher zu erkennen ist, welcher Operator zu welchem Objekt gehört. z.B.:

```
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```
- Leerzeilen
 - Top-Level Funktionen und Klassen-Definitionen sollten mit jeweils zwei Leerzeilen umfasst sein.
 - Einfache Methoden-Definitionen in Klassen sollten mit jeweils einer Leerzeile umgeben werden.
 - Zusätzliche Leerzeilen können, wenn auch sparsam, genutzt werden um die logische Struktur von z.B.: Gruppen von Funktionen oder Sektionen mit ähnlichem/zusammengehörigem Inhalt zu verdeutlichen.
- Kommentare
 - Kommentare sollten immer über dem Objekt stehen welches sie beschreiben.

2.2 Leerzeichen

- vermeiden
 - Sie sollten direkt nach öffnenden Klammern oder vor schließenden Klammern vermieden werden. z.B.:

```
spam(ham[1] , {eggs: 2})
spam( ham[ 1 ] , { eggs: 2 } )
```
 - Sie sollten nicht vor Kommas, Semikolons oder Doppelpunkten stehen (wobei das nicht für alle Doppelpunkte gilt, siehe “Slice-Expressions”).
 - Zwischen “nachfolgenden” Kommas und schließenden Klammern sollten sie vermieden werden.
 - Auch direkt vor öffnenden Klammern, welche Argumentlisten eröffnen, bzw. Indizes oder Slices beginnen, sollten sie vermieden werden.
- nutzen
 - Sie sollten genutzt werden um die folgenden binäre Operatoren zu umgeben: assignment (`=`), augmented assignment (`+=`, `-=`, etc.), comparisons (`==`, `<`, `>`, `!=`, `<>`, `<=`, `>=`, `in`, `not in`, `is`, `is not`), Booleans (`and`, `or`, `not`), Slicing (`:`).
 - Auch sollte der `->`-Operator bei Funktionsannotationen mit jeweils einem Leerzeichen umgeben sein.
 - Kommentare innerhalb von Methoden oder Funktionen sollen mindestens mit zwei Leerzeichen vom Ausdruck getrennt sein welchen sie beschreiben.
 - Nach einem Komma, Semikolon, Doppelpunkt sollte ein Leerzeichen folgen.

2.3 Kommentare

Kommentare sollten in Englisch gehalten werden und immer aus kompletten Sätzen bestehen. Dabei ist zu beachten, dass sie den Inhalt des Objektes, auf welches sie sich beziehen, so kurz wie möglich, aber trotzdem verständlich für alle beschreiben. Bei Blockkommentaren oder allgemein längeren Textabschnitten bietet es sich zu besseren Strukturierung an Paragraphen zu nutzen.

2.4 Namenskonventionen

Namen sollten den “mixedCase“-Standard nutzen, d.h. bestehen sie aus mehreren Wörtern, dann werden sie zusammen geschrieben, beginnen mit einem Buchstaben und jedes weitere Wort beginnt mit einem Großbuchstaben: z.B.: `nameOfFunction`. Nur Namen von Klassen beginnen mit einem Großbuchstaben, alle anderen klein. Werden Akronyme oder Abkürzungen benutzt, so werden diese immer komplett groß geschrieben: z.B.: `errorInTCPServer`.

3 Testkonzept

Durch die besondere Verantwortung des THW hat die Zuverlässigkeit der Software einen außerordentlich hohen Stellenwert. Um diese zu gewährleisten muss das umfangreiche Testen der Software automatisiert ablaufen, wobei eine Testcoverage von 100% angestrebt wird.

3.1 Komponententests

Python Als Testing Framework findet in diesem Projekt das etablierte und in der Python Standard Bibliothek integrierte Python Unit Testing Framework Verwendung. Komponententests sind nach folgenden Konventionen zu erstellen. Für jede Klasse existiert eine Testklasse und für jede Funktion eine Testfunktion. Bei der Namensgebung wird im Dateinamen vor der Dateiendung ein `.test` eingeschoben. So erhält die Datei *MittringEmulator.py* ihre Testdatei *MittringEmulator.test.py*. Testdateien befinden sich im gleichen Ordner wie zu testende Dateien. Beispiel:

```
1 class MittringEmulator:
2     def add(self, a, b):
3         return (a + b);
```

```
1 import unittest
2 from MittringEmulator import MittringEmulator
3
4 class TestMittringEmulator(unittest.TestCase):
5     #Instantiate Object to be tested
6     def setUp(self):
7         self.MittringEmulator = MittringEmulator()
8
9     #TestFunction
10    def testAdd(self):
11        #First TestCase, checking if add(3,4) results in 7
12        self.assertEqual(self.MittringEmulator.add(3,4),7)
13        #checking that add(-1,-1) does not result in 0
14        self.assertNotEqual(self.MittringEmulator.add(-1,-1),0)
15
16    #Calling unittest, which does actual Testing
17    if __name__ == '__main__':
18        unittest.main()
```

JavaScript Als JavaScript Testing Framework findet in diesem Projekt Jest Verwendung. Komponententests sind nach folgenden Konventionen zu erstellen: Für jede Datei existiert eine Testdatei und für jede Funktion mindestens eine Testfunktion. Bei der Namensgebung wird im Dateinamen vor der Dateiendung ein `.test` eingeschoben. So erhält die Datei `exampleModule.js` ihre Testdatei `exampleModule.test.js`. Testdateien befinden sich im gleichen Ordner wie zu testende Dateien.

```

1  // very basic module to be tested
2
3  // multiplication
4  function product (a, b) {
5    return a * b
6  }
7  // module.export.functionName = functionName allows the testmodule to use our function/class/etc.
8  module.exports.product = product
9
10 // returns String
11 function getGreatBarockTitle () {
12   return 'Canone per 3 Violini e Basso'
13 }
14 module.exports.getGreatBarockTitle = getGreatBarockTitle
15
16 // returns Array
17 function getShoppingList () {
18   return ['Broccoli', 'Cottage Cheese', 'Hummus', 'Broccoli']
19 }
20 module.exports.getShoppingList = getShoppingList
21
22 // simple cube class
23 class Cube {
24   constructor (length) {
25     this.length = length
26     this.color = null
27   }
28 }
29 module.exports.Cube = Cube
30
31 // returns instance of cube with color set to 'red' and custom length
32 function getRedCube (length) {
33   var c = new Cube(length)
34   c.color = 'red'
35   return c
36 }
37 module.exports.getRedCube = getRedCube

```

```

1  /*
2  import /Projekt/src/frontend/src/examples/module.js
3  @ already substitutes for /Projekt/src/frontend/src/ and fileendings are not necessary
4  */
5  import module from '@test-examples/module'
6
7  /*
8  testing a function by:
9  test('String describing the test', () => {
10    expect(module.function(args)).matcher(expectedValue)
11  })
12  matchers allow to test values in different ways

```

```
13  matchers used in this example are:
14  toBe - checks that a value is as expected
15  toBeGreaterThan - checks that a value exceeds a certain minimum
16  toMatch - checks that a String matches a regular expression
17  not.toMatch - checks that a String does not match a regular expression
18  toContain - checks that an Array contains a certain Object
19  arrayContaining - checks that an Array contains all Objects in a certain Array
20  toBeInstanceOf - checks that an Object is an Instance of a certain class
21  not.toBeNull - checks that a value is unequal to null
22
23  For a complete list of matchers visit: https://facebook.github.io/jest/docs/en/expect.html
24  */
25
26
27  //Testing if product(2,2) returns 4
28  test('product(2,2)=4', () => {
29    expect(module.product(2, 2)).toBe(4)
30  })
31
32  //Testing if the product of a positive integer and a negative integer is less than 1
33  test('product(negative,positive)<1', () => {
34    for (let a = 1; a < 10; a++) {
35      for (let b = -1; b > -10; b--) {
36        expect(module.product(a, b)).not.toBeGreaterThan(0)
37      }
38    }
39  })
40
41  //Testing if a String contains the 'Bass'
42  test('Bass in getGreatBarockTitle', () => {
43    expect(module.getGreatBarockTitle()).toMatch(/Bass/)
44  })
45
46  //Testing if a String does not contain 'autotune' (not case sensitive)
47  test('autotune not in getGreatBarockTitle (not case sensitive)', () => {
48    expect(module.getGreatBarockTitle().toLowerCase()).not.toMatch(/autotune/)
49  })
50
51  //Testing if an array contains 'broccoli'
52  test('getShoppingList contains broccoli', () => {
53    expect(module.getShoppingList()).toContain('Broccoli')
54  })
55
56  //Testing if an array contains 'Broccoli' and 'Hummus'
57  test('getShoppingList contains hummus twice', () => {
58    expect(module.getShoppingList()).toEqual(expect.arrayContaining(['Broccoli', 'Hummus']))
59  })
60
61  //Testing if the object returned by getRedCube is an instance of Cube
62  test('getRedCube returns instance of Cube', () => {
63    expect(module.getRedCube(10)).toBeInstanceOf(module.Cube)
64  })
65
66  //Testing if the property color, from the object returned by getRedCube is not null
67  test('getRedCube returns instance of Cube', () => {
68    expect(module.getRedCube(10).color).not.toBeNull()
69  })
```

3.2 GUI Tests

Durch die Nutzung von vue-js zur Frontenderstellung bietet sich auch der Einsatz der vue-test-utils an, welche das automatisierte Testen der GUI Funktionalitäten erlauben. Für jede Komponente wird eine Testdatei im selben Ordner erstellt. Bei der Namensgebung wird im Dateinamen vor der Dateieindung ein .spec eingeschoben. So erhält die Datei *counter.js* ihre Testdatei *counter.spec.js*.

```

1  // Implementation of Button, able to count how often it has been clicked
2  export default {
3    template: `
4      <div>
5        <span class="count">{{ count }}</span>
6        <button @click="increment">Increment</button>
7      </div>
8    `,
9
10   // return Number of times button has been clicked
11   data () {
12     return {
13       count: 0
14     }
15   },
16
17   // called whenever button is being clicked, increments count
18   methods: {
19     increment () {
20       this.count++
21     }
22   }
23 }
```

```

1  /*
2   * mounting allows the component to be tested in isolation while also rendering its child components
3   * if rendering childs is not necessary use 'import { shallow } from '@vue/test-utils' ' and 'const wrapper = shallowMount(Component)'
4   */
5  import { mount } from '@vue/test-utils'
6
7  /*
8   * import /Projekt/src/frontend/src/examples/counter.js
9   * @ already substitutes for /Projekt/src/frontend/src/ and fileendings are not necessary
10  */
11  import Counter from '@test-examples/counter.js'
12
13  /*
14   * describing test for Counter:
15   * testing a gui-object by:
16   * it('String describing the test', () => {
17   *   expect(wrapperproperty).matcher(value to match against)
18   * })
19   * for more info visit "https://vue-test-utils.vuejs.org/en/guides/getting-started.html"
20   */
21  describe('Counter', () => {
22
23    //mounted components are returned inside a wrapper
24    //wrapper allows for easy manipulation, querying, etc.
25    const wrapper = mount(Counter)
26  })
```



```
27  //first test case, checking if HTML output is as expected
28  it('renders the correct markup', () => {
29    expect(wrapper.html()).toContain('<span class="count">0</span>')
30  })
31
32  //checking if button exists
33  it('has a button', () => {
34    expect(wrapper.contains('button')).toBe(true)
35  })
36
37  //checking if buttonclick increments count
38  it('button click should increment the count', () => {
39    expect(wrapper.vm.count).toBe(0)
40    const button = wrapper.find('button')
41    button.trigger('click')
42    //wrapper.vm returns vue instance
43    expect(wrapper.vm.count).toBe(1)
44  })
45  })
```

Darüber hinaus werden auch manuelle Tests der Benutzeroberfläche durchgeführt, welche Unannehmlichkeiten in der Bedingbarkeit aufdecken sollen.

4 Continuous Integration

Um die zahlreichen Vorgaben aus Dokumentationskonzept und Coding Standard automatisiert zu überprüfen, das Testkonzept umzusetzen und all dies einfach und zentral zu verwalten, wird das Prinzip der Continuous Integration angewandt. Hierzu wird GitLab CI verwendet, welches diese Aufgaben auf einfache und effiziente Weise der gewohnten Versionsverwaltung voranstellt und so die Anzahl der mangelhaften Dokumente in der Versionsverwaltung minimiert.