# 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 Integrationstests . . . 4
3.3 GUI Tests . . . 4

**4 Continuous Integration** **5**

# 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 zuverfassen.

## 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.
  - Längere Textabschnitte, wie z.B: Kommentare oder Dokumentarstrings, sind auf 72 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, genutz 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 genutz 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

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 dem Funktions- bzw. Klassennamen ein test bzw Test vorangestellt. Beispiel:

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

```
1  import unittest
2  from Projekt.MittringEmulator import MittringEmulator
3
4  class TestMittringEmulator(unittest.TestCase):
5      def setUp(self):
6          self.MittringEmulator = MittringEmulator()
7
8      def testAdd(self):
9          self.assertEqual(self.MittringEmulator.add(3,4),7)
10         self.assertNotEqual(self.MittringEmulator.add(-1,-1),0)
11
12 if __name__ == '__main__':
13     unittest.main()
```

Die Ordnerstruktur ist wie folgt anzulegen:

```
main
├── Projekt
│   ├── __init__.py
│   └── Quellcode.py
└── TestsProjekt
    ├── __init__.py
    └── TestQuellcode.py
```

Wobei die beiden __init__.py Dateien leer sind und lediglich zur Identifikation der Ordner als Python Paket dienen.

### 3.2 Integrationstests

Integrationstests gewährleisten das korrekte Zusammenspiel voneinander abhängiger Softwarekomponenten und erfolgen nach dem Bestehen der Komponententests. Hierbei werden vor allem Schnittstellentests durchgeführt, welche überprüfen ob die Kommunikation der einzelnen Komponenten fehlerfrei verläuft.

### 3.3 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 erstellt, wobei die Namensgebung den Komponententests gleicht. Beispiel:

```
export default {
  template: `
    <div>
      <span class="count">{{ count }}</span>
      <button @click="increment">Increment</button>
    </div>
  `,

  data () {
    return {
      count: 0
    }
  },

  methods: {
    increment () {
      this.count++
    }
  }
}
```

```
import { mount } from '@vue/test-utils'
import Counter from './Projekt/Counter.js'

describe('Counter', () => {
  const wrapper = mount(Counter)

  it('renders the correct markup', () => {
    expect(wrapper.html()).toContain('<span class="count">0</span>')
  })

  it('has a button', () => {
```

```
12      expect(wrapper.contains('button')).toBe(true)
13    })
14
15    it('button click should increment the count', () => {
16      expect(wrapper.vm.count).toBe(0)
17      const button = wrapper.find('button')
18      button.trigger('click')
19      expect(wrapper.vm.count).toBe(1)
20    })
21
22  })
```

Die Ordnerstruktur ist wie folgt anzulegen:

```
main
├── Projekt
│   └── Quellcode.js
└── TestsProjekt
    └── TestQuellcode.js
```

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.