Clean Code

Im Buch ***Clean Code***[1] von Robert C. Martin wird sehr ausführlich an Beispielen erläutert, weshalb das Einhalten von grundlegenden Regeln beim Schreiben vom Programmtexten von entscheidender Bedeutung für einen langfristigen Erfolg bei der Softwareentwicklung ist.

Einrückungen kennzeichnen Strukturebenen

Eine der grundlegenden Regeln, das konsequente Einhalten von Einrückungen für das Kennzeichnen von Strukturebenen des Programms ist bei Python selbstverständlich. Da Python keine Klammern o.ä. kennt und allein die Einrückungen zur Definition der Struktur verwendet, funktioniert ein Python-Programm nicht, wenn man die Einrückungen nicht richtig macht. Darauf macht dann an vielen Stellen bereits der Editor aufmerksam.
Das sieht bei Java-Programmen anders aus und Schülerinnen und Schüler reagieren oft unwillig auf die Forderung, die Einrückungen einzuhalten. "Das funktioniert doch auch so." ist die typische Antwort, die man immer wieder erhält und sie kennzeichnet den Kern des Problems, um den es bei Clean Code geht:

"Es geht doch" reicht nicht

Es mag sein, dass schlechter Programmtext irgendwie funktioniert. Er ist jedoch für Andere nur mit Mühe oder überhaupt nicht verständlich. Selbst der Entwickler selbst wird nach einiger Zeit nicht mehr wissen, was er sich dabei gedacht hat. Änderungen lösen ungeahnte Folgen aus und nach mehreren Änderungsschritten sind chaotische Programme oft nicht mehr wartbar.
Es besser zu machen bedeutet also, die Grundlage für eine langfristig nutzbare Software zu schaffen.

Anforderungen

Es geht daher darum, auch bei der Softwareentwicklung im Unterricht grundlegende Anforderungen kennen zu lernen und ihre Beachtung bei den Schülerinnen und Schülern einzufordern. Dabei gilt es z.T. auch schlechte Angewohnheiten von kommerziellen Entwicklern[2] zu korrigieren, die ihre Ursache in traditionellen Angewohnheiten haben.

Aussagekräftige Namen

Die Zeiten, in denen Variablennamen nur kurz sein durften, sind längst vorbei. Java-Programmzeilen wie

```
for (int i=0; i<n; i++) a[i]=b[i]*c[i];
```

sollten nicht mehr auftauchen, da die verwendeten Namen von Variablen für den Leser keinen Bezug zum Problem haben. Bei der folgenden Python-Programmzeile weiß man dagegen sofort, was gemacht wird[3]:

```
for tag in tage:
    gesamtArbeitszeit+=tage.getArbeitsZeit()
```

Das Beispiel zeigt einen weiteren Fall von klarer Kennzeichnung. Get- und Set-Methoden sollten eindeutig als solche gekennzeichnet werden.
Allerdings taucht die Frage auf, ob man das in *"Denglisch"* macht, wie im o.a. Beispiel

1 Robert C. Martin: Clean Code ISBN 978-3-8266-5548-7 Deutsche Übersetzung
2 ggf. auch bei Kolleginnen und Kollegen
3 Na ja, vielleicht bis auf das kryptische +=

oder nicht besser in Deutsch mit **`holeArbeitsZeit()`** oder **`gibArbeitsZeit()`**.
Man sollte innerhalb eines Programms aber immer bei derselben Schreibweise bleiben.

Anforderungen stichwortartig

Zu weiteren Anforderungen verweise ich auf das Buch von Martin. Stichwortartig seien hier aber genannt:

- keine Namen verwenden, die in anderen Kontexten festgelegt sind
- keine zu ähnlichen Namen verwenden, es sei denn …
- bei zusammenhängenden Begriffen ein gemeinsamer Namensteil
- bei Klassen Substantive und bei Methoden Verben als Grundlage verwenden
- Verwechslungen vermeiden; der große Buchstabe O kann bei vielen Zeichensätzen mit der Zahl 0 verwechselt werden und das kleine l mit der Zahl 1.[1]
  `O ≠ 0 und l ≠ 1`

Funktionen, Prozeduren, Methoden

Martin verwendet in seinem Buch nur den Begriff Funktionen. Da uns die Unterschiede funktionaler Programmierung und deklarativer Programmierung wichtig sind, gehen wir vielleicht auch auf den Begriff der Prozeduren ein und in einer der wichtigsten funktionalen Sprache, Scheme [Racket], heißen ausgerechnet die Funktionen procedure.

Allen gemeinsam ist, dass sie selbstständige Programmabschnitte sind, die einen Namen haben, Parameter übergeben bekommen und einen eigenen Namensraum definieren.

Üblicherweise sprechen wir von Funktionen, wenn sie einen Wert zurückgeben und von Prozeduren, wenn sie Arbeitsschritte beschreiben, die in diesem Sinne Nebeneffekte darstellen.

Bei der OO spricht man allgemein von Methoden und da bei Java immer der Typ des Rückgabewertes deklariert werden muss, erkennen wir Methoden, die nicht funktional sind, am Rückgabewert **`void`**. Python erkennt Funktionen und funktionale Methoden am **`return`**.

Kohäsion

Die wichtigste Forderung im Zusammenhang mit Clean Code an Funktionen finden wir auch hervorgehoben im Buch von Barnes und Kölling[2].

> Kohäsion: Jede Klasse oder Funktion ist für genau eine Aufgabe zuständig.

Das hat automatisch zur Folge, dass Funktionen immer nur sehr kurz sind, im Prinzip so klein wie möglich.
Ein gezielt ins Projekt eingebautes Musterbeispiel für den Verstoß gegen diese Prinzip ist die Methode **`GibFigur()`** im Anfangsprojekt zum Raumplaner:

1 Wie schwierig das manchmal sein kann, zeigt das Buch selbst. Der Übersetzer hat an einigen Stellen die Zahl 0 mit dem Begriff **`null`** übersetzt, der in den betrachteten Programmiersprachen mit dem nicht existierenden Objekt verknüpft ist.

2 Barnes und Kölling:Java lernen mit BlueJ ISBN 978-3-8273-7152-2

```
    def GibFigur(self):
        """definiert und transformiert die zu zeichnende Figur"""
        gc = Zeichenflaeche.GibZeichenflaeche().GibGC()
        path = gc.CreatePath()

        path.AddRectangle(0, 0, self.b, self.t)

        gc.PushState()
        gc.Translate(self.x+self.b/2, self.y+self.t/2)
        gc.Rotate(radians(self.w))
        gc.Translate(-self.b/2, -self.t/2)
        transformation = gc.GetTransform()
        gc.PopState()
        path.Transform(transformation)
        return path
```

Diese (funktionale) Methode definiert zunächst für das Objekt **`tisch`** die Grundfigur als path und transformiert ihn dann.
Schon in dieser Formulierung erkennt man die beiden Aufgaben. Daher sollte man -auch aus anderen Gründen- die Transformation in eine eigene Funktion ausgliedern.
Dazu ein weiteres Beispiel aus dem Buch zu Python von Peter Kaiser und Johannes Ernesti, das sie im Abschnitt zur Einführung der OO Programmierung einsetzen. Die Funktion

```
def geldtransfer(quelle, ziel, betr[1]):
    # Hier erfolgt der Test, ob der Transfer möglich ist
    if(quelle["Kontostand"] < betr or
       quelle["UmsatzHeute"] + betr > quelle["MaxTagesumsatz"] or
       ziel["UmsatzHeute"] + betr > ziel["MaxTagesumsatz"]):

        return False # Transfer unmöglich
    else:
        # Alles OK - Auf geht's

        quelle["Kontostand"] -= betr
        quelle["UmsatzHeute"] += betr
        ziel["Kontostand"] += betr
        ziel["UmsatzHeute"] += betr
        return True
```

bei der noch nicht OO gearbeitet wird, greift auf Konten vom Typ Dictionary zu. Das soll uns hier aber nicht interessieren, statt dessen beschäftigen wir uns zunächst mit den verknüpften Bedingungen im **`if`**. Hier werden mehrere Prüfungen durchgeführt, die nicht als solche eindeutig zu erkennen sind. Klarer lesbarer Code entsteht, wenn man die Aufgaben ausgliedert. Der entsprechende Abschnitt im Programm kann dann so aussehen:

```
    if( kontoStandAusreichend(quelle, betrag) or
       tagesUmsatzZulaessig(quelle, betrag) or
       tagesUmsatzZulaessig(ziel, betrag) ):
```

Abgesehen davon, dass dieser Aufruf bei OO Programmierung[2] verständlicher aussieht, wird nun im Programmtext schon deutlich, was geprüft wird.
Die entsprechenden ausgegliederten Funktionen sind sehr einfach mit einer einzigen Strukturebene:

1 Bitte nicht **`betr`**, sondern **`betrag`** ausschreiben. Groß oder Kleinschreibung ist eine grundsätzliches Problem, das ich hier nicht diskutiere.
2 OO-Version: quelle.kontoStandAusreichend(betrag)

```
def kontoStandAusreichend(konto, betrag):
    return (konto["Kontostand"] < betrag)

def tagesUmsatzZulaessig(konto, betrag):
    return (konto["UmsatzHeute"] + betrag > konto["MaxTagesumsatz"])
```

Bei der Definition der beiden Funktionen braucht man natürlich nicht zu wissen, ob sie für ein Quellkonto oder ein Zielkonto geschrieben werden, was ein zusätzlicher Vorteil im Sinne einer Verallgemeinerung ist.

Der Name der Funktion soll die Aufgabe eindeutig beschreiben

Welche Aufgabe hat die Funktion eigentlich? Soll sie den Geldtransfer durchführen oder prüfen, ob er zulässig ist?
Das dies nicht offensichtlich klar ist, erkennt man auch daran, dass der Programmentwickler es für nötig befunden hat, jeweils einen Kommentar ins Programm einzufügen.
Wir haben also wieder zwei Aufgaben in einer Funktion. Das Programm sollte daher auch zwei getrennte Funktionen zum Prüfen und zum Ausführen des Geldtransfers haben.
Bei der Prüffunktion würde man sich wünschen, dass Java und Python das Fragezeichen in Funktionsnamen zulassen. Um eine Prüfung zu kennzeichnen, kann es daher sinnvoll sein, alle Namen von Prüffunktionen mit **`ist`** beginnen zu lassen, also **`istKontoStandAusreichend(...)`**.
Das komplette Programm kann dann so aussehen:

```
def versucheGeldtransfer(quelle, ziel, betrag):
    if istGeldtransferZulaessig(quelle, ziel, betrag):
        geldtransferAusfuehren(quelle, ziel, betrag)
    else:
        print "Transfer ist unzulässig."

def istGeldtransferZulaessig(quelle, ziel, betrag):
    return (istKontoStandAusreichend(quelle, betrag) and
       istTagesUmsatzZulaessig(quelle, betrag) and
       istTagesUmsatzZulaessig(ziel, betrag))

def geldtransferAusfuehren(quelle, ziel, betrag):
    quelle["Kontostand"] -= betrag
    quelle["UmsatzHeute"] += betrag
    ziel["Kontostand"] += betrag
    ziel["UmsatzHeute"] += betrag

def istKontoStandAusreichend(konto, betrag):
    return (konto["Kontostand"] > betrag)

def istTagesUmsatzZulaessig(konto, betrag):
    return (konto["UmsatzHeute"] + betrag < konto["MaxTagesumsatz"])
```

Zu viele Strukturebenen

Wir kommen hier auf die Strukturebenen zurück, da viele Strukturebenen in einer Funktion darauf hindeuten, dass man Aufgaben unterer Strukturebenen in eigene kleine Funktionen ausgliedern könnte. Das folgende Beispiel zeigt die Möglichkeiten, dabei allerdings auch einige kleine Nachteile der Aufgliederung in die Teilfunktionen.

```
haus=[['Keller', ['Heizung'], ['Müll']],
      ['Erdgeschoss', ['Küche', 'Anne'],
      ['Wohnzimmmer', 'Ria' ,'Katze'],
      ['Flur', 'Jan']],
      ['1. Etage', ['Schlafzimmer'], ['Bad']]]

def durchsucheHaus(haus, suchObjekt):
    for etage in haus:
        for raum in etage[1:]:
            for objekt in raum[1:]:
                if objekt==suchObjekt:
                    print objekt, 'ist im', etage[0], 'im', raum[0]
```

Die zweite Variante mit dem Abbau der Strukturebenen:

```
def durchsucheHaus(haus, suchObjekt):
    for etage in haus:
        durchsucheEtage(etage, suchObjekt)

def durchsucheEtage(etage, suchObjekt):
    for raum in etage[1:]:
        durchsucheRaum(raum, suchObjekt)

def durchsucheRaum(raum, suchObjekt):
    for objekt in raum[1:]:
        if objekt==suchObjekt:
            print objekt, 'ist im', raum[0]
```

Formatierung des Quellcodes

Bei der Gestaltung des Programmtextes sollte man dringend auf sorgfältige Formatierungen achten. Es verbessert sehr die Lesbarkeit, wenn darauf geachtet wird, zusammengehörende Blöcke auch optisch kenntlich zu machen. Dazu dienen Leerzeilen zwischen solchen Blöcken, die Einordnung von [in Java Variablendeklarationen und] Variablendefinitionen an den üblicherweise erwarteten Stellen, also im Kopf der Klasse und nicht irgendwo mittendrin und das Vermeiden von unnötigen Kommentaren.

Kommentare sind in der Regel unnötig

Robert C. Martin weist sehr eindringlich darauf hin, dass es nur an wenigen Stellen nötig sei, Kommentare einzufügen. Insbesondere rügt er das standardmäßige Einfügen von Kommentaren, wie es z.B. BlueJ in seinen Standarddateien pflegt.
Besser ist es seiner Meinung nach, Programmtexte so zu schreiben, dass sie selbsterklärend sind. Was man dafür tun kann, ist oben beschrieben.
Die Methode **`BewegeHorizontal()`** aus unserem Projekt ist dafür ein gutes Beispiel.

Kommentare sind sinnvoll

Ich weise bewusst an dieser Stelle darauf hin, dass andererseits an vielen Stellen Kommentare sinnvoll und wichtig sind. So kann man beispielsweise unter dem Klassennamen einen Dokumentationsstring einfügen, auf den durch <Objektname>.__doc__ oder durch <Klassenname>.__doc__ zugegriffen werden kann.

Beispiel:

```
class Stuhl(Moebel):
    """Klasse Stuhl
    ermoeglicht das Zeichnen und Bearbeiten eines
    Stuhl-Symbols für den Raumplaner"""
```

und der Zugriff

```
>>> print stuhl.__doc__
Klasse Stuhl
    ermoeglicht das Zeichnen und Bearbeiten eines
    Stuhl-Symbols für den Raumplaner
```

Der Text dieses Kommentares könnte ggf. noch verbessert werden, indem man die Bearbeitungsmöglichkeiten nennt. Im verwendeten Projektbeispiel lassen sich nämlich mit passenden Methoden nur die Position und Orientierung des Symbols verändern, nicht aber seine Farbe und Abmessungen.

Verbindung von aufrufender und aufgerufener Funktion

Das Beispielprogramm zum Geldtransfer (s.o.) zeigt eine weitere Möglichkeit, die Lesbarkeit zu verbessern. Im Sinne einer Top-down-Entwicklung werden die von einer Funktion aufgerufenen Funktionen möglichst unmittelbar anschließend eingefügt. Ziel ist dabei, möglichst nicht umfangreich in Programmtexten blättern zu müssen.

Testen von Programmcode

Mit JUnit existiert ein inzwischen gut bekanntes Framework zum Testen von Java-Programmen, das in sehr einfacher Weise das Testen von Programmcode ermöglicht. Bei BlueJ ist es inzwischen integriert, so dass von BlueJ wie bei normalen Klassen auch zu Testklassen schon ein Gerüst bereitgestellt wird.
Bei Python ist nicht nur die Ausnahmebehandlung in die Standardsprache integriert, sondern auch die Anweisung **`assert`** mit der zugehörigen **`AssertionError`** – Ausnahme. Außerdem gibt es mit der flexiblen Anweisung **`apply`** die Möglichkeit, die Anwendung einer Funktion oder Methode auf eine Parameterliste durchzuführen und ihr Ergebnis mit **`assert`** zu überprüfen. So kann man unter Python ebenfalls einfach Tests durchführen.

"Ausnahmebehandlung [Quelle: Wikipedia]

Python nutzt ausgiebig die Ausnahmebehandlung (engl. exception handling) als ein Mittel, um Fehlerbedingungen zu testen. Dies ist so weit in Python integriert, dass es teilweise sogar möglich ist, Syntaxfehler abzufangen und zur Laufzeit zu behandeln.
Ausnahmen haben einige Vorteile gegenüber anderen beim Programmieren üblichen Verfahren der Fehlerbehandlung (wie z. B. Fehler-Rückgabewerte und globale Statusvariablen). Sie sind Thread-sicher und können leicht bis in die höchste Programmebene weitergegeben oder an einer beliebigen anderen Ebene der Funktionsaufruffolge behandelt werden.
Der korrekte Einsatz von Ausnahmebehandlungen beim Zugriff auf dynamische Ressourcen erleichtert es zudem, bestimmte auf Race Conditions basierende Sicherheitslücken zu vermeiden, die entstehen können, wenn Zugriffe auf bereits veralteten Statusabfragen basieren."

Test-driven-development

Eine konsequente Verfolgung des Gedankens des umfassenden Testens von Software hat zum Konzept des Test-driven-development geführt. Dabei definiert man die Testklassen bzw. Testprogramme **vor** dem Schreiben des Programmcodes und entwickelt erst dann den Programmcode so weit, bis er die Tests erfolgreich besteht.
Der Vorteil eines solchen Vorgehens liegt an zwei Stellen:

1. Es wird kein Programmcode entwickelt, der zur Lösung der gestellten Aufgabe nicht nötig ist.
2. Nach Änderungen oder Erweiterungen des Programmcodes braucht man nur den ursprünglichen Test heranzuziehen, um herauszufinden ob dadurch Fehler der ursprünglichen Lösung entstanden sind.

Obwohl nach meiner Erfahrung in der Schule keine große Begeisterung für ein solches Vorgehen zu erreichen ist, sollten die Schülerinnen und Schüler das Konzept kennen gelernt haben.
Etwas fraglich ist aus meiner Sicht, ob dafür die Entwicklung eines OO Grafiksystems ein gutes Projekt ist; vielleicht habe ich die Möglichkeiten dafür aber auch nicht ausreichend analysiert. Ich würde also eher ein kleines eigenständiges Projekt dafür heranziehen.