

Wissenschaftliche Softwareentwicklung

12. Funktionen und Methoden

Till Biskup

Physikalisch-Technische Bundesanstalt

06.11.2023





- 🔑 Funktionen sollten so kurz wie möglich sein.
Übersichtlichkeit erleichtert intellektuelle Beherrschbarkeit.
- 🔑 Funktionen sollten genau eine Sache tun, die dafür aber richtig.
Unix-Prinzip: „*Do one thing, and do it well.*“
- 🔑 In einer Funktion sollte nur eine Abstraktionsebene vorherrschen.
Das fördert die Übersichtlichkeit und Verständlichkeit.
- 🔑 Je weniger Parameter eine Funktion hat, desto besser.
Mehr als drei Parameter sollten nie auftreten.
- 🔑 Doppelungen im Code sollten grundsätzlich vermieden werden:
„*Don't Repeat Yourself*“ (DRY).

“ *Functions are the first line of organization in any program.*

– Robert C. Martin

- Programme sind inhärent komplex.
 - Alle Programmierkonzepte zielen letztlich auf die intellektuelle Beherrschbarkeit der Programmierung.
- Struktur und Organisation sorgen für Lesbarkeit.
 - Lesbarkeit ist die Grundvoraussetzung für qualitativ hochwertige Programme.
- Zuständigkeitshierarchien sind weit verbreitet.
 - Expertenwissen erfordert Fokussierung auf ein Thema.

“ *Every system is built from a domain-specific language designed by the programmers to describe that system. Functions are the verbs of that language, and classes are the nouns. [...]*
The art of programming is, and has always been, the art of language design.

– Robert C. Martin

- ☛ Funktionen helfen dabei, eine Sprache zu entwickeln, in der die Lösung zu einem gegebenen Problem verständlich formuliert werden kann.

Länge: So kurz wie möglich

Fokus: Immer nur eine Aufgabe

Parameter: Je weniger, desto besser

Modularität: Don't Repeat Yourself

Wie kurz ist kurz genug?

Zwei Regeln – und ein paar Argumente



Zwei Regeln

- 1 Funktionen sollten kurz sein.
- 2 Funktionen sollten kürzer als kurz sein.

Argumente

- Übersichtlichkeit
 - Die gesamte Funktion sollte auf den Bildschirm passen.
- Fokussierung
 - Die menschliche Aufnahmefähigkeit ist beschränkt.
- Fehlerrate
 - Kurze Funktionen enthalten weniger Fehler.

These

Eine Funktion, die sich über mehr als ca. 20 (Code-)Zeilen erstreckt, ist zu lang und sollte aufgeteilt werden.

- Dokumentation ist nicht mitgerechnet.
 - (Öffentliche) Funktionen sollten einen Dokumentationskopf haben.
 - Mehr Zeilen für Dokumentation als für Code ist *kein* Problem.
- Lange Zeilen sind keine Lösung.
 - Horizontales Scrollen ist viel schlimmer als vertikales.
 - Die Zeilenlänge beeinflusst die Lesefähigkeit.
- Quellcode hat eine hohe Informationsdichte.
 - Die menschliche Auffassungsgabe ist beschränkt.
 - Faustregel: Fokussierung auf 7 ± 2 Aspekte

📌 Argument

Die harte Beschränkung der Länge von Funktionen führt zu einer unübersichtlich großen Zahl kleiner Funktionen.

Entgegnung

- Organisation ist unumgänglich.
 - Viele kleine Funktionen sind eine Möglichkeit.
- Aufteilung sorgt für Fokussierung.
 - Das menschliche Aufnahmevermögen ist begrenzt.
- Modularität sorgt für Wiederverwendbarkeit.
 - nur durch Aufteilung und Fokussierung erreichbar
- 👉 Gute Editoren/IDEs unterstützen
(Autovervollständigung, Springen im Code)

Länge: So kurz wie möglich

Fokus: Immer nur eine Aufgabe

Parameter: Je weniger, desto besser

Modularität: Don't Repeat Yourself

“ After all, the reason we write functions is to decompose a larger concept (in other words, the name of the function) into a set of steps at the next level of abstraction.

– Robert C. Martin

- generelles Vorgehen bei der Programmierung
 - Aufteilung eines Problems in Teilprobleme
 - iterativer Prozess: solange wiederholen, bis die Lösung für ein Teilproblem offensichtlich ist
- Hierarchie von Verantwortlichkeiten
 - Jede Funktion ist für genau eine Ebene verantwortlich.
 - erleichtert das Benennen der Funktion ungemein

☛ Unix-Prinzip: „*Do one thing, and do it well.*“

Was ist „genau eine Aufgabe“?

Hinweise, dass eine Funktion mehr als eine Aufgabe erfüllt



- Sind alle Befehle auf einer Abstraktionsebene?
 - Einheitlichkeit erleichtert das Verständnis.
- Lassen sich Teilaufgaben delegieren?
 - Kriterium: Der Name der Unterfunktion wiederholt nicht lediglich die Codezeile, die er ersetzt.
 - Bsp.: komplexe Abfragen für `if`-Bedingungen
- Gibt es einzelne Blöcke?
 - Oft gliedern einzelne Kommentarzeilen eine Funktion.
 - Hinweis auf delegierbare Teilaufgaben
- Macht die Funktion mehr, als ihr Name impliziert?
 - führt zu Nebenwirkungen (ungewolltem Verhalten)
 - schwer zu findende und zu behebende Fehler

- Funktionen sind Abstraktionen.
 - Die Aufgabe einer Funktion sollte sich in einem Satz beschreiben lassen.
- Abstraktionsebene eine Stufe unter dem Funktionsnamen
 - setzt gute Benennung der Funktion voraus
- Mischung von Abstraktionsebenen verwirrt
 - Unterscheidung zwischen essentiellm Konzept und (relativ) unwichtigem Detail geht verloren.
 - führt zur Ansammlung weiterer Details
 - Code wird auf Dauer unlesbar.
- Mischung von Abstraktionsebenen bläht den Code auf.
 - Funktionen sollten kurz und übersichtlich sein.

Beispiel: Funktion zur Datenauswertung

- 1 Lies die Daten ein.
- 2 Vorverarbeite die Daten.
- 3 Analysiere die Daten nach festen Kriterien.
- 4 Präsentiere das Ergebnis.
- 5 Speichere das Ergebnis.

Listing: Pseudocode einer Funktion zur Analyse von Daten

```
def analyse_my_data():  
    data = read_data()  
    data = preprocess_data(data)  
    result = analyse_data(data)  
    present_result_of_analysis(result)  
    save_result_of_analysis(result)
```

Listing: Funktion, die Abstraktionsebenen mischt

```
def _import_data(self):
2   with open(self.source) as file:
        content = file.read()
4   raw_data = np.genfromtxt(
        io.BytesIO(content.replace(',', ' ').encode()),
6       skip_header=2
    )
8   self.dataset.data.data = raw_data[:, 1]
        self._get_header()
10  self._import_metadata()
```

Listing: Funktion mit nur einer Abstraktionsebene

```
def _import(self):
2   self._import_data()
        self._get_header()
4   self._import_metadata()
```

- ☛ Der Import der eigentlichen Daten wurde ausgelagert, die Funktion (Methode) umbenannt.

- Programmierung: Abbildung realer Probleme in Code
 - Reale Probleme sind zu komplex, um sie direkt in Code übersetzen zu können.
- verbale Beschreibung einer Funktion in einem Satz
 - „Um A zu erreichen, muss B, C und D ausgeführt werden.“
 - „A“ spiegelt sich im Funktionsnamen wider.
 - „B“, „C“ und „D“ sind die aufzurufenden Unterfunktionen.
- iterativer Vorgang
 - Zerlegung solange durchführen, bis die Umsetzung in Code offensichtlich ist
- Hilfsmittel
 - Ablauf einer Funktion in Kommentaren festhalten
 - Blöcke im zweiten Schritt in Funktionen auslagern

Nebenwirkung (*side effect*)

unerwartete Auswirkung eines Funktionsaufrufs,
die nicht aus dem Funktionsnamen hervorgeht

- mögliche Folge: zeitliche Kopplung
 - Aufrufreihenfolge bestimmt das Programmverhalten
 - unerwartetes, „erratisches“ Verhalten
 - schwer zu findender Fehler
- Lösung: genau eine Aufgabe pro Funktion
 - Aufgabe spiegelt sich im Funktionsnamen wider
 - setzt Disziplin bei der Programmierung voraus

Listing: Funktion mit Nebenwirkung

```
def check_password(username="", password=""):  
    user = User(name=username)  
    stored_password = user.get_password()  
    if password == stored_password:  
        session.initialise()
```

- Problem
 - Funktion initialisiert eine Sitzung (letzte Zeile)
 - Funktion nicht in beliebiger Situation aufrufbar
 - (mögliche) Lösung
 - Passwortüberprüfung und Rückgabe Booleschen Wertes
- 👉 Vereinfachtes Beispiel: Passwörter *niemals* im Klartext speichern!

Länge: So kurz wie möglich

Fokus: Immer nur eine Aufgabe

Parameter: Je weniger, desto besser

Modularität: Don't Repeat Yourself

- Übersichtlichkeit
 - je mehr Parameter, desto unübersichtlicher
 - Reihenfolge der Parameter sollte offensichtlich sein – oder egal (Schlüssel–Wert-Zuweisungen)
- Vorteile objektorientierter Programmierung
 - reduziert die Parameterzahl durch den gegebenen Kontext
 - Eigenschaften müssen nicht explizit übergeben werden
- Parameter dienen der Eingabe
 - Rückgabe über Rückgabewerte, nicht über Parameter
 - Eingabeparameter sollten nicht verändert werden.
 - Viele Sprachen unterstützen nur einen Rückgabeparameter (beliebigen Typs, also auch Listen).

- Idealfall: kein Parameter (*niladisch*)
 - nur möglich, wenn der Kontext vorgegeben ist
 - Stärke objektorientierter Programmierung
 - häufiger Fall: ein Parameter (*monadisch*)
 - klarer Zusammenhang zwischen Funktion und Parameter
 - seltenerer Fall: zwei Parameter (*dyadisch*)
 - Zusammenhang zwischen Funktion und Parameter oft nicht aus dem Funktionsnamen ersichtlich
 - erschwert das Lesen von Code
 - sehr seltener Fall: drei Parameter (*triadisch*)
 - nur sehr sparsam einsetzen
- ☛ Strategien zur Reduzierung der Parameterzahl

- kein Parameter (*niladisch*)
 - `timer_start()`
 - `break()`
- ein Parameter (*monadisch*)
 - `file_exists(filename)`
 - `sin(x)`
- zwei Parameter (*dyadisch*)
 - `assertEqual(expected, actual)` (**problematisch!**)
 - `write_parameter_to_file(parameter, filename)`
- drei Parameter (*triadisch*)
 - `assertEqual(expected, actual, precision)`

Zwei Standardformen

- Parameter abfragen
 - häufig Abfrage des Typs oder der Existenz
 - Bsp.: `file_exists("filename") -> bool`
- auf Parameter operieren
 - Das Ergebnis wird als Rückgabewert zurückgegeben.
 - Bsp.: `open_file("filename") -> StringIO`

Dritte Form

- Ereignis senden
 - sorgt für Zustandsänderung des Systems
 - in der Regel ohne Rückgabewert
 - Bsp.: `sleep(5) -> None`

- Listen
 - Zusammenfassung gleichartiger Parameter
 - Bsp.: `print`
- Objekte
 - Zusammenfassung von Parametern beliebigen Typs
 - Abstraktion, die auch der Compiler/Interpreter versteht
 - semantische Information

Anzeichen für mehr als eine Aufgabe

- optionale Schlüssel-Wert-Paare
- Boolesche Werte
- ☛ Aufteilung in (Unter-)Funktionen
- ☛ Ggf. eine „Wrapper“-Funktion als Nutzerschnittstelle

- Zielstellung: Lesbarkeit von Code
 - Funktions- und Parameternamen sollten einen logischen und lesbaren Zusammenhang bilden.
 - bei der Benennung berücksichtigen
- offensichtliche Reihenfolge der Parameter
 - Funktionen mit mehr als einem Parameter sollten durch ihren Namen deren Reihenfolge offensichtlich machen.
- Signaturen von Funktionen dienen der Dokumentation.
 - Moderne Editoren zeigen meist die Signatur an.
 - Gut gewählte Parameternamen ersparen Blick in die Hilfe.
- zwei unterschiedliche Kontexte
 - Deklaration einer Funktion
 - Aufruf einer Funktion

Funktions- und Parameternamen

Der Wert guter Editoren/IDEs: Vorschau und Hilfestellung



```
filename: str = "", entries: list = None, sort: bool = False, after: str = ""
)
utils.add_to_toctree()
index_filename = os.path.join('docs', 'api', 'gui', 'index.rst')
```

```
filename: str = "", entries: list = None, sort: bool = False, after: str = ""
)
utils.add_to_toctree(f)
index_filename = c
if not os.path.exi
    return
modules = ['app',
package = self.cor
utils.add_to_toctr
    filename=index
    entries=[f'{pa
    sort=True
)
def _create_mainwindow
```

filename=

- fcntl
- format(__value, __format_spec) builtins
- ftplib
- flake8
- filter builtins
- float builtins
- frozenset builtins
- function builtins
- faulthandler
- filecmp
- fileinput

Strg+Unten and Strg+Oben will move caret down and up in the editor. Next Tip

Länge: So kurz wie möglich

Fokus: Immer nur eine Aufgabe

Parameter: Je weniger, desto besser

Modularität: Don't Repeat Yourself

“ *Duplication may be the root of all evil in software. Many principles and practices have been created for the purpose of controlling or eliminating it. [...]*

It would appear that since the invention of the subroutine, innovations in software development have been an ongoing attempt to eliminate duplication from our source code.

– Robert C. Martin

- Information genau einmal an genau einem Ort ablegen
- Manuelle Synchronisation ist zum Scheitern verurteilt.

- Faulheit
 - Problem einmal lösen, dafür richtig
 - kostet nur anfänglich mehr Zeit
 - führt zu tieferem Verständnis des Problems
- Wartbarkeit
 - Modularisierung erleichtert die Wiederverwendbarkeit.
 - Isolation: Fehler müssen nur einmal behoben werden.
- Effizienz
 - Wiederverwendbarkeit beschleunigt die Entwicklung.
 - Wiederverwendbarkeit skaliert, Neuschreiben nicht.
- Ausdrucksstärke
 - Funktionen sind die Verben der Sprache.
 - Je mehr Vokabeln verfügbar sind, desto verständlicher lässt sich die Lösung eines Problems formulieren.

- Erweiterbarkeit
 - kreative Kombination vorhandener Bausteine
 - Stichwort: LEGO (die alten, einfachen Steine)
 - Wissenschaft: immer wieder neue Fragestellungen
- Testbarkeit
 - Voraussetzung: definierte Vor- und Nachbedingungen
 - Modularität sorgt für Testbarkeit, und Testbarkeit für Modularität

Parallele zu den Wissenschaften

- Abstraktion und Verständnis der Problemstellung
 - gute Modelle abstrahieren und erklären
- Kombination bekannter Bausteine zu etwas Neuem
 - Wissenschaft baut (fast) immer auf Bekanntem auf

- „*You Ain't Gonna Need It*“ (YAGNI)
 - Nicht jede Verallgemeinerung ist auch sinnvoll.
 - Pragmatismus ist das Gebot der Stunde.
- Kontext nicht aus den Augen verlieren
 - Unterschiedliche Ziele erfordern einen unterschiedlichen Grad an Abstraktion.
 - Ein Framework ist abstrakter als ein konkretes Programm.
 - Abstraktion ist auch eine Frage der Erfahrung.
- Code und Anforderungen sind nicht statisch.
 - Eine Funktion ist nicht beim ersten Mal perfekt.
 - schrittweise Anpassungen, iteratives Vorgehen

👉 Doppelungen genau dann entfernen, wenn sie auftreten



- 🔑 Funktionen sollten so kurz wie möglich sein.
Übersichtlichkeit erleichtert intellektuelle Beherrschbarkeit.
- 🔑 Funktionen sollten genau eine Sache tun, die dafür aber richtig.
Unix-Prinzip: „*Do one thing, and do it well.*“
- 🔑 In einer Funktion sollte nur eine Abstraktionsebene vorherrschen.
Das fördert die Übersichtlichkeit und Verständlichkeit.
- 🔑 Je weniger Parameter eine Funktion hat, desto besser.
Mehr als drei Parameter sollten nie auftreten.
- 🔑 Doppelungen im Code sollten grundsätzlich vermieden werden:
„*Don't Repeat Yourself*“ (DRY).