

# Programmierkonzepte in der Physikalischen Chemie

## 14. Funktionen

Albert-Ludwigs-Universität Freiburg



**UNI  
FREIBURG**

Dr. Till Biskup

Institut für Physikalische Chemie  
Albert-Ludwigs-Universität Freiburg  
Wintersemester 2017/18



- 🔑 Funktionen sollten so kurz wie möglich sein.  
Übersichtlichkeit erleichtert intellektuelle Beherrschung.
- 🔑 Funktionen sollten genau eine Sache tun, die aber richtig.
- 🔑 Innerhalb einer Funktion sollte nur eine Abstraktionsebene vorherrschen.
- 🔑 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.

# Wie kurz ist kurz genug?

Harte Zahlen sind schwierig – aber...



## These

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

- ▶ Quellcode hat eine hohe Informationsdichte.
  - Die menschliche Auffassungsgabe ist beschränkt.
  - Faustregel: Fokussierung auf  $7 \pm 2$  Aspekte
- ▶ Dokumentation ist nicht mitgerechnet.
  - Funktionen sollten einen Dokumentationskopf haben.
- ▶ Lange Zeilen sind keine Lösung.
  - Horizontales Scrollen ist viel schlimmer als vertikales.

# Wird das nicht unübersichtlich?

Ein Plädoyer für viele kleine Funktionen



## 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

“ *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

# 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 Nebeneffekten (ungewolltem Programmverhalten)
  - 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 essentiellen 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.

- ▶ 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

- ▶ Fehlerbehandlung häufig über `try...catch`
  - führt zu zusätzlicher Einrückung
  - verringert die Übersichtlichkeit
  
- ▶ Fehlerbehandlung in eigener Funktion
  - Funktion reduziert auf `try` und `catch`
  - Innerhalb des `try`-Blocks steht nur ein Funktionsaufruf.
  - Die aufgerufene Funktion erledigt die „eigentliche Arbeit“.
  - Fehlerbehandlung über Ausnahmen (*exceptions*)
  
- ▶ Unterfunktionen eng bei Hauptfunktion
  - private Funktionen eines Objektes
  - Unterfunktionen innerhalb einer Funktion
  - Unterfunktionen nach Möglichkeit nur lokal aufrufbar

## Nebeneffekt

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

- ▶ eine Aufgabe pro Funktion
  - Zustand abfragen *oder* verändern
- ▶ keine booleschen Rückgabewerte für „Setter“
  - führt zu schwer lesbarem, da uneindeutigem Code

### Listing 1: Mehrdeutigkeit durch Vermischung von Befehl und Abfrage

```
if (set(attribute, value)) ...
```

---

### Listing 2: Lösung: zwei Funktionen mit klaren Namen

```
if (attributeExists(attribute))  
    setAttribute(attribute, value)
```

---

- ▶ 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).

# Nie mehr als drei Parameter

oberstes Gebot: Übersichtlichkeit und Lesbarkeit

- ▶ 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

# Nie mehr als drei Parameter

## Beispiele für niladische bis triadische Funktionen

- ▶ kein Parameter (*niladisch*)
  - `timerStart()`
  - `break()`
  
- ▶ ein Parameter (*monadisch*)
  - `fileExists(filename)`
  - `sin(x)`
  
- ▶ zwei Parameter (*dyadisch*)
  - `assertEquals(expected, actual)` (problematisch!)
  - `writeParameter2File(parameter, filename)`
  
- ▶ drei Parameter (*triadisch*)
  - `assertEquals(expected, actual, precision)`

## Zwei Standardformen

- ▶ Parameter abfragen
  - häufig Abfrage des Typs oder der Existenz
  - Bsp.: `boolean fileExists("fileName")`
- ▶ auf Parameter operieren
  - Das Ergebnis wird als Rückgabewert zurückgegeben.
  - Bsp.: `InputStream openFile("fileName")`

## Dritte Form

- ▶ Ereignis senden
  - sorgt für Zustandsänderung des Systems
  - in der Regel ohne Rückgabewert

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

## Anzeichen für mehr als eine Aufgabe

- ▶ optionale Schlüssel-Wert-Paare
- ▶ Boolesche Werte
- 👉 Aufteilung in (Unter-)Funktionen

- ▶ 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

# Don't Repeat Yourself!

## Zur Bedeutung von Doppelungen im Code

“ *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.

### ▶ Ausdrucksstärke

- Funktionen sind die Verben der Sprache.
- Je mehr Vokabeln verfügbar sind, desto verständlicher lässt sich die Lösung eines Problems formulieren.

- ▶ „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 Beherrschung.
- 🔑 Funktionen sollten genau eine Sache tun, die aber richtig.
- 🔑 Innerhalb einer Funktion sollte nur eine Abstraktionsebene vorherrschen.
- 🔑 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).