

# Erfolgreiche Anfängerausbildung im Programmieren

Marcus Crestani  
Universität Tübingen  
crestani@informatik.uni-tuebingen.de

Michael Sperber  
DeinProgramm  
sperber@deinprogramm.de

**Abstract:** Das Tutorial gibt einen Überblick über Inhalte und Techniken zur effektiven Anfängerausbildung im systematischen Programmieren. Der im Tutorial vermittelte Ansatz baut auf den *design recipes* bzw. *Konstruktionsanleitungen* des TeachScheme!- bzw. DeinProgramm-Projekts auf – einem System expliziter Anleitungen zur systematischen Konstruktion von Programmen. Diese Anleitungen werden durch eine Progression spezieller auf Anfänger zugeschnittener Programmiersprachen ermöglicht, die in einer Programmierumgebung, die ebenfalls speziell auf Anfänger zugeschnitten ist. Ein über viele Jahre entwickeltes Übungskonzept bringt das Konzept zur vollen Entfaltung. Dieser Artikel beschreibt Motivation und wichtige Bestandteile des Lehransatzes und demonstriert die Anwendung der Konstruktionsanleitungen an einem Beispiel. Zum Lehransatz sind ein universitäres Lehrbuch, *Die Macht der Abstraktion*, ein wachsendes Schulbuch, umfangreiche Software sowie weiteres Lehrmaterial (darunter eine umfangreiche Aufgabensammlung) verfügbar.

## 1 Einleitung

Programmieranfänger haben andere Bedürfnisse als fertig ausgebildete professionelle Programmierer. Traditionelle Programmierkurse tragen diesen Bedürfnissen nur unzureichend Rechnung: Das Ergebnis sind oft Absolventen, die nicht oder nur unsystematisch und fehlerhaft programmieren können [BDH<sup>+</sup>08a].

Der im Tutorial vermittelte Ansatz baut auf den *design recipes* [FFFK04] bzw. *Konstruktionsanleitungen* des TeachScheme!/DeinProgramm-Projekts [KS07] auf. Das Konzept ist die Grundlage für die Vorlesungen „Informatik I“ an den Universitäten Tübingen und Freiburg und wird dort seit mehreren Jahren erfolgreich praktiziert und verbessert [BDH<sup>+</sup>08b]. Es hat sich durch mehrere Evaluationen über verschiedene Dozenten als robust erwiesen und wird kontinuierlich aktualisiert und verbessert.

Die Konstruktionsanleitungen fangen mit einer Analyse der Daten eines Problems an und liefern Schritt für Schritt die Elemente des Programms, die sich aus der Struktur der Daten ergeben. Die systematische Konstruktion wird durch eine Progression spezieller, auf Anfänger zugeschnittener Programmiersprachen ermöglicht, die eine seichte Lernkurve mit direktem Anschluss an den Mathematikunterricht über den Umgang mit Daten hin zu

umfangreicheren Programmen (z.B. reaktiven Animationen) ermöglichen. Diese laufen in einer Programmierumgebung, die speziell auf Anfänger zugeschnitten ist. Ein über viele Jahre entwickeltes Übungskonzept bringt den Ansatz zur vollen Entfaltung.

## 2 Traditionelle Programmierausbildung

Die traditionelle Programmierausbildung tut sich oft schwer, das Programmieren erfolgreich zu vermitteln:

- Professionelle Programmiersprachen (wie z.B. C++, Java und Pascal/Delphi) sind sehr umfangreich; die Behandlung ihrer Sprachelemente, der Syntax und idiosynkratischen Aspekte der Semantik nimmt einen Großteil der Ausbildung in Anspruch. Damit bleibt wenig Zeit, die Grundkonzepte des Programmierens zu behandeln [Köl99a].
- Programmierumgebungen, die für professionelle Software-Entwickler entwickelt wurden, setzen sie vieles voraus, das Anfänger erst noch lernen müssen. Außerdem liefern sie für Anfänger ungenügendes Feedback, wenn es Probleme gibt. [Köl99b, FCF<sup>+</sup>02].
- Damit bleiben frühe Erfolgserlebnisse aus, die für die Freude am eigenständigen Lernen wichtig sind. Anfänger, welche auf dem Weg zur Lösung nicht schnell sichtbare Fortschritte machen, geben oft auf und schreiben ab [BDH<sup>+</sup>08a].
- Da Konzepte nicht systematisch vermittelt werden, sind diese nur autodidaktisch zugänglich. Damit verkümmert der Unterricht zu einer reinen Polarisierung zwischen den Autodidakten – die zudem oft nur unsystematisch programmieren lernen – und denjenigen Studierenden, die den Unterricht ohne Gewinn wieder verlassen. (Obwohl das Problem schon lange bekannt ist [Gri74], ist die heutige Ausbildung erstaunlich frei von Lösungen.)

## 3 Effektive Programmierausbildung für Anfänger

Die folgenden, aufeinander abgestimmten Bausteine, lösen gemeinsam die Probleme der traditionellen Ausbildung:

**Explizite Anleitung** Die *design recipes* in der Terminologie von Felleisen [FFFK01] bzw. *Konstruktionsanleitungen* bei Klaeren/Sperber [KS07] bieten *explizite* Anleitungen zur systematischen Programmieren. Abschnitt 4 beschreibt, wie die Konstruktionsanleitungen das systematische Problemlösen ermöglichen.

**Programmiersprache** Der Ansatz verwendet eine Lehrsprache basierend auf der funktionalen Sprache Scheme [SDFvS10], die es erlaubt, im direkten Anschluss an den

Mathematikunterricht alle grundlegenden Konzepte des Programmierens nachvollziehbar darzustellen. Vollständige Programme werden bereits in der ersten Unterrichtsstunde geschrieben. Die Sprachen sind sehr klein, womit Dozenten nur wenig Zeit der spezifischen Behandlung der Sprachen widmen müssen.

**Programmierungsumgebung** Die Programmierungsumgebung *DrRacket* (früher *DrScheme*) ist speziell auf Anfänger zugeschnitten [FCF<sup>+</sup>02]: Sie ist interaktiv und ermöglicht das schrittweise sofortige Testen und visualisiert den Programmablauf. DrRacket liefert präzise Fehlermeldungen, die Anfänger aktiv bei der Fehlersuche unterstützen [CS10]. In DrRacket ist aber auch die Entwicklung anspruchsvoller Programme mit grafischer Benutzeroberfläche bis hin zu professionellen Applikationen möglich.

**Übungen** Im Rahmen der traditionellen Übungsorganisation bieten wir eine umfangreiche, annotierte Aufgabensammlung. Zudem haben wir neue Ansätze zur Betreuung („Betreutes Programmieren“) und innovative Konzepte für Gruppenarbeit („Gruppentestate“) [BDH<sup>+</sup>08a] entwickelt, die den Lernstoff besonders effektiv vermitteln. Ein Online-Kursinformationssystem erleichtert die organisatorische Umsetzung und unterstützt den Übungsbetrieb.

## 4 Systematische Problemlösung nach Konstruktionsanleitung

Die Konstruktionsanleitungen unterscheiden unseren Ansatz von traditioneller Programmierausbildung, die mit Beispielen funktionierender Programme oder „design patterns“ arbeiten. (Die mächtigen Abstraktionsmöglichkeiten unserer Lehrsprachen machen „design patterns“ weitgehend überflüssig.) Die Konstruktionsanleitungen trennen die systematischen von den kreativen Aspekten des Problemlösungsprozesses und erlauben damit Anfängern, sich auf den kreativen Teil zu konzentrieren. Im Folgenden stellen wir die Problemlösung mit Konstruktionsanleitung anhand einer einfachen Beispielaufgabe vor. Die Erarbeitung der Aufgabenlösung folgt den Schritten der Konstruktionsanleitungen:

Ein Computer besteht vereinfacht gesehen aus Prozessor, Hauptspeicher und Festplatte. Die Taktfrequenz des Prozessors gibt man in Ghz, die Größe des Hauptspeichers und der Festplatte in GB an.

Programmieren Sie Computer! Schreiben Sie ein Programm, in dem es möglich ist, den Gesamtspeicher eines Computers zu berechnen!

**Kurzbeschreibung** Die Kurzbeschreibung fasst die Aufgabe der Prozedur in einer einzelnen Zeile zusammen:

```
; Gesamtspeicher eines Computers berechnen
```

**Datenanalyse** Die Datenanalyse ist der zentrale Teil der Konstruktionsanleitung. Sie klassifiziert die Arten von Informationen, die in der Aufgabenstellung auftauchen: In der Aufgabenstellung taucht die Größe „Computer“ auf. Der Begriff „Computer“ kann anhand der Aufgabenstellung weiter ausgeführt werden:

Ein Computer hat einen Prozessor mit Taktfrequenz in Ghz, einen Hauptspeicher mit Größe in GB und eine Festplatte mit Größe in GB. Diese Formulierung ist eine *Datendefinition*. Eine Formulierung der Form „x hat“ identifiziert ein Objekt als *zusammengesetzte Daten*: Es besteht aus mehreren Bestandteilen. Solch eine Datendefinition wird als Kommentar formuliert und dann direkt in Code in Form einer Record-Definition übersetzt:

```
; Ein Computer besteht aus:
; - Prozessor (in Ghz)
; - Hauptspeicher (in GB)
; - Festplatte (in GB)
(define-record-procedures computer
  make-computer
  computer?
  (computer-processor computer-ram computer-drive))
(: make-computer (rational rational rational -> computer))
```

Diese Form definiert die Signatur `computer` für Computer sowie mehrere Prozeduren für den Umgang mit Record-Werten: den Konstruktor `make-computer`, um Computer herzustellen, das Prädikat `computer?`, um Computer von anderen Sorten von Werten zu unterscheiden und die Selektoren `computer-processor`, `computer-ram` und `computer-drive`, die für einen Computer jeweils den Prozessor, den Hauptspeicher und die Festplatte liefern.

Schließlich wird noch die Signatur für die Konstruktorprozedur `make-computer` deklariert: `make-computer` akzeptiert drei Zahlen (Taktfrequenz des Prozessors und Größen von Hauptspeicher und Festplatte). Die Signatur `rational` für rationale Zahlen ist eingebaut.

**Signatur** Die Signatur ähnelt einer Typsignatur der Prozedur. Sie enthält den Namen der Prozedur, welche Arten von Daten die Prozedur erwartet und welche Art sie zurückgibt. Aus der Aufgabenstellung ist ersichtlich, dass die Prozedur, die den Gesamtspeicher berechnet, einen Computer konsumiert und eine Zahl zurückgibt:

```
(: total-memory (computer -> rational))
```

**Testfälle** Die Testfälle werden vor der Prozedur geschrieben und erlauben den Studierenden, erste Korrektheitskriterien aufzustellen. Der erste Schritt für die Testfälle ist die Erstellung von Beispielen für die auftretenden Daten. Diese Beispiele sollten durch Kommentare die Beziehung zwischen den Daten und der repräsentierten Information klarstellen. In diesem Fall könnten Beispiele so aussehen:

```
; Computer mit 1 Ghz, 2 GB RAM, 500 GB Festplatte
(define c1 (make-computer 1 2 500))
; Computer mit 2 Ghz, 4 GB RAM, 1000 GB Festplatte
(define c2 (make-computer 2 4 1000))
```

Mit diesen Beispielen lassen sich folgende Testfälle formulieren:

```
(check-expect (total-memory c1) 502)
(check-expect (total-memory c2) 1004)
```

Die Form `check-expect` akzeptiert zwei Operanden, deren Werte gleich sein sollen. Sind sie es nicht, werden Testfall-Fehlschläge in einem Protokoll festgehalten.

**Gerüst** Der erste Schritt zur Konstruktion der eigentlichen Prozedur ist das Gerüst, das sich direkt aus der Signatur ergibt: Die Studierenden müssen lediglich die Namen für die Parameter wählen.

```
(define total-memory
  (lambda (c)
    ...))
```

Das `define` benennt einen Wert mit dem Namen `total-memory` – der `lambda`-Ausdruck liefert eine Prozedur mit dem Parameter `c` (für den Computer). Die Ellipse steht für den noch zu schreibenden Rumpf der Prozedur.

**Schablone** Die Schablone leitet sich rein aus der Form der Daten her, ohne die Bedeutung oder Zweck der Prozedur in Betracht zu ziehen. Die Studierenden können an der Form der Daten die Programmelemente ablesen, die im späteren Rumpf der Prozedur auftauchen. In den meisten Fällen kommen mehrere Schablonen zum Einsatz:

1. Der Parameter `c` muss im Rumpf auftauchen:

```
(define total-memory
  (lambda (c)
    ... c ...))
```

2. Da es sich bei `c` um zusammengesetzte Daten handelt, wird eine Prozedur die Bestandteile des Computers benötigen. Die Schablone wird also erweitert:

```
(define total-memory
  (lambda (c)
    ... c ...
    ... (computer-processor c) ...
    ... (computer-ram c) ...
    ... (computer-drive c) ...))
```

**Rumpf** Erst in diesem Schritt ist problemspezifisches Wissen nötig, um die Lücken der Schablone im Rumpf der Prozedur zu füllen – bis hierher ist das Programm systematisch aus der Datenanalyse der Problemstellung entstanden. Zunächst spielt für die Berechnung des Gesamtspeichers eines Computers die Prozessor-Komponente keine Rolle. Dann geht es noch darum, die Verknüpfung der im Rumpf verbliebenen Größen zu finden, in diesem Fall die Addition. Damit ist die Konstruktion des Rumpfes abgeschlossen, der vollständig so aussieht:

```
(define total-memory
  (lambda (c)
    (+ (computer-ram c)
       (computer-drive c))))
```

**Testen** Der letzte Schritt ist das Testen: Die vorher erstellten Testfälle werden ausgeführt und Fehler werden behoben, die durch die Testfälle aufgedeckt wurden.

## 5 Aufruf zur Zusammenarbeit

Wir laden alle Interessenten ein, den Ansatz auszuprobieren und gemeinsam zu verbessern.

### Literatur

- [BDH<sup>+</sup>08a] Annette Bieniusa, Markus Degen, Phillip Heidegger, Peter Thiemann, Stefan Wehr, Martin Gasbichler, Marcus Crestani, Herbert Klaeren, Eric Knauel und Michael Sperber. Auf dem Weg zu einer robusten Programmierausbildung. In Andreas Schwill, Hrsg., *3. Workshop des GI-Fachbereichs Hochschuldidaktik der Informatik*, Potsdam, Germany, Dezember 2008.
- [BDH<sup>+</sup>08b] Annette Bieniusa, Markus Degen, Phillip Heidegger, Peter Thiemann, Stefan Wehr, Martin Gasbichler, Marcus Crestani, Herbert Klaeren, Eric Knauel und Michael Sperber. HtDP and DMdA in the Battlefield. In Frank Huch und Adam Parkin, Hrsg., *Functional and Declarative Programming in Education*, Victoria, BC, Canada, September 2008.
- [CS10] Marcus Crestani und Michael Sperber. Growing Programming Languages for Beginning Students. In Stephanie Weirich, Hrsg., *Proceedings International Conference on Functional Programming 2010*, Baltimore, Maryland, USA, September 2010. ACM Press, New York.
- [FCF<sup>+</sup>02] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul A. Steckler und Matthias Felleisen. DrScheme: A Programming Environment for Scheme. *Journal of Functional Programming*, Seiten 159–182, Marz 2002.
- [FFFK01] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt und Shriram Krishnamurthi. *How to Design Programs*. MIT Press, 2001.
- [FFFK04] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt und Shriram Krishnamurthi. The TeachScheme! Project: Computing and Programming for Every Student. *Computer Science Education*, Marz 2004.
- [Gri74] David Gries. What should we teach in an introductory programming course? In *Proceedings of the Fourth SIGCSE Technical Symposium on Computer Science Education*, Seiten 81–89, Detroit, Michigan, USA, 1974.
- [KS07] Herbert Klaeren und Michael Sperber. *Die Macht der Abstraktion*. Teubner Verlag, 1st. Auflage, 2007.
- [Köl99a] Michael Kölling. The Problem of Teaching Object-Oriented Programming, Part 1: Languages. *Journal of Object-Oriented Programming*, 11(8):8–15, January 1999.
- [Köl99b] Michael Kölling. The Problem of Teaching Object-Oriented Programming, Part 2: Environments. *Journal of Object-Oriented Programming*, 11(9):6–12, April 1999.
- [SDFvS10] Michael Sperber, R. Kent Dybvig, Matthew Flatt und Anton van Straaten, Hrsg. *Revised[6] Report on the Algorithmic Language Scheme*. Cambridge University Press, 2010.