Erfolgreich Programmieren lernen und lehren

Marcus Crestani, Universität Tübingen
Michael Sperber, DeinProgramm

Kurzfassung: Die meisten Lehransätze für die Anfängerausbildung im Programmieren bauen auf unsystematisch konstruierten Beispielen auf und bereiten damit Anfänger nur unzureichend darauf vor, eigenständig Programme zu schreiben. Die Lehre im Programmieren sollte stattdessen klar identifizierbare, systematische Techniken zur Programmkonstruktion vermitteln. Wir demonstrieren das Problem anhand eines konkreten Beispiels aus einem populären Lehrbuch zur objektorientierten Programmierung und zeigen dann, wie durch Anwendung von *Konstruktionsanleitungen* – explizite Techniken zur Programmkonstruktion – und spezieller, rein funktionaler Lehrsprachen eine systematische Lösung für das gleiche Problem entsteht.

1 Einleitung

Techniken, um erfolgreich programmieren zu lernen, unterscheiden sich nicht grundlegend von Techniken, andere hochstehende Tätigkeiten erfolgreich zu lernen: Lernende müssen über einen längeren Zeitraum *bewusst üben* – mit dem Ziel, besser zu werden. Es reicht nicht, nur ein Skript zu verstehen oder in der Vorlesung gut zuzuhören. Zum effektiven bewussten Üben gehören im Allgemeinen (Ericsson, Krampe, & Tesch-Romer, 1993):

- Anstreben guter technische Ausführung (nicht nur Resultaten)
- Setzen spezifischer Ziele
- Einholen und Benutzen von zeitnahem und präzisem Feedback

Leider ist die Anwendung dieser Prinzipien auf die Programmierausbildung in der Praxis erstaunlich schwierig. Die wenigen Untersuchungen, die es zum Erfolg traditioneller Anfängerausbildung gibt, sind meist von Misserfolgen geprägt (Börstler, Nordström, Kallin Westin, Moström, & Eliasson, 2008). Dies liegt daran, dass die Informatik bisher wenig Augenmerk darauf gelegt hat, Lehrmethoden zu entwickeln, die Studierende zu erfolgreichem, eigenständigem Üben überhaupt erst befähigen.

Die herkömmliche Anfängerausbildung funktioniert meist über Beispiele: Es werden fertige Programme erläutert, ohne ihren Entstehungsprozess so genau zu beschreiben, dass die Studierenden ihn selbständig zu Hause nachvollziehen können – geschweige denn, ihn auf neue Aufgaben übertragen. Das Resultat ist oft, dass die Anfänger sich nach anfänglichen Misserfolgen eigenständiges Programmieren nicht mehr zutrauen, das Üben einstellen und damit nachhaltig scheitern – die Veranstaltung verlassen oder durch Abschreiben den Erfolg vortäuschen.

Dar Lehransatz der Anfängervorlesungen an den Universitäten Tübingen und Freiburg unterscheidet sich radikal von der traditionellen Ausbildung: Zentral sind ein didaktisches Konzept, das die Studierenden explizit anleitet, systematisch zu programmieren (anstatt nur Beispiele zu präsentieren), sowie eine speziell auf die Anforderungen der Anfänger zugeschnittene Serie angepasster Programmiersprachen (im Gegensatz zu den üblichen Sprachen, die auf die Anforderungen professioneller Programmierer zugeschnittenen sind). In diesem Papier demonstrieren wir Problem und Lösung anhand eines realen Unterrichtsbeispiels.

2 Prozess und Programmieren

Zur Anfängerausbildung im Programmieren gehören naturgemäß Beispielprogramme. Sie allein reichen jedoch nicht aus, Anfänger zu befähigen, eigenständig Programme zu schreiben. Für die erfolgreiche Lehre ist es nötig, den Anfängern zumindest zu beschreiben, *wie* ein Programm zustandegekommen ist. Hier ist ein Fragment eines begleitenden Beispiels aus einem populären Buch für die Anfängerausbildung in objektorientierter Programmierung mit dem Ansatz “objects first” (Barnes & Kölling, 2008):

```
public class Circle
{
    private int diameter;
    private int xPosition;
```

```
    private int yPosition;
    private String color;
    private boolean isVisible;
    ...
    /* Slowly move the circle horizontally by 'distance' pixels. */
    public void slowMoveHorizontal(int distance)
    {
        int delta;
        if(distance < 0)
        {
            delta = -1;
            distance = -distance;
        }
        else
        {
            delta = 1;
        }

        for(int i = 0; i < distance; i++)
        {
            xPosition += delta;
            draw();
        }
    }
    ...
}
```

Das Beispiel wird im Buch verwendet aber nicht erläutert. Es ähnelt aber vielen anderen Programmen aus dem Buchtext selbst, welche die gleichen Probleme aufweisen und ist damit für unsere Diskussion repräsentativ. Einige Aspekte des Programms widersprechen offensichtlich einer systematischen Konstruktion:

- Das Attribut `isVisible` gehört zur Klasse `Circle` – aber ist ein unsichtbarer Kreis noch ein Kreis? (Insbesondere hätte er dann keine Farbe mehr.)
- Obwohl das Beispiel im Rahmen eines Lehransatzes „objects first“ präsentiert wird, ist die Methode nicht objektorientiert programmiert und benutzt imperative Schleifen.
- Die Manipulation des Vorzeichens von `distance`, um die erst später folgende Schleife einfach zu halten, ist bestenfalls eine Optimierung, eigentlich aber ein Hack.

Das Design der Methode offenbart ein noch tiefergehendes Problem, nämlich die selbstverständliche Verwendung von Zuweisungen zur Modellierung von Zeit in der Abbildung des Kreises: Die Aufgabe, welche die Methode löst, ist, die Position des Kreises von der Zeit abhängig zu machen. Dieser funktionale Zusammenhang ist im Code allerdings nur

implizit vorhanden. Schlimmer noch: die korrekte Abbildung des funktionalen Zusammenhangs ist von der Reihenfolge der Zuweisung an `xPosition` und des Aufrufs von `draw` abhängig.[1]

Um es zusammenzufassen: Das Beispiel ist schlechter Code. Die Verwendung von Zuweisung und Objektidentität von Anfang an führt zusätzlich zu Schwächen in der Erläuterung im Buch:

Das Buch versteht unter "state" "the set of values of all attributes defining an object" (S. 8), unterscheidet also nicht zwischen Zustand und Eigenschaften. Zuweisungen werden so erklärt (S.29): "Assignment statements work by taking the value of what appears on the right-hand side […] and copying that value into a variable [...]". Diese Erklärung ist zumindest verwirrend, da die Zuweisung in Java bei Objekten nur die Referenz kopiert, nicht das Objekt – der Begriff der Referenz, der kritisch für das Verständnis ist, fehlt aber in der Erläuterung. Die Verwendung von Zustand zur Modellierung von Zeit und damit die Verwendung von Objektidentität verursacht zusätzliche Verwirrung: Es kann zwei Kreis-Instanzen an der gleichen Position mit der gleichen Farbe geben – diese sind intuitiv gleich, aber die Zuweisungen unterscheiden zwischen beiden.

Dass die systematische Programmkonstruktion ausbleibt, ist also insbesondere ein Resultat des für den Anfang gewählten imperativen Programmierparadigmas. Entsprechend führt das Buch die imperative `for`-Schleife auch nicht über bestimmte Probleme ein, zu deren Lösung sie notwendig wäre, sondern erklärt einfach die Semantik des Konstrukts.

Das Beispiel, sowie das Buch, in dem es erscheint, stehen nicht allein: Die meisten Bücher, die in die Programmierung einführen, lehren über solche unsystematisch entstandenen Beispiele und können dementsprechend auch keine systematischen Techniken beschreiben, mit denen sie zustandegekommen sind. (In vieler Hinsicht ist (Barnes & Kölling, 2008) besser als viele andere.) Zwar wird in vielen Veröffentlichungen zur Anfängerausbildung die Bedeutung des Programmierprozesses betont, jedoch geben die meisten von ihnen keine systematische Anleitung für alle

[1]Dieses Problem wird dann sichtbar, wenn die Aufgabenstellung dahingehend geändert wird, dass der Kreis diagonal bewegt werden soll: Zwischen der Zuweisung von `xPosition` und der dann fälligen Zuweisung an `yPosition` entsteht ein inkorrekter Zwischenzustand.

Schritte des Programmierprozesses an – insbesondere nicht für die Konstruktion von Funktionen/Prozeduren/Methoden, die den Löwenanteil der Programmierung ausmachen (Caspersen & Kölling, 2009).

3 Programmieren mit Konstruktionsanleitungen

Die Anfängerausbildung im Programmieren an den Universitäten Tübingen und Freiburg benutzt ein System expliziter Anleitungen zur systematischen Konstruktion von Programmen, das den Programmierprozess vollständig abbildet (Felleisen, Findler, Flatt, & Krishnamurthi, 2003; Klaeren & Sperber, 2007): Die *Konstruktionsanleitungen* befähigen Anfänger, selbständig funktionierende Programme zu schreiben und damit zielgerichtet und erfolgreich eigenständig zu üben. Jeder Schritt der Konstruktionsanleitung produziert ein Element des späteren fertigen Programms. Die Konstruktionsanleitungen beginnen mit einer Analyse der Daten eines Problems und liefern Schritt für Schritt die Elemente des Programms, die sich aus der Struktur der Daten ergeben. Diese systematischen Aspekte des Problemlöseprozesses werden durch die Konstruktionsanleitungen von den kreativen Aspekten getrennt, die meist leicht fallen, wenn die Vorarbeit der Konstruktionsanleitungen getan ist.

Zu unserem Ansatz gehört eine Reihe speziell auf die Anforderungen der Anfänger zugeschnittene Serie angepasster Programmiersprachen (Crestani & Sperber, 2010)auf der funktionalen Programmiersprache Scheme (Sperber, Dybvig, Flatt, & van Straaten, 2009). Programmieranfänger können bereits nach der ersten Unterrichtsstunde vollständige Programme schreiben und kommen dadurch früh zu Erfolgserlebnissen. Die Lehrsprachen sind außerdem am Anfang *rein funktional*, was die oben angesprochenen Probleme mit Zustand und Objektidentität von vornherein vermeidet. Die rein funktionale Programmierung knüpft außerdem direkt an die Schulalgebra an, was das Lernen zusätzlich erleichert. Als Programmierumgebung verwenden wir das speziell für Anfänger entwickelte DrRacket (früher DrScheme) (Findler et al., 2002).

Im Folgenden programmieren wir das Beispiel aus Abschnitt 2 erneut unter Verwendung von Konstruktionsanleitungen. Der Platz reicht leider nicht aus, um alle Aspekte der Konstruktionsanleitungen oder die Lehrsprachen ausführlich zu erläutern; diese sind anderweitig dokumentiert

(Börstler & Sperber, 2008; Klaeren & Sperber, 2007; Felleisen et al., 2003). Wir nehmen dazu folgende Aufgabenstellung an:

> Schreiben Sie ein Programm, das einen Kreis horizontal von links nach rechts über eine zweidimensionale Zeichenfläche bewegt. Ein Kreis besteht aus Mittelpunkt, Radius und Farbe.

Die im Folgenden vorgestellte Lösung benutzt die DrRacket-Sprachebene *Die Macht der Abstraktion – Anfänger* und die Teachpacks `image2` und `universe` zum Zeichnen und Animieren.

Der zentrale Teil der Konstruktionsanleitungen ist die *Datenanalyse*: In der Aufgabenstellung taucht die Größe „Kreis“ auf. Der Begriff „Kreis“ kann anhand der Aufgabenstellung weiter ausgeführt werden: Ein Kreis hat einen Mittelpunkt, einen Radius und eine Farbe. (Wir korrigieren gleich den Fehler des ursprünglichen Beispiels und lassen `isVisible` weg.) Diese Formulierung ist eine *Datendefinition*, die direkt in die Definition eines Record-Typs und die Signatur ihres Konstruktors übersetzt werden kann:

```
; Ein Kreis besteht aus:
; - Mittelpunkts (Punkt)
; - Radius (Nichtnegative Zahl)
; - Farbe (Zeichenkette)
(define-record-procedures circle
  make-circle circle?
  (circle-center circle-radius circle-color))
(: make-circle (vector natural string -> circle))
```

Wir benutzen Vektoren (noch zu definieren), um Koordinaten zu repräsentieren und – aus Platzgründen – Zeichenketten wie `"red"` und `"green"` für die Repräsentation von Farben. Die Ebene ist zweidimensional, ein Vektor hat also X- und eine Y-Koordinate. Es handelt sich daher auch hier um zusammengesetzte Daten. Daraus entstehen nach derselben Konstruktionsanleitung eine Daten- und eine Record-Definition:

```
; Ein Vektor besteht aus:
; - X-Koordinate
; - Y-Koordinate
(define-record-procedures vector
 make-vector vector?
 (vector-x vector-y))
(: make-vector (integer integer -> vector))
```

Hier sind einige Beispiele für Vektoren und Kreise. Auch ihre Erstellung gehört zur Datenanalyse; die Kommentare machen die Beziehung zwischen Information und Daten deutlich:

```
(define v1 (make-vector 0 0)) ; Ursprung
(define v2 (make-vector 10 10)) ; X-Koordinate 10, Y-Koordinate 10
(define c1 (make-circle 20 v1 "red"))
  ; roter Kreis mit Radius 20 und Mittelpunkt im Ursprung
(define c2 (make-circle 10 v2 "blue"))
  ; blauer Kreis mit Radius 10 und Mittelpunkt bei (10,10)
```

Aus der Sicht der Schulmathematik findet die Bewegung des Kreises in Abhängigkeit von der Zeit statt – für den Mittelpunkt $\vec{p}$ ergibt sich also die Gleichung $\vec{p}=\vec{p}_0+t\cdot\vec{d}$. Dabei ist t die Zeit und $\vec{d}$ die Bewegungsrichtung und -geschwindigkeit. Die Prozedur, die den Kreis bewegt, braucht also drei Argumente: den Kreis, die Zeit und den Richtungsvektor. Dies führt zur folgenden *Kurzbeschreibung* und *Signatur* der Prozedur:

```
; Kreis zeitabhängig bewegen
(: move-circle (circle real vector -> circle))
```

Hier ein exemplarischer *Testfall*:

```
(check-expect (move-circle c2 5 (make-vector 1 2))
              (make-circle 10 (make-vector 15 20) "blue"))
```

Das *Gerüst* der Prozedur folgt direkt aus der Signatur:

```
(define move-circle
  (lambda (c t v)
    ...))
```

Die Prozedur akzeptiert neben Zeit und Richtung die Repräsentation eines Kreises und liefert die Repräsentation des bewegten Kreises. Da es sich bei Kreisen um zusammengesetzte Daten handelt, greifen zwei Konstruktionsanleitungen, um eine sogenannte *Schablone* zu konstruieren. Schablonen liefern Programmfragmente, die dem Aufbau der Daten folgen und die somit im Rumpf der Prozedur auftauchen sollten. Für den Eingabe-Kreis müssen die Selektoren aufgerufen werden, da mit den Komponenten gerechnet wird. Der zurückgegebene Kreis muss durch einen Aufruf des Konstruktors erzeugt werden. Diese Elemente, die nur den Daten folgen, aber keinerlei tiefes Verständnis der Lösung der Aufgabe voraussetzen, werden vermerkt:

```
(define move-circle
  (lambda (c t v)
    (make-circle ... ... ...)
    ... (circle-radius c) ...
    ... (circle-center c) ...
    ... (circle-color c) ...))
```

Der Lernende ist durch den unvollständigen Konstruktoraufruf gehalten, folgende Fragen zu beantworten:

- Was ist der Radius des bewegten Kreises?
- Was ist der Mittelpunkt des bewegten Kreises?
- Was ist die Farbe des bewegten Kreises?

Die Antworten auf die erste und die letzte Frage sind trivial (Radius und Farbe des ursprünglichen Kreises bleiben unverändert), die zweite Frage wird durch die obige Formel beantwortet, die ausdrücklich an Bereichswissen aus der Mathematik anknüpft. In der Formel kommen die Addition zweier Vektoren und die Multiplikation einer Zahl mit einem Vektor vor. Die Summe und das Produkt sind *Zwischengrößen*, die in Hilfsprozeduren berechnet werden. Diese Hilfsprozeduren werden zunächst durch *Wunschdenken* vorausgesetzt und dann später programmiert. (Ihre Konstruktion mussten wir aus Platzgründen weglassen – sie folgt aber den gleichen Prinzipien.) Wir müssen die Formel nur noch in Code übersetzen:

```
(define move-circle
  (lambda (c t v)
    (make-circle (circle-radius c)
                 (add-vectors (circle-center c)
                              (scale-vector t v))
                 (circle-color c))))
```

Um den Kreis in Bewegung zu setzen und grafisch darzustellen, brauchen wir noch eine Prozedur, welche die Repräsentation eines Kreises in das Bild eines Kreises umwandelt. Dazu wird die eingebaute Prozedur `circle` verwendet:

```
; Zeichnet einen Kreis
(: draw-circle (circle -> image))

(define draw-circle
  (lambda (c)
     (circle (circle-radius c) "solid" (circle-color c))))
```

Auch hier kam die Konstruktionsanleitung für zusammengesetzte Daten zum Einsatz. Schließlich soll das Kreisabbild noch auf der zweidimensionalen Zeichenfläche platziert und angezeigt werden. Dazu gehören die Größe der Zeichenfläche, der Kreis sowie Richtungsvektor und Zeit. Hier Kurzbeschreibung und Signatur:

```
; Kreis auf Bild der zweidimensionalen Ebene platzieren
(: circle-scene (natural natural circle vector real -> image))
```

Für die Konstruktion der Ebene verwenden wir die Prozedur `place-image`, die ein Bild an spezifizierten Koordinaten in der Ebene platziert. Auch diese Prozedur können wir schrittweise systematisch durch die Anwendung der Konstruktionsanleitung für zusammengesetzte Daten sowie das Herausziehen eines mehrfach vorkommenden Ausdrucks in eine lokale Variable konstruieren. Das Ergebnis sieht so aus:

```
(define circle-scene
  (lambda (w h c v t)
    (let ((newc (move-circle c t v)))
      (place-image (draw-circle newc)
                   (vector-x (circle-center newc))
                   (vector-y (circle-center newc))
                   (empty-scene w h)))))
```

Die vom `universe`-Teachpack bereitgestellte Prozedur `animate` animiert diese Szenen. `Animate` akzeptiert als Argument eine Prozedur, die abhängig von der verstrichenen Zeit ein Bild erzeugt und zeigt die erzeugten Bilder als kleinen Film in einem neuen Fenster:

```
; Bewegt einen Kreis in einer Animation von links nach rechts
(animate (lambda (t)
           (circle-scene 100 100 c2 (make-vector 1 0) t)))
```

In diesem Fall bewegt sich der Kreis horizontal, aber auch vertikale und diagonale Bewegung ist möglich. Das Programm ist fertig.

4 Zusammenfassung

Viele Autoren und Dozenten wählen für die Programmierausbildung Beispiele, deren Konstruktionsprinzipien sie nicht vollständig erklären können. Dies hindert die Lernenden daran, eigenständig Programme zu konstruieren. Stattdessen würden Autoren von Lehrbüchern und Dozenten besser daran tun, ihre Unterrichtsbeispiele konsequent daran auszurichten, dass sie sich systematisch konstruieren lassen: Jedes Element des Programms sollte auf eine identifizierbare, explizit gelehrte Technik zurückzuführen sein. Einerseits stärkt diese Maxime den didaktischen Wert der Beispiele; in vielen Fällen führt sie zusätzlich zu besserem Code. Dass dies möglich ist, zeigen die erfolgreichen Anfängerausbildungen an den Universitäten Tübingen und Freiburg zusammen mit dem dazugehörigen Lehrmaterial.

5 Literatur

Barnes, D. J., & Kölling, M. (2008). *Objects First with Java - A Practical Introduction using BlueJ*. Pearson Education.

Börstler, J., Nordström, M., Kallin Westin, L., Moström, J., & Eliasson, J. (2008). Transitioning to OOP--A Never Ending Story. In *Reflections on the Teaching of Programming*, Lecture Notes in Computer Science (pp. 80-97). Springer, Berlin.

Börstler, J., & Sperber, M. (2008). Systematisches Programmieren in der Anfängerausbildung. *informatica didactica*, *8*. Retrieved from http://www.informatica-didactica.de/cmsmadesimple/index.php?page=Boerstler2010

Caspersen, M. E., & Kölling, M. (2009). STREAM: A First Programming Process. *The Transactions of Computing Education*.

Crestani, M., & Sperber, M. (2010). Experience report: Growing Programming Languages for Beginning Students. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming - ICFP '10* (p. 229). Presented at the the 15th ACM SIGPLAN international conference, Baltimore, Maryland, USA. doi:10.1145/1863543.1863576

Ericsson, K. A., Krampe, R. T., & Tesch-Romer, C. (1993). The Role of Deliberate Practice in the Acquisition of Expert Performance. *Psychological Review*, *100*(3), 363-406.

Felleisen, M., Findler, R. B., Flatt, M., & Krishnamurthi, S. (2003). *How to Design Programs*. MIT Press.

Findler, R. B., Clements, J., Cormac Flanagan, Flatt, M., Krishnamurthi, S., Steckler, P. A., & Felleisen, M. (2002). DrScheme: A Programming Environment for Scheme. *Journal of Functional Programming*, *March*, 159-182.

Klaeren, H., & Sperber, M. (2007). *Die Macht der Abstraktion: Einführung in die Programmierung* (1st ed.). Vieweg+Teubner.

Sperber, M., Dybvig, R. K., Flatt, M., & van Straaten, A. (Eds.). (2009). *Revised[6] Report on the Algorithmic Language Scheme*. Cambridge University Press.