Form over Function

Teaching Beginners How to Construct Programs (Distilled Tutorial)

Michael Sperber
Active Group GmbH
michael.sperber@active-group.de

Marcus Crestani
University of Tübingen
crestani@informatik.uni-tuebingen.de

Abstract

Teaching beginners how to program is hard: As knowledge about systematic construction of programs is quite young, knowledge about the didactics of the discipline is even younger and correspondingly incomplete. Developing and refining an introductory-programming course for more than a decade, we have learned that designing a successful course is a comprehensive activity and teachers should consider and question all aspects of a course. We doubt reports of sweeping successes in introductory-programming classes by the use of just one single didactic device—including claims that "switching to Scheme" magically turns a bad course into a good one. Of course, the choice of individual devices (including the use of Scheme) does matter, but for teaching an effective course the whole package counts. This paper describes the basic ideas and insights that have driven the development of our introductory course. In particular, a number of conclusions about effective teaching were not as we had originally expected.

Categories and Subject Descriptors D.2.10 [*Software Engineering*]: Design—Methodologies; K.3.2 [*Computers and Education*]: Computer and Information Science Education—Computer Science Education

General Terms Design, Languages

Keywords Introductory Programming

1. Introduction

In this tutorial, we distill the basic ideas and insights that have driven the development of our introductory course, which we started in 1999 at the University of Tübingen and has since been adopted by the Universities of Freiburg and Kiel.

Teaching the introductory programming course starts with a clear idea of what we want to teach: We believe that much of programming should be done *systematically*, with a clearly defined path from problem to program. Matthias Felleisen's group pioneered the *Program by Design* approach [7], and we have been following in their footsteps [9]. Everything we do flows from our desire to enable students to construct programs systematically. However, this has been exceedingly difficult: The very idea of programming systematically is controversial among practitioners as well as educators, and the idea of doing *anything* systematically is anathema to many of our students as they begin the course.

Teaching the course, we have tried to observe and measure the effectiveness of our teaching. This resulted in many depressing conclusions, unexpected insights and startling successes. In particular, a number of conclusions about effective teaching were not as we had originally expected. In no particular order:

- Continual evaluation of teaching success is important.
- Personal supervision is important.
- Traditional teaching evaluations are almost worthless.
- Form matters more than working programs.
- Our students are not like us.
- Cheating is a significant problem.
- Grade pressure works.
- The requirements for a teaching language are different than the requirements for a production language.
- Students have to like neither us nor Scheme to learn successfully.
- Telling students that what they are learning is worthwhile does not matter.

One theme has emerged from these disparate observations: Most teachers, when evaluating their students' performance on homework and exams focus on the *correctness of function*: A student receives credit when a program written as an answer to a problem statement fulfills its purpose correctly. In our experience, this is not enough: For many students, this approach fails at enabling them to write functioning code, and, even when they do, that code is often atrocious in structure. One of the most visible and, to visitors, surprising aspects of our course is that we insist that students follow form. We do this not just for the code produced by the students, but also for the learning habits we try to impress on our students. We have found that moving from the previous, "function-oriented" approach to "form-oriented" has improved the effectiveness of our teaching significantly. In this paper, we describe the most important aspects of our course.

Overview The paper is structured as follows: Section 2 introduces the program-by-design approach we use as the basis for our course. Section 3 discusses programming-language issues, and how Scheme fits in the picture. Section 4 gives an overview over the design recipes. Section 5 describes a sample teaching session, followed by notes explaining some of the salient aspects of that

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACM SIGPLAN Workshop on Scheme and Functional Programming September 9, 2012, Copenhagen, Denmark

Copyright © 2012 ACM [to be supplied]. . . $10.00

session in section 6. Section 7 describes the important factors in making the students succeed at learning the material. Section 8 describes some of the organizational measures that we use to help in this endeavor. Section 9 has some notes on follow-up courses. Section 10 contains a summary.

2. Program by Design

At the heart of our course is the idea that students (or anyone, for that matter) can program *systematically*: following a known, explicit system of instructions on how to convert a problem statement into a program that implements the solution. The *Program by Design* project (`http://www.programbydesign.org`) calls this system *design recipes*. Programming systematically may seem like an obvious aspect of a programming course. However, many professional programs written by graduates of a programming course have just as obviously not been written systematically. In reality, teaching systematic programming is much harder than it seems at first because most programmers use intuition and experience to guide their problem-solving strategy, and often take shortcuts in their heads before writing code down—and so do most teachers. Actually verbalizing intuition and experience to the point that the result serves as an explicit, teachable system is significant work, pioneered in introductory programming by the book *How to Design Programs* (*HtDP*) [7].

In HtDP (and our own *DMdA* [9]), the development of *every single program* follows from the design recipes. This is a quality of these books that is not immediately apparent to the casual reader, to whom they may seem merely pedantic and boring: They repeat the application of the same set of design recipes over and over, without taking into account "obvious" shortcuts or optimizations. However, it is precisely this quality that makes these books appropriate for their target audience—the students—rather than the teachers.

The pedantic nature of HtDP/DMdA is a testament to one of the fundamental problems of the teacher—the *curse of knowledge* [10]: The teacher looks at textbooks through the eyes of an experienced computer scientist, and is likely to prefer a book with clever ideas and daring examples—for example, one with brilliant algorithms, clever abstractions and data structures with unexpected properties. These are wonderful examples for the ingenuity of clever computer scientists of the past. Unfortunately, by their very nature, these examples were not developed systematically, and are thus not systematically teachable. Forcing them upon beginning students does not help them develop programming prowess of their own.[1]

Consequently, we follow the design recipes slavishly when doing "live coding" in classes, despite the strong temptation to be "clever" and take shortcuts. Even knowing what the process is in principle, it is easy for teachers to underestimate the rigor involved in this procedure.

3. Programming Languages for Beginners

The programming language used in an introductory course is a means to an end. In our case, it serves two needs:

- as an expository device for the design recipes, and
- as a vehicle for directed practice.

Fulfilling these needs is, unfortunately, not a matter of merely choosing the right language. In particular, standard Scheme fulfills none of the above requirements: Not every element of the design recipes has a corresponding language construct, and the full generality of Scheme syntax is difficult to use for beginners, at least in the conventional Scheme IDEs and editors made for professional programmers.

Just creating a new language to meet the needs of the course is tempting, especially to people in the programming language community. However, there is again the real danger of the curse of knowledge: A language that is attractive to the teacher is not necessarily an appropriate language for the student. The key, therefore, is not so much to design a language but to *evolve* it, continually observing how students fare when they use it, and making improvements based on those observations.

This is where the Racket system and its Scheme heritage come in: Scheme allows us to add new procedures and syntactic forms that appear just like the existing ones, as well as "removing" existing forms, essentially by developing library modules. This is at least an order of magnitude easier than making the corresponding changes in a language that lacks Scheme's flexibility.

Scheme and Racket programmers, of course, have always known this when they write programs professionally. In fact, this is also a crucial quality when evolving a programming system for beginners: As we were implementing our teaching languages (replicating the experience of PLT [3]), we were able to ship updated versions as libraries to the students on a weekly basis, always incorporating improvements that came from insights we had gained by looking over students' shoulders. (To this day, we update the languages based on feedback, but at a slower pace.) So Scheme/Racket allowed us to innovate on language design without being hampered by an underlying fixed language implementation or standard.

The second crucial factor in the development of the teaching languages is the programming environment: Emacs or Eclipse are hardly appropriate tools for beginners. DrRacket (formerly DrScheme) [6] not only provides a vastly simpler user interface, it also allows us to tailor the user experience of beginners in various ways. This includes improved feedback for errors, and graphical tools for visualizing program execution and testing results.

The result of this development—at this point—is a set of language levels that are recognizably Scheme dialects, but also differ from Scheme in various ways: The language levels impose various restrictions that help beginners avoid common mistakes. They also add language constructs—specifically record-type definitions, signatures and test cases—that correspond to elements of the design recipes.

Note that the syntax of our languages is still parenthesis-based, maybe the most obvious invitation of criticism. As Scheme programmers, we know how important the syntax is to us, but, again this does not necessarily mean the syntax is appropriate for beginners. Our experience has shown, however, that the parenthesis-based syntax works just fine for beginners. (We even eschew the square brackets that HtDP uses to differentiate certain constructs.)

Another controversial choice is the omission of static typing, which may be surprising as the design recipes are essentially a types-based method of program construction. Type errors are often hard to understand for beginners, especially as they provide no examples for concrete evaluations that may go wrong. Moreover, static type systems reject some perfectly correct and systematically written programs, and explaining the artifacts of the type system that are responsible would detract from the main purpose of the course. Instead, we use *signatures* [3], which fulfill a similar role to types: They allow students to state the shape of data, and are checked automatically, albeit at run time.[2]

[1] A case in point is the famous *Structure and Interpretation of Computer Programs* (or *SICP*) [1], one of the greatest computer-science books of all time. However, SICP introduces programming mostly through a free-wheeling set of brilliant examples without explaining methodology, and does little to enable beginning students to construct programs similar to those examples.

[2] A future experiment we are planning is to add an advanced language level that interprets signature declarations as type declarations.

4. Using the Design Recipes

A typical lecture of our beginner's course features live coding. We present a problem and we show our students how to systematically solve the problem, *as if they were doing it themselves*. We either use the blackboard or a computer for live coding. (In the teaching language, substantial programs fit on a blackboard or a computer screen.)

We teach systematic programming by using a data- and test-driven top-down design, formalized in design recipes. The universal design recipe for writing a procedure is the following:

> Construct a procedure in the following order:
>
> **short description** write a one-line description
>
> **data analysis** analyze the involved data and determine the sorts of data
>
> **signature** choose a name for the procedure and write the signature
>
> **test cases** write a number of test cases
>
> **skeleton** derive the skeleton of the procedure from the signature
>
> **template** derive the template from the signature and the data analysis by applying the corresponding design recipes
>
> **body** complete the body of the procedure
>
> **test** make sure your test cases run successfully

First, we focus on the actual problem by writing a short description.

Then, we identify and define the data. Our students learn to document and build the implementation of the data by using specific design recipes. We teach our students the following standard categories of data:

primitive data like numbers, strings, or booleans

composite data that consists of several components

mixed data that could be one of several kinds of data

For each case, there is a specific design recipe that leads us from the informal data definition explicitly to a working representation in code. When the data structures are implemented, we write down a *signature* for the procedure that establishes the procedure's name and the kinds of input and output data. The next step is writing down concrete examples for the implemented data structures and test cases for the procedure.

Then, we start the implementation of the procedure via what we call the *skeleton*, which is a straightforward translation of the procedure's signature. We implement the procedure by using the data analysis. For every kind of data, we provide a design recipe that contributes a code *template* to the procedure depending on the sort of data. The recipes make the construction of a procedure's body a systematic effort that is driven by the data.

The skeleton and the various templates are snippets of code with placeholders. (We uses ellipses `....`) A program with ellipses is an intermediate result, the placeholders indicate what we need to do next. To make the construction of a program traceable, it is important to have each step produce a specific piece of the final result.

The design recipes separate the systematic aspects of problem solving from domain-specific or "creative" aspects of programming. Once the systematic aspects are in place, the creative aspects become manageable. In our approach, once there is no more applicable design recipe, the procedure contains ellipses that we need to fill. Only then, we need to use a deeper understanding of the problem and domain-specific knowledge to fill in the missing bits. (This is often shockingly easy.)

When we teach our students, we always stick to the systematic approach. We never combine several "trivial" steps in a single one. Even though we are often tempted to instantly replace some ellipses on the fly with the "obvious" solution, we avoid doing so. It is crucial to explicitly write down the intermediate results to visualize where we are in our problem solving process.

5. Sample Teaching Session

In this section we give an example problem and show how we solve the problem by following the design recipes. We do this in some detail, to illustrate the steps a teacher would go through when explaining how to solve the problem. We use a well-worn problem statement originally formulated by Matthias Felleisen:

> A geometric shape in the two-dimensional plane is one of the following:
>
> - a circle
> - a square (parallel to the axes)
> - an overlay of two geometric shapes
>
> Implement geometric shapes! Write a procedure that checks whether a given point is inside or outside a geometric shape!

We solve the problem with the language level *Die Macht der Abstraktion* [3] that comes with DrRacket.

We start with the data analysis. The first step of the data analysis is to collect all the different sorts of data. The problem statements yields the following:

- shapes
- circles
- squares
- overlays
- points

(Arguably, there is also the two-dimensional plane, but it—like air—does not require explicit representation.)

The next step is to consider the structure of each sort. The wording "one of the following" in the problem statement identifies geometric shapes as mixed data. Here is the design recipe for mixed data, which we either put on a projector or have handed out to the students on paper:

> When your data analysis contains mixed data, write a data definition starting with the following:
>
> ```
> ; An x is one of the following:
> ; - sort₁ (sig₁)
> ; ...
> ; - sortₙ (sig₂)
> ```
>
> This leads to the following signature definition:
>
> ```
> (define sort
>   (signature
>     (mixed sort₁ ... sortₙ)))
> ```

As a prelude, we ask students to identify *how many* different sorts comprise a mixed-data definition, and to let that guide the process of filling out the above template. This leads to the data definition:

```
; A geometric shape is one of the following:
```

```
; - a circle (circle)
; - a square parallel to the axes (square)
; - an overlay of two geometric shapes (overlay)
```

Writing the code for shapes along this recipe is straightforward:

```
(define shape
  (signature
    (mixed circle square overlay)))
```

The forms `signature` and `mixed` are part of our language: `shape` is defined to be a signature for a mixed data type. We use `circle`, `square`, and `overlay` as signatures representing circles, squares, and overlays that have yet to be defined.

On with the data analysis: The next sort in the problem statement is circles. We need to decide what kind of data would represent a circle—several choices are possible and valid, which we can discuss with the students. Here, we choose to describe a circle through its center and radius. Thus, a circle has two attributes or parts—*compound data*. We refer to the design recipe for compound data, which goes like this:

> When your data analysis contains compound data, identify the signatures of the components. Then write a data definition starting with the following:
>
> ```
> ; An x consists of / has:
> ; - field_1 (sig_1)
> ; ...
> ; - field_n (sig_2)
> ```
>
> Then translate the data definition into a record definition:
>
> ```
> (define-record-procedures sig
>   constr pred?
>   (select_1 ... select_n)))
> ```
>
> Write a constructor signature of the following form:
>
> ```
> (: constr (sig_1 ... sig_n -> sig))
> ```
>
> Also, write signatures for the predicate and the selectors:
>
> ```
> (: pred? (any -> boolean))
> (: select_1 (sig -> sig_1))
> ...
> (: select_n (sig -> sig_n))
> ```

As with the mixed data, we informally ask students to start filling in this template by considering *how many* components the compound data has. So, for the circle we ask *How many components does a circle have?* We conduct a brief show of hands, possibly asking students who show something other than two to explain their reasons. Once we agree on "two," we can write down the data definition.

```
; A circle consists of:
; - center (point)
; - radius (real)
```

The data definition translates straightforwardly into a record definition and signatures for the procedures defined by it:

```
(define-record-procedures circle
  make-circle circle?
  (circle-center circle-radius))
(: make-circle (point real -> circle))
(: circle? (any -> boolean))
(: circle-center (circle -> point))
(: circle-radius (circle -> real))
```

The form `define-record-procedures` defines the signature `circle` and the procedures needed for manipulating records: The `make-circle` procedure constructs a new circle, `circle?` is a predicate to distinguish circles from other sorts, and the selectors `circle-center` and `circle-radius` access the components of a circle.

The form `:` defines the signatures for the procedures, the arrow `->` separates the sorts of input parameters from the sort of the output parameter. The notation is hopefully easy enough to understand. Some signatures are predefined: `any` for arbitrary values, `boolean` for booleans, and `real` for real numbers. The signature `point` that represents points in the plane has yet to be defined. (All signatures but the one for `make-signature` are redundant—the next section describes the rationale.)

The next piece of data on our list is squares. Again, the representation of a square is not detailed in the problem statement, we therefore need to discuss and choose how to represent a square as data. Here, we choose to represent a square with a lower left corner and a size. Again, we ask *How many components?* Again, we may have a discussion on the correct answer. Once we agree on "two", we can fill out the template, yielding this data definition:

```
; A square consists of:
; - corner (point)
; - size (real)
```

Then we can translate the data definition into a record definition and signatures:

```
(define-record-procedures square
  make-square square?
  (square-corner square-size))
(: make-square (point real -> square))
(: square? (any -> boolean))
(: square-corner (square -> point))
(: square-size (square -> real))
```

Somewhat arbitrarily, we choose to handle points next. Since the shapes are on a two-dimensional plane, it is natural to use a cartesian representation with an x and a y coordinate. We get this:

```
; A point consists of:
; - x coordinate (real)
; - y coordinate (real)
(define-record-procedures point
  make-point point?
  (point-x point-y))
(: make-point (real real -> point))
(: point? (any -> boolean))
(: point-x (point -> real))
(: point-y (point -> real))
```

Next on our list are overlays. According to the problem statement an overlay consists of two shapes. Therefore, it is compound data, resulting in this data definition:

```
; A overlay consists of:
; - top (shape)
; - bottom (shape)
```

Again, we can translate the data definition into code:

```
(define-record-procedures overlay
  make-overlay overlay?
  (overlay-top overlay-bottom))
(: make-overlay (shape shape -> overlay))
```

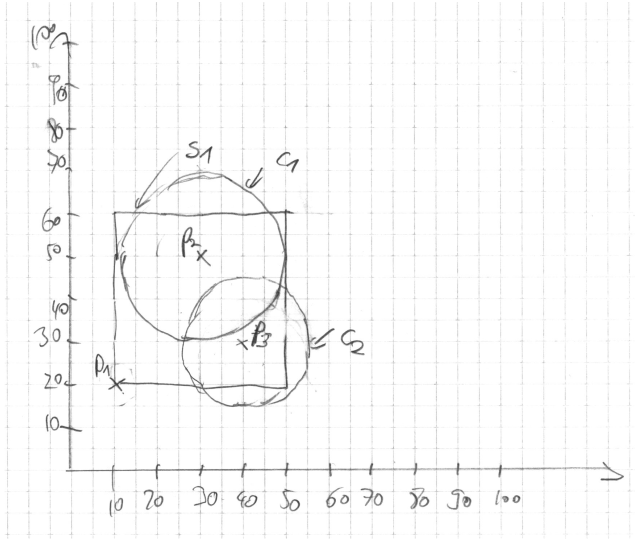

Figure 1. Shape examples

```
(: overlay? (any -> boolean))
(: overlay-top (overlay -> shape))
(: overlay-bottom (overlay -> shape))
```

We note that overlays contain two *self-references*: An overlay contains two shapes, which can themselves be overlays.

This concludes the first part of our solution—data analysis and data definitions. Our recipes led from informal data definitions to code that implements the data.

Next, we write a number of examples for our defined data types that we can use in our test cases. Since we are dealing with geometry, it is helpful to draw a picture. A doodle is sufficient, figure 1 shows an example.

Using the doodle, we write code that creates the corresponding objects. To make the connection between information and code apparent, the examples contain comments that describe the represented information.

```
; point at x=10, y=20
(define p1 (make-point 10 20))
; point at x=30, y=50
(define p2 (make-point 30 50))
; point at x=40, y=30
(define p3 (make-point 40 30))
; square with corner at point p1, size 40
(define s1 (make-square p1 40))
; circle around point p2, radius 20
(define c1 (make-circle p2 20))
; overlay of circle c1 and square s1
(define o1 (make-overlay c1 s1))
; Circle around point p3, radius 15
(define c2 (make-circle p3 15))
; Overlay of overlay o1 and circle c2
(define o2 (make-overlay o1 c2))
```

Now that we have defined the data, we need to write the procedure that checks whether a given point is inside or outside a geometric shape. The universal design recipe for procedures, detailed in the previous section, leads the way:

First, we write a one-line description—*exactly* one line:

```
; Is a point within a shape?
```

Then, we choose a name for the procedure and we write the signature. A signature establishes the procedure's name and the kinds of input and output data. From the problem statement it is apparent that the procedure accepts a point and a shape and returns a boolean, therefore the signature is this:

```
(: point-in-shape? (point shape -> boolean))
```

Next up are *test cases*, for which we can use the examples. (We insist on doing the tests first.) The teaching language contains the form `check-expect` that accepts two operands that we expect to be equal. If they are not equal, the test cases fail.

```
(check-expect
 (point-in-shape? p2 c1) #t)
(check-expect
 (point-in-shape? p3 c2) #t)
(check-expect
 (point-in-shape? (make-point 51 50) c1) #f)
(check-expect
 (point-in-shape? (make-point 11 21) s1) #t)
(check-expect
 (point-in-shape? (make-point 49 59) s1) #t)
(check-expect
 (point-in-shape? (make-point 9 21) s1) #f)
(check-expect
 (point-in-shape? (make-point 11 19) s1) #f)
(check-expect
 (point-in-shape? (make-point 51 59) s1) #f)
(check-expect
 (point-in-shape? (make-point 49 61) s1) #f)
(check-expect
 (point-in-shape? (make-point 40 30) o2) #t)
(check-expect
 (point-in-shape? (make-point 0 0) o2) #f)
(check-expect
 (point-in-shape? (make-point 30 65) o2) #t)
(check-expect
 (point-in-shape? (make-point 40 17) o2) #t)
```

Finally, we start with the actual procedure definition by writing down the *skeleton*. The skeleton derives directly from the signature; we make use of the procedure name and the number of arguments. The only addition are the names for the arguments:

```
(define point-in-shape?
  (lambda (p s)
    ...))
```

Note that this is exactly the way we present the skeleton. Specifically, we *write down the ellipsis*. (And we expect the students to do likewise.)

Now we begin constructing the body of the procedure by repeatedly adding relevant *templates*. First, we expect to use the arguments in its body:

```
(define point-in-shape?
  (lambda (p s)
    ... p ... s ...))
```

Again, we write down the ellipses, clearly noting that the procedure is incomplete, and where we need to add more code.

Since `p` is compound data, the procedure likely needs the components of the point. There is a design recipe that covers the occurrence of compound data:

> When your data analysis finds compound data, add `(`*`select`* *`x`*`)` for each field of the compound data to the procedure, where *`select`* is the selector of a field and *`x`* is the name of the procedure's parameter.

We therefore add selector applications for the point's fields to the template:

```
(define point-in-shape?
  (lambda (p s)
    ... p ... s ...
    ... (point-x p) ... (point-y p) ...))
```

Since s is mixed data, the procedure needs to distinguish the various sorts of shapes. The design recipe for mixed data provides a template:

> If the predicates for the sorts are called *pred?*$_1$ ... *pred?*$_n$, a procedure that accepts mixed data as input has the form:
>
> ```
> (: proc (sig -> ...))
> (define proc
>   (lambda (a)
>     (cond
>      ((pred?_1 a) ...)
>      ...
>      ((pred?_n a) ...))))
> ```
>
> The right-hand sides are completed according to the design recipes of the particular sorts.[3]

When applying the design recipe, we refer back to the data definition for shapes. We ask: *How many different sorts comprise shapes?* When we agree on "three", we start typing out the cond form with three branches, only then filling out the predicate applications. This yields:

```
(define point-in-shape?
  (lambda (p s)
    ... p ... s ...
    ... (point-x p) ... (point-y p) ...)
    (cond
      ((circle? s) ...)
      ((square? s) ...)
      ((overlay? s) ...)))
```

Next, we complete the right-hand sides for the different branches of the cond statement. In each case—circle, square, overlay—we have to deal with compound data, therefore the design recipes for compound data suggests that we may need to access the compound data's fields. We add selector applications:

```
(define point-in-shape?
  (lambda (p s)
    ... p ... s ...
    ... (point-x p) ... (point-y p) ...)
    (cond
      ((circle? s) ... (circle-center s)
                   ... (circle-radius s) ...)
      ((square? s) ... (square-corner s)
                   ... (square-size s) ...)
      ((overlay? s) ... (overlay-top s)
                    ... (overlay-bottom s) ...)))
```

Finally we make use of the fact that overlays contain self-references: overlay-top and overlay-bottom yield shapes. Of course, there is a design recipe for self-referential data:

> When your data analysis finds self-referential data, wrap it with a recursive call.

Therefore, we recursively call point-in-shape? on the components of the overlay:

```
(define point-in-shape?
  (lambda (p s)
    ... p ... s ...
    ... (point-x p) ... (point-y p) ...)
    (cond
      ((circle? s) ... (circle-center s)
                   ... (circle-radius s) ...)
      ((square? s) ... (square-corner s)
                   ... (square-size s) ...)
      ((overlay? s)
       ... (point-in-shape? (overlay-top s))
       ... (point-in-shape? (overlay-bottom s))
       ...)))
```

Now we have all the elements of the program in place that we were able to systematically derive from the procedure's input and output data.

Only now we need domain-specific knowledge to combine the fragments in a way that the procedure does what we expect, i.e. that the previously written test cases work. We tackle the branches of the cond statement one at a time:

- For circles, we already have the radius and the center in the skeleton. The student needs the insight that a point is within a circle if the distance between the point and the center is smaller than the radius. The distance is an *intermediate result* that deserves its own procedure:

  ```
  ; Calculate the distance between two points
  (: distance (point point -> real))
  ```

  For now, we pretend that we have already written the procedure (using a design recipe called *wishful thinking*) and we use it in the case for circles:

  ```
  (cond
    ((circle? s)
     (<= (distance p (circle-center s))
         (circle-radius s)))
  ```

- For squares, the coordinates of the point have to be compared to the coordinates of the square's corner. All these components are already present in the skeleton. The connection is not particularly hard:

  ```
  (cond
    ...
    ((square? s)
     (and (>= (point-x p)
              (point-x (square-corner s)))
          (<= (point-x p)
              (+ (point-x (square-corner s))
                 (square-size s)))
          (>= (point-y p)
              (point-y (square-corner s)))
          (<= (point-y p)
              (+ (point-y (square-corner s))
                 (square-size s)))))
  ```

[3] Usually, we expect to cover all possible cases with explicit tests, rather than have the last one be a fall-through else clause: This makes students aware what data they actually need to handle in a branch. We do teach how else works, but most examples that contain an "else" are binary ifs.

(In class, this part of the template is taught using more than one step.)

- For overlays, we already have two recursive calls that solve the problem for the top and bottom shape separately. We just have to connect both answers to solve the problem for the overlay itself. The logical `or` does this:

```
(cond
  ...
  ((overlay? s)
   (or (point-in-shape? p (overlay-top s))
       (point-in-shape? p (overlay-bottom s))))
```

Now the body of our procedure is complete. We clean up the remaining ellipses and fragments that we did not need. The completed procedure looks like this:

```
(define point-in-shape?
  (lambda (p s)
    (cond
      ((circle? s)
       (<= (distance p (circle-center s))
           (circle-radius s)))
      ((square? s)
       (and (>= (point-x p)
                (point-x (square-corner s)))
            (<= (point-x p)
                (+ (point-x (square-corner s))
                   (square-size s)))
            (>= (point-y p)
                (point-y (square-corner s)))
            (<= (point-y p)
                (+ (point-y (square-corner s))
                   (square-size s)))))
      ((overlay? s)
       (or (point-in-shape? p (overlay-top s))
           (point-in-shape? p (overlay-bottom s))))
```

Before we are able to finally run our program and make sure that all our test cases work, we need to write the procedure `distance` that we have used to write the code for circles. We leave this (ellipses and all) as an exercise to the reader.

6. Notes on Teaching

The previous section may seem excruciatingly pedantic to the reader. (It was painful to write.) Yet, being pedantic and tedious about this is precisely the point: We follow the design recipes to at least this level of detail over and over—with every single example that we present. This—and only this—qualifies us to ask the same of the students. Once we become sloppy and take shortcuts, the students feel licensed to do the same—with the crucial difference that they usually fail.

In particular, we insist that the students perform each intermediate step of the program separately. (Some homework exercises only ask for certain steps of the sequence to drive home that point.) Writing down the ellipses during live coding makes this quite explicit.

The informal data definitions are redundant with the code. However, we discovered that the redundancy helps the students practice the connection between information and the data that represents it.

We also insist, for record definitions, on the full set of signatures, even though all but the constructor signature are redundant. This was not always the case: When we originally introduced signatures, we only wrote down the constructor signature. However, the connection between the constructor's signature and the other procedure's signatures was not initially obvious to our students. *The students* asked us to write down all other signatures, and make them a requirement of the design recipe. It becomes more obvious to a beginner how to use the predicate or the selectors when they explicitly have written down their signatures and they are able to look them up later. We were happy to follow our students' suggestion.

Students make surprising mistakes considering the number of components of compound data and the number of cases for mixed data. During live coding, we always have the students determine the count of components, and we may ask the students to assist with other steps of the design recipes. This interaction stresses the importance and makes them aware of possible difficulties and lets them carefully reason about this step.

The order of the steps the universal design recipe for procedures mandates aims to help our students to fill the missing gaps after there is no more applicable design recipe. First, the format of the short description makes them focus on what precisely the procedure is supposed to do. Then, writing test cases before implementing the procedure forces students to actually think about what constitutes a correct solution of the problem. The insight they gain in this step is the problem-specific knowledge that translates directly to how they are supposed to connect the code templates in the procedure's body.

Self-references and recursion is a topic in a beginner's course that is considered difficult, and therefore hard to teach and even harder to understand by the students. With the design recipes, it comes naturally. In the example problem we implemented a self-referential data structure and a recursive procedure in passing, thanks to the design recipes: An overlay contains two shapes and in turn is a shape itself. The design recipe handles this by having us wrap every self-referential data with a recursive procedure call.

7. Learning Habits

The classroom exposition described earlier is the first step towards teaching systematic programming. As every teacher knows, however, teaching does not equal learning. Just as we take a principled approach to presentation—refined by experience—we organize the rest of the course to fulfill the promise of the material.

Two insights on the nature of effective learning have shaped the organization of our course. Foremost is the role of *practice* in learning. To become good at any sufficiently complex activity, learners need to engage in deliberate, directed practice [5]. The "directed" part cannot be overemphasized: In order to improve their competence, students must engage in practice activities with the specific goal of getting better. Effective practice also includes:

- making an effort to achieve good technical proficiency (not just valid results),
- setting specific goals, and
- seeking and using feedback (which, to be effective, must be prompt and precise).

If students do not engage in directed practice, even excellent material presented in a clear and understandable fashion will not make a significant impact on the students' competence.

The DrRacket environment, the design-recipe approach to teaching [7, 9], and the HtDP/DMdA teaching languages [3] all support deliberate practice: The design recipes establish a standard of technical proficiency; DrRacket provides excellent feedback; programs written in the Scheme-based teaching-languages are small compared to, say, Java programs, which enables the students to practice more per time unit.

However, the software and excellent material are necessary, but not sufficient prerequisites for enabling directed practice. Early

iterations of our course were not as successful as we would have liked at getting students to engage in deliberate practice, despite the use of DrRacket and the design recipes as well as vast efforts at explaining the benefits of practice, trying to provide positive motivation and establishing mutual trust [2].

One possible explanation for this problem is the mindset that students use in approaching learning activities, in particular whether competence is due to fixed, innate talent, or a process of learning and deliberate practice [4]. While we did not conduct any study on the prevalence of one or the other mindset, our students at times exhibited a startling distrust in their abilities to improve their competence over time. ("It's very frustrating to see that other students finish the lab exercises sooner than I do.")

In other words: Many students believe they *cannot* master the material, *ever*, because their talent is not sufficient. From this, students often conclude that practice opportunities provided as part of the course are useless. At that point, a student faces the options of either dropping out or trying to obtain credit through plagiarism. Plagiarism is a huge problem: Not only does it inhibit learning, it poisons any evaluation results a teacher may obtain about the effectiveness of the course.

Thus, our efforts go towards providing opportunities and incentives that encourage students to engage in effective learning habits. Clearly, we cannot assume that the students possess those habits coming out of high school. The biggest challenge is breaking through the "fixed mindset," the belief a student has that programming abilities are either innate or cannot be acquired. To do this, we try to provide all students with a successful learning experience early on in the course. This, in turn, we do with homework or lab exercises that we somehow try to "get" the students to complete successfully.

This "getting" is exceedingly difficult with students that have a fixed mindset, which comes with low tolerance for frustration. Those students give up on exercises at the slightest hint of trouble—or often just at the belief that, surely, there will be trouble at some point. To break through this mindset, we tried means traditionally associated with effective teaching—positive motivation, enthusiastic presentation, explaining the relevance of what we were doing—with little success. These students just do not believe what we know about effective learning: they are different from us.

As a result, we have since adopted a somewhat contrarian approach: We try to *make* the students follow effective learning habits, through personal supervision and a rigid set of rules that award credit only for practicing those habits. Conversely, we try to *make* the students succeed by providing help at every step—through personal supervision, the teaching assistants providing various help channels, a discussion forum, wearing dress shirts and suits instead of Mickey-Mouse t-shirts, and generally by being obliging, formal, and professional when dealing with our students. We adjust those measures with each iteration of the course, partly because of changes in the bureaucratic framework at the University, partly because we keep looking for new ways of encouraging effective habits. These measures have improved the effectiveness of the course measurably [2].

8. Organizational Measures

This section summarizes an assortment of organizational measures we have used and found effective to encourage practice and self-improvement:

Fight plagiarism We make the students aware of plagiarism and the consequences (we use a two-strikes policy, where the first strike already discards some homework credit) by having them sign a form stating that they would not plagiarize. We dedicate one of our teaching assistants to the sole task of detecting plagiarism.

Placement test In the first lecture we give our students a placement test with 10th-grade-level math problems. Since we do not announce the placement test, the students are not prepared. The results of the test are not part of the final grade. Instead, we use the results to get a general idea of their math skills, to assemble heterogeneous study groups, and to be able to correlate the results of the final exam with the students' abilities at the beginning of the class.

Weekly exercises We publish weekly homework exercises that the students have to solve within one week. The sheets are graded by our teaching assistants and discussed in weekly exercise sessions guided by our teaching assistants.

Assisted programming Students have to solve programming exercises under the supervision of teaching assistants in our computer lab. The supervisors ensure that the students use the design recipes when programming and they help over the humps that inevitably occur. The students use a restricted login environment that does not allow anything else than working on their programming problems.

Mandatory exercises Two of the weekly exercises, one towards the beginning and one towards the end of a course, are mandatory exercises. Each student has to individually present the solution to the teaching assistant. Mandatory exercises do not contribute to the final score; a student needs to pass each mandatory exercise to be able to pass the course.

Group exercises Twice during the semester we partition our students into heterogeneous groups of four, using the results of the placement test. Each group has to solve a larger problem. Only one of the team members gets to present the result; the grade of that one person becomes the grade of everyone in the group. (When bureaucratically possible, we make these exercises mandatory—a group needs to pass for its members to pass the course.)

Exams and grades At the end of the class, there is a final exam. The grade of the exam and the grade of all the exercises throughout the semester determine the final grade that is relevant for the degree.

Teaching assistants Our teaching assistants are extremely important to the success of the course: In addition to holding tutorial sessions and providing help to the students, they supervise the students in the assisted-programming sessions, and grade the various exercises throughout the semester. We hold a "training boot camp" (typically two days) for our teaching assistants before the course starts and we meet them each week for ohe hour. We also provide them with grading instructions and we distribute additional material that they can use in their exercise sessions.

In-class quizzes During many lecture sessions, we ask our students a few simple quiz questions, giving them five minutes or so to discuss the solution with their neighbors. We then ask the students who have come to conclusions different from those of their neighbors to identify themselves, and we use this to start short discussions.

These measures are labor-intensive and expensive. Moreover, writing and debugging good exercises takes time. We feel they are well-invested, however, given how they make the difference between success and failure for a significant proportion of our students.

9. Beyond the Intro Course

In Tübingen, the intro course that is covered by this tutorial is followed by a second-semester course on object-oriented programming based on PLT's *How to Design Classes* [8], using Java. *How to Design Classes* is a sequel to HtDP, so the transition is quite seamless.

Using Java in the second-semester course does come with a number of problems though: While the language does in principle support the design recipes, its verbosity and complexity often get in the way of concise examples. Moreover, the lack of proper tail recursion means that the programmer has to use Java's poorly designed loop constructs `while` and `for`. Using these loop constructs in turn requires imperative programming using accumulators, which is significantly more difficult than structural recursion.

Another problem with the use of Java is that the lecturers of follow-up courses invariably assume we have taught a "Java course" where more emphasis is put on the ability to use, e.g., `java.util.Hashtable` than on the ability to properly design programs. For example, a consistent feedback we have had over the years is that students use recursion where lecturers expect loops. We have thus expanded our coverage of loops in the course. However, time is at a premium when only two semesters are available for basic training in program design, which creates a mismatch between what lecturers of follow-up courses expect and what we teach.

We see no easy way to resolve this problem: Even if we spent all of our time on a "Java course," it would not be sufficient to cover the entire language and its standard libraries. Our tentative conclusion is to move away from Java for a future course, both to improve the teaching experience, to more clearly define expectations that lecturers of follow-up courses can have, and to offer them a clean slate on which to root Java.

Having said that, feedback from students reporting on their ability to work on the programming exercises of follow-up courses has been uniformly positive.

10. Summary

Here is a summary of the insights and techniques that have helped us improve our course:

Program systematically We write example programs systematically, with a clear path from a problem to a program, using the design recipes, driven by data analysis. Thus, students can trace every single step of the programming process to an explicit instruction. This enables even weak students to successfully write substantial programs.

Avoid eureka moments We carefully and explicitly separate problems that we solve using the design recipes from those that require a eureka-like insight—i.e. clever algorithms or data structures, and we avoid the latter. While most problems can be solved completely by just using the design recipes, actually doing so requires commitment and discipline.

Practice, practice, practice To learn programming successfully, students need as much practice as they can get. While the systematic approach provides a guide as to *what* to practice, we provide significant encouragement and guidance on *how* to practice.

Make students succeed We use every incentive we can think of that is permissible in the bureaucratic and legal environment of a University course to foster directed practice. We try to enforce the use of the design recipes and proper learning habits through a rigid set of rules and credit incentives. Moreover, we support students as they work in supervised lab exercises to ensure they have positive learning experiences.

Language is a means to an end The programming language is only a means to the end of teaching students how to program systematically. (As is everything else.) If we can think of ways to change it that improve the effectiveness of learning, we make the change.

In conclusion, we stress the form of what we do and what we make students do to help them succeed. Merely insisting on correct outcomes is not enough. The form we use for a particular aspect of the course is subject to continual improvement. In particular, the programming language we use is a descendant of Scheme, which we continue to refine. Reassuringly, the core of Scheme has served us well, and its flexible nature allows us to express most of our refinements within the language.

Acknowledgments Listing everyone who has been instrumental in making our course a success—specifically the teaching assistants and students—would by itself fill the space of this paper. In particular, we thank Herbert Klaeren for providing the environment for developing the course, Matthias Felleisen and the members of PLT for the pioneering work that made our course possible, as well as the lecturers who have taught the course: Martin Gasbichler, Eric Knauel, Andreas Schilling, Peter Thiemann, Michael Hanus, Jan-Georg Smaus, and Torsten Grust.

References

[1] H. Abelson, G. J. Sussman, with J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., second edition, 1996.

[2] A. Bieniusa, M. Degen, P. Heidegger, P. Thiemann, S. Wehr, M. Gasbichler, M. Crestani, H. Klaeren, E. Knauel, and M. Sperber. HtDP and DMdA in the battlefield. In F. Huch and A. Parkin, editors, *Functional and Declarative Programming in Education*, Victoria, BC, Canada, Sept. 2008.

[3] M. Crestani and M. Sperber. Growing programming languages for beginning students. In S. Weirich, editor, *Proceedings of the International Conference on Functional Programming 2010*, Baltimore, Maryland, USA, Sept. 2010. ACM Press, New York.

[4] C. Dweck. *Mindset: The New Psychology of Success*. Ballantine Books, Dec. 2007.

[5] K. A. Ericsson, R. T. Krampe, and C. Tesch-Romer. The role of deliberate practice in the acquisition of expert performance. *Psychological Review*, 100(3):363–406, 1993.

[6] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. The DrScheme project: An overview. *SIGPLAN Notices*, 33(6):17–23, June 1998.

[7] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs*. MIT Press, 2001.

[8] M. Felleisen, M. Flatt, R. B. Findler, K. E. Gray, S. Krishnamurthi, and V. K. Proulx. How to Design Classes. `http://www.ccs.neu.edu/home/matthias/htdc.html`, Feb. 2011.

[9] H. Klaeren and M. Sperber. *Die Macht der Abstraktion*. Teubner Verlag, 1st edition, 2007.

[10] C. Wiemann. The "curse of knowledge," or why intuition about teaching often fails. *APS News*, 16(10), 2007. `http://www.aps.org/publications/apsnews/200711/backpage.cfm`.