From Principles to Practice with Class in the First Year

Sam Tobin-Hochstadt   David Van Horn

Northeastern University
Boston, Massachusetts, USA

{samth,dvanhorn}@ccs.neu.edu

We propose a bridge between functional and object-oriented programming in the first-year curriculum. Traditionally, curricula that begin with functional programming transition to a professional, usually object-oriented, language in the second course. This transition poses obstacles for students, and often results in confusing the details of development environments, syntax, and libraries with the fundamentals of OO programming that the course should focus on. Instead, we propose to begin the second course with a sequence of custom teaching languages which minimize the transition from the first course, and allow students to focus on core ideas. After working through the sequence of pedagogical languages, we then transition to Java, at which point students have a strong command of the basic principles. We have 3 years of experience with this course, with notable success.

1   Introduction

Many universities and colleges aim to teach their students proficiency in an industrial object-oriented programming language by the end of the students' first year. The most common approach to achieve this aim is to teach an industrial language in the first course, often Java, starting on the first day. Other curricula take a more indirect route by teaching functional programming in the first semester, followed by a second semester in Java. The latter approach is an improvement over the first, as pointed out by numerous observers [9, 23, 7, 22], but both suffer serious flaws.

As an example, Northeastern University teaches introductory programming in the first semester using *How to Design Programs* [8], followed by object-oriented programming using *How to Design Classes* [11] in the second semester. This sequence was designed to provide a smooth path for incoming students with a competence in high-school level algebra to reach proficiency in Java by the end of their first year [9]. It was a major improvement over the previous Java-first curriculum in terms of student success, attrition, and preparation for subsequent courses [21]. However, significant problems remain; in particular, the second semester course violates the designers' own principles (as recalled in [5]):

1. *introduce only those language constructs that are necessary to teach programming principles*, and

2. *choose a language with as few language constructs as possible, and one in which they can be introduced one at a time*.

The problem is that the first semester ends with an advanced pedagogical functional language and the second semester starts with Java, although it focuses on a side-effect free subset. Despite this focused subset, this transition is too abrupt to meaningfully bridge the gap between functional and object-oriented programming, because several other significant transitions happen in concert:

- from a highly regular and minimal syntax to a complicated irregular syntax,

- from an untyped language to a typed language,

- from a pedagogical programming environment (DrRacket) to a professional programming environment (Eclipse),

- from a language with numeric values corresponding to mathematical objects to a language with numeric values corresponding to common machine representations,[1]

- from a language with image literals and graphical libraries to one in which graphical programming is tedious,

- from an interaction-oriented language and tool suite to a compiled, batch-oriented language.

This abrupt transition has several negative consequences which we have experienced first-hand: the principles of object-oriented programming are obscured and de-emphasized, struggling with the programming environment is frustrating and can cause potentially good students to leave the program, it favors students with prior exposure to the particular tools (a set that is demographically skewed), it inhibits students from experimenting by allowing them to rely upon past skills, and it creates the false impression that courses are discrete units of instruction that can be discarded after successful completion rather than being part of a continuous and cumulative educational experience.

We contribute an alternative approach to the second semester that overcomes these problems and provides a gradual introduction to object-oriented programming. Our approach starts the second semester by introducing *only* the concept of programming with objects, while all other aspects of the course remain where they were left off in the previous semester. This allows other concepts to be introduced at the point at which they are relevant and motivated. Despite this more gradual approach, the course accomplishes the goal of reaching industrial language competence by the end of the semester, covering a super-set of the concepts and topics covered in the *How to Design Classes*-based course.

Outline   The remainder of this paper is organized as follows: section 2 provides background on *How to Design Programs* and the context and constraints involved in the first year at Northeastern. Section 3 describes our approach to the second semester, which starts with a small shift in perspective to bridge the gap between functional programming and object oriented programming. Section 4 describes the path from our pedagogical languages to an industrial object-oriented programming language. Section 5 discusses the relation to existing work and section 6 concludes.

2   Background: the context at Northeastern

At Northeastern, the College of Computer & Information Science (CCIS) requires a four course introductory sequence for Computer Science majors in the first year. The first semester features both a course on discrete mathematics and an introduction to programming following the *How to Design Programs* curriculum. The second semester follows with a course on object-oriented programming and one featuring formal reasoning about programs, both on paper and with the ACL2 theorem prover [17].

After the first year, students take a wide variety of follow-up courses, ranging from a required course in "Object-oriented design" to architecture, operating systems, robotics, and programming languages. No standard language is used in these courses.

More significantly, Northeastern distinctively emphasizes experiential education, with almost all Computer Science majors participating in a 6 month "co-op" internship after their third semester. These co-ops take place at a wide variety of companies, and while most students do some software development, there is no single platform or set of tools that commands majority use. In particular, there is wide

---

[1]While this may seem like a minor point, details of numeric representation can crop up quickly in a Java-based course—for example, 1/3 cannot be represented by any Java numeric types.

variation in the languages students use while on co-op. This combination sets the constraints under which we designed our approach.

2.1 A first course on *How to Design Programs*

In the first semester, students are introduced to the "design recipe", a step-by-step process for going from English descriptions of problems to working programs. The design recipe involves six steps:

1. Analyze the information involved in the problem and express how to represent it as data.

2. Write down a function signature, a summary of the purporse of the function, and a function stub.

3. Illustrate the signature and the purpose statement with some functional examples.

4. Take an inventory of the input data that can be used to compute an answer.

5. Write the code for the function.

6. Verify the behavior of the program against the functional examples given earlier.

Students explore program design using this process in the context of a series of successively richer pedagogical programming language levels [9, 8] that are included in the DrRacket (formerly DrScheme) programming environment [12]. The language and environment include several tools in support of the design recipe. For example, functional examples can be written as executable tests by writting `check-expect` expression [14]; an algebraic stepper and REPL are provided to interact with programs at each step of the design process.

Finally, the first semester course makes extensive use of a library for developing interactive animations and games using functional programming and functional graphics [10, 4].

2.2 The goal of the second course

After the second course, students should both (a) be prepared for subsequent courses in the curriculum, which expect familiarity with Java and standard Java libraries, (b) be prepared for co-ops in which they will use professional-grade languages and tools which will almost certainly be object-oriented. More significantly, we aim to teach the key insights behind the object-oriented approach to program design.

These constraints, while in detail specific to Northeastern and the CCIS curriculum, are broadly similar to the requirements for the first year at many universities. Our course also attends to smaller and more idiosyncratic elements of our curriculum, ranging from formal reasoning to algorithmic analysis, as described in the following sections.

3 A small shift of focus

On the first day of the second semester, we introduce a single linguistic concept to an otherwise unchanged context of the previous semester: the idea of an object. An object is a new kind of value that can, as a first cut, be understood as a pairing together of two familiar concepts: data and functionality.

- An object is like a structure in that it has a fixed number of fields, thus an object (again, like a structure) can represent compound data. But unlike a structure, an object contains not just data, but functionality too;

- An object is like a (set of) function(s) in that it has behavior—it computes; it is not just inert data.

This suggests that objects are a natural fit for well-designed programs since good programs are organized around data definitions and functions that operate over such data. An object, in essence, packages these two things together into a single programming apparatus. This has two important consequences:

1. Students already know how to design programs oriented around objects.

   Since objects are just the combination of two familiar concepts that students already use to design programs, they already know how to design programs around objects, even if they have never heard the term "object" before.

2. Objects enable new kinds of abstraction and composition.

   Although the combination of data and functionality may seem simple, objects enable new forms of abstraction and composition. That is, objects open up new approaches to the construction of computations. By studying these new approaches, we can distill new design principles. Because we understand objects are just the combination of data and functionality, we can understand how all of these principles apply in the familiar context of programming with functions.

3.1   The basics of objects

To begin with, we introduce the notion of a *class definition*, which can be thought of at first as a structure definition in that it defines a new class of compound data. A class is defined using the `define-class` form:

```
(define-class posn (fields x y))
```

This is similar to the `define-struct` form of the first semester, used as follows:

```
(define-struct posn (x y))
```

An *object* is a value that is a member of this class of data, which can be constructed with the `new` keyword, a class name, and the appropriate number of arguments for the fields of the object:

```
(new posn 3 4)
```

An object understands some set of *messages*. Simple structure-like objects understand messages for accessing their fields and messages are sent by using the `send` keyword, followed by an object, a message name, and some number of arguments:

```
(send (new posn 3 4) x) ;=> 3
(send (new posn 3 4) y) ;=> 4
```

The `send` notation is simple, but syntactically heavy. Once students are comfortable with `send`, we introduce shorthand to make it more convenient, writing (x . m) for (`send` x m). The *dot notation* can be nested, so (x . m . n) is shorthand for (`send` (`send` x m) n). (The approach of introducing a simple, uniform syntax and later introducing a convenient shorthand that would have been confusing to start with follows the approach of first introducing `cons` and then later `list` and `quote` in the first semester.)

It is possible to endow objects with functionality by defining *methods*, which extend the set of messages an object understands. A method definition follows the same syntax as a function definition, but is located inside of a class definition. Here is a more complete development of the `posn` class with two methods:

```
;; A Posn is a (new posn Number Number),
;; which represents a point on the Cartesian plane
(define-class posn (fields x y)

  ;; dist : Posn -> Number
  ;; Distance between this posn and that posn
  (check-expect ((new posn 0 0) . dist (new posn 3 4)) 5)
  (define (dist that)
    (sqrt (+ (sqr (- (this . x) (that . x)))
             (sqr (- (this . y) (that . y))))))

  ;; dist-origin : -> Number
  ;; Distance of this posn from the origin
  (check-expect ((new posn 0 0) . dist-origin) 0)
  (check-expect ((new posn 3 4) . dist-origin) 5)
  (define (dist-origin)
    (this . dist (new posn 0 0))))
```

This class definition defines a new class of values which are posn objects. Such objects are comprised of two numeric values and understand the messages x, y, dist, and dist-origin. Unit tests have been included with each method definition, following the principles of the design recipe studied in the first semester. Although check-expect forms can appear within class definitions, they are lifted to the top-level when a program is run.

Methods can be defined to consume any number of arguments, but they are implicitly parameterized over this, the object that received the message.

3.2   Where did the cond go?

Unions, and recursive unions in particular, are a fundamental kind of data definition that students are well-versed in from the previous semester. A fundamental early lesson is how to represent (recursive) unions using classes and how to write recursive methods. As an example, figure 1 defines binary trees of numbers (an archetypal recursive union data definition) using the Beginning Student language (BSL) as used at the start of the first semester, and also using the Class language of our course.

The structure of this program is analogous to the approach of the previous semester but this example brings to light an important difference with the functional approach. The method for computing the sum of a leaf is defined in the leaf class, while the method for computing the sum of a node is in the node class. When a tree object is sent the sum method, there is no function with a conditional to determine whether the object is a leaf—instead, the object itself takes care of computing the sum based on its own sum method. This shift in perspective is at the core of object-orientation: objects contain their own behavior and the case analysis previously done in functions is eliminated.

3.3   Worlds and animations

At Northeastern, Programming in the first semester is often oriented around interactive event-driven video games. The basic design of a video game involves defining a data representation for states of the game and functions for transitioning between states based on events such as clock ticks, keyboard input, or mouse events. The design of a game thus involves the design of data and operations on that data;

```
#lang bsl                                  #lang class/1
;; A Tree is one of:                       ;; A Tree is one of:
;; - (make-leaf Number)                    ;; - (new leaf Number)
;; - (make-node Tree Number Tree)          ;; - (new node Tree Number Tree)
(define-struct leaf (v))                   ;; and implements
(define-struct node (left v right))        ;; sum : -> Number
                                           ;; sums the elements of this tree
;; sum : Tree -> Number
;; sums the elements of the given tree      (define-class leaf
(define (sum a-tree)                         (fields v)
  (cond [(leaf? a-tree) (leaf-v a-tree)]     (define (sum) (this . v)))
        [else
         (+ (sum (node-left  a-tree))       (define-class node
            (node-v a-tree)                   (fields left v right)
            (sum (node-right a-tree)))]))      (define (sum)
                                               (+ (this . left . sum)
                                                  (this . v)
                                                  (this . right .sum))))


(check-expect (sum (make-leaf 7)) 7)        (check-expect ((new leaf 7) . sum) 7)
(check-expect                               (check-expect
  (sum (make-node                             ((new node
        (make-leaf 1)                              (new leaf 1)
        5                                          5
        (make-node (make-leaf 0)                   (new node (new leaf 0)
                   10                                        10
                   (make-leaf 0))))                          (new leaf 0))))
   16)                                        . sum)
                                             16)
```

Figure 1: Binary tree sum in Beginning Student and in the Class language

in other words, the game involves the design of objects. We therefore continue in the second semester with the use of programming video games but supplement the course with a library for doing so in an object-oriented style. Figure 2 gives an example written in both the functional style and object-oriented style.

The key difference between these two programs is that the functional program uses the 2htdp/universe library, which provides a big-bang form that consumes the initial state of the world and has a declarative form of associating event-handler functions, while the object-oriented program uses an alternative library developed for our class: class/universe. It also provides a big-bang form but it consumes a single argument, the initial state of the world represented as an object. Event handlers are simply methods of this object; for example, clock ticks trigger the on-tick method.

The program on the left is the first program of the first semester, while the one on the right is the first program of the second semester. Our approach is able to make the conceptual connection between functional and object-oriented programming quite clear while appealing to the familiar event-driven interactive programs developed throughout the year.

```
#lang bsl                                      #lang class/1
(require 2htdp/image 2htdp/universe)           (require 2htdp/image class/universe)


;; A World is a Number                         ;; A World is a (new world Number)
                                               (define-class world
                                                 (fields n)
;; on-tick : World -> World
(define (tick w)                                 ;; on-tick : -> World
  (add1 w))                                      (define (on-tick)
                                                   (new world (add1 (this . n)))))

;; to-draw : World -> Image
(define (draw w)                                 ;; to-draw : -> Image
  (place-image                                   (define (to-draw)
    (circle 10 "solid" "red")                      (place-image
    w 200 (empty-scene 400 400)))                   (circle 10 "solid" "red")
                                                    (this . w) 200 (empty-scene 400 400)))

;; on-key : KeyEvent World -> World
(define (on-key k w) 10)                          ;; on-key : KeyEvent -> World
                                                 (define (on-key k) (new world 10)))

(big-bang 10
          [to-draw draw]                       (big-bang (new world 10))
          [on-tick tick])
```

Figure 2: World programs

The move to object-oriented style immediately and naturally leads to designs that are enabled by organizing programs around objects. For example, the state pattern [13] becomes useful almost immediately. The programs in figure 2 animate a rocket (rendered as a circle in this example) taking off. An illustrative follow-up exercise is to animate a rocket that *lands*. The natural design is to have two variants for states of the rocket: one for descending rockets and one for landed rockets (an example is given in appendix A). While in the functional approach it is easy to use the state-pattern for the *data* representing a rocket, it is more difficult to have states of *behavior*. The typical solution adds conditionals to all of the event handlers. In the object-oriented approach, states of behavior are just as natural as data. It is therefore straightforward to design programs with easy to observe invariants such as "a landed rocket never changes position." In the functional approach, even such simple properties are more involved to establish, because all event handlers must be inspected.

This approach also leads naturally to discussion of inheritance. Often programs with multiple states wish to share the implementation of some methods. We first show that this can be accomplished at the cost of minor boilerplate with delegation, and then show how inheritance allows the programmer to avoid duplication and boilerplate entirely. Once inheritance is able to group identical methods, overriding is a natural next step when some but not all of the implementations are identical across the variants.

3.4 Language levels

Our introduction to object-oriented programming is built on a series of "language levels", each of which introduces additional features, adding complexity to the programming model and expressiveness to the programs. Each language is class/*N* for some *N*, with features appearing in the following order.

0. Classes and Objects

1. Abbreviated notation for method calls

2. Super classes

3. Overriding

4. Constructors

Several commonalities run through all of these languages. First, they are all purely functional; we do not introduce imperative I/O or side-effects until after transitioning to Java in the second half of the course. Second, they all are a super set of the *Intermediate Student* language from *How to Design Programs*, meaning that they support higher-order functional programming and lists.

One key principle that we adhere to in the design of the language levels is that no features of the language are added purely to support "software engineering" concerns such as specification mechanisms. Not only does that language not support declaring types or contracts, but interfaces are described purely in comments.

This is not to say that interfaces and contracts are optional; in fact, they are mandatory. But the focus of the first part of the course is on the fundamentals of object-orientation. Teaching the use of software engineering tools such as type systems, while vital, is a topic which we defer to the second half of the course when we transition to Java.

We made this decision after experience in which students were confused about the relationship between explicit interface specifications, type systems, and the informal data definitions and contracts which students are required to write for all methods. After removing interfaces from the language and making them purely a specification construct, this confusion disappeared.

4   From principles to industrial languages

The transition from custom teaching languages to a professional language takes place about half-way through the course. At this point, students already have experience with many of the essential concepts of object-oriented programming. In particular: objects, classes, fields and methods, dynamic dispatch, inheritance, and overriding.

From this point, almost any language that students might encounter in future co-op positions, summer internships, or upper-level courses would be an appropriate follow-up. Our course transitions to Java, but C#, Python, Ruby, Eiffel, or JavaScript would all work naturally. The key lesson of the transition is that the fundamental principles underlying object-oriented programming remain the same between languages, and that learning a new language is primarily a matter of mapping these concepts to specific constructs in the new language. Of course, particular languages also use unique specific mechanisms which need to be taught to use the language effectively, but these are rarely as vital as the cross-language principles.

We chose the half-way point as the time for transition based on experience with earlier versions of this course. In particular, we found that a later transition, while allowing us to present additional concepts in a controlled environment, did not give students sufficient time and experience with Java. Subsequent classes found that students were strong on fundamentals but weak on Java practice. The other alternative, transitioning earlier, would not provide sufficient time to cover the fundamental topics before the transition.

4.1 Functional Java

The transition begins with replicating the object-oriented style of our teaching languages in Java. In particular, we do not introduce mutation, for loops, or mutable data structures such as arrays or `ArrayLists` until later in the semester. Instead, students design data representations using classes, with `interfaces` representing unions of data. Additionally, we avoid mention of the distinction between primitive and other values in Java, which is made easier by not using standard libraries early. An example of this style of programming is presented in figure 3, repeating the binary tree sum from the previous section.

Comparing this figure to the previous example illustrates a number of the differences that students are exposed to upon transition to Java.

1. Explicit representation of unions and interfaces in the language. Previously, interfaces were simply described in stylized comments, following the *How to Design Programs* approach.

2. Types are now specified as part of the program and are (statically) enforced. Data definitions and interfaces can be transformed from the stylized comments into `interface` definitions and method signatures annotated with types. Students are taught the benefits of type systems, which impose syntactic restrictions sufficient to prove objects meet (the structural aspects of) their interface definitions. Students also encounter the downside of types when they discover the type system cannot always follow valid reasoning about program invariants and may reject perfectly good programs.

3. Java syntax is substantially different and more verbose. For example, constructors must be defined explicitly.

4. The testing environment is somewhat different, and requires additional boilerplate, although we are able to use the JavaLib framework [20] to support testing with structural equality.

There are other differences which cannot be seen from a code snippet.

5. Students must use a new development environment and compiler. In class, we primarily develop in a text editor and run the Java compiler at the command line. In labs and on homeworks, students typically use the Eclipse IDE.

6. Installing and configuring libraries is now required. Because we use a custom library for testing, students must cope with library installation and class paths on the first day.

All but the first two of these changes are unrelated to the fundamental lessons we hope to teach—the rest merely present additional hurdles for students. However, at this point in the semester, the students are far better equipped to meet these challenges. They are already familiar with objects, classes, and the other concepts we have covered. They are also fully engaged in the class, instead of making the transition in the midst of the transition between semesters. Finally, they have now been programming for 50% longer than they had at the start of the semester.

4.2 Traditional Java

Thanks to the preparation in the first half of the course, we can cover OO programming in a functional subset of Java in a just a few lectures. We then increase the subset of the language we use to encompass mutation, loops, and mutable data structures and introduce the underlying design principles. We present `ArrayLists`, followed briefly by arrays. Students use, and then implement, hash tables as well as other mutable and immutable data structures. Conventional input and output are treated only very briefly, as we focus instead on both fundamentals and exercises making use of real APIs such as hashing functions

```
import tester.*;

interface Tree {
    // sums the elements of this tree
    Integer sum();
}

class Leaf implements Tree {
    Integer v;
    Leaf(Integer v) { this.v = v; }
    public Integer sum() { return this.v; }
}

class Node implements Tree {
    Tree left; Integer v; Tree right;
    Node(Tree l, Integer v, Tree r) {
        this.left = l;
        this.v = v;
        this.right = r;
    }

    public Integer sum() {
        return this.left.sum() + this.v + this.right.sum();
    }
}

class Examples {
    void test_tree(Tester t) {
        t.checkExpect(new Leaf(7).sum(), 7);
        t.checkExpect(new Node(new Leaf(1),
                               5,
                               new Node(new Leaf(0), 10, new Leaf(0))).sum(),
                      16);
    }
}
```

Figure 3: Binary tree sum in the style of *How to Design Classes*

or Twitter posting. Finally, `while`, `for`, and for-each loops are presented, following the methodology of *How to Design Classes* which connects loops to stylized use of recursive functions with accumulators, a technique the students now have used for two semesters.

4.3   Beyond Traditional Java

Finally, at the end of the course, we are able to build on the two major segments to examine less-well-explored topics in object-oriented programming. Typically, we cover the basics of representing objects in a functional language, advanced OO techniques such as mixins and prototypes, and a new OO language such as Ruby or JavaScript. Additionally, we emphasize the ability to embed functional programming in an OO context, using techniques such as the command pattern and the visitor patterns. Again, the key message is the transferability of concepts across languages.

5   Related work

Teaching programming principles in a functional style has a long history, with Abelson and Sussman's *Structure and Interpretation of Computer Programs* [1] being a prominent example. Our work follows in the tradition of the *Program by Design* (PbD) project[2] (previously known as the *TeachScheme!* project), which emphasizes a systematic approach to program construction.

Since the introduction of functional-first curricula, and more specifically in the Program by Design framework, numerous courses have tackled the problem of transition. Typically they, as we, transition to Java in the second course. We discuss first the approach developed by some of the principal creators of PbD, and then other approaches.

5.1   Program by Design and ProfessorJ

The Program by Design project initially focused only on the first course, with the second course typically taught in Java in institution-specific ways. Subsequently, the pedagogical approach was extended to Java, but without the tool support and textbook of the first course. An example of this approach is described by Bloch [5], who presents the experience integrating these courses at Adelphi. He reports that "many of Java's concepts could be introduced more easily in a second course than a first."

With these lessons in mind, the PbD project set out to apply the lessons of teaching languages and IDE support to Java, as well as to present the approach to object-oriented programming in textbook form. ProfessorJ [15] is the resulting system, accompanying the draft textbook *How to Design Classes* [11]. In parallel to our course, Northeastern teaches the remainder of its computer science majors following this approach.

ProfessorJ and *How to Design Classes* maintain many of the ideas of the first course. In particular, ProfessorJ brings language levels to Java, in an attempt to smooth the transition for students from the first course and provide more helpful feedback. ProfessorJ is also embedded in the DrRacket IDE, increasing familiarity for the students and supporting tools such as an interactive read-eval-print loop.

However, the "day 1" transition from the student languages used with *How to Design Programs* to ProfessorJ is too abrupt and too large. Most significantly, changing languages from the first semester immediately rather than simply adding a new concept confuses too many issues for students. On the first day of a *How to Design Classes*-based course, students see object-orientation, a new programming

---

[2]http://www.programbydesign.org/

paradigm; Java, a new language with new syntax, and a static type system, a crucial but orthogonal concept. In contrast, our course presents just one of these concepts on the first day, but covers all of them by the end of the semester.

ProfessorJ also takes on the the dual challenges of implementing Java as well as subsetting it. This ultimately resulted in both a limited Java environment as well as the eventual abandoning of the tool since it was too difficult to maintain, let alone keep up with advances in Java.

Committing to Java on the first day, regardless of the environment provided to students, has significant limitations. First, the syntactic and semantic heaviness of Java is a burden for beginning students, and discourages interactive and experimental programming. The very first chapter of *How to Design Classes* discusses the fixed size of Java integers, a topic avoided entirely in the first course. Second, by committing to a particular industrial-strength language, it closes off possibilities in the curriculum. Third, it commits entirely to the new paradigm, making it more difficult for students to compare the approaches.

Since ProfessorJ is no longer available, students are faced with an even starker change on the first day. Even with a student-oriented environment such as DrJava or BlueJ [2, 16], students must learn an entirely new tool, along with new libraries. If the course uses a typical professional development environment such as Eclipse, students must also contend with compilation, loss of interactivity, and subtle issues such as classpaths, none of which are fundamental to the concepts that the course focuses on.

5.2 Other transitions

Not every curriculum that begins with *How to Design Programs* transitions to Java after the first course. Ragde [22] describes a second course that includes both more advanced work in Scheme beyond teaching-oriented languages as well as low-level programming in C, taught to computer science majors at University of Waterloo. Radge's course intentionally does not use student-oriented languages, although the recently-developed C0 language [19] could provide such a language. Other discussions of functional programming in the first year [7] do not discuss the problems of transition.

5.3 Other approaches to Java

The problems of teaching Java in introductory courses have been well-explored; we mention only a few related directions here. DrJava [2] and BlueJ [16, 18] are introductory environments for Java, which alleviate some but not all of the drawbacks we have outlined. For example, both of these systems immediately present students with (1) type systems and (2) Java syntax, and (3) do not support the image values and exact numeric values that we rely on in our course.

Several teaching-oriented graphics libraries for Java have been proposed [6, 3], but these are significantly more complex than the graphics and interaction libraries we are able to use in the introductory language we present.

6 Experience and outlook

We have now completed the third iteration of this course, teaching approximately 35 students each time. Our experience has been uniformly positive, and the students have gone on to significant success in the subsequent courses, despite the curriculum differing from what the bulk of Northeastern University computer science majors take. Anectodally, the class has also had notable success in the recruitment and

retention of female students, as compared to the other versions of the second-semester course. However, the classes are sufficiently different as to make a precise comparison impossible.

The course has provided a vantage point to introduce topics that will be taken up later in the curriculum. We present types, contracts, invariants, and properties of functions, all of which tie into both the concurrent course on logic and computation, as well as later classes on formal methods. The emphasis on representation-independence and interfaces both tie into later classes on software engineering, as well as preparing students for algorithms and data structures courses. Finally, the use of interactive and distributed systems connects to later courses on operating systems and networks.

Despite our success, much remains to be done. Type systems are a fundamental concept, but their introduction accompanies the rest of Java. Developing a typed version of our introductory languages would allow a smoother introduction of this idea.

Our class's use of Eclipse could also be improved by first transitioning to a pedagogically-oriented Java environment, but we have not evaluated the specific options. Alternatively, introducing Java-like syntax for the teaching languages we have developed would help tease apart the difficult transitions still present in the course.

Finally, the Java portion of the class does not continue the use of "World"-style interactive graphical programming, although a version of the "World" library has been developed for Java [20]. Instead, our course focuses on coverage of standard Java libraries, as well as introductory algorithmic and data structure topics. Continuing to use World-style programming in motivating examples might be valuable for continuity between the two halves of the course.

References

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1996.

[2] Eric Allen, Robert Cartwright, and Brian Stoler. DrJava: a lightweight pedagogic environment for java. *SIGCSE Bull.*, 34(1):137–141, 2002.

[3] Carl Alphonce and Phil Ventura. Using graphics to support the teaching of fundamental object-oriented principles in CS1. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 156–161. ACM, 2003.

[4] Ian Barland, Matthew Flatt, and Robby Findler. The design of a functional image library. In *Workshop on Scheme and Functional Programming (SFP)*, 2010.

[5] Stephen A. Bloch. Scheme and java in the first year. *J. Comput. Sci. Coll.*, 15(5):157–165, 2000.

[6] Kim B. Bruce, Andrea Danyluk, and Thomas Murtagh. A library to support a graphics-based object-first approach to CS 1. In *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, volume 33 of *SIGCSE '01*, pages 6–10. ACM, 2001.

[7] Manuel M. T. Chakravarty and Gabriele Keller. The risks and benefits of teaching purely functional programming in first year. *J. Funct. Program.*, 14(1):113–123, 2004.

[8] Matthias Felleisen, Robert B. Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to design programs: an introduction to programming and computing*. MIT Press, 2001.

[9] Matthias Felleisen, Robert B. Findler, Matthew Flatt, and Shriram Krishnamurthi. The structure and interpretation of the computer science curriculum. *Journal of Functional Programming*, 14(4): 365–378, 2004.

[10] Matthias Felleisen, Robert B. Findler, Matthew Flatt, and Shriram Krishnamurthi. A functional I/O system or, fun for freshman kids. In *ICFP '09 Proceedings of the 14th ACM SIGPLAN International Conference on Functional programming*, pages 47–58. ACM, 2009.

[11] Matthias Felleisen, Matthew Flatt, Robert Bruce Findler, Kathryn E. Gray, Shriram Krishnamurthi, and Viera K. Proulx. *How to Design Classes (Draft)*. 2012. URL `http://www.ccs.neu.edu/home/matthias/htdc.html`.

[12] Robert B. Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. Drscheme: a programming environment for Scheme. *JFP*, 12 (02):159–182, 2002.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[14] Kathryn E. Gray and Matthias Felleisen. Linguistic support for unit tests. Technical Report UUCS-07-013, University of Utah, 2007.

[15] Kathryn E. Gray and Matthew Flatt. ProfessorJ: a gradual introduction to java through language levels. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 170–177. ACM, 2003.

[16] James I. Hsia, Elspeth Simpson, Daniel Smith, and Robert Cartwright. Taming java for the classroom. *SIGCSE Bull.*, 37(1):327–331, 2005.

[17] Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[18] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. The BlueJ system and its pedagogy. *Journal of Computer Science Education*, 13(4), 2003.

[19] Frank Pfennig. C0 reference, 2011. URL `http://c0.typesafety.net/doc/c0-reference.pdf`.

[20] Viera K. Proulx. Javalib, 2012. URL `http://www.ccs.neu.edu/javalib/`.

[21] Viera K. Proulx and Kathryn E. Gray. Design of class hierarchies: an introduction to OO program design. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education*, SIGCSE '06, pages 288–292. ACM, 2006.

[22] Prabhakar Ragde. The chilling descent: making the transition to a conventional curriculum. In *Proceedings of the 2008 international workshop on Functional and declarative programming in education*, FDPE '08, pages 13–20. ACM, 2008.

[23] Joel Spolsky. The Perils of JavaSchools, 2005. URL http://www.joelonsoftware.com/articles/ThePerilsofJavaSchools.html.

A   Worlds and the State pattern

```
#lang class/1
(require 2htdp/image class/universe)

;; A World is one of
;; - (new landed-world)
;; - (new downworld Number)
(define-class landed-world

  ;; to-draw : -> Image
  (define (to-draw)
    (place-image
     (circle 10 "solid" "red")
     390 200 (empty-scene 400 400))))

(define-class downworld
  (fields n)

  ;; on-tick : -> World
  (define (on-tick)
    (cond [(zero? (this . n))
           (new landed-world)]
          [else
           (new world (sub1 (this . n)))]))

  ;; to-draw : -> Image
  (define (to-draw)
    (place-image
     (circle 10 "solid" "red")
     (this . w) 200 (empty-scene 400 400)))

  ;; on-key : KeyEvent -> World
  (define (on-key k) (new world 400)))

(big-bang (new downworld 400))
```