Design of Classes I

Marco T. Morazán
Seton Hall University
morazanm@shu.edu

The use of functional programming languages in the first programming course at many universities is well-established and effective. Invariably, however, students must progress to study object-oriented programming. This article presents how the first steps of this transition have been successfully implemented at Seton Hall University. The developed methodology builds on the students' experience with type-based design acquired in their previous introduction to programming courses. The transition is made smooth by explicitly showing students that the design lessons they have internalized are relevant in object-oriented programming. This allows for new abstractions offered by object-oriented programming languages to be more easily taught and used by students. Empirical evidence collected from students in the course suggests that the approach developed is effective and that the transition is smooth.

1 Introduction

Designed based introductory courses to programming, inspired by the approach put forth in the textbook *How to Design Programs* (HtDP) [2, 3], are now well-established and effective [4, 15, 17]. This approach focuses on problem solving using type-based program design. That is, solutions to problems are designed based on the data that needs to be processed. One of its most salient features is the use of *design recipes*. A design recipe is a series of steps that take a programmer from a problem statement to a working and tested solution. Each step has a concrete outcome that students and instructors can verify. This approach forces students to focus on problem analysis, data representation, and testing. Mistakenly, many believe that the approach is limited to problems solved using structural recursion. Although a design recipe facilitates designing programs using structural recursion, there are also design recipes, albeit less prescriptive, for programs based on generative and accumulative recursion [2, 3], distributed programming [15], and imperative while loops [16]. That is, a design recipe is more about problem solving than it is about prescriptive steps for a given type instance to be processed. Nonetheless, the pivotal role of types give rise to different (not all) design recipes.

Students that start with a design-based curriculum using a functional programming language eventually must move on to other courses that introduce them to an object-oriented (OO) programming language. How is this effectively done? How do we move beginners from program design using a functional programming language to program design using an OO programming language? They keyword here is *design*. We certainly should not have students migrate to writing programs without designing them. This article presents how this transition starts at Seton Hall University within the context of the first OO-based programming course. At the heart of the approach is type-oriented programming and the use of design recipes to guide students during development. The goal is introduce students to abstractions offered by OO programming languages without starting from scratch and to make a smooth transition from using a functional programming language to an OO programming language. That is, our aim is to build on and reinforce the program design skills that students have internalized. A natural question that may arise in the mind of some readers is: Why run another course using design recipes if students are already familiar

P. Achten & E. Machkasova (Eds.): Trends in
Functional Programming in Education (TFPIE) 2021/22
EPTCS 363, 2022, pp. 1–21, doi:10.4204/EPTCS.363.1

with design recipes? The answer is twofold. First, design skills do not magically transfer from functional programming to object-oriented programming (OOP) automatically in beginners. It is important to show beginning students how to apply their design skills in a new context. Otherwise, it is all too easy for beginning students to set aside the lessons of design and resort to trial-and-error hacking that leads to programs that students cannot convincingly argue are correct and that instructors have difficulty comprehending. Second, the use of design recipes brings OOP to familiar territory for students. Thus, facilitating a smooth transition to OOP and giving students a strong sense that lessons learned in previous courses are relevant and useful when using a programming language that is popular in industry.

This article outlines how students are introduced, at the beginning of their first OOP course, to objects, interfaces, generic programming, polymorphic dispatch, inheritance, λ-expressions, and abstract classes using Java[1]. Unlike most textbooks, the emphasis is not on syntax. Instead, the emphasis is to build on the design experience students have accumulated and bring to the course. The article is organized as follows. Section 2 discusses related work. Section 3 presents the students' background. Section 4 presents how classes are motivated using the students' experience with structures. Section 5 motivates interfaces, generic programming, polymorphic dispatch, inheritance, and λ-expressions building on students' design experience with union types. Section 6 motivates abstract classes based on students' experience with functional abstraction. Section 7 presents student feedback. Finally, Section 8 draws conclusions and presents directions for future work.

2 Related Work

Broadly speaking, there are two approaches to introduce beginners to OOP. The most closely related to the approach described in this article are those approaches that teach design principles using a DSL. The other approach is more widely spread and is based on teaching the syntax of a language. There are strong advocates in both camps each with valid points to make. The advocates for teaching design put forth that design principles are language agnostic and may be applied regardless of the syntax used. Currently, a major problem with this approach is the lack of textbooks available that students and instructors may follow. The advocates for a syntax-based approach put forth that there is a long history of teaching syntax and numerous textbooks that students and instructors may rely on. Furthermore, they observe that most new faculty members face a steep learning curve when adopting a design-based approach. This article will not settle this debate, but it is fairly clear that a popular syntax today is left behind when a new programming language becomes fashionable. Therefore, an emphasis on design rather than syntax is likely to be more beneficial to students especially in the long term.

2.1 DSL-Based Approaches

The unpublished textbook How to Design Classes (HtDC) is closely coupled with a DSL known as ProfessorJ [7]. It first introduces students to classes as a mechanism to define a type of data that may have many fields just like a structure may have many fields. A field type in a class, unlike a field type in a HtDP structure, must be part of the program's code. As a structure, a class has a constructor that initializes the fields and produces an object–an instance of the type defined by the class. This class constructor is explicitly written by the programmer. The design of a class starts by understanding how data is to be represented. Union types are introduced to motivate the need for interfaces. In essence, an interface is used to define a type and glue together its subtypes. Interfaces are particularly useful

[1] Latter parts of the course cover mutation, arrays, ArrayLists, while-loops, design patterns, iterators, and hash tables.

to define data of arbitrary size (e.g., a list) and, therefore, programs based on structural recursion. The design of methods is not tackled until students develop some expertise defining classes that only have a single constructor. The design of a method follows the design recipe approach found in HtDP. Testing of methods is done using syntax as the following:

```
check item.price() expect 31
```

HtDC has students engage in problem and data representation analysis before developing methods. Furthermore, students are encouraged to write tests before writing a method. Similar to HtDC, the approach described in this article starts with parallels between structures and classes with no subtypes and then moves to union types and abstraction. In contrast, however, we start directly with Java syntax and rely quite heavily on the knowledge students acquired designing functions using a functional programming language. For example, students have been introduced to generic types and these are introduced early in the course. In this manner, for instance, students reason and design based on a list of X instead of a list of geometric shapes, a list of points, or a list of strings.

Another design-based approach developed at Northeastern University teaches students (that have used HtDP in their first course) using a hybrid approach [19]. The beginning of the second course proposes the use of several DSL s embedded in Racket before making the full transition into Java. The motivation for such an approach is that students first develop a command of basic OOP principles before tackling Java's syntax, types, and a compiler (as opposed to an interpreter). This approach also relies heavily of the design recipe to provide students (and instructors) with a framework for program design and discussion. From the start objects are described as consisting of data and functions. A class is first thought of as a structure and its fields are accessed using a dot notation. For example, `(point . x)` denotes that the message, `x`, on the right hand of the dot is used to request a value from the object, `point`, on the left hand side of the dot. Union types and design based on structural recursion are introduced as requiring a distinct class for each variety in the union. The transition to Java occurs in the middle of the semester. This transition is made to functional Java first to make programs as similar to those developed using the DSLs embedded in Racket. Later mutation and looping constructs are presented. In a similar spirit, the work presented in this article introduces objects as data and functions. Objects are introduced using functional Java to have students focus on the principles of OO design instead of properly making mutations. In contrast, the work described here is presented to students after two semesters (not one) designing using a functional programming language. It relies much more on student programming-maturity that allows them to tackle a new syntax (i.e., Java) from the beginning of the course. Furthermore, this experience allows us to quickly and smoothly introduce students to generic programming, interfaces, λ-expressions, and abstract classes.

2.2 Java-Based Approaches

Most OOP textbooks for beginners (and, therefore, probably most instructors) follow a syntax-based approach (e.g., [5, 6, 18, 20]). The expectation is that somehow students will learn how to successfully write OO programs based on syntax, examples, and in-class warnings about common pitfalls. Although many readers of this article, surely, "learned" how to program in this manner, it is not a stretch to say that OO abstractions are not effectively taught in this manner. Simply stated, programmers need to understand what an abstraction represents and how it was developed. This kind of insight is unlikely to be discovered by novices learning syntax by example. In addition, beginning students need a systematic way to design large software systems–one of the principle goals of OOP. Students must learn to design a (hierarchical) class system and understand how components in a system are expected to interact. In contrast to this

approach, the work described in this article motivates the existence of abstractions (such as classes, interfaces, and abstract classes) and how to design programs using them.

DrJava is designed to gently introduce students to writing Java programs and to provide support for developing programs [1]. It is an IDE that provides students with a definitions and an interactions pane that hide the details of interacting with a Java compiler. Students are exposed to full Java syntax from the beginning including types, classes, and methods. In addition, DrJava provides support for interactions with error messages. Like DrJava, the approach described here has students use Java syntax from the start. Unlike the DrJava approach, we allow students to directly start wrestling with the error messages of an industrial compiler. This is based on the observation that the work described here takes place in their third programming course. Therefore, there is more programming maturity among our students than among beginners in a first semester course.

BlueJ is an IDE specifically developed to teach OOP to beginners [10]. Its interface shields programmers from interacting directly with the compiler and presents beginners with a main window displaying a class diagram to visualize the application's structure. Programmers can then, for example, click on a class to edit it. Its pedagogy starts with objects first, but never with a blank screen. Students read relatively large Java code from the beginning and are asked to modify it. As already mentioned above, like BlueJ, the work presented here has students work directly with Java from the beginning. In contrast, students are trained to design classes from the beginning instead of working with an unfamiliar design. This decision was made because classes are central in OO design. Methods inside a class only make sense if the overall design of the system is understood. That is, methods only make sense if it is understood why they are encapsulated in a given class.

At Northeastern University the course *Fundamentals II Introduction to Class-based Program Design* uses Java in conjunction with HtDP-inspired design recipes [11]. As the approach presented in this article students are taught systematic design from the beginning using Java. Among other things, this means that contracts immediately become part of the code to define the types of variables and methods. Another similarity is that students are quickly introduced to testing, interfaces and abstract classes. In contrast, tests for methods to access fields are not introduced. The work presented emphasizes these because beginning students eagerly insist in not writing such methods which becomes a problem when fields become private. Another contrasting feature is that the Northeastern course does not introduce students to λ-expressions in Java. As λ-expressions become part of idiomatic Java it is important to introduce students to their use and the convenience they provide.

3 Student Background

At SHU, the first two introductory Computer Science (CS) courses focus on problem solving using a computer [12, 14]. The languages of instruction are the successively richer subsets of Racket known as the student languages which are tightly-coupled with HtDP [2, 3]. No prior experience with programming is assumed. The first course starts by familiarizing students with primitive data (e.g., numbers, strings, booleans, symbols, and images), primitive functions, and library functions to manipulate images. During this introduction, students are taught about variables, defining their own functions, and the importance of writing signatures, purpose statements, and unit tests. Through the development of signatures students are introduced to types albeit as comments in their code. The course then introduces students to data analysis and programming with compound data of finite size (i.e., structures). At this point, students are exposed to their first design recipe. Students gain experience in developing data definitions, type instances, function templates, and tests for all the functions they write. Building on this experience,

students develop expertise with processing compound data of arbitrary size such as lists, natural numbers, and trees. In this part of the course, students learn to design functions using structural recursion. After structural recursion, students are introduced to functional abstraction and the use of higher-order functions such as `map` and `filter`. This part of the course also introduces students to generic types as these are necessary to write contracts for higher-order functions. The first semester ends with a module on distributed programming [13, 15].

In the second course, students are exposed to generative recursion, accumulative recursion, and mutation [14]. The course starts with generative recursion and transitions to accumulative recursion. The course then introduces students to mutation. This part of the course includes a module on interfaces as a mechanism to encapsulate state variables and functions on these state variables. These functions are called constructors (to build a new instance of the type), observers (to compute a value from a given type instance), and mutators (to assign a new value to a state variable). Message-passing is used to request a service from an interface. In essence, an instance of an interface is an object. This part of the course also includes a module on `while` loops [16]. This module introduces students to the proper sequencing of mutations to obtain a computational goal using loop invariants and Hoare Logic [8, 9].

Each course is for a semester (15 weeks). There are two weekly 75-minute lectures that students are required to attend. The typical classroom has between 15 to 25 students. In addition to the lectures, the instructor is available to students during office hours and there are about 30 hours of tutoring each week which the students may voluntarily attend. The tutoring hours are conducted by undergraduate students handpicked and trained by the lecturer.

The use of different type-based design recipes is emphasized throughout both courses. We can outline these design recipes as follows:

```
1. Problem Analysis                    5. Write Function Header
2. Data Analysis                       6. Write Unit Tests
3. Function Template Development       7. Write Function Body
4. Signature and Purpose Statement     8. Run Tests
```

Students first outline how to solve a problem. They then proceed to create data definitions for the required data's representation (i.e., they define types). Based on the type, a function template is developed that outlines how to process instances. After these initial steps, problem-specific steps follow. Students start with a signature, a purpose statement, and a function header. Armed with these students proceed to write unit tests before writing the function's body. Finally, students run the unit tests and redesign if necessary.

4 From Structures to Classes

Students arrive to the course having familiarity with structures and the design of functions to process a structure. This type of data does not have varieties (i.e., no subtypes). In other words, it is not a union type. We leverage this knowledge to motivate the need for classes.

4.1 Students' Structure Knowledge

To illustrate a student's background consider solving the following problem:

```
A student has a name, grade point average, and the number of credits
completed. Write a function that takes as input a student and that
returns the given student's year level: Freshman, Sophomore,
Junior, or Senior.
```

```
;; student is a structure
;;     (make-student string 0≤ ℝ ≤4.0 natnum)
;; with a name, a grade point average, and a number of completed credits.
(define-struct student (name gpa credits))
;; Sample students
(define STUDENT1 (make-student "Spiderman" 3.5  16))
(define STUDENT2 (make-student "Batwoman"  3.9  43))
(define STUDENT3 (make-student "Superman"  3.7  75))
(define STUDENT4 (make-student "Ironman"   4.0  120))
;; student → string
;; Purpose: Determine the given student's year level
(define (student-level a-student)
  (cond [(<= (student-credits a-student) 30) "Freshman"]
        [(<= (student-credits a-student) 60) "Sophomore"]
        [(<= (student-credits a-student) 90) "Junior"]
        [else "Senior"]))
(check-expect (student-level STUDENT1) "Freshman")
(check-expect (student-level STUDENT2) "Sophomore")
(check-expect (student-level STUDENT3) "Junior")
(check-expect (student-level STUDENT4) "Senior")
```

Figure 1: Function to Determine a Student's Year Level

As part of the first step of the design recipe students outline how to solve the problem as follows:

```
A student with 30 credits or less is a freshman
A student with 31-60 credits is a sophomore
A student with 61-90 credits is a junior
Otherwise, the student is a senior
```

This analysis already informs the student that a conditional expression is required to solve this problem.

Given that a `student` is compound data of finite size it may be represented using structure:

```
(define-struct student (name gpa credits))
```

The structure has 3 fields: a string for the name, a nonnegative real number for the grade point average, and a natural number for the number of completed credits. Once a data definition and structure definition are written students define sample instances. In this example, at least four are required given that a `student` may be at one of four levels. Students know that the above structure definition creates a constructor (`make-student`) and four observers (`student-name`, `student-gpa`, `student-credits`, and `student?`)[2].

The function template for Step 3 must capture all the commonalities that all functions that process a `student` may have. This step is problem-agnostic and may be used (as the result of Step 2) to solve other problems that require processing a `student`. A `student`-processing function needs at least a `student` as input. Given that a student structure has three fields there are 3 selector expressions in the body of the definition template that may be useful. In addition, there must be at least one test given that there is only one variety of `student` (i.e., there are no subtypes). The template for a function on a `student` is:

[2]They are also aware that a mutator for each field is created.

```
;; student ...  → ...     Purpose: ...
(define (f-on-student a-student)
  ...(student-name a-student)...(student-gpa a-student)...
  ...(student-credits a-student)...)
(check-expect (f-on-student ...) ...) ...
```

The solution with the rest of the design recipe steps completed is displayed in Figure 1. The output of the function may be represented as a string (e.g., `"Freshman"`). We briefly observe that, as expected, the function returns a string, the body of the function is a conditional expression, and there is one test for each possible outcome[3].

4.2 Classes and Objects

Students are explained that a type may be described as data and the operations that are valid on instances of the type. Code is organized in a class and instances of the class are called objects. Unlike structures, however, the functions (i.e., methods) implementing the valid operations must all be written by the programmer. In addition, unit tests are written in a separate file.

This means that the basic design recipe must be updated to encompass this perspective. Students are explained that the methods implementing the valid operations on the type being defined must all be encapsulated in the class along with the data that may vary from one instance to the next. The following design recipe is presented:

1. Problem Analysis
2. Data Analysis
3. Class Template Development
4. Tests Template Development
5. Write Tests
6. Method Development
   (a) Write Signature and Purpose Statement
   (b) Write Method Header
   (c) Write Unit Tests
   (d) Write Method Body
7. Run Tests

Problem analysis remains unchanged. Data analysis now uses an object, instead of a structure, to describe compound data of finite size. The next two steps outline the basic structure of any class and testing class for the type being defined. Once students have a testing template they develop tests for the valid operations. Method development for the valid operations proceeds in a manner similar to functions using a functional programming language. A major difference is that signatures (i.e., types) are part of the code and not comments. Finally, students must run their tests and redesign if necessary. Observe that each step of the design recipe has a specific outcome that may be verified by a student or instructor reading the code.

To make the steps of the design recipe concrete let us revisit the student problem from Section 4.1. Problem analysis remains unchanged. That is, a `student`'s year level is determined the same way. Data analysis defines a `Student` as an object:

[3]Technically, the function returns an instance of an enumerated type (containing 4 strings), but such details are omitted here in the interest of brevity.

```
Student is an object, new student(String 0≤ℝ≤4.0 natnum),
with a name, a grade point average, and a number of completed credits.
```

The following operations are valid on a student:

Constructors

Student String double int → Student

Observers

getName : → String
getGpa : → double
getCredits : → int
isStudent : → boolean

Observe that the description of the operations includes the signatures for each method. For example, `isStudent` takes no input and returns a `boolean`. It is noteworthy that this constant `true` method is included to have students develop the habit of developing predicates which become useful when there is variety in the data. In addition, it is consistent with student expectations moving from `Racket` structures to classes.

Class template development uses the signatures from the previous step to outline any class that implements `Student`. The template must include instance variables and method signatures. At this point we do not yet separate specification from implementation. This allows students to more easily develop classes. For our `Student` example, therefore, the class template is[4]:

```
class Student
{String name;     double gpa;   int credits;
 //               0≤gpa≤4.0     credits≥0
 Student(String nm, double g, int c)
 {name = nm;     gpa = g;     credits = c; }
  String  getName()    { return(name);  }
  double  getGpa()     { return(gpa); }
  int     getCredits() { return(credits);  }
  boolean isStudent()  { return(true); }}
```

Observe that when specification is not separated from implementation, the class template contains full implementations of the valid operations. For students, this approach brings their familiarity with structures and function design to bear on class design. It is also noteworthy that `name` $\neq$ `null` is not listed as an invariant. This is because students have not been exposed to `null` yet.

Test template development has students outline the type instances and the tests that must be implemented. For compound data of finite size there must be at least one instance and there must be at least one test for each valid operation. For our `Student` example the testing class template is:

```
class TestStudent
{ @Test
  public void testStudentMethods()
 { Student S1 = new Student(...);
         ⋮
   Student Sn = new Student(...);
```

[4]In the interest of saving space `Java` code formatting and naming conventions are not always followed in this article.

```
public class TestStudent
{ @Test
  public void testStudentMethods()
  {Student S1 = new Student("Spiderman" 3.5  16);
   Student S2 = new Student("Batwoman" 3.9  43);
   Student S3 = new Student("Superman"   3.7  75);
   Student S4 = new Student("Ironman"  4.0  120);
   assertEquals(S1.getName(),    "Spiderman");
   assertEquals(S1.getGpa(),     3.5, 0.01);
   assertEquals(S1.getCredits(), 16);
   assertEquals(S1.isStudent(),  true);
   assertEquals(S1.getLevel(),   "Freshman");
   assertEquals(S2.getLevel(),   "Sophomore");
   assertEquals(S3.getLevel(),   "Junior");
   assertEquals(S4.getLevel(),   "Senior"); }}
```

Figure 2: Tests for Student Class

```
assertEquals(....getName(), ...);
assertEquals(....getGpa(), ..., ...);
assertEquals(....getCredits(), ...);
assertEquals(....isStudent(), ...);
      ⋮ }}
```

The template states that first instances must be developed. This is followed by tests for the observers using the defined instances. At this point, students are told that `public` is part of the required syntax.

The next step of the design recipe has students specialize the testing template. There are two important observations to make. The first is that at this point students are not wondering what needs to be tested. The second is that tests are written before specializing the class template. Sequencing design-recipe steps in this manner becomes useful when an actual problem is solved. Writing tests commonly provides insights into how a value may be computed. The `Student` tests may be specialized as displayed in Figure 2.

At this point students have a `Student` implementation and may run the tests to validate the implementation allowing them to now solve problems. This requires designing methods and tests that are added to the existing `Student` and `TestStudent` classes. For our year level example this requires performing Steps 5-7 of the design recipe given that problem and data analysis as well as template development have already been done. In other words, students build on the work they have done. The following tests are added for Step 5:

```
assertEquals(S1.getLevel(), "Freshman");
assertEquals(S2.getLevel(), "Sophomore");
assertEquals(S3.getLevel(), "Junior");
assertEquals(S4.getLevel(), "Senior");
```

Students observe that these are the same tests written for the structure-based version albeit using different syntax. The following method is added to the `Student` class:

```
Purpose: Determine the given student's level
```

```
String getLevel()
{if (credits <= 30) {return("Freshman");}
 else if (credits <= 60) {return("Sophomore");}
      else if (credits <= 90) {return("Junior");}
           else {return("Senior");} }
```

Once again, students appreciate that in essence this is the same function as in the structure-based version albeit using different syntax. It is worth noting that students do comment on the more cumbersome conditional statement syntax.

5 From Union Types to Interfaces

Problem solving using union types is used to introduce students to interfaces, generic programming, λ-expressions, and polymorphic dispatch. Once again, these topics are not covered from scratch. Instead, we build on the knowledge students bring to the classroom. Specifically, we build on knowledge of parameterized data definitions and functional interfaces using message passing.

5.1 Students' Background

Students arrive knowing how to abstract over data definitions to obtain parameterized (or generic) data definitions in which type variables are used instead of concrete types. For example, consider the following data definitions:

```
;; A list of number (lon) is one of: empty or (cons number lon)
;; A list of string (los) is one of: empty or (cons string los)
```

Students abstract over them to obtain the following parameterized data definition:

```
;; A (listof X), loX, is one of: empty or (cons X loX)
```

Here `X` is a type variable. Given that this is a union type (i.e., the data has subtypes) students know that a conditional is needed to distinguish between the subtypes in the body of any function that processes a `(listof X)`. Such data definitions are used to write generic functions like:

```
;; [X Y]: (X → Y) (listof X) → (listof Y)
;; Purpose: To apply the given function to the given list
(define (mapf f a-lox)
  (if (empty? a-lox)
      '()
      (cons (f (first a-lox)) (mapf f (rest a-lox)))))
(check-expect (mapf add1 '())        '())
(check-expect (mapf add1 '(1 2 3))   '(2 3 4))
```

`X` and `Y` are type variables whose values are unknown until `mapf` receives its arguments. Signatures are parameterized with the unknown types. Students understand that functions parameterized in this manner are generic.

Students also arrive with experience designing functional interfaces using message-passing. An interface defines the operations that are valid on the type defined. As an example, consider the following small interface, `Ilox`, for `(listof X)`:

```
(define (mtList)
 (local
  [(define (mycons an-x)
    (consList an-x service-manager))

   (define (equals L)
     (L 'empty?))


   ;; [Y]: (X → Y) → I(listof Y)
   (define (map f) service-manager)



   ;; [A]: (X → A) A → A
   (define (foldl f acc) acc)

   (define (service-manager m)
    (cond [(eq? m 'first)  (error ...)]
          [(eq? m 'rest)   (error ...)]
          [(eq? m 'empty?)  #true]
          [(eq? m 'cons)    mycons]
          [(eq? m 'equals)  equals]
          [(eq? m 'map)     map]
          [(eq? m 'foldl)   foldl]
          [else (error ...)]))
    service-manager))
```

(a) Empty-List Interface

```
(define (consList first rest)
 (local
  [(define (mycons an-x)
     (consList an-x service-manager))

   (define (equals L)
     (and (equal? first (L 'first))
          ((rest 'equals) (L 'rest))))

   ;; [Y]: (X → Y) → I(listof Y)
   (define (map f)
     (consList (f first)
               ((rest 'map) f)))

   ;; [A]: (X → A) A → A
   (define (foldl f acc)
     ((rest 'foldl) f (f first acc)))
   (define (service-manager m)
     (cond [(eq? m 'first)   first]
           [(eq? m 'rest)    rest]
           [(eq? m 'empty?)  #false]
           [(eq? m 'cons)    mycons]
           [(eq? m 'equals)  equals]
           [(eq? m 'map)     map]
           [(eq? m 'foldl)   foldl]
           [else (error ...)]))]
   service-manager))
```

(b) Cons-List Interface

Figure 3: `Ilox` Implementation.

```
An IloX is an interface that provides the following services:
      first:  X throws error          rest: Ilox throws error
     empty?: boolean                  cons: (X → Ilox)
     equals: Ilox → boolean       [Y] map: (X → Y) → IloY
  [A] foldl: (X A → A) A → A
```

In an interface a service that requires more input returns a function that consumes that input to compute the answer. Otherwise, it returns a value. An interface definition specifies the type of the returned value. Functions like `map` and `foldl` are only parameterized with one type variable, `Y` and `A` respectively, because they are encapsulated inside the interface `ILoX`. That is, `X` is known in any instance of `ILoX`. Finally, students understand that since `(listof X)` is a union type `Ilox` must be implemented twice (once for each subtype). Sample implementations are displayed in Figure 3. Observe that students have been exposed to polymorphic dispatch. There is no need for a conditional expression, for example, to implement functions such as `foldl` or `map` because each subtype interface encapsulates its own version

of these functions. Finally, we note that `equals` is for extensional equality.

5.2 Union Types and Classes

Students are presented with the following design recipe to implement union types in an `OO` language:

1. Problem Analysis
2. Data Analysis Reveals Need for a Union Type
3. Design Interface
4. Develop Unit Tests for Interface
5. Implement the Interface for each Subtype Using a Class
6. Method Development for each Class
   (a) Write Purpose Statement
   (b) Write Method Header
   (c) Write Unit Tests
   (d) Write Method Body
7. Run Tests

Students perform problem and data analysis as they are accustomed. If data analysis reveals the need for a union type then they must implement an interface. Here is where specification is separated from implementation. An interface contains the purpose statements and the method headers for the valid operations. If the data definition is parameterized then the interface must be parameterized. If an operation is parameterized then the method implementing it must also be parameterized.

The next step has students develop unit tests. Tests must be written for every operation for each subtype. The tests, once again, are written before implementing the interface in order to gain insight into how to implement the operations.

After writing tests, students proceed with the implementation of the interface. They must implement the interface for each subtype in the union type in a separate class. That is, the number of classes needed is equal to the number of subtypes in the union type. Each class must implement all the operations in the interface. The implementation in each class is specific to a subtype. This step is reminiscent to students of writing functions for each subtype for functional interfaces. Finally, each class is bound to the interface using, for example, `implements` in `Java`.

Once the interface is implemented there is a good opportunity to talk to students about polymorphic dispatch and why conditionals are not required to distinguish among subtypes. After all this is accomplished, students may engage in implementing the solution to the problem. Students are presented with two choices:

1. Extend the interface
2. Write a program that uses an instance of the union type

The first option is chosen when a new operation is needed (or requested by a client). In this case, a method solving the problem for each subtype must be added to each class. This method must also be added to the interface. The second option is chosen when the problem being solved is not a required operation for the union type. In this case a separate class is written and a conditional is required in the method to distinguish among the subtypes.

```
public interface ILIST<X>
{ILIST<X> cons(X val);
 // Purpose: add given X to front of this list

 boolean isEmpty();
 // Purpose: Determine if list is empty

 X first() throws Exception;
 // Purpose: Return the first list element

 ILIST<X> rest() throws Exception;
 // Purpose: Return rest of this list

 public boolean equals(IList<X> l);
 // Purpose: Is this = given list

 <Y> ILIST<Y> map(IFun<X,Y> f);
 // Purpose: Apply given function to this

 <A> A foldl(IFun2<X, A> f, A res);
 // Purpose: Accumulate given function application from left to right
}
```

Figure 4: Java Interface for (listof X).

To make the steps of the design recipe concrete consider implementing `(listof X)`. Given that there is variety in the data an interface is needed. We may implement the interface as displayed in Figure 4. It is highlighted to students how the types become part of the code. The interface is parameterized with `X` because the data definition is parameterized with `X`. The methods `map` and `foldl` are parameterized with the types, respectively `Y` and `A`, not known to `this` list. Finally, any operation that throws an error must be implemented by a method that throws an `Exception` and the types `IFUN` and `IFUN2` are wrapper interfaces to define, respectively, a one-input and two-input function.

The next step of the design recipe asks students to write unit tests for the interface. Students know that the tests for each operation must cover all the varieties in the union type. Writing tests for `ILIST<X>` provides us with the opportunity to expose students for the first time to casting and to λ-expressions in `Java`. Casting is explained in the context, for example, of letting the compiler know that we mean `Integer` 3 and not the `int` 3. The first exposure to λ-expressions is done untyped. Nonetheless, students are advised that sometimes these expressions must be typed by the programmer. One important thing to note is that `exceptions` are not tested. Figure 5 displays a sample testing file developed in class following student advice. Students find the syntax verbose and awkward at times (i.e., λ-expressions) but they do not feel overwhelmed by the material given that it is introduced in a context that is familiar to them from designing using a functional language.

The next step of the design recipe has students implement the interface. Students are reminded that an interface for a union type must be implemented in the form of a `class` for each subtype. Each method in `ILIST<X>` must be implemented for the `empty` and the `cons` list. This implementation is displayed in

```
class ListTests2
{@Test
  public void test()
 {IList<Integer> N  = new MTLIST2<Integer>();
  IList<Integer> N1 = N.cons(4).cons(6).cons(3);
  IList<Integer> N2 = N.cons(4).cons(6).cons(3);
  IList<Integer> N3 = N.cons(3).cons(6).cons(4);
  IList<String>  E  = new MTLIST2<String>();
  IList<String>  L0 = E.cons("pal!").cons("there ").cons("Hi ");
  try
  {assertEquals(N1.first(), (Integer) 3);
   assertEquals(N1.rest().equals(N.cons(4).cons(6)), true);
   assertEquals(N.isEmpty(), true);
   assertEquals(N1.isEmpty(), false);
   assertEquals(N.cons(10).equals((new mtList<Integer>()).cons(10)), true);
   assertEquals(
   N1.cons(0).equals((new mtList<Integer>()).cons(4).cons(6).cons(3).cons(0)),
   true);
   assertEquals(N.equals(N1), false);
   assertEquals(N1.equals(N2), true);
   assertEquals(E.map(s -> s.length()).equals(N), true);
   assertEquals(L0.map(s -> s.length()).equals(N1), true);
   assertEquals(N.foldl((n, r) -> r.cons(n), N).equals(N), true);
   assertEquals(N2.foldl((n, r) -> r.cons(n), N).equals(N3), true);   }}}
```

Figure 5: `JUnit` Tests for `ILIST<X>`.

Figure 6. The reader can appreciate how remarkably similar the functional interfaces and the classes are. There are some points that must be highlighted to students:

this The concept of `this` is introduced and substitutes references to `service-manager` in Figure 3.

Dot Composition Messages are passed to objects in the same manner as in functional interfaces. For example, we write `((rest 'map) f))` for `map` in Figure 3b. We write `this.rest().map(f)` in Figure 6b.

The final step of the design recipe has students write code to solve a problem. This exercise is similar to `getLevel` for `Student` in Section 4.2. In the interest of brevity the repetition of such a development is omitted. Students end the steps of the design recipe by running the tests and, if necessary, redesigning.

6 Abstraction

Program design may result in code repetition. This commonly occurs when functions are developed for similar types. One way to eliminate the repetition is to develop parameterized data definitions. Students usually face two challenges that prevent them from having repetitive code. The first is that they do not realize the significant similarities between data definitions. These result in functions that are very similar. The second is that possible similarities may not be immediately obvious to students. The first may be

```
class mtList<X>
       implements ILIST<X>

{
 mtList() {}

 ILIST<X> cons(X v)
 {return((new consList<X>(v,this)));}

 X first() throws Exception
 {throw new Exception("...");}

 ILIST<X> rest()
        throws Exception
 {throw new Exception("...");}

 boolean isEmpty() {return(true);}

 boolean equals(IList<X> l)
 {return(l.isEmpty());}

 <Y> ILIST<Y> map(IFun<X, Y> f)
 {return(new mtList<Y>());}

 <A> A foldl(IFun2<X, A> f, A ac)
 {return(ac);} }
```

(a) Empty-List Class

```
class consList<X>
       implements ILIST<X>
{X f;  ILIST<X> r;
 consList(X v, ILIST<X> rest)
 {f = v;  r = rest;}

 ILIST<X> cons(X v)
 {return((new consList<X>(v, this)));}
 X first() {return(f);}
 ILIST<X> rest(){return(r);}
 boolean isEmpty() {return(false);}

 boolean equals(IList<X> l)
 {try {return(
    (this.first().equals(l.first())) &&
     this.rest().equals(l.rest()));}
  catch(Exception e)
  {System.out.println(...);
   return(false); }}
 <R> ILIST<R> map(IFun<X, R> f)
 {return(
   new consList<R>(f.f(this.first()),
                this.rest().map(f)));}
 <R> R foldl(IFun2<X,R> fn, R ac)
 {return(this.rest().foldl(
        fn, fn.f(this.first(), ac)));}}
```

(b) Cons-List Class

Figure 6: `ILIST<X>` Interface Implementation.

resolved using functional abstraction. The second may be resolved using code refactoring (a topic they have not studied before). Abstraction in an `OOP` course is used to motivate abstract classes, inheritance, and code refactoring.

6.1 Student Background

Students arrive to the classroom knowing how to do functional abstraction. A typical student may be presented the following functions to square a list of numbers and to extract the names of a list of inventory records (abbreviated `(listof ir)`):

```
;; (listof number) → (listof number)
;; Purpose: Square the given list of numbers
(define (sqr-list L)
  (cond [(empty? L) '()]
        [else (cons (sqr (first L)) (sqr-list (rest L)))]))
```

```
;; (listof ir) → (listof string)
;; Purpose: Extract the name in the given list of ir
(define (names a-lir)
  (cond [(empty? L) '()]
        [else (cons (ir-name (first L)) (names (rest L)))]))
```

The student immediately observes that these functions are very similar and are candidates for abstraction. They perform this task by applying the design recipe for abstraction [3]:

```
1. Compare functions and mark differences
2. Define the abstract function with the differences as parameters
3. Define and test the original functions using the abstract function
4. Write the signature for the abstract function
```

Applying the design recipe results in the following code[5]:

```
;; [X Y] (listof X) (X → Y) → (listof Y)
;; Purpose: Apply the given function to the given list's elements
(define (mapf L f)
  (cond  [(empty? L) '()]
         [else (cons (f (first L))  (mapf (rest L) f))]))
;; (listof number) → (listof number)
;; Purpose: Square the given list of numbers
(define (sqr-list L) (mapf L sqr))
;; (listof ir) → (listof string)
;; Purpose: Extract the name in the given list of ir
(define (names ls) (mapf L ir-name))
```

Students appreciate the reduction in repetition and the elegance of the resulting code (especially if the programming language provides the abstract function they discovered).

6.2 Abstract Classes and Inheritance

Students invariably notice repetitions and ask for abstraction. For example, they notice that in Figures 6a to 6b the `cons` method is exactly the same. This provides the opportunity to motivate abstract classes and inheritance. Students are presented with the following design recipe for abstraction:

```
1. Compare classes and mark similarities
2. Define an abstract class containing the similarities
3. Have classes extend abstract class to inherit similarities
4. Refactor code to create similarities
```

This design recipe is strikingly similar to the design recipe for functional abstraction, which puts students at ease. In Step 1, instead of looking for differences students must look for similarities. In Step 2, the similarities are migrated to an abstract class. These similarities may include methods and instance variables of the same type. If the original classes implement an interface then the abstract class must implement the same interface. In step 3, students associate the original classes with the new abstract class. In `Java` this is done by using `extends`. That is, the original classes extend the abstract class. In this manner, all the methods and the instance variables in the abstract class are inherited and available in the original classes. It is important to explain to students that abstract classes can not be instantiated.

[5]Tests are omitted due to limited space.

Applying the first three steps of the design recipe to the classes in Figures 6a to 6b yields code structured as follows:

```
abstract class AList<X> implements IList<X>
{IList<X> cons(X v)
 {return(new NMTLIST<X>(v, this));}}

class mtList<X> extends AList<X>
{ ... }

class consList<X> extends AList<X>
{ ... }
```

The only changes in the classes is the migration of `cons` to the abstract class and the use of `extends` in the class headers.

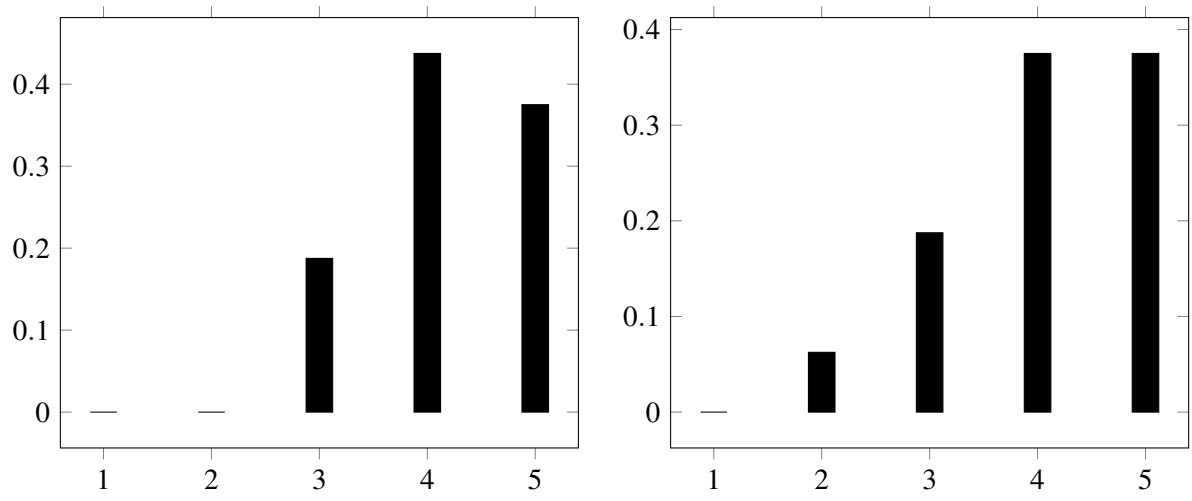
6.3 Code Refactoring

At a first glance there are no more similarities among the classes after abstracting away `cons`. At this point students are introduced to code refactoring. Students are explained that code refactoring changes existing code without changing the semantics. In our context, refactoring is used to create similarities among classes so that they can be abstracted away. Students are explained that they may write methods in terms of other methods much like `sqr-list` and `names` above are rewritten using `mapf` (or the built-in `map`). If refactoring a method across all classes in a union type yields repetitions then they may be abstracted away.

As an example students are asked to consider the implementation of `equals` in Figures 6a to 6b. The code clearly does not look the same. We can observe, however, that if the type of `this` and the given list are different then the answer is `false`. Otherwise, `foldl` may be used to test for equality. Now that the equals method in both classes is the same it may be migrated to the abstract class to yield:

```
abstract class AList<X> implements IList<X>
{IList<X> cons(X v)
 {return(new NMTLIST<X>(v, this));}

 boolean equals(IList<X> l)
 {if ((this.isEmpty() && !l.isEmpty()) ||
       (!this.isEmpty() && l.isEmpty()))
  {return(false);}
  else {return(this.foldl((x, r) ->
                          {try {return(x.equals(l.first()) && r);}
                           catch(Exception e)
                           {System.out.println("..." + e.getMessage());
                            return(false);}},
        true));}}}
```



(a) Better Problem Solver.

(b) Feel Empowered.

Figure 7: Student Feedback I.

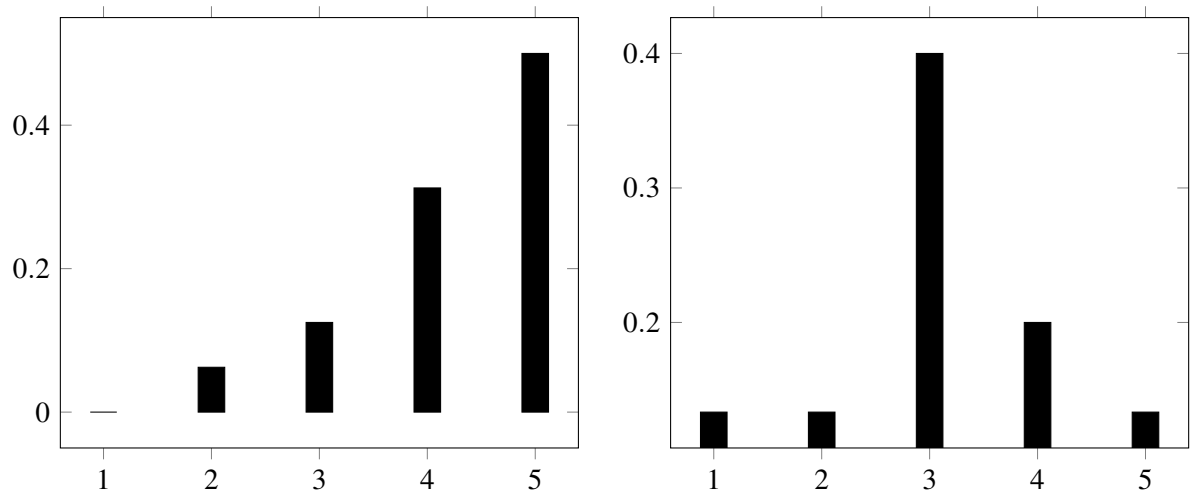
7 Student Feedback

Both quantitative and qualitative data was gathered from students in the course at the end of the semester. The class consisted of 16 students (all male given that, unfortunately, we had no female enrollment in the course). The average age of the students is 19.56. All the students are Computer Science majors except one majoring in English.

Students were asked *Do you feel you are a better problem solver by knowing how to design and implement `OO` programs?* on a scale from 1 (Not at all) to 5 (Very much so). Figure 7a summarizes the responses. We observe that all students responded positively (range 3-5) affirming that they feel they are better problem solvers. In fact, students felt quite strongly about this with 81% of students in the 4-5 range. Closely related to this students were asked *Do you feel empowered by knowing how to design and implement `OO` programs?* on a scale from 1 (Not at all empowered) to 5 (Very much empowered). The bar chart in Figure 7b summarizes the responses. We observe that students overall feel empowered with 94% of responses in the 3-5 range. In fact, students feel quite empowered as evidenced by 75% of responses in the 4-5 range. This data attests that students feel that they have acquired new and useful skills through the material presented.

As mentioned in Section 1 a potential concern was students becoming bored by having a follow-up design-based course that relies heavily on design recipes. After all, it is possible that students may not be interested in rehashing design recipes albeit in a different programming paradigm. To ascertain students' feelings they were asked *How intellectually stimulating is `OO` programming?* on a scale from 1 (Not at all stimulating) to 5 (Very Stimulating). The responses are displayed in Figure 8a. Students find the material intellectually stimulating with 94% responding in the 3-5 range. Once again, we may observe that students in general feel very strongly as may be observed with 81% of the responses in the 4-5 range. This strongly suggests that students are not bored by studying applications of design recipes in a new paradigm. In fact, it confirms that students need to see how the design skills they have internalized apply in a different programming context (i.e., a different programming language and paradigm).

Another concern mentioned in Section 1 is whether or not this material represented a smooth tran-



(a) Intellectually Stimulating.

(b) Smooth Transition.

Figure 8: Student Feedback II.

sition from 2 first-year courses a la `HtDP` to an `OOP` course. To ascertain students' attitudes they were asked *How smooth is the transition from the Racket student languages to Java?* on a scaled from 1 (Not at all smooth) to 5 (Extremely smooth). The responses are summarized in Figure 8b. Overall, students feel the transition is smooth with 73% of responses in the 3-5 range. Only one third of the students feel very strongly (responses in the 4-5 range) that the transition was smooth. About a quarter of the respondents (26%) feel that the transition was not relatively smooth. The biggest problem students had was navigating `Java` error messages, which they did not find very useful. This was followed by the lack of much feedback on failed tests and the syntax of the language (especially λ-expressions).

On the qualitative side students were asked *What was the worst part of the course? What did you like the least?* Sample responses are:

- Learning a new syntax though it gets easier with time.
- Lambda functions in `Java`. They're much more difficult to understand and implement.
- Everything we learned I hated it all. I loved programming in AP CS, but I hate this.

These responses confirm the instructor's impressions that students did not like `Java`'s verbosity and really did not like the syntax for λ-expressions. The third response above is not typical, but does occur sometimes with students that took a `Java` programming course in high school. Some of these students expect to be given code to edit or programming problems that are tightly-coupled with an example done in class. Asking them to design the solution to a problem from scratch does not fulfill their expectations and they require more attention to help them overcome their frustration. Helpful techniques to address their needs include encouraging them to work in groups and to attend tutoring. Rather surprisingly, it is worth noting that students did not complain or raise concerns about having a `Java` textbook that is tightly-coupled with design-based programming. Students do inquire about such a textbook when they get stuck, but this shortcoming was not elevated to the level of a complaint in this group of students.

Students were also asked *What was the best part of this course? What did you like best?*. Sample responses are:

- Abstraction was awesome. Interfaces were cool. Creating objects was cool.

- Learning to implement our knowledge into object oriented design.
- Solving problems with code.

The first response above is common in spirit among other responses given. It points to students truly feeling they understand the technology (i.e., programming constructs) that they are using. The second response is also, in spirit, similar to other responses. Students welcomed being taught how their design experience is applicable to program development using a popular language in industry. The third comment above is not uncommon. In general, students find problem solving fun when they have steps (i.e., a design recipe) to guide them.

8 Concluding Remarks and Future Work

This article presents part of the methodology developed to introduce students to some abstractions and programming constructs available in an object-oriented programming language. In particular, it discusses how to introduce students to classes, interfaces, generic programming, inheritance, polymorphic dispatch, λ-expressions, and abstract classes. Although the approach was developed to provide a smooth transition from a design-based introduction to programming using functional programming to design-based object-oriented programming, the methodology has proven successful with transfer students (i.e., students that transfer from other universities without the benefit of a designed-based course in their background). The data collected from students to date indicates that the approach is well-received and that students find the mentioned transition to be smooth.

Future work includes developing a methodology to introduce students to modifiers that restrict access to methods and instance variables, to Iterators, and to subtype specializations. In addition, future work includes developing a methodology to introduce students to design patterns as a mechanism to abstract away frequent steps OO programmers perform during software development.

References

[1] Eric Allen, Robert Cartwright & Brian Stoler (2002): *DrJava: A Lightweight Pedagogic Environment for Java. SIGCSE Bull.* 34(1), p. 137–141, doi:10.1145/563517.563395.

[2] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt & Shriram Krishnamurthi (2001): *How to Design Programs: An Introduction to Programming and Computing*, First edition. MIT Press, Cambridge, MA, USA.

[3] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt & Shriram Krishnamurthi (2018): *How to Design Programs: An Introduction to Programming and Computing*, Second edition. MIT Press, Cambridge, MA, USA.

[4] Kathi Fisler (2014): *The Recurring Rainfall Problem*. In Quintin I. Cutts, Beth Simon & Brian Dorn, editors: *International Computing Education Research Conference, ICER 2014, Glasgow, United Kingdom, August 11-13, 2014*, ACM, pp. 35–42, doi:10.1145/2632320.2632346.

[5] William Ford & William Topp (1996): *Data Structures with C++*, first edition. Prentice Hall.

[6] Michael T. Goodrich, Roberto Tamassia & Michael H. Goldwasser (2014): *Data Structures and Algorithms*, 6 edition. JohnWiley & Sons, Inc.

[7] Kathryn E. Gray & Matthew Flatt (2003): *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, Association for Computing Machinery, New York, NY, USA, p. 170–177, doi:10.1145/949344.949394.

[8] C. A. R. Hoare (1969): *An Axiomatic Basis for Computer Programming*. *Commun. ACM* 12(10), pp. 576–580, doi:10.1145/363235.363259. Available at `http://doi.acm.org/10.1145/363235.363259`.

[9] C.A.R. Hoare & H. Jifeng (1998): *Unifying Theories of Programming*. Prentice Hall series in computer science, Prentice Hall.

[10] Michael Kölling, Bruce Quig, Andrew Patterson & John Rosenberg (2003): *The BlueJ System and its Pedagogy*. *Comput. Sci. Educ.* 13(4), pp. 249–268, doi:10.1076/csed.13.4.249.17496.

[11] Ben Lerner (2021): *Fundamentals II: Introduction to Class-based Program Design*. Available at `https://course.ccs.neu.edu/cs2510/`. Accessed 2021-06-10.

[12] Marco T. Morazán (2011): *Functional Video Games in the CS1 Classroom*. In Rex Page, Zoltán Horváth & Viktória Zsók, editors: *Trends in Functional Programming: 11th International Symposium, TFP 2010, Norman, OK, USA, May 17-19, 2010. Revised Selected Papers*, Lecture Notes in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 166–183, doi:10.1007/978-3-642-22941-1_11.

[13] Marco T. Morazán (2014): *Functional Video Games in CS1 III*. In Jay McCarthy, editor: *Trends in Functional Programming: 14th International Symposium, TFP 2013, Provo, UT, USA, May 14-16, 2013, Revised Selected Papers*, *Lecture Notes in Computer Science* 8322, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 149–167, doi:10.1007/978-3-642-45340-3_10.

[14] Marco T. Morazán (2015): *Generative and Accumulative Recursion Made fun for Beginners*. *Comput. Lang. Syst. Struct.* 44, pp. 181–197, doi:10.1016/j.cl.2015.08.001.

[15] Marco T. Morazán (2018): *Infusing an HtDP-based CS1 with distributed programming using functional video games*. *Journal of Functional Programming* 28, p. e5, doi:10.1017/S0956796818000059.

[16] Marco T. Morazán (2020): *How to Design While Loops*. *Electronic Proceedings in Theoretical Computer Science* 321, p. 1–18, doi:10.4204/eptcs.321.1.

[17] Emmanuel Schanzer, Kathi Fisler & Shriram Krishnamurthi (2018): *Assessing Bootstrap: Algebra Students on Scaffolded and Unscaffolded Word Problems*. In Tiffany Barnes, Daniel D. Garcia, Elizabeth K. Hawthorne & Manuel A. Pérez-Quiñones, editors: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education, SIGCSE 2018, Baltimore, MD, USA, February 21-24, 2018*, ACM, pp. 8–13, doi:10.1145/3159450.3159498.

[18] Robert Sedgewick & Kevin Wayne (2008): *Introduction to Programming in Java: An Interdisciplinary Approach*, 6 edition. Pearson Education, Inc.

[19] Sam Tobin-Hochstadt & David Van Horn (2013): *From Principles to Practice with Class in the First Year*. In Philip K. F. Hölzenspies, editor: *Proceedings Second Workshop on Trends in Functional Programming In Education, TFPIE 2013, Provo, Utah, USA, 13th May 2013*, *EPTCS* 136, pp. 1–15, doi:10.4204/EPTCS.136.1.

[20] Paul T. Tymann & G. Michael Schneider (2004): *Modern Software Development Using Java*. Thomson Brooks/Cole.