Introduction to Functional Classes in CS1

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

Students introduced to programming using a design-based approach and a functional programming language become familiar with first-class functions. They rarely, however, connect first-class functions to objects and object-oriented program design. This is a missed opportunity because students inevitably go on to courses using an object-oriented programming language. This article describes how students are introduced to objects within the setting of a design-based introduction to programming that uses a functional language. The methodology exposes students to interfaces, classes, objects, and polymorphic dispatch. Initial student feedback suggests that students benefit from the approach.

1 Introduction

Design-based introduction to programming courses using a functional programming language, as put forth in the textbook *How to Design Programs* (HtDP), are well-established and have become popular. In such courses students learn type-based design through the use of *design recipes*–a series of steps, each with a concrete result, that take the student from a problem statement to a well-designed and tested solution. Among other things, students are exposed to first-class functions. That is, students learn that functions may be data (i.e., functions may be passed to and returned by a function). For instance, the following Racket function returns a function for the composition of, f and g, two given functions:

```
;; (X → Y) (Y → Z) → (X → Z)
;; Purpose: Return a function for (f ∘ g)(x)
(define (compose-f-g f g)
  ;; X → Z
  ;; Purpose: Return (f ∘ g)(x) for the given X-value
  (λ (an-x) (f (g an-x))))

;; number → number
;; Purpose: Add 2 to the given number
(define add2 (compose-f-g add1 add1))

;; number → number
;; Purpose: Return the given number
(define id   (compose-f-g add1 sub1))

(check-expect (add2 3)  5)
(check-expect (id 10)  10)
```

The signature states that the first input is a function that maps an element of type X to an element of type Y, that the second input is a function that maps an element of type Y to an element of type Z, and that the returned value is a function that maps an element of type X to an element of type Z. The function header has two parameters `f` and `g` in accordance with the signature. The body is a λ-expression that defines a function that takes as input a value of type `X` and returns a value of type `Z`. This λ-expression implements `(f ∘ g)(x)` as stated in its purpose statement. It has a parameter, `an-x`[1], and applies `f` to the result of applying `g` to `x`. Observe that the returned function satisfies `compose-f-g`'s signature. Tests are written for functions created using `compose-f-g`. In the example above, two functions are defined: one to add 2 to its input and the identity function for numbers. Test are written for these functions using `check-expect`[2].

The other side of the coin is rarely explored to any depth. That is, thinking of data as a function is not emphasized. This is a missed opportunity to introduce beginners to object-oriented concepts that they will invariably see in future courses. Such an introduction is important because it facilitates the transition to object-oriented programming (`OOP`) and design. Furthermore, it helps address the concerns of `OOP` instructors that worry about beginners not being exposed to object-oriented programming in their introduction courses. At the heart of the idea is having a returned function implement an interface and use message-passing to provide services. Students are started with the implementation of compound data of finite size that does not have variety (i.e., no subtypes) and then are moved onto implementing union types (i.e., with subtypes) including compound data of arbitrary size.

The article is organized as follows. Section 2 briefly discusses related work. Section 3 discusses the students' background. Section 4 discusses how students are introduced to interfaces, classes, and objects by implementing structures using functions. Section 5 presents a design recipe for interfaces. Section 6 discusses the implementation of union types and polymorphic dispatch using the design recipe for interfaces. Section 7 presents initial student feedback. Finally, Section 8 presents conclusions and directions for future work.

2 Related Work

The most common source students have to introduce them to objects are `OOP` textbooks. Such textbooks traditionally emphasize that classes and objects combine code and data to be manipulated as illustrated by the following examples:

- An object is a structure for incorporating data and the procedures for working with that data [1].
- A class is a group of objects that share common state and behavior. A class is an abstraction or description of an object. An object, on the other hand, is a concrete entity that exists in space and time [13].
- Every object is an instance of a class, which serves as the type of the object and as a blueprint, defining the data which the object stores and the methods for accessing and modifying that data [5].
- Objects are used to combine data with procedures that operate on that data [11].

As the reader can appreciate `OOP` textbooks emphasize the encapsulation of data and methods to define classes and objects. These textbooks then proceed with examples illustrating how to use classes and how to create objects. The unstated assumption is that beginning students absorb design principles by studying code samples. Like the approach taken by many `OOP` textbooks the work presented in this

[1] Students are encouraged to use variable names that help readers of the code understand what is being represented. In this case, `an-x` suggests a value of type `X`.

[2] The forms `check-expect` and `check-within`, among others, are used by students to write unit tests in the `Racket` student languages.

article emphasizes the definition of behavior. Students are taught to combine data and functions using encapsulation to create a package called a class. In contrast, however, program design and the proper use of OOP abstractions are emphasized from the beginning. That is, behavior is defined in an interface, an implementation of an interface is called a class, and an instance of a class is an object. Observe that such an approach truly separates specification from implementation in the mind of beginners.

The unpublished textbook `How to Design Classes` (HtDC) [4] is closely coupled with a domain specific language (DSL) known as `ProfessorJ` [6]. It first introduces students to classes as a mechanism to define a compound datatype that may have many fields (similar to a structure). Students are explained that every class has a constructor that initializes the fields and produces an object–an instance of the type defined by the class. 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. The design of methods is not tackled until students develop some expertise defining classes that only have a single constructor. 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. In contrast, however, we use `Racket` syntax (specifically, the intermediate student language with `lambda`) and rely quite heavily on the familiarity students have developed with this syntax to introduce them to OOP concepts. That is, a new specialized DSL is not required.

Another design-based approach developed at Northeastern University teaches students (that have used HtDP in their first course) using a hybrid approach [12]. The beginning of the second course proposes the use of several DSLs 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 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 that define behavior. 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 subtype. The transition to `Java` occurs in the middle of the semester. This transition is made to functional `Java` first to make programs as similar as possible to those developed using the OOP DSLs embedded in `Racket`. Later mutation and looping constructs are presented. In a similar spirit, the work presented in this article introduces objects as encapsulated data and functions. In addition, it also starts with parallelism between structures and classes and then moves to union types. In contrast, the work presented here is not part of an OOP course. Instead, the work presented is intended to introduce students to object-oriented abstractions within the context of a first programming course using the program by design methodology put forth in HtDP.

3 Student Background

At Seton Hall University, the introductory Computer Science 2-course sequence focuses on problem solving using a computer [7, 9]. 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, primitive functions, and library functions to manipulate images (i.e., the image teachpack). During this introduction, students are taught about variables, defining their own functions, and the importance of writing signatures, purpose statements, and tests. The next step of the course introduces students to data analysis and programming with compound data of finite size (i.e., structures). At this point, students are introduced to the first design recipe. Building on this experience, students develop expertise in 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. The first semester ends with a module on distributed programming [8, 10].

As part of the module on functional abstraction students are introduced to λ-expressions and curried functions. Both of these topics are pivotal to introducing students to interfaces and objects. Students understand that a λ-expression evaluates to a function that consumes input and returns a value. With this understanding students are introduced to curried functions and how they provide the convenience of not consuming all of their input at once. For example, consider a set of functions as the following to scale a list of numbers:

```
(define (scale-by-2 L)
  (map (λ (x) (* 2 x)) L))

(define (scale-by-5 L)
  (map (λ (x) (* 5 x)) L))

(define (scale-by-3 L)
  (map (λ (x) (* 3 x)) L))
        ⋮
```

Observe that there is a great deal of repetition among the functions. Students learn to eliminate such repetitions using functional abstraction. For the set of functions above, the result of functional abstraction is a curried function that takes as input the scaling factor and returns a function to scale a list by that factor as follows:

```
;; number → ((listof number) → (listof number))
(define (make-list-scaler sc)
  (λ (L) (map (λ (x) (* sc x)) L)))
```

The original definitions may be refactored and tested as follows:

```
(define scale-by-2 (make-list-scaler  2))

(define scale-by-5 (make-list-scaler  5))

(define scale-by-3 (make-list-scaler  3))
    ⋮

(check-expect (scale-by-2 '()) '())
(check-expect (scale-by-2 '(1 2 3)) '(2 4 6))
(check-expect (scale-by-5 '()) '())
(check-expect (scale-by-5 '(4 0)) '(20 0))
(check-expect (scale-by-3 '()) '())
(check-expect (scale-by-3 '(-2 8)) '(-6 24))
    ⋮
```

```
;; A 3Dposn is a structure: (make-3Dposn number number number)
(define-struct 3Dposn (x y z))

;; Sample instances of 3Dposn
(define ORIGIN  (make-3Dposn 0 0 0))
(define A3DPOSN (make-3Dposn 2 3 5))

;; 3Dposn → number
;; Purpose: Return the distance to the origin of the given 3Dposn
(define (dist-origin a-3dposn)
  (sqrt (+ (sqr (3Dposn-x a-3dposn))
           (sqr (3Dposn-y a-3dposn))
           (sqr (3Dposn-z a-3dposn)))))

;; Sample expressions for dist-origin
(define ORIGIND  (sqrt (+ (sqr (3Dposn-x ORIGIN))
                          (sqr (3Dposn-y ORIGIN))
                          (sqr (3Dposn-z ORIGIN)))))
(define A3DPOSND (sqrt (+ (sqr (3Dposn-x A3DPOSN))
                          (sqr (3Dposn-y A3DPOSN))
                          (sqr (3Dposn-z A3DPOSN)))))

;; Tests for dist-origin
(check-within (dist-origin ORIGIN)  ORIGIND  0.01)
(check-within (dist-origin A3DPOSN) A3DPOSND 0.01)
(check-within (dist-origin (make-3Dposn 10 20 30))  37.42 0.01)
```

Figure 1: Distance to the Origin Program for a `3Dposn`.

Most functional programmers are likely to consider `make-list-scaler` a function that returns a function. This is technically correct but it can also be described using `OOP` lingo. We may also say that `make-list-scaler` is a *class* that returns `objects` that scale a list of numbers by a given scalar. This connection is not automatically made by students in the course. It is, therefore, necessary to make this connection explicitly in class. The result is twofold: students gain more interest in the techniques being taught given that they see how they apply beyond their first course and students arrive at their first `OOP` course with an understanding of some of the abstractions object-oriented languages emphasize.

4 Structures as Interfaces

Students are reminded that a data definition defines a type and in the case of compound data, like a structure, it also defines the types of the components. It states nothing about the valid type operations. The valid operations on an instance of the type may be defined in an *interface*. An interface defines the behavior of a defined type. In other words, an interface specifies the operations that are valid on a type. Students are asked to consider, for example, the problem of computing the distance to the origin for a given point on a three-dimensional plane. Following the steps of the design recipe yields the program displayed in Figure 1. There are a few unstated assumptions in the program. First, it is assumed that there is a function to construct a `3Dposn`. Second, it is assumed that there are selector functions for the components of a `3Dposn`. Third, the values of `x`, `y`, `z`, and the constructor/selector functions are stored separately from the function `dist-origin`. The fact that students know that constructor and selector functions exists is only because they have been told they are created for them when they define a structure.

In addition to defining a type, an interface is developed to explicitly define the expected behavior. An interface outlines the valid operations and the returned type. For a `3Dposn` the interface is:

Request `x`**:** `number`

Request `y`**:** `number`

Request `z`**:** `number`

Request `distance`**:** `number`

The interface makes it clear to any reader or user which are the valid operations on a `3Dposn`. Observe that an interface says nothing about how a data type and its valid operations are implemented.

Students are asked to take a moment to ponder what has just been done. The data definition and the interface together explicitly relate `x`, `y`, `z`, `3Dposn-x`, `3Dposn-y`, `3Dposn-z`, and `dist-origin`. If they are related then we ought to be able to encapsulate them into a single package. Whenever a `3Dposn` is constructed the package returned ought to be able to perform all the operations in the interface. Encapsulation is a practice that students are familiar with given that they have previously been exposed to `local`-expressions.

Students are explained that to provide functionality a technique called *message-passing* is used. An interface is implemented by a constructor function called a *class* that returns a message-processing function. This is a curried function that receives as input a message requesting a service. For example, this function may get the message `'getx` requesting the `x` value of the `3Dposn`. An interface, therefore, must specify the messages used to request a service. We can now refine the `3Dposn` interface to be:

```
'getx: number
'gety: number
'getz: number
 'd2o: number
```

Observe that there is a unique message (in this case a symbol) associated with each service. The idea is that the message-processing function determines what value to compute by examining the message it gets as input. Note that embedded in the interface definition is a data definition for a `message`. A message is an enumeration type: either `'getx`, `'gety`, `'getz`, or `'d2o`.

Students are explained that a constructor function (i.e., class) that implements an interface encapsulates the values of and the operations on a type. It defines a constructor for instances of a type. The value returned by a class is called an *object*. An object is an instance of an interface and knows how to perform all the services in the interface using message-passing. The `3Dposn` class is displayed in Figure 2. The class takes as input 3 numbers and returns a `3Dposn`. It is named `make-3Dposn` to easily identify its role as a constructor for `3Dposns`. Its body is a `local`-expression that defines an auxiliary function for any value that needs to be computed (in this case only distance to origin) and the message-processing function called `manager`. The `manager` takes as input a message and returns (the value of) a service defined in the interface. The body of `manager` is a `match`-expression to distinguish the message varieties. If a service requires no computation a value is directly returned. If computation is required

```
(require 2htdp/abstraction)

;; number number number → 3Dposn
;; Purpose: Return a 3Dposn object
(define (make-3Dposn x y z)
  (local [;; 3Dposn → number Purpose: Return distance to origin
          (define (dist-origin x y z) (sqrt (+ (sqr x) (sqr y) (sqr z))))
          ;; message → 3Dposn service throws error
          ;; Purpose: To manage messages for a 3Dposn
          (define (manager m)
            (match m
              ['getx x]
              ['gety y]
              ['getz z]
              ['d2o  (dist-origin x y z)]
              [else (error (string-append "Unknown message to 3Dposn: "
                                          (symbol->string m)))]))]
    manager))

;; Sample 3Dposn objects
(define ORIGIN  (make-3Dposn 0 0 0))
(define A3DPOSN (make-3Dposn 2 3 5))

;; Tests for 3Dposn
(check-within (ORIGIN  'getx) 0    0.01)
(check-within (A3DPOSN 'gety) 3    0.01)
(check-within (ORIGIN  'getz) 0    0.01)
(check-within (A3DPOSN 'd2o)  6.16 0.01)
(check-error (A3DPOSN 'move-r)
             "Unknown message to 3Dposn: move-r")
```

Figure 2: Interface Implementation for 3Dposn.

(like computing the distance to the origin) a local function is called. In this case, `manager` is a guarded function that throws an error when the received input is not a `message`. We can observe that `manager` is also an object given that it knows how to compute all the services in the interface and, therefore, the `local`-expression returns it.

Testing interfaces requires defining one or more objects and writing tests to check that services are correctly provided. In Figure 2 two `3Dposn` objects are defined. The tests check the result obtained from passing each message to an object. For example, the x-coordinate of `ORIGIN` is obtained using `(ORIGIN 'getx)`–passing the message `'getx` to `ORIGIN` (akin to the `Java` syntax: `ORIGIN.getx()`). The expected value is 0.

Students suddenly realize that a `3Dposn`, which they have always thought of as data, is a function. Specifically, it is an instance of the curried function `manager` (i.e., an object) that is specialized for the values of `x`, `y`, `z` given to `make-3Dposn`. `ORIGIN` is a `3Dposn` object in which `x` = `y` = `z` = 0. `A3DPOSN`

```
;; 3Dposn → number
;; Purpose: Return the x of the given 3Dposn
(define (3Dposn-x a-3dposn) (a-3dposn 'getx))

;; Tests for 3Dposn-x
(check-within (3Dposn-x ORIGIN)  0  0.01)
(check-within (3Dposn-x A3DPOSN) 2 0.01)
(check-within (3Dposn-x (make-3Dposn 10 20 30)) 10 0.01)

;; 3Dposn → number
;; Purpose: Return the y of the given 3Dposn
(define (3Dposn-y a-3dposn) (a-3dposn 'gety))

;; Tests for 3Dposn-y
(check-within (3Dposn-y ORIGIN)  0  0.01)
(check-within (3Dposn-y A3DPOSN) 3  0.01)
(check-within (3Dposn-y (make-3Dposn 10 20 30)) 20 0.01)

;; 3Dposn → number
;; Purpose: Return the z of the given 3Dposn
(define (3Dposn-z a-3dposn) (a-3dposn 'getz))

;; Tests for 3Dposn-z
(check-within (3Dposn-z ORIGIN)  0  0.01)
(check-within (3Dposn-z A3DPOSN) 5 0.01)
(check-within (3Dposn-z (make-3Dposn 10 20 30)) 30 0.01)

;; 3Dposn → number
;; Purpose: Return distance to origin of given 3Dposn
(define (dist-origin a-3dposn) (a-3dposn 'd2o))

;; Tests for dist-origin
(check-within (dist-origin ORIGIN)  0    0.01)
(check-within (dist-origin A3DPOSN) 6.16 0.01)
(check-within (dist-origin (make-3Dposn 10 20 30)) 37.42 0.01)
```

Figure 3: Wrapper Functions for `3Dposn`.

is a `3Dposn` object in which `x` = 2, `y` = 3, and `z` = 5. Students clearly see that, just like functions can be data, data can be a function and an object *is* a function.

4.1 Improving the Human Interface

Message-passing may reduce the readability of the code. For example, does `(A3DPOSN 'd2O)` communicate to others that this expression represents `A3DPOSN`'s distance to the origin? Unless you are intimately familiar with the message-passing protocol it is likely that this expression is meaningless. Furthermore, it is unlikely that any programmer (including the student that wrote it) will permanently remember the message-passing protocol in the near and far future. This will make it unnecessarily more difficult to refine the program.

To mitigate this problem, *wrapper functions* for the services provided by an interface may be written. A wrapper function hides the details of the implementation. In this case, it hides the details of message-passing. The idea is to allow programmers to use 3Dposns without forcing them to know how they are implemented. A wrapper function is needed for each service in the interface. It takes as input an object (and any additional inputs if any) and its body applies the object to the appropriate message. Wrapper functions are designed following the steps of the design recipe.

Figure 3 displays the wrapper functions for 3Dposn. Adding this code to the one displayed in Figure 2 allows programmers to use a nicer version of the defined interface. Instead of explicitly using message-passing, they can use the wrapper functions. Observe that now programmers have the same interface as the one used in Figure 1. Writing wrapper functions does not provide a programmer with new computational powers, but it is an abstraction that liberates a programmer from the details of a message-passing protocol.

4.2 Services that Require More Input

After an interface is implemented it may be necessary to add more services. This means expanding the message-processing function and, if necessary, designing (local) auxiliary functions. If a value may be computed using only the information stored in an object then adding a service is the same as what is done for dist-origin for 3Dposn. For example, adding a service to determine if a given 3Dposn object is on the x-axis may de done by comparing the x-value to 0.

If a service requires further input the answer cannot be computed using only the values stored in an object. That is, the object providing the service (usually referred to as this) needs information beyond that which it stores. For instance, consider adding a service that computes the distance to a given 3Dposn object. In addition to this object another 3Dposn object is needed. This other object is unknown when this is constructed and, therefore, cannot be provided as input. In this regard, it is similar to receiving a message given that there is no way to know which messages will actually be received as input. The solution for messages is to make a curried function that consumes the extra input (i.e., a message). The same design tactic may be employed to add services that require extra input. The interface must return a function that consumes the extra input.

To illustrate the technique let us add a service to compute the distance of this 3Dposn to a given 3Dposn. The first step is to update the interface as follows:

```
'getx: number
'gety: number
'getz: number
 'd2o: number
 'd2p: 3Dposn →  number
```

The data definition of a message is expanded to include 'd2p for the new distance-computing service. Given that extra input is needed the interface returns a function that consumes the extra input, a 3Dposn, and that returns a number for the distance between this and the given 3Dposn.

The next step is to refine the manager function to include the new service. This means adding a stanza for the new service to the match-expression as follows:

```
;; message → 3Dposn service throws error
;; Purpose: To manage messages for a 3Dposn
(define (manager m)
  (match m
    ['getx x]
    ['gety y]
    ['getz z]
    ['d2o  (dist-origin x y z)]
    ['d2p  distance]
    [else
     (error
      (string-append "Unknown message to 3Dposn: " (symbol->string m)))]))
```

The new stanza matches the new message and returns the `distance` function (yet to be written). The `distance` function must satisfy the return type specified in the interface definition.

The `distance` function may now be designed and implemented. Keep in mind that this is a local function inside the `3Dposn` class. Therefore, this function has in scope all the variables declared in the class. This is where the power of currying is exploited. The previous inputs (`x`, `y`, and `z`) are used to compute the distance to the `3Dposn` received as input. After following the steps of the design recipe the following local function is added to the `3Dposn` class:

```
;; 3Dpson → number
;; Purpose: Compute distance from this to given 3Dposn
(define (distance a-3dposn)
  (sqrt (+ (sqr (- x (3Dposn-x a-3dposn)))
           (sqr (- y (3Dposn-y a-3dposn)))
           (sqr (- z (3Dposn-z a-3dposn))))))
```

Observe that this function uses the coordinates of `this` and of the given `3Dposn` to compute the distance.

The final step is to develop a wrapper function for the new distance service. As before, this is done following the steps of the design recipe. The resulting function is:

```
(define B3DPOSN (make-3Dposn 1 1 1))

;; 3Dposn 3Dposn → number
;; Purpose: Return distance between given 3Dposns
(define (3Dposn-distance p1 p2) ((p1 'd2p) p2))

;; Sample expressions for 3Dposn-distance
(define ORIGINDP  ((ORIGIN  'd2p) ORIGIN))
(define A3DPOSNDP ((A3DPOSN 'd2p) B3DPOSN))

;; Tests for 3Dposn-distance
(check-within (3Dposn-distance ORIGIN ORIGIN) ORIGINDP 0.01)
(check-within (3Dposn-distance A3DPOSN B3DPOSN) A3DPOSNDP 0.01)
(check-within (3Dposn-distance (make-3Dposn 10 20 30)
                               (make-3Dposn 2  3  4))
              32.07
              0.01)
```

Students are encouraged to first write sample expressions that use the interface because they are new to message passing. They can then perform abstraction over the sample expressions to write the function much like they have done to discover, for example, `map` and `filter`.

The reader can appreciate that students are exposed to many prevalent concepts in `OOP`: interfaces, classes, objects, constructors, observers, and message-passing. The only difference here is that a different syntax is used. Students walk away with an understanding of elements emphasized in `OOP`.

5 A Design Recipe for Interfaces

Building on the work above, the systematic steps to design the implementation of an interface may now be enumerated. The design recipe for an interface is:

1. Identify the values that must be stored and the services that must be provided.
2. Develop an interface data definition and a data definition for messages.
3. Develop a function template for the class that consumes the values that must be stored and whose body is a `local`-expression returning the message-processing function.
4. Specialize the signature, purpose, class header, and message-processing function.
5. Design and implement local auxiliary functions needed by the message-passing function.
6. Write and test a wrapper function for each service.

The first step is problem analysis. It asks to identify the information that specializes each object and the services that an object must provide. For every piece of information identified there must be a parameter. The services must at least include a selector for each piece of information that specializes an object. The second step asks you to develop an interface data definition and a message data definition. There must be a message for each service identified in Step 1.

The third step asks for the development of a function template for the class. Its parameters correspond to the values identified in Step 1 to specialize an object. The name of the class is also the name of the constructor. The function's body is a `local`-expression that encapsulates all needed functions and that returns the message-processing function. The fourth step has you specialize the definition template for local message-passing function. This function must contain an expression that distinguishes the messages defined in Step 2. The fifth step asks you to develop all the auxiliary functions needed by the message-processing function and make them local. Each auxiliary function is developed using the design recipe.

The sixth step asks for a wrapper function for each service in the interface developed in Step 2. These functions take as input the object that is providing the service and the extra input, if any, required. The body of each wrapper function applies the object providing the service to the appropriate message. If extra input is needed the function returned by the object is applied to the extra input.

6 Implementing Union Types

The implementation of a union type is used to illustrate the steps of the design recipe in practice. We shall use the following data definitions:

```
;; A square, sq, is a an object
;;     (make-sq number symbol symbol)
;; with a length, a mode, and a color.
;; A rectangle, rect, is an object
;;     (make-rect number number symbol symbol)
;; with a width, length, a mode, and a color.

;; A circle, circ, is an object
;;     (make-circ number symbol symbol)
;; with a radius, a mode, and a color.
```

Based on these definitions a union type for geometric shapes may be defined as follows:

```
;; A geometric shape, gs, is either:
;;  1. sq
;;  2. rect
;;  3. circ
```

Following the steps of the design recipe from Section 5 students are shown how to implement this union type using an interface. A point that is emphasized is that designing interfaces for a union type requires individually reasoning about each subtype. This is not a huge intellectual leap for students given that individually reasoning about each subtype is how functions to process a union type are designed.

6.1 Step 1: Values and Services

To implement gs three classes are needed: one for each subtype. The sq class needs to store a number and two symbols. The rect class needs to stores two numbers and two symbols. The circ class needs to store a number and two symbols. This type of analysis is the same as designing an implementation using structures.

The new component is defining the behavior for a union type. The following services are offered by a gs:

- Determine if the gs ia a sq, a rect, or a circ
- Compute the gs's area
- Determine if the area of this gs is larger than a given gs

6.2 Step 2: Interface and Message Definitions

Based on the services outlined in Step 1, 5 return types and 5 message varieties need to be defined. The interface for a gs, including the messages, is:

```
;; A gs is an interface offering:
;;    'is-sq?:  Boolean
;;  'is-rect?:  Boolean
;;  'is-circ?:  Boolean
;;      'area:  number
;;   'bigger?:  gs → Boolean
```

Students easily observe that to compare areas more input is needed and, therefore, a function must be returned.

```
;; ... → gs
;; Purpose: Return a gs object
(define (make-gs ...)
  (local
    [   ⋮
     ;; gs → Boolean
     ;; Purpose: Determine if this' area is larger than given gs' area
     (define (is-this-bigger? a-gs) ...)

     ;; message → service throws error
     ;; Purpose: Provide service for the given message
     (define (manager m)
       (match m
         ['is-sq? ...]
         ['is-rect?  ...]
         ['is-circ?   ...]
         ['area   ...]
         ['bigger?    ...]
         [else
           (error
             (format "Unknown gs service requested: ~s" m))]))]
   manager))
```

Figure 4: The Class Function Template for a gs.

6.3 Step 3: Class Function Template

The class template captures all the similarities that any gs must have. These include the 5 services offered by the interface developed in Step 2. The class function template for a gs is displayed in Figure 4. The signature and purpose statement clearly establish that this class returns a gs object. The body of the class is a `local`-expression that at the very least must encapsulate a function to determine if the area of `this` object is bigger than the area of a given gs and a manager function to process messages. A function is needed for the 'bigger? service because more input is required. Experienced students are free to inline a λ-expression in the body of `manager` to implement this function if they so desire. The body of the `local`-expression returns the `manager` function which is the object that is capable of providing all the services in the interface.

In essence, the template in Figure 4 is the roadmap students follow to implement the needed classes. In this example, students are explained that it is used to write three classes: one for each gs subtype.

6.4 Steps 4 and 5: Class Function Template Specialization

Step 4 of the design recipe asks students to specialize the signature, purpose, class header, and message-processing function of the class function template. Step 5 asks students to write the auxiliary functions needed by the `manager` function. The results of these steps for `sq` are displayed in Figure 5. The class' signature reflects the three types of arguments needed to build a `sq`. The purpose statement is specialized to explicitly state that a `sq` is returned. Observe that this is consistent with the overall design because `sq`

```
(require 2htdp/abstraction)

;; number symbol symbol → sq
;; Purpose: Return a sq object
(define (make-sq length mode color)
  (local
    [;; gs → Boolean
     ;; Purpose: Determine if this' area is larger than given gs' area
     (define (is-this-bigger? a-gs) (> (sqr length) (gs-area a-gs)))

     ;; message → service throws error
     ;; Purpose: Provide service for the given message
     (define (manager m)
       (match m
         ['get-length length]
         ['get-mode   mode]
         ['get-color  color]
         ['is-sq?     #true]
         ['is-rect?   #false]
         ['is-circ?   #false]
         ['area       (sqr length)]
         ['bigger?    is-this-bigger?]
         [else
           (error (format "Unknown gs service requested: ~s" m))]))]
    manager))
```

Figure 5: The sq Class.

is a subtype of gs. The class header defines the constructor's name, make-sq, and has three parameters as suggested by the signature. The specialization of the manager function is the most detailed part of Step 4. Its match-expression processes messages for the components of a sq. Each of these only needs to return the requested value. The rest of the stanzas implement the gs interface. The stanza for 'area returns the area of this square by squaring its length. The stanza for 'bigger? returns a function because more input is required. The auxiliary function is named is-this-bigger?. This completes Step 4 of the design recipe.

For Step 5 of the design recipe only one auxiliary function (i.e., is-this-bigger?) needs to be written. Students design this function following the steps of the design recipe for function development. The most salient feature for the purposes of this article is that the wrapper function to compute the area of a gs, gs-area, is used to compute the area of the given gs. This function, of course, is developed in the next step of the design recipe for interfaces. Students are discouraged to follow their first instinct that tells them to use a conditional to implement this service.

The development of the rect and circ classes follows in the same manner. In the interest of brevity their development is omitted. Once all three classes for geometric shapes are implemented students are explained about dynamic dispatch. They are able to see how a service, like area, is always correctly provided because each class locally defines its implementation.

```
(define SQR1  (make-sq   5 'outline 'green))
(define RECT1 (make-rect 2 3 'solid 'blue))
(define CIRC1 (make-circ 7 'outline 'red))

;; gs → Boolean
;; Purpose: Determine if given gs is a sq
(define (gs-sq? a-gs) (a-gs 'is-sq?))

;; gs → Boolean
;; Purpose: Determine if given gs is a rect
(define (gs-rect? a-gs) (a-gs 'is-rect?))

;; gs → Boolean
;; Purpose: Determine if given gs is a circ
(define (gs-circ? a-gs) (a-gs 'is-circ?))

;; gs → Boolean
;; Purpose: Compute area of given gs
(define (gs-area a-gs) (a-gs 'area))

;; gs gs → Boolean
;; Purpose: Determine if first given gs is bigger than second given gs
(define (gs-bigger? this that) ((this 'bigger?) that))

;; Tests for gs interface
(check-expect (gs-sq? SQR1)    #true)
(check-expect (gs-sq? RECT1)   #false)
(check-expect (gs-sq? CIRC1)   #false)
(check-expect (gs-rect? SQR1)  #false)
(check-expect (gs-rect? RECT1) #true)
(check-expect (gs-rect? CIRC1) #false)
(check-expect (gs-circ? SQR1)  #false)
(check-expect (gs-circ? RECT1) #false)
(check-expect (gs-circ? CIRC1) #true)
(check-expect (gs-area SQR1)   25)
(check-expect (gs-area RECT1)  6)
(check-within (gs-area CIRC1)  153.93 0.01)
(check-expect (gs-bigger? SQR1 RECT1)  #true)
(check-expect (gs-bigger? RECT1 CIRC1) #false)
(check-expect (gs-bigger? CIRC1 SQR1)  #true)
```

Figure 6: Wrapper Functions and Tests for gs.

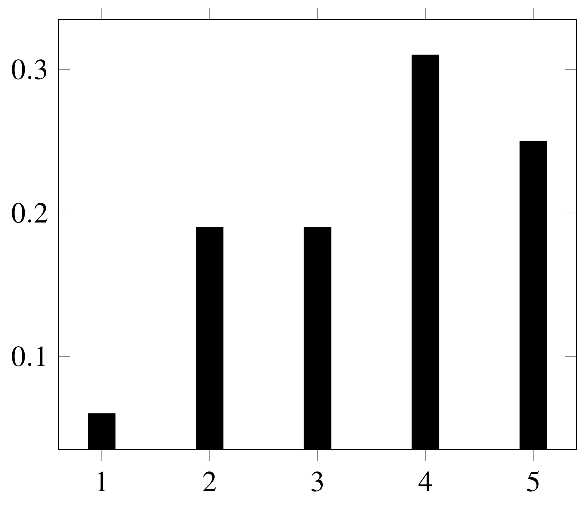

Figure 7: Distribution of Answers for Effective Transition.

6.5 Step 6: Wrapper Functions and Tests

The wrapper functions are designed following the steps of the design recipe for functions. Students must remember that the input to each wrapper function must contain at least one object (in our case a `gs` object). The result of this step is displayed in Figure 6. In the body of a wrapper function an object must be applied to the appropriate message. If extra input is required then the function returned by the object is applied to said input. This is the case for `gs-bigger?`. Students also develop wrapper functions to access the fields of each `gs`-subtype. Observe that this is exactly what needs to be done when fields are private when developing a class, for instance, in `Java`.

Students are encouraged to define sample instances of each `gs-subtype`. These are then used to write the tests for the `gs` interface. This is the strategy followed for the tests in Figure 6. Testing must be thorough and all tests must pass. If any tests fails or testing is lacking students must refine their design.

7 Student Feedback

This section presents the first data obtained from students regarding the transition from an `HtDP`-based course to a `Java`-based `OOP` course. Students that took the `HtDP`-based course in the Fall of 2018 were followed-up on after they took their first `OOP` course in the Fall of 2019 or Fall of 2020. A total of 16 students made such a transition and were asked:

How effective do you feel it is to start with How to Design Programs and then move to learning about how to design classes?

Students answered on a scale from 1 (not at all effective) to 5 (extremely effective).

Figure 7 displays the distribution of responses. We can observe that the distribution is fairly normal (mean = 3.5 and median = mode = 4) with a slight majority of students giving a response above the mean. Overall, students feel that starting with `HtDP` and then moving to `OOP` is effective. 75% of respondents expressed the approach was effective (responses 3-5). More than half, 56% of the students felt strongly about the effectiveness of the approach (responses 4-5). We can also observe that the distribution is fairly homogeneous with an IQR = 1.5 (Q1 = 2.75 and Q3 = 4.25). This indicates that there is little variation among the respondents in relation to the effectiveness of the approach.

Qualitatively, students expressed appreciation for being exposed to classes and objects before *learning `Java`* (their nomenclature). Students that arrived with prior `OOP` experience from high school did express feeling frustrated with the approach at the beginning, but now feel they understand `OOP` much better and are appreciative of their experience.

8 Conclusions and Future Work

This article presents a methodology to introduce beginning students to `OOP` concepts in an `HtDP`-based course before enrolling in their first `OOP` course. At the heart of the idea is to expose beginning students to object-oriented abstractions using the design concepts and syntax that they have learned. To this end, students are introduced to the idea that a data definition and the valid operations on the defined data are called an interface, that the implementation of an interface is a class, and that an instance of a class is an object. In this manner, specification is separated from implementation. A class is implemented using a message-passing function. The use of functions is natural for students following a program by design curriculum and is intended to foreshadow that classes are functions (as exemplified by λ-expressions in `Java`). Students learn to implement union types as the process of implementing a class for each subtype. This leads to their initial exposure to polymorphic dispatch. Initial data collected from students suggests that they find the transition from an `HtDP`-course to `OOP` effective.

Future work includes finding how to make the material more fun for students. Specifically, the implementation of a small video game or simulation would be ideal to spike student interest. In addition, we are exploring how to expose students to abstract classes and inheritance.

References

[1] D. Clark (2008): *An Introduction to Object-Oriented Programming with Visual Basic .NET*. .NET developer series, Apress.

[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] Matthias Felleisen, Matthew Flatt, Robert Bruce Findler, Kathy Gray, Shriram Krishnamurthi & Viera K. Proulx (2012): *How to Design Classes*. Accessed 2022-06-22.

[5] Michael T. Goodrich, Roberto Tamassia & Michael H. Goldwasser (2014): *Data Structures and Algorithms in Java (6. ed.)*. Wiley.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] Bruno R. Preiss (1999): *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. Wiley.

[12] 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.

[13] P.T. Tymann & G.M. Schneider (2004): *Modern Software Development Using Java: A Text for the Second Course in Computer Science*. Thomson-Brooks/Cole.