FP vs. OOP: case studies in misunderstandings

Panicz Maciej Godek
godek.maciek@gmail.com

March 26, 2019

Abstract

Paradigms. Some people say that studying them increases our knowledge and programming abilities. On the other hand, we sometimes witness "paradigm wars", which possibly shows that the knowledge of paradigms doesn't necessarily increase our abilities to understand one another.

A paradigm is a conceptual framework that distinguishes between legitimate questions and nonsensical ones. Paradigms are sometimes also called "thought pattenrs", that is, the ways of thinking that our minds are accustomed to.

We usually arrive at these patterns not by the way of deliberate construction, but by our innate abilities to copy other people's behaviors. It should therefore be no surprise that we're often not even aware that our thinking represents some particular set of patterns, or that we're making some implicit assumptions.

Some people who consider themselves functional programmers and object oriented programmers sometimes claim that "their paradigm" represents fundamental approach to programming. OOP zealots often claim that the way they model systems is "natural", because "the real world" consists of objects that interact with each other and are organized in various forms of hierarchies.

Functional programmers are reluctant towards talking about the mythical "real world", and focus more on the notion of abstraction.

Yet, there is a number of assumptions that most programmers share without questioning them, and often without even realizing them.

How can we discover these assumptions? To what extent are they a limitation to us and to the systems that we build? What alternatives can we propose? How can we deconstruct or transcend the paradigms that are familiar to us?

While I don't know any systematic answer to those questions, I believe that there are certain important aspects of computation that we've been missing so far. With the help of Racket, I'm going to deconstruct some fundamental concepts of functional and object oriented programming and show programming techniques and application prototypes that transcend the traditional way of thinking about software and programming languages.

1 Paradigms

The title of this talk contains some acronyms that are, I believe, commonly known to programmers. Those few letters are typically used to designate some of the so called "paradigms of programming", so before I move on to the specifics, I want to say a few words about paradigms in general.

When we learn new things, we typically imitate other people in their behaviors. This is especially true for languages, where we learn to speak first by repeating other people's words, and only later we analyze those words and discover the rules that govern their use.

Of course, this isn't only true for languages. In our lives we struggle to produce anything of value, and when we succeed and look back at the road that lead us there, the road often turns out to be long and winding. When we explore the territory that we don't know, and if we also don't know exactly the thing we are looking for, it would be naive to expect that we'll find that thing easily and without erring.

Wikipedia defines paradigm as

> a distinct set of concepts or thought patterns, including theories, research methods, postulates, and standards for what constitutes legitimate contribution to a field. [8]

I don't really like this definition, because it misses the "struggle" part of the human endeavour: in order to truly learn, we always need to step into the darkness.

It even may be a mistake to define paradigm, because paradigm is, by its nature, something vague, something that has no strict boundaries.

I believe that it would be better to explain paradigm using a metaphor of a compass or a map or a flashlight that helps us get through some unknown territory.

A compass can be disrupted or inapplicable to our situation, a flashlight can run out of batteries, and a map can be inaccurate. But people rarely say that a compass is better than a flashlight, or that it is more flexible. We understand that these are just tools that have their purpose, and that they all complement one another.

The title of this talk is "FP vs. OOP", as if FP and OOP were mutually exclusive, or even mutually antagonistic.

2 Functional programming

"Functional programming" could be defined as a style of programming, where a programmer is allowed to only define pure functions, by composing them from other pure functions

This is a definition. Like the definition of paradigm, it isn't very helpful. It doesn't tell us what the benefits of functional programming are, nor where it is desired to use it.

I sometimes hear that functional programming is advantageous, because "all your functions are pure", or because "there are no side effects", or because "your expressions are referentially transparent", as if those were some actual values, or things that people are longing for.

The book "Structure and Interpretation of Computer Programs" explains the benefits well when it discusses the costs of using the assignment operation in your programs.

It says:

> As we have seen, the `set!` operation enables us to model objects that have local state.
>
> However, this advantage comes at a price. Our programming language can no longer be interpreted in terms of the substitution model of procedure application [...].
>
> So long as we do not use assignments, two evaluations of the same procedure with the same arguments will produce the same result,
>
> so that procedures can be viewed as computing mathematical functions.
>
> Programming without any use of assignments, as we did throughout the first two chapters of this book, is accordingly known as functional programming.[1]

This is a useful characterization, because it shows both benefits and limitations of functional programming.

It is probably worth mentioning that you can imagine programs that are not functional in the aforementioned sense, but which can still be analyzed in terms of substitutions.

Consider the following "expression compiler"

```
(define (compile expression result)
  (match expression
      (`(,<*> ,a ,b)
       (let ((a+ (name a))
             (b+ (name b)))
         `(,@(compile a a+)
           ,@(compile b b+)
           (,result = ,a+ ,<*> ,b+))))
      (_
  '())))
```

It contains a call to the `name` procedure, which generates unique identifiers. The `name` procedure could be defined in the following way:

```
(define name
  (let ((counter 0))
    (lambda (expression)
```

```
      (cond ((symbol? expression)
             expression)
            (else
             (set! counter (+ counter 1))
             (symbol-append 't (number->symbol counter)))))))
```

The result of the expression `(compile '(* (+ a b) (/ c d)) 'result)` could then be derived in terms of substitutions:

```
(match '(* (+ a b) (/ c d))
    (`(,<*> ,a ,b)
     (let ((a+ (name a))
           (b+ (name b)))
       `(,@(compile a a+)
         ,@(compile b b+)
         (,result = ,a+ ,<*> ,b+))))
    (_
  '()))

(let ((a+ (name '(+ a b)))
      (b+ (name '(/ c d))))
  `(,@(compile '(+ a b) a+)
    ,@(compile '(/ c d) b+)
  (result = ,a+ * ,b+)))
```

We cannot derermine the order of evaluation of bindings in the `let` form, so we can choose arbitrarily:

```
(let ((a+ 't1)
      (b+ 't2))
  `(,@(compile '(+ a b) a+)
    ,@(compile '(/ c d) b+)
    (result = ,a+ * ,b+)))

`(,@(compile '(+ a b) t1)
  ,@(compile '(/ c d) t2)
  (result = t1 * t2))
```

It should be easy to see that the value eventually becomes

```
'((t1 = a + b)
  (t2 = c / d)
  (result = t1 * t2))
```

Of course, it could be a slightly different expression. The names of identifiers could be different, and they could be produced in a different order. But this wouldn't break the overall correctness of the program.

Consider another example:

```
(define (shuffle l)
  (let ((n (length l)))
    (if (is n < 2)
       l
       (let ((left right (split-at l (+ 1 (random (- n 1))))))
         (if (= (random 2) 1)
            `(,@(shuffle right) ,@(shuffle left))
            `(,@(shuffle left) ,@(shuffle right)))))))
```

It relies on the result of the `random` procedure, which can return a different value each time it is evaluated.

For example, `(shuffle '(a b c))` is equivalent to

```
(let ((n 3))
  (if (is 3 < 2)
     l
     (let ((left right (split-at '(a b c) (+ 1 (random 2)))))
       (if (= (random 2) 1)
          `(,@(shuffle right) ,@(shuffle left))
          `(,@(shuffle left) ,@(shuffle right))))))
```

which is equivalent to

```
(let ((left right (split-at '(a b c) (+ 1 (random 2)))))
  (if (= (random 2) 1)
    `(,@(shuffle right) ,@(shuffle left))
    `(,@(shuffle left) ,@(shuffle right))))
```

You can see that it contains two occurrences of the expression `(random 2)`. It cannot be refactored to

```
(let* ((number (random 2)) ;; wrong!
       (left right (split-at '(a b c) (+ 1 number))))
  (if (= number 1)
    `(,@(shuffle right) ,@(shuffle left))
    `(,@(shuffle left) ,@(shuffle right))))
```

because this would yield a different program.

On the contrary, since both expressions can evaluate to either 0 or 1, we get four possible expansions:

```
(let ((left right (split-at '(a b c) 1)))
  (if (= 0 1)
    `(,@(shuffle right) ,@(shuffle left))
    `(,@(shuffle left) ,@(shuffle right))))
```

```
(let ((left right (split-at '(a b c) 2)))
  (if (= 0 1)
    `(,@(shuffle right) ,@(shuffle left))
    `(,@(shuffle left) ,@(shuffle right))))
```

```
(let ((left right (split-at '(a b c) 1)))
  (if (= 1 1)
    `(,@(shuffle right) ,@(shuffle left))
    `(,@(shuffle left) ,@(shuffle right))))

(let ((left right (split-at '(a b c) 2)))
  (if (= 1 1)
    `(,@(shuffle right) ,@(shuffle left))
    `(,@(shuffle left) ,@(shuffle right))))
```

each of which can be reduced to

```
`(,@(shuffle '(a)) ,@(shuffle '(b c)))

`(,@(shuffle '(a b)) ,@(shuffle '(c)))

`(,@(shuffle '(b c)) ,@(shuffle '(a)))

`(,@(shuffle '(c)) ,@(shuffle '(a b)))
```

It should now be easy to see that each of those expressions also has two possible expansions:

```
'(a b c) '(a c b)

'(a b c) '(b a c)

'(b c a) '(c b a)

'(c a b) '(c b a)
```

The expressions consisting of functions like `random` or `name` are "referentially opaque", but they can still be analyzed in terms of substitutions.

Strictly speaking, those programs are non "functional". One could say that they are "quasi-functional", or perhaps "expression oriented".

The term "expression oriented" is, in its spirit, more similar to "object oriented" than to "functional", because it is vague. It only conveys a rough idea on how to organize your programs. "Functional programming", on the other hand, is a discipline: either your program is functional or it is not.

This is not to say that the demarcation line of functional programming is sharp. Consider the following Haskell program:

```
main :: IO ()
main = do {
  name <- getLine;
  putStrLn ``Hello ``++name;
}
```

Some people argue that this is a purely functional program. When you frown at them persistently enough, they explain that this is a purely functional program which constructs an imperative program that is meant to be consumed by Haskell runtime.

It may be worth to note that this "program" does not depend on any input: each time it is evaluated, it will produce the same imperative program.

So, clearly, there is a problem of attribution here. I believe that most programmers would say that this is a program that reads an input line and prints it to the output, prepended with the string "Hello".

The key difference between functional programs and imperative ones is that imperative programs "do things", while functional programs "mean things".

I think that many functional programmers consider object oriented programming to be a form of imperative programming.

It may be hard to deliver a conclusive argument, because, unlike functional programming, object oriented programming truly is a paradigm, and not a discipline. There is no strict boundary between a program that is object oriented and the one that is "no longer object oriented".

It also seems possible to reconcile object orientation with functional programming. Consider the following program:

```
2 + 2
```

A Haskell programmer will say that `+` is a function that takes two numbers and produces some result.

3 Object oriented programming

A Smalltalk programmer, on the other hand, will interpret the number `2` on the left hand side as an object that is being sent a message `+ 2`, and that will respond to that message somehow.

We are accustomed to thinking that objects modify their state in reaction to messages. This doesn't have to be true for all objects. Numbers in Smalltalk are "platonic objects" in the sense that they are immutable and have no "internal state".

So at least the arithmetics in Smalltalk is an example of something that is both functional and object oriented at the same time.

But perhaps we shouldn't talk about object oriented programming without defining it first, or otherwise characterizing it.

One cannot talk about object orientation without mentioning Alan Kay.

In "The Early History of Smalltalk", he claimed that

> Smalltalk is the recursion on the notion of computer itself [4]

The programming model that he proposes is computers that send messages to other computers, and if we discount biology, which allegedly was the inspiration

for the concept, then the Internet is probably the biggest and most complex OO system created by humans.

Alan Kay also made the remark that he regrets having coined the term that way, and that he should have instead used the phrase "message-oriented programming". But if you listen to him carefully, you'll probably conclude that his concept of object orientation is very far from the mainstream.

It seems that, in many ways, Kay's vision of OO is much more progressive and groundbreaking than anything that most programmers have ever experienced.

Yet, I wanted to stop for a second by the phrase *recursion on the notion of computer itself*, and the idea that there are objects that receive messages from other objects and react to them in a way they please.

I wanted to show a series of simple programs that demostrate this idea and its limitations.

I will be using the Scheme programming language for this. It's interesting that Scheme was actually conceived in an attempt to understand Carl Hewitt's "actor model", which was itself inspired by the ideas from Smalltalk, so expressing this style of object orientation in Scheme comes rather naturally.

3.1 Passing messages

Let's begin with a simple program where you have colorful rectangles spread across the screen, and you can drag them around.

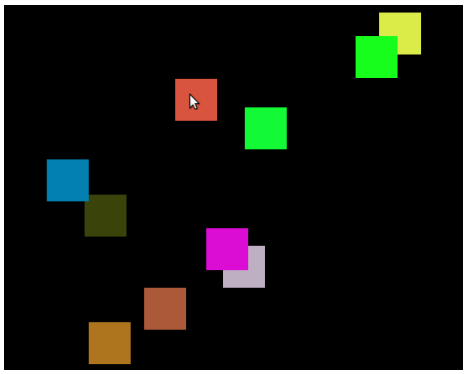

Figure 1: Rectangles that can be dragged around the screen

The main type of objects that we need for expressing this problem is 'draggable-rectangle'. It has its position, size and color.

```
(define (draggable-rectangle left top width height color)
  (let ((dragged? #false)
        (image (rectangle width height color)))
    (lambda message
```

```
    (match message
      (`(position)                 `(,left ,top))
      (`(mouse-down ,x ,y)         (set! dragged? #true))
      (`(mouse-up ,x ,y)           (set! dragged? #false))
      (`(mouse-move ,x ,y ,dx ,dy) (when dragged?
                                             (set! left (+ left dx))
                                        (set! top (+ top dy))))
      (`(embraces? ,x ,y) (and (is left <= x <= (+ left width))
                               (is top <= y <= (+ top height))))
      (`(as-image)                 image)
      (`(mouse-out)               (set! dragged? #false))
      (_ #false)))))
```

We also need one "singleton" object called "stage", which keeps track of all the rectangles and sends them messages. It's going to keep a list of the rectangles.

```
(define stage
  (let* ((`(,width ,height) (screen-size))
         (elements (map (lambda (_)
                          (draggable-rectangle
                                   (random (- width 50))
                                   (random (- height 50))
                                   50 50 (random #x1000000)))
                        (range 10)))
         (image (rectangle width height 0))
         (hovered-element #false))

    (lambda message
      (match message
        (`(as-image)
         (fill-image! image 0)
         (fold-right (lambda (element image)
                       (let ((`(,x ,y) (element 'position)))
                         (draw-image! (element 'as-image)
                                      x y image)
                         image))
                     image
                     elements)
         image)

        (`(mouse-down ,x ,y)
         (when hovered-element
           (hovered-element 'mouse-down x y)))

        (`(mouse-up ,x ,y)
```

```
    (when hovered-element
      (hovered-element 'mouse-up x y)))

   (`(mouse-move ,x ,y ,dx ,dy)
    (let ((hovered (find (lambda (_) (_ 'embraces? x y)) elements)))
      (unless (eq? hovered hovered-element)
        (when hovered-element (hovered-element 'mouse-out))
        (when hovered (hovered 'mouse-over))
        (set! hovered-element hovered))
      (when hovered (hovered 'mouse-move x y dx dy))))
   ))))
```

3.2 Hierarchical objects

This simple program could be generalized: we could allow the rectangles to contain other rectangles inside.

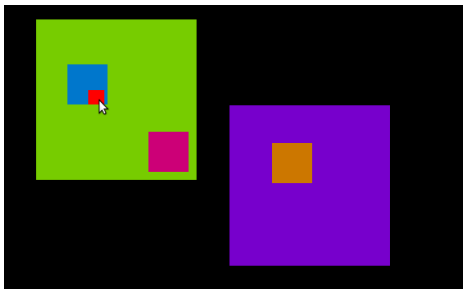

Figure 2: Rectangles that can contain inner rectangles

The 'box' class blends some features of the "rectangle" class and "stage" object:

```
(define (box #:left left #:top top #:width width #:height height
             #:background-color color #:draggable? [draggable? #true]
             . elements)
  (let ((dragged? #false)
        (hovered-element #false)
        (image (rectangle width height color)))
    (lambda message
      (match message
        (`(position) `(,left ,top))

        (`(mouse-down ,x ,y)
         (if hovered-element
             (hovered-element 'mouse-down (- x left) (- y top))
             (when draggable?
```

```
        (set! dragged? #true))))

      (`(mouse-up ,x ,y)
       (when hovered-element
         (hovered-element 'mouse-up (- x left) (- y top)))
       (set! dragged? #false))

      (`(mouse-move ,x ,y ,dx ,dy)
       (cond (dragged?
              (set! left (+ left dx))
              (set! top (+ top dy)))
             (else
              (let ((hovered (find (_ 'embraces? (- x left) (- y top))
                                   elements)))
                (unless (eq? hovered hovered-element)
                  (when hovered-element
                    (hovered-element 'mouse-out))
                  (when hovered
                    (hovered 'mouse-over))
                  (set! hovered-element hovered))
                (when hovered
                  (hovered 'mouse-move (- x left) (- y top) dx dy))))))

      (`(embraces? ,x ,y) (and (is left <= x <= (+ left width))
                               (is top <= y <= (+ top height))))
      (`(as-image)
       (fill-image! image color)
       (fold-right (lambda (element image)
                     (let ((`(,x ,y) (element 'position)))
                       (draw-image! (element 'as-image) x y image)
                       image))
                   image
                   elements))
      (`(mouse-over) #false)
      (`(mouse-out) (set! dragged? #false))))))
```

The 'stage' object consists of all of the considered objects

```
(define stage
  (let ((`(,w ,h) (screen-size)))
    (box #:left 0 #:top 0 #:width w #:height h
         #:background-color #x000000 #:draggable? #false
         (box #:left 10 #:top 10 #:width 200 #:height 200
              #:background-color #x77cc00
              (box #:left 10 #:top 10 #:width 50 #:height 50
                   #:background-color #x0077cc
```

```
                (box #:left 5 #:top 5 #:width 20 #:height 20
                     #:background-color #xff0000))
           (box #:left 140 #:top 140 #:width 50 #:height 50
                #:background-color #xcc0077
                #:draggable? #false))
      (box #:left (- w 210) #:top (- h 210) #:width 200 #:height 200
           #:background-color #x7700cc
           (box #:left 140 #:top 140 #:width 50 #:height 50
                #:background-color #xcc7700)))))
```

I think that this example is interesting in that you can think of each of those boxes as if they were some virtual machines that interpret incomming signals. In this sense, I believe that this program is a good embodyment of the phrase "recursion on the notion of computer itself".

3.3 Beyond messaging

Now let's have a look at another program:

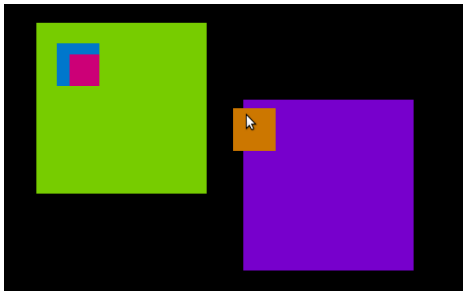

Figure 3: Moving rectangles between rectangles

As you can see, we can now take boxes out from other boxes and put them in yet another boxes.

You should note that there is a huge leap between the previous program and this one: previously we were only transferring messages, that is, information, whereas now we are transferring ownership or identity.

The key difference is that when you give some information to someone, you don't loose that information, whereas if you give some thing to someone, you no longer have it.

It may seem strange that there aren't many mainstream programming languages that recognize the issue of ownership (most notably Rust, and - to some extent - C++).

```
(define (box #:left left #:top top #:width width #:height height
             #:background-color color . elements)
  (let ((dragged-element #false)
```

```
    (hovered-element #false)
    (image (rectangle width height color)))
(define (self . message)
  (match message
    (`(position) `(,left ,top))

    (`(mouse-down ,x ,y)
     (let ((acquired (self 'acquire-hovered-element!)))
       (when acquired
         (set! dragged-element acquired))))

    (`(mouse-up ,x ,y)
     (when dragged-element
       (self 'install-element! dragged-element)
       (set! dragged-element #false))
     (self 'mouse-move x y 0 0)
     (when hovered-element
       (hovered-element 'mouse-up (- x left) (- y top))))

    (`(mouse-move ,x ,y ,dx ,dy)
     (when dragged-element
       (dragged-element 'move-by! dx dy))
     (let ((hovered (find (_ 'embraces? (- x left) (- y top))
                          elements)))
       (unless (eq? hovered hovered-element)
         (when hovered-element
           (hovered-element 'mouse-out))
         (when hovered
           (hovered 'mouse-over))
         (set! hovered-element hovered))
       (when hovered
         (hovered 'mouse-move (- x left) (- y top) dx dy))))

    (`(embraces? ,x ,y) (and (is left <= x <= (+ left width))
                             (is top <= y <= (+ top height))))
    (`(as-image)
     (fill-image! image color)
     (fold-right (lambda (element image)
                   (let ((`(,x ,y) (element 'position)))
                     (draw-image! (element 'as-image) x y image)
                     image))
                 image
                 `(,@(if dragged-element
                         `(,dragged-element)
                         '())
                   ,@elements)))
```

```
    (`(move-by! ,dx ,dy)
     (set! left (+ left dx))
     (set! top (+ top dy)))

    (`(acquire-hovered-element!)
     (and hovered-element
          (or (and-let* ((acquired (hovered-element
                                    'acquire-hovered-element!))
                         (`(,x ,y) (hovered-element 'position)))
                (acquired 'move-by! x y)
                acquired)
              (let ((acquired hovered-element))
                (set! hovered-element #false)
                (set! elements (filter (lambda (_) (isnt _ eq? acquired))
                                       elements))
                acquired))))

    (`(install-element! ,element)
     (if hovered-element
         (let ((`(,x ,y) (hovered-element 'position)))
           (element 'move-by! (- x) (- y))
           (hovered-element 'install-element! element))
         (set! elements `(,element . ,elements))))
    (_ #false)))
  self))
```

But I guess that Alan Kay's view on OOP is itself far from the main stream, and we should look for a more popular definition, like the one given by Grady Booch and his co-authors in the book "Object-Oriented Analysis and Design with Applications":

> Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationship. [2]

This is a very specific definition, which emphasizes the idea of organizing software in classes that relate to each other through inheritance.

Many expositions of object orientation mention the "four pillars of OOP":

- abstraction
- encapsulation
- inheritance

- polymorphism

I tried to find the origin of those "pillars". Someone claimed that they were formulated in the aforementioned book by Grady Booch, but all I found there was a list of "major principles on which object-oriented development is founded":

- abstraction
- encapsulation
- modularity
- hierarchy

I don't want to focus on all these concepts, but I'd like to talk a bit about abstraction and inheritance.

3.4 Abstraction

I have seen many programmers struggling to define what abstraction is.

For example, John De Goes, a celebrity Scala-turned-Haskell programmer, proposes the following definition:

> An abstraction is a set of types, operations on these types, and laws that give meaning to the operations across all contexts.
>
> In functional programming, we call abstractions type classes (or sometimes modules), and contrast them with design patterns.
>
> [`https://twitter.com/jdegoes/status/1041769867341443078`]

Eric Normand who runs LispCast wrote an essay where he struggles to define abstraction, and instead gave a bunch of quotes that contain the word "abstraction"[6].

The definition that can be found in Booch's *Object-Oriented Analysis and Design* is no better:

> An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer [2]

This definition of abstraction is surprising, given that before formulating it, Booch makes the following reference:

> 'Dahl, Dikstra and Hoare suggest that 'abstraction arises from a recognition of similarities between certain objects, situations or processes in the real world, and the decision to concentrate upon these similarities and to ignore for the time being the differences [2]

While this is not a definition, but rather a description of the typical way by which we arrive at abstractions, it clearly has nothing to do with any "essential charactersitics of an object that distinguish it from all other kinds of objects".

It may be instructive to take a look at a picture, like one of the "find 6 differences" pictures that you sometimes find in kids' magazines:

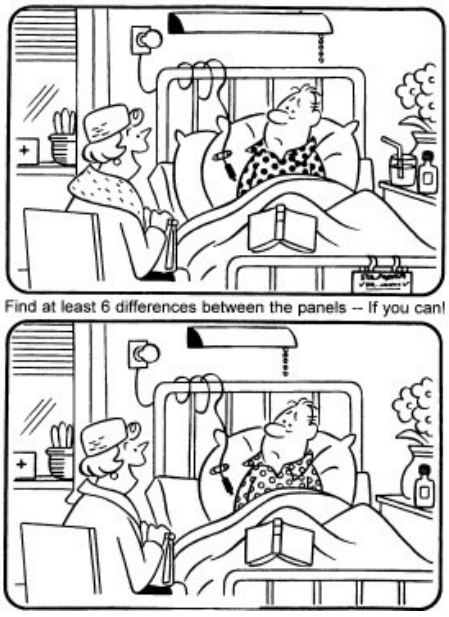

Figure 4: Find 6 differences

Once you spot the differences, you can "abstract from them" by forming an incomplete picture that contains six holes which can be filled with appropriate content:

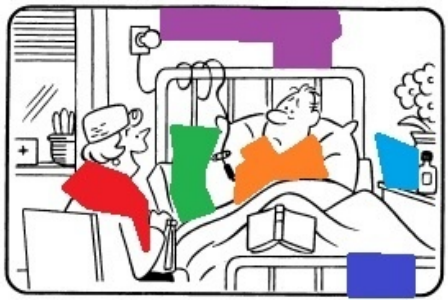

Figure 5: An image containing holes (an abstraction)

This of course begs the question, how do we distinguish this incomplete

picture from a picture that contains some colorful spots in it.

The answer is that we can explicitly enumerate the holes:

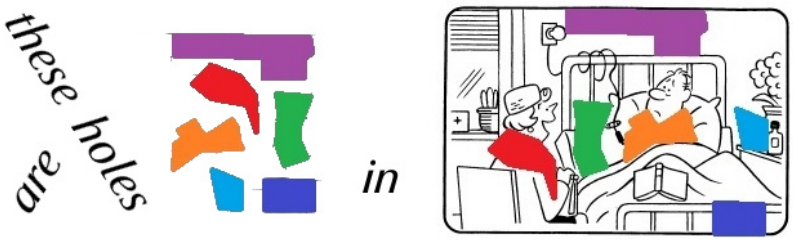

Figure 6: The holes are explicitly pointed out

This pattern should be familiar to functional programmers. It is no coincidence that that expressions in lambda-calculus that begin with the lambda operator are called *lambda-abstractions*.

You can also think of the shapes of the holes as types. In an untyped system, all the holes and images could be rectangular, for example.

Of course, this is just an illustration, and not a definition. We can try defining abstraction as *a process of intentionally omitting (leaving out) certain detail in some mental representation*. Obviously, this only defines it as a verb, whereas many people use the word "abstraction" as a noun.

We can define the noun "abstraction" as the product of the process of abstraction. It's hard to do it directly without resorting to metaphors, especially if we would like to have the "normal" type of definition, where we give the "genus proximum" and "differentia specifica": it begs for an explanation such as:

> "abstraction is a mental representation which contains *holes* that are intended to be filled with some content (concretized)"

It may not be a perfect definition. The use of the metaphor of *holes* may be off-putting, and probably the best definition of abstraction is the one that comes from the lambda-calculus.

Also, there has been a huge debate in Philosophy regarding the question whether abstract objects exist independently of minds, but it seems that we have no problem abstracting mental representations from minds themelves.

I think that no one has any doubts how fundamental abstraction is for programming (and for thinking in general). However, I find it worrying that some important authorities seem to be so confused about this very basic concept.

3.5 Inheritance

I am even more worried that the proponents of object orientation often attribute the same importance to abstraction as they do to another "pillar" that I wanted to talk about today, namely - inheritance.

Let's begin by analyzing the justifications that various authors provide for this concept.

Chamond Liu, the author of *Smalltalk, Objects and Design*, wrote this:

> Inheritance [...] means that you may specify [...] that a class is a special kind of another class [...]. The underlying idea is [...] that of classifying things by increasing degrees of specialization.
>
> The botanist Carl von Linne, better known as Linneus, popularized this way of thinking about plants (as well as animals) in the eighteenth century. [...] the idea extended the very way in which people could think about the world [5]

I am not aware of the influence that Linneus had on the way "people think about the world", but I don't think that the fact that something becomes popular means that it is a good idea.

Brad J. Cox and Andrew Novobilski wrote in their *Object Oriented Programming: An Evolutionary Approach*:

> It would be possible to stop with the encapsulation concept and build a language that provides classes and instances, but not inheritance. Some languages do exactly that (e.g. Ada). Inheritance is not a necessary feature of an object-oriented language, but it is certainly an extremely desirable one.
>
> Also, the simple linear scheme to be described there is not the only way to provide inheritance. For example, some languages provide multiple inheritance, which means that classes can have more than one superclass. These languages provide more generality, solving, for example, problems like expressing what a toy truck is. Is it a kind of toy, or is it a kind of truck? With multiple inheritance, it can be both. [3]

I have to stop here for a second.

When I read this fragment for the first time, I imagined someone working in logistics. "Sure, we have 5 trucks available, I can dispach them to your company right away".

Seriously, a toy truck is not a kind of truck. A sculpture of a woman is not a woman. Those examples only suggest that inheritance, or at least multiple inheritance, is a tool for making problems, not solving them.

But let's read on:

> Inheritance is a tool for organizing, building, and using reusable classes. Without inheritance, every class would be a free-standing

> unit, each developed from the ground up. Different classes would bear no relationship with one another, since the developer of each provides methods in whatever manner he chooses. Any consistency across classes is the result of discipline on the part of programmers.
>
> Inheritance makes it possible to define new software in the same way we introduce any concept to a newcommer, by comparing it with something that is already familiar [3]

Recall now what we said earlier about abstraction: that it allows us to emphasize similarities and ignore differences. So it is possible that a bunch of classes could all be concetizations of a single abstract class. There is no reason to think that "every class needs to be a free-standing unit".

All that we need is abstraction done right.

Some people have pointed out many flaws that can arise from the use of inheritance. In fact, in the OO circles you can often hear the mantra "favor composition over inheritance".

Functional programmers tend to use the word "composition" to refer to "function composition", which is different from the usage that the proponents of OOP have in mind.

Grady Booch explained in the aforementioned book, that he uses the word "inheritance" to refer to the "is a" relationship, and the word "composition" to refer to the "has a" relationship. Probably "inclusion" would be a better word for this.

So far we have been using the word "class" without defining it. It may be safe to assume that most programmers have an idea what a class is.

The essential characteristic of a class is that it can be instantiated. Some programming languages have a concept of "abstract classes" that cannot be instantiated directly, but I don't think it is a very important concept.

A mixin is a function that operates at a class level. We could imagine its counterpart that would operate at the instance level. Let's call this counterpart a "lag":

		mixin	
class:	A	→	B
	↓		↓ instantiation
instance:	a	→	b
		lag	

Of course, we shouldn't be talking about inheritance and similar mechanisms without mentioning the problems that they try to solve.

Let's recall the program where we could move boxes between boxes.

Suppose that we'd also like to be able to resize the boxes when we drag them at their edges.

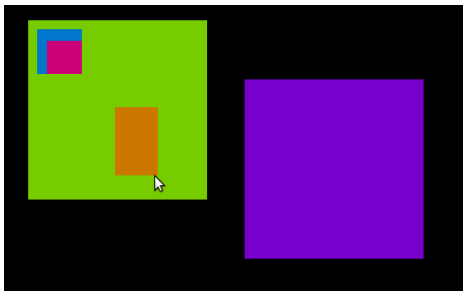

Figure 7: Rectangles can be resized by grabbing their edges

How on earth would we do that in the original program?

The main class in the original program consists of about 100 lines of code, spread across around 10 methods. This gives us on average 10 lines per method, which may not seem much. However, in order to extend the class with new capabilities, we would need to review each of those methods to make sure that we do not inject any undesired interactions.

This would be tedious, error-prone and would lead to code that would itself be even harder to maintain.

The truth is, that the "box" class that I presented before already blends several concern together: its instances can be dragged from one box to another, and it can itself store other boxes.

The base class is only capable of providing position, size and the colorful rectangle to be displayed on the screen. We can also change the position and the size of the box:

```
(define (bit #:left left #:top top
             #:width width #:height height
             #:background-color color)
  (let ((image (rectangle width height color)))
    (define (self . message)
      (match message
        (`(position)  `(,left ,top))

        (`(size)      `(,width ,height))

        (`(extents)   `(,left ,top ,(+ left width)
                                   ,(+ top height)))
        (`(resize-by! ,dx ,dy)
         (set! width (max 0 (+ width dx)))
         (set! height (max 0 (+ height dy)))
         (set! image (rectangle width height color)))

        (`(move-by! ,dx ,dy)
         (set! left (+ left dx))
```

```
       (set! top (+ top dy)))

      (`(embraces? ,x ,y)
        (and (is left <= x <= (+ left width))
             (is top <= y <= (+ top height))))

      (`(as-image)
       (fill-image! image color)
       image)

      (_ #false)))
  self))
```

As mentioned earlier, we want our objects to be able accept and provide other boxes:

```
(define (containing origin elements)
  (let ((hovered-element #false))
    (lambda message
      (let ((`(,left ,top) (origin 'position)))
        (match message

          (`(mouse-down ,x ,y)
           (if hovered-element
               (hovered-element 'mouse-down (- x left) (- y top))
               #false))

          (`(mouse-up ,x ,y)
           (when hovered-element
             (hovered-element 'mouse-up (- x left) (- y top))))

          (`(mouse-move ,x ,y ,dx ,dy)
           (let ((hovered (find (_ 'embraces? (- x left) (- y top))
                                elements)))
             (unless (eq? hovered hovered-element)
               (when hovered-element
                 (hovered-element 'mouse-out))
               (when hovered
                 (hovered 'mouse-over))
               (set! hovered-element hovered))
             (when hovered
               (hovered 'mouse-move (- x left) (- y top) dx dy))
             hovered-element))

          (`(add! ,element)
           (set! elements `(,element . ,elements)))
```

```
          (`(remove! ,element)
           (set! elements (filter (lambda (_) (isnt _ eq? element)) elements)))

          (`(as-image)
           (let ((image (origin 'as-image)))
             (fold-right (lambda (element image)
                           (let ((`(,x ,y) (element 'position)))
                             (draw-image! (element 'as-image) x y image)
                             image))
                         image
                         elements)))

          (_
           (apply origin message)))
        ))))
```

This allows us to define the original `box` class in the following way:

```
(define (box #:left left #:top top
             #:width width #:height height
             #:background-color color . elements)
 (transferable
  (containing
   (bit #:left left #:top top
        #:width width #:height height
        #:background-color color)
   elements)))
```

Now, what we'd like to do is to write a `resizable` lag, which would allow us to redefine box in the following way:

```
(define (box #:left left #:top top
             #:width width #:height height
             #:background-color color . elements)
  (resizable
   (transferable
    (containing
     (bit #:left left #:top top
          #:width width #:height height
          #:background-color color)
  elements))))
```

The most apparent difference between the use of lags and classical OO inheritance is that we can see the whole "lag hierarchy" in the glimpse of an eye, or that "inheritance" can be expressed as function application, which is neat.

4 Beyond textual programming

So far I have covered the two "big" concepts of OO: abstraction and inheritance.

I believe that the mechanism of "lags" shows that "inheritance" doesn't add anything that isn't expressible with plain abstraction. It also shows that perhaps instead of focusing - like Linneus did - on the place of something in a fixed, global hierarchy, it may be better to just focus on the relation of that something to its surroundings.

You can sometimes hear - in the Internet - voices that claim that "FP is going to take over OOP", or that "OOP will remain the dominating paradigm".

Perhaps we should step back a bit and ask ourselves: "which paradigm is actually dominating?".

But the fact that the "FP vs. OOP" debate is hearable suggests that none of those appraches is truly "dominating". The dominating paradigm is the one that is questioned by no one, or by very few people, and is perceived by most as "the natural order of things" or "necessity".

In 1987, Phil Wadler, one of the designers of the Haskell programming language, wrote "a critique" of the book that I quoted at the beginning, "Structure and Interpretation of Computer Programs".

He made some valid points in that paper. He compared some programs written in Scheme with their counterparts written in Haskell's predecessor Miranda, and claimed that the Scheme programs are "harder to read because of the syntax (or, rather, lack of syntax) of Lisp."

He also made a few less valid points. He presented a Lambda-calculus evaluator in Miranda and in Scheme and observed that the natural representations of lambda-expression in Miranda are more cumbersome than the ones in Scheme, but that if you work hard enough, you can make the Scheme representations equally cumbersome.

Then he concluded that

> a better approach would be to write parsers and unparsers, to convert between a convenient notation for reading and writing programs-as-data and a convenient notation for manipulating them. This clearly requires more work, but also yield more benefit. Parsers and unparsers are interesting subjects in their own right, [...]. [7]

I was once having a discussion with Jon Harrop, who claimed that "SICP is a seriously out of date book". In the course of the discussion, which obviously touched the question of syntax, I said:

> I'm not sure whether parsing would actually bring any value to the topics that were covered by SICP. How would it be helpful to their logic circuit simulator, or to the picture language? How would it help with symbolic differentiation? How would they justify introduction of syntax? "Hey, listen, we have this straightforward way of representing expressions, and although we could get used to it and go on with our presentation of ideas, it looks a bit alien to us, so we

> will devote the next chapter to parsing, so that we could be writing it so that it would resemble things that we are more accustomed to"?

Jon's reaction was the following:

> What a bizarre question. Syntax is defacto standard. Syntax is everywhere, in every major language. Rather than asking for justification for the inclusion of syntax you should be trying to justify its exclusion.

This is an interesting observation.

Later today, Jesse is going to teach us about "making languages". My bet is that some part of his tutorial is going to be devoted to the design of syntax.

But before it happens, let me ask: what is it that the syntax buys us?

One thing that is certain is the ability to make syntax errors. When we write data on disks, for example, we trust our file system drivers and operating system interface that they will take care of representing the tree structures that we want to be represented. Yet, as programmers, we stubbornly insist on preserving the ability to make syntax errors.

But I bet that this is not the answer that you would hear from most programmers. My bet is that they would rather say that syntax gives them "freedom of expression", or the ability to "express their programs in the way they want to".

We are the fish, and we all swim in the water that is called UNIX. We are all convinced, deep in our hearts, that everything is a file, that is, a sequence of bytes. We think that programs need to be text, or that text is the fundamental representation of the things that we make.

Of course, there are some visual programming languages available, like - for example - MIT Scratch or LabVIEW. But those are just toys or highly domain-specific tools. Let's be honest - how many professional Scratch developers do you know?

Lisp is specific, because it is closest to the surface of the water what we are all swimming in. But it is still under water. It is still based on text. People who expand the acronym "LISP" as "lots of irritating superfluous parentheses" are right - Lisp is weird.

The colorful boxes that I've been showing you weren't just a futile exercise in object-oriented concepts.

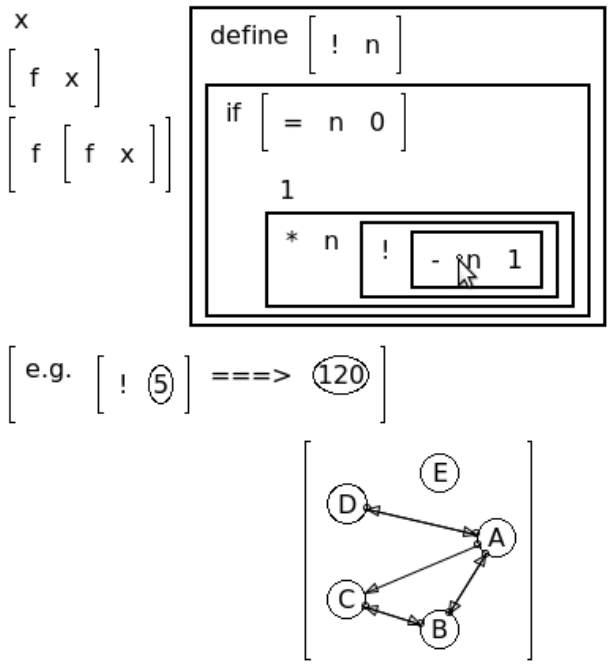

Figure 8: Lots of Intriguing Movable Boxes

It suffices to display those boxes slightly differently. I called this representation LIMB, for "lots of intriguing movable boxes", to prevent the evil people from making bad puns on this name.

The important property of this representation is that it is based on graphics, not on text, so it should give much more freedom of expression.

Of course the main point of this representation is not that it makes your programs "look even weirder", but that it allows, in principle, to define specialized editors for particular kinds of data, such as graphs, chemical compounds, Feynman diagrams, clouds of points, flow charts, state machines, so that the visualisation can be embedded in the very source of the program.

It is also well adjusted to touchs screen editing. Actually, I conceived it when I was playing with the "Simple Scheme" interpreter on my Android-based smartphone. I quickly discovered that using an on screen keyboard for writing Lisp programs is horribly painful and inefficient.

The program is far from complete, but I hope that some of you may find it inspiring.

References

[1] Abelson, Harold and Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, Second Edition, MIT Press, 1996, ISBN 0-262-01153-0
`https://mitpress.mit.edu/sicp/full-text/book/book.html`

[2] Booch, Grady et al., *Object-Oriented Analysis and Design with Applications*, 3rd edition, Addison Wesley 2007

[3] Cox, Brad J and Andrew Novobilski, *Object Oriented Programming: An Evolutionary Approach*, Addison Wesley, 1991

[4] Alan Kay, *The Early History of Smalltalk*,
`http://worrydream.com/EarlyHistoryOfSmalltalk/`
access 25/03/2019

[5] Chamond Liu, *Smalltalk, Objects and Design*,
iUniverse, 2000

[6] Eric Normand, *What is Abstraction*
`https://lispcast.com/what-is-abstraction/`
access 25/03/2019

[7] Wadler, Phillip, *A critique of Abelson and Sussman - or - Why calculating is better than scheming*, 1987
`https://www.cs.kent.ac.uk/people/staff/dat/miranda/wadler87.pdf`

[8] Wikipedia, *Functional Programming*,
`https://en.wikipedia.org/wiki/Functional_programming`
access 25/03/2019