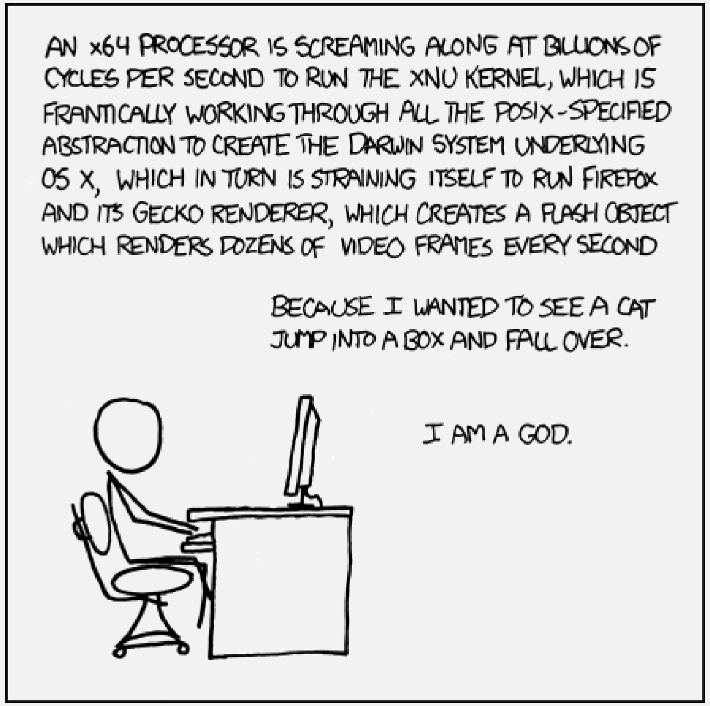
UMass Boston Computer Science

CS450 High Level Languages (section 2)

Abstraction

Monday, October 7, 2024



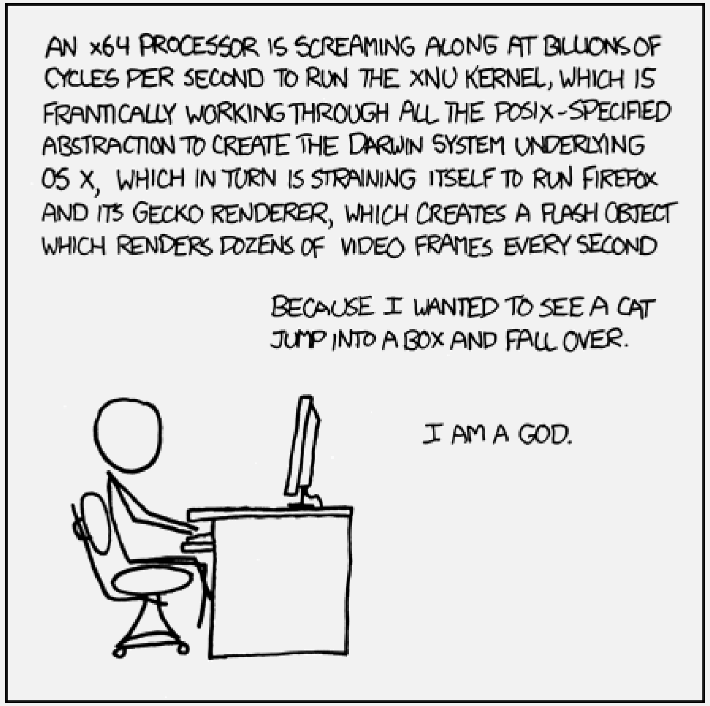
Logistics

- HW 4 in
    - ~~due: Mon 10/7 12pm (noon) EST~~

- HW 5 out
    - due: Mon 10/14 12pm (noon) EST

- HW 6
    - out: Mon 10/14 12pm (noon) EST
    - due: Mon 10/21 12pm (noon) EST

- **NOTE:** no class next Monday 10/14



Last Time

List (Recursive) Data Definition 1

```
;; A ListofInt is one of:
;; - empty
;; - (cons Int ListofInt)
```

Last Time

List (Recursive) Data Definition 1: Fn Template

Recursive call matches recursion in data definition

```
;; A ListofInt is one of:
;; - empty
;; - (cons Int ListofInt)
```

```
;; TEMPLATE for list-fn
;; list-fn : ListofInt -> ???
(define (list-fn lst)
  (cond
    [(empty? lst) ....]
    [(cons? lst) .... (first lst) ....
           .... (list-fn (rest lst)) ....]))
```

Extract pieces of compound data

cond clause for each itemization item

Last Time

Recursive List Fn Example 1: `inc-list`

Function design recipe:
1. Name
2. Signature
3. Description
4. Examples
5. Template

...

```
;; inc-list : ListofInt -> ListofInt
;; increments each list element by 1
(define (inc-lst lst)
  (cond
    [(empty? lst) ....]
    [(cons? lst) .... (first lst)  ....
          ....  (inc-lst (rest lst)) ....]))
```

```
(check-equal?
  (inc-list (list 1 2 3))
            (list 2 3 4))
```

Last Time

Recursive List Fn Example 1: **inc-list**

```
;; inc-list : ListofInt -> ListofInt
;; increments each list element by 1
(define (inc-lst lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst) .... (first lst)  ....
           ....  (inc-lst (rest lst)) ....]))
```

Empty input produces empty output
(look at signature for help if needed)

Last Time

Recursive List Fn Example 1: **inc-list**

```
;; inc-list : ListofInt -> ListofInt
;; increments each list element by 1
(define (inc-lst lst)
  (cond
    [(empty? lst) empty]
    [else ....  (add1 (first lst)) ....
          ....  (inc-lst (rest lst)) ....]))
```

Call another function to process (first) (Int) list element

Last Time

Recursive List Fn Example 1: `inc-list`

```
;; inc-list : ListofInt -> ListofInt
;; increments each list element by 1
(define (inc-lst lst)
  (cond
    [(empty? lst) empty]
    [else (cons (add1 (first lst))
                (inc-lst (rest lst)))]))
```

Figure out how to “combine” with recursive call result
(look at signature for help if needed)

Last Time

List (Recursive) Data Definition 2

```
;; A ListofBall is one of:
;; - empty
;; - (cons Ball ListofBall)
```

Last Time

List (Recursive) Data Definition 2: Fn Template

Recursive call matches recursion in data definition?

```
;; A ListofBall is one of:
;; - empty
;; - (cons Ball ListofBall)
```

```
;; TEMPLATE for list-fn
;; list-fn : ListofBall -> ???
(define (list-fn lst)
  (cond
    [(empty? lst) ....]
    [(cons? lst) .... (first lst)  ....
           ....  (list-fn (rest lst)) ....]))
```

cond clause for each itemization item?

Extract pieces of compound data?

Last Time

Recursive List Fn Example 2: `next-world`

Function design recipe:

1. Name
2. Signature
3. Description
4. Examples
5. Template

...

```
;; next-world: ListofBall -> ListofBall
;; Updates position each ball by one tick
(define (next-world lst)
  (cond
    [(empty? lst) ....]
    [(cons? lst) .... (first lst)  ....
          ....  (next-world (rest lst)) ....]))
```

Last Time

Recursive List Fn Example 2: `next-world`

```
;; next-world: ListofBall -> ListofBall
;; Updates position each ball by one tick
(define (next-world lst)
  (cond
   [(empty? lst) empty]
   [(cons? lst) .... (first lst)  ....
         ....  (next-world (rest lst)) ....]))
```

Empty input produces empty output
(look at signature for help if needed)

Last Time

Recursive List Fn Example 2: `next-world`

```
(check-equal? (next-world (list (make-ball 0 0 1 1)))
                          (list (next-ball (make-ball 0 0 1 1))))
```

```
;; next-world: ListofBall -> ListofBall
;; Updates position each ball by one tick
(define (next-world lst)
  (cond
    [(empty? lst) empty]
    [else ....  (??? (first lst)) ....
          ....  (next-world (rest lst)) ....]))
```

Call another function to process (`first`) list element?

Ball

Last Time

Recursive List Fn Example 2: `next-world`

```
;; next-world: ListofBall -> ListofBall
;; Updates position each ball by one tick
(define (next-world lst)
  (cond
    [(empty? lst) empty]
    [else ....  (next-ball (first lst)) ....
          ....  (next-world (rest lst)) ....]))
```

Call another function to process (`first`) (`Ball`) list element

Last Time

Recursive List Fn Example 2: `next-world`

```
;; next-world: ListofBall -> ListofBall
;; Updates position each ball by one tick
(define (next-world lst)
  (cond
    [(empty? lst) empty]
    [else (cons (next-ball (first lst))
                (next-world (rest lst)))]))
```

Figure out how to "combine" with recursive call result
(look at signature for help if needed)

Last Time

Comparison 1

Differences?

```
;; inc-lst: ListofInt -> ListofInt
;; Returns list with each element incremented
(define (inc-lst lst)
  (cond
    [(empty? lst) empty]
    [else (cons (add1 (first lst))
                (inc-lst (rest lst)))]))
```

```
;; next-world : ListofBall -> ListofBall
;; Updates position of each ball by one tick
(define (next-world lst)
  (cond
    [(empty? lst) empty]
    [else (cons (next-ball (first lst))
                (next-world (rest lst)))]))
```

Last Time

Abstraction: Common List Function #1

Make the difference a parameter of a (function) abstraction

```
(define (lst-fn1 fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (lst-fn1 (rest lst)))]))
```

Abstraction Recipe

1. Find **similar patterns** in a program
   - Minimum: **2**
   - Ideally: **3+**
2. Identify **differences** and **make them parameters**
3. Create a **reusable abstraction** with the **discovered parameters**
   - E.g., **a function**(al) **abstraction**

Abstraction: Common List Function #1

```
;; lst-fn1: (?? -> ??) Listof?? -> Listof??
;; Applies the given fn to each element of given lst
```

```
(define (lst-fn1 fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (lst-fn1 (rest lst)))]))
```

Abstraction of Data Definitions

```
;; A ListofInt is one of
;; - empty
;; - (cons Int ListofInt)
```

```
;; A ListofBall is one of
;; - empty
;; - (cons Ball ListofBall)
```

Abstraction Recipe

1. Find **similar patterns** in a program
   - Minimum: **2**
   - Ideally: **3+**
2. Identify **differences** and **make them parameters**
3. Create a **reusable abstraction** with the **discovered parameters**
   - E.g., **a function(al) abstraction**

Abstraction of Data Definitions

```
;; A ListofInt is one of
;; - empty
;; - (cons Int ListofInt)
```

```
;; A ListofBall is one of
;; - empty
;; - (cons Ball ListofBall)
```

Abstraction Recipe

1. Find similar patterns in a program
   - Minimum: 2
   - Ideally: 3+
2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction

Abstraction of Data Definitions

```
;; A ListofInt is one of
;; - empty
;; - (cons Int ListofInt)
```

```
;; A ListofBall is one of
;; - empty
;; - (cons Ball ListofBall)
```

parameter

```
;; A Listof<X> is one of
;; - empty
;; - (cons X Listof<X>)
```

Abstraction: Common List Function #1

NOTE: textbook writes it like this
(both are ok, just follow data definition)

```
;; lst-fn1: [X -> Y] [Listof X] -> [Listof Y]
;; Applies the given fn to each element of given lst
```

```
;; lst-fn1: (X -> Y) Listof<X> -> Listof<Y>
;; Applies the given fn to each element of given lst
```

```
(define (lst-fn1 fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (lst-fn1 (rest lst)))]))
```

Abstraction Recipe

1. Find similar patterns in a program
   - Minimum: 2
   - Ideally: 3+
2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
4. Use the abstraction, by giving concrete arguments for parameters

Abstraction: Common List Function #1

```
;; lst-fn1: (X -> Y) Listof<X> -> Listof<Y>
;; Applies the given fn to each element of given lst
```

```
(define (lst-fn1 fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (lst-fn1 (rest lst)))]))
```

```
(define (inc-lst lst) (lst-fn1 add1 lst))
(define (next-world lst) (lst-fn1 next-ball lst))
```

Q: Do these functions follow the design recipe (template)?

A: They do. Because “arithmetic” is always allowed.

```
(define (inc-lst lst) (lst-fn1 add1 lst))
(define (next-world lst) (lst-fn1 next-ball lst))
```

Common List Function #1

```
;; lst-fn1: (X -> Y) Listof<X> -> Listof<Y>
;; Applies the given fn to each element of given lst
```

```
(define (lst-fn1 fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (lst-fn1 (rest lst)))]))
```

```
(define (inc-lst lst) (lst-fn1 add1 lst))
(define (next-world lst) (lst-fn1 next-ball lst))
```

Common List Function #1: **map**

```
;; map: (X -> Y) Listof<X> -> Listof<Y>
;; Applies the given fn to each element of given lst
```

```
(define (map fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (map (rest lst)))]))
```

```
(define (inc-lst lst) (map add1 lst))
(define (next-world lst) (map next-ball lst))
```

Abstraction Recipe

1. Find similar patterns in a program
   - Minimum: 2
   - Ideally: 3+

> Abstractions should have a "clear and concisely defined task"

2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
   - The abstraction must have a short, clear name and "be logical"
4. Use the abstraction by giving concrete "arguments" parameters

Abstraction Recipe

1. <u>Find</u> **similar patterns** in a program
   - Minimum: **2**
   - Ideally: **3+**

Not all “similar patterns” should be abstracted

2. <u>Identify</u> **differences** and **make them parameters**
3. <u>Create</u> a **reusable**

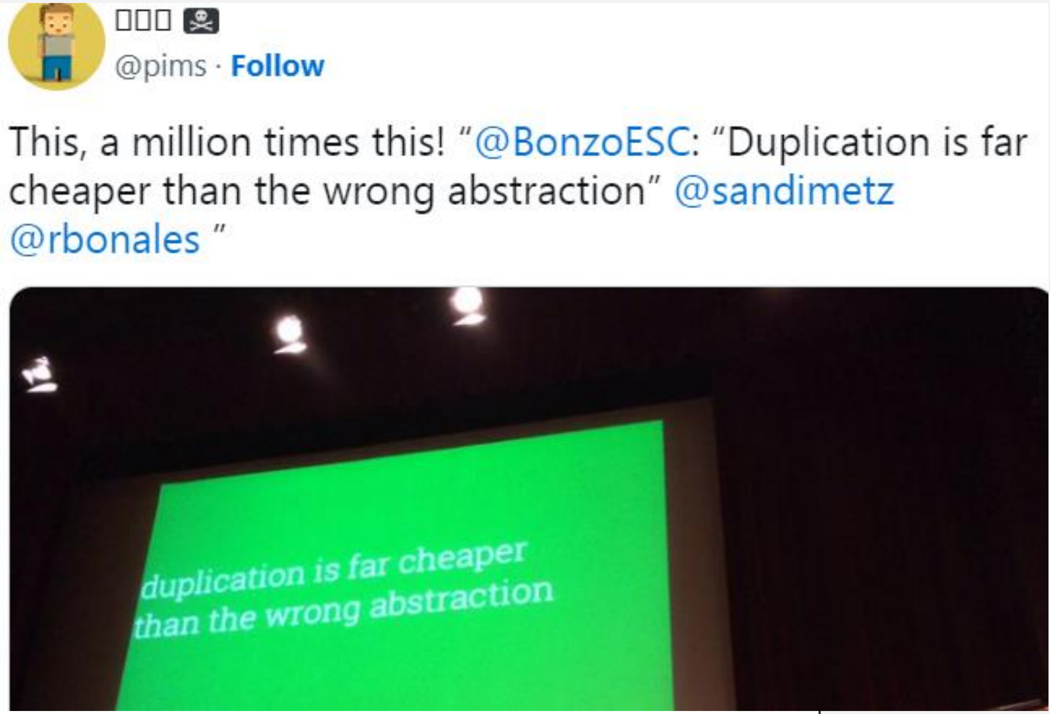
Creating **<u>Bad</u> Abstractions** is **<u>Dangerous</u>**

   - E.g., **a function(al) abstrac**
   - E.g., **a data abstraction**

Creating **<u>Good</u> Abstractions** is **<u>Hard</u>**

   - The **abstraction** <u>must</u> have a **short, clear name** and **“be logical”**
4. <u>Use</u> the **abstraction** by **giving concrete “arguments” parameters**

Abstraction Warning Story



I came to see the following pattern:

1. Programmer A sees duplication.
2. Programmer A extracts duplication and gives it a name.
   This creates a new abstraction.
3. Programmer A replaces the duplication with the new abstraction.
   Ah, the code is perfect. Programmer A trots happily away.

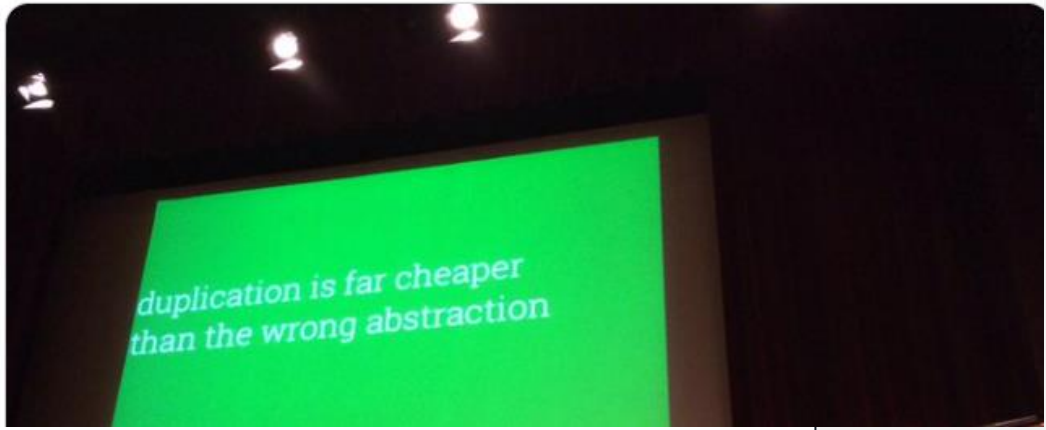
4. Time passes ...

https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction

Abstraction Warning Story

□□□ ☠
@pims · Follow

This, a million times this! "@BonzoESC: "Duplication is far cheaper than the wrong abstraction" @sandimetz @rbonales "



I came to see the following pattern:

1. <u>Programmer A</u> sees duplication.
2. <u>Programmer A</u> extracts duplication and gives it a name.
   This creates a new abstraction.
3. <u>Programmer A</u> replaces the duplication with the new abstraction.
   Ah, the code is perfect. Programmer A trots happily away.

4. Time passes …

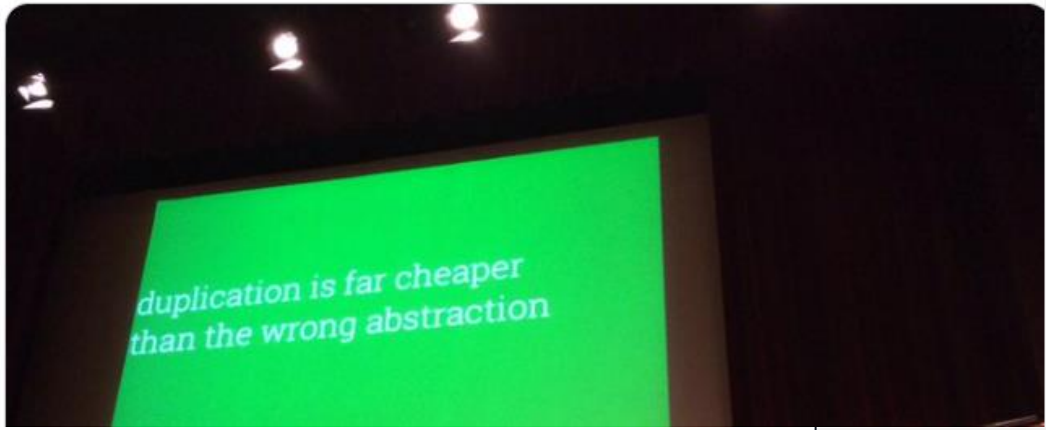
5. A new requirement appears for which the current abstraction is *almost* perfect.
6. <u>Programmer B</u> gets tasked to implement this requirement.
   <u>Programmer B</u> tries to retain the existing abstraction, but it's not perfect, so they alter the code to take a parameter, and then add extra logic that is conditionally based on the value of that parameter.

https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction

Abstraction Warning Story

@pims · Follow

This, a million times this! "@BonzoESC: "Duplication is far cheaper than the wrong abstraction" @sandimetz @rbonales "



I came to see the following pattern:

1. Programmer A sees duplication.
2. Programmer A extracts duplication and gives it a name.

   ...raction.

3. Programmer A replaces the duplication with the new abstraction.

   Ah, the code is perfect. Programmer A trots happily away.

How to avoid?

Always be thinking about the data

4. Time passes ...

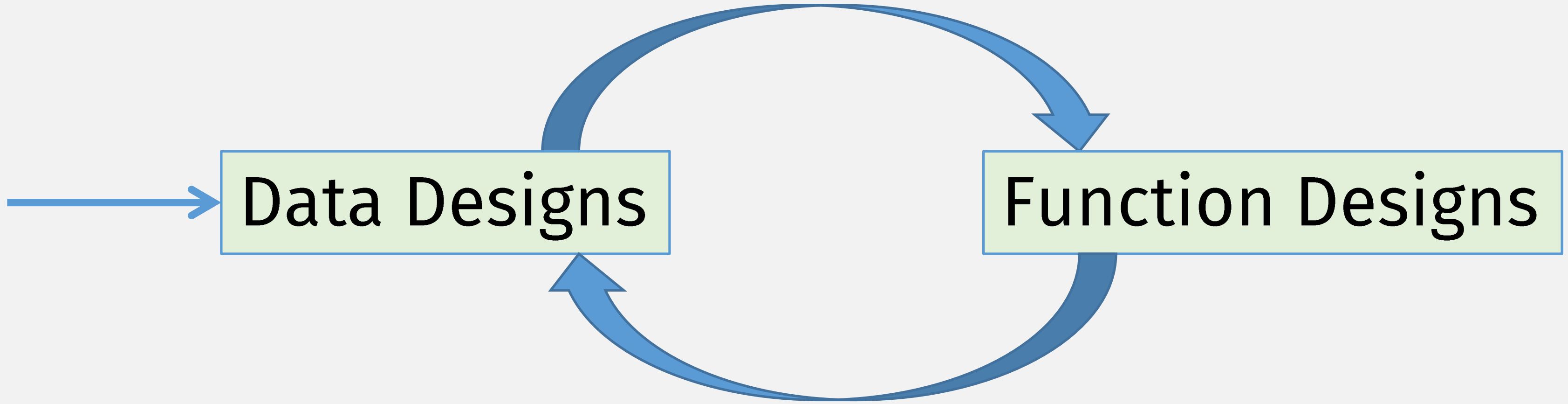
5. A new requirement appears for which the current abstraction is *almost* perfect.
6. Programmer B gets tasked to implement this requirement.

   Programmer B tries to retain the existing abstraction, but it's not perfect, so they alter the code to take a parameter, and then add extra logic that is conditionally based on the value of that parameter.

7. Another new requirement arrives. And a new *Programmer X, who adds an additional parameter and a new conditional. Loop until **code becomes incomprehensible**.*
8. You appear in the story about here, and your life takes a dramatic turn for the worse.

https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction

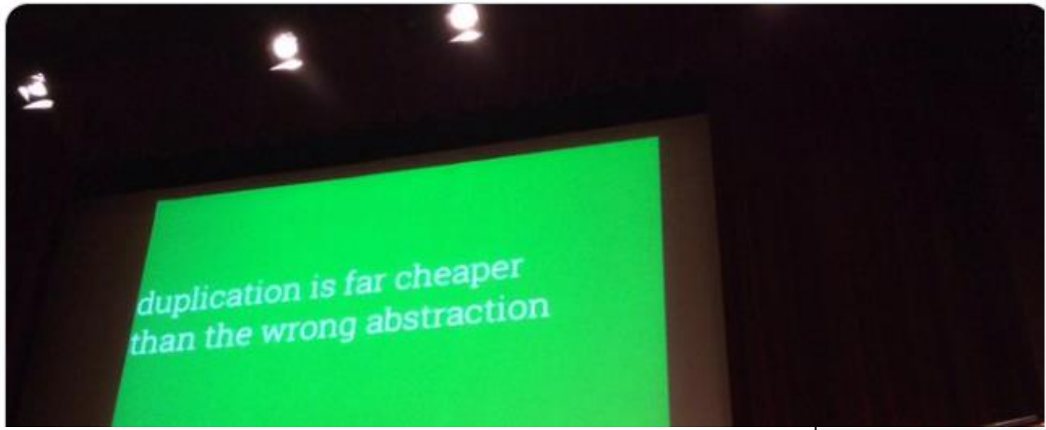
Program Design Recipe



Abstraction Warning Story

□□□ ☠
@pims · Follow

This, a million times this! "@BonzoESC: "Duplication is far cheaper than the wrong abstraction" @sandimetz @rbonales "



I came to see the following pattern:

1. Programmer A sees duplication.
2. Programmer A extracts duplication and gives it a name.
   ...raction.
3. Pr... ...uplication with the new abstraction.
   Ah, the code is perfect. Programmer A trots happily away.
4. Time passes ...
5. A ... *...most* perfect.
6. Programmer B gets tasked to implement this requirement.
   Programmer B *tries ... to take a parameter, and then* ***add extra logic*** *that is conditionally based on the value of that parameter.*
7. Another new requirement arrives. And a new *Programmer X, who adds an additional parameter and a new conditional. Loop until* ***code becomes incomprehensible.***
8. You appear in the story about here, and your life takes a dramatic turn for the worse.

How to avoid?

Always be thinking about the data

Don't focus only on "getting the code working"

These programmers only cared about "getting the code working"

https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction

Last Time

Common List Function #2: ???

Last Time

Comparison #2

```
;; sum-lst: ListofInt -> Int
(define (sum-lst lst)
  (cond
   [(empty? lst) 0]
   [else (+ (first lst)
            (sum-lst (rest lst)))]))
```

```
;; render-world : ListofBall -> Image
(define (render-world lst)
  (cond
   [(empty? lst) EMPTY-SCENE]
   [else (place-ball (first lst)
                     (render-world (rest lst)))]))
```

Abstraction Recipe

1. Find **similar patterns** in a program
   - Minimum: **2**
   - Ideally: **3+**
2. Identify **differences** and **make them parameters**
3. Create a **reusable abstraction** with the **discovered parameters**
   - E.g., **a function(al) abstraction**
   - E.g., **a data abstraction**
   - The **abstraction** must have a **short, clear name** and **“be logical”**
4. Use the **abstraction** by **giving concrete “arguments” parameters**

Last Time

Comparison #2

```
;; sum-lst: ListofInt -> Int
(define (sum-lst lst)
  (cond
   [(empty? lst) 0]
   [else (+ (first lst)
            (sum-lst (rest lst)))]))
```

```
;; render-world : ListofBall -> Image
(define (render-world lst)
  (cond
   [(empty? lst) EMPTY-SCENE]
   [else (place-ball (first lst)
                     (render-world (rest lst)))]))
```

Common List Function #2

X = Type of list element

Y = Result Type

```
;; list-fn2 : (X Y -> Y) Y Listof<X> -> Y
(define (lst-fn2 fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (lst-fn2 fn initial (rest lst)))]))
```

Abstraction Recipe

1. Find similar patterns in a program
   - Minimum: 2
   - Ideally: 3+
2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
   - The abstraction must have a short, clear name and “be logical”
4. Use the abstraction by giving concrete “arguments” parameters

Common List Function #2: **foldr**

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

Also **called "reduce"**
Because a **list of values** is **"reduced"** to **one value**

Abstraction Recipe

1. Find similar patterns in a program
   - Minimum: 2
   - Ideally: 3+
2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
   - The abstraction must have a short, clear name and “be logical”
4. Use the abstraction by giving concrete “arguments” parameters

Common List Function #2: `foldr`

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

```
;; sum-lst: ListofInt -> Int
(define (sum-lst lst) (foldr + 0 lst))
;; render-world: ListofBall-> Image
(define (render-world lst) (foldr place-ball EMPTY-SCENE lst))
```

Do we always want to start at the right?

For some functions, order doesn't matter, but for others, it does?

```
(foldr + 0 (list 1 2 3)) = (1 + (2 + (3 + 0)))
```

```
(1 + (2 + (3 + 0))) = (((1 + 0) + 2) + 3)
```

(Addition is associative)

```
(1 - (2 - (3 - 0))) ?!=? (((1 - 0) - 2) - 3)
```

Need List Function #2b: **`foldl`** (start from left)

Challenge:

- **Change `foldr` to `foldl`**
- so that the **function is applied from the left** (first element first)

```
(define (foldr fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

```
(1 + (2 + (3 + 0)))
```

```
(1 - (2 - (3 - 0)))
```

```
(define (foldl fn initial lst)
  (cond
   [(empty? lst) ....]
   [else .... (first lst) .... (foldl fn initial (rest lst)) ....]))
```

```
(((1 + 0) + 2) + 3)
```

```
(((1 - 0) - 2) - 3)
```

Need List Function #2b: **foldl** (start from left)

Y = Result Type

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

Expressions with needed “result” type:
- initial
- fn call
- recursive call itself

(look at signature to help)

```
;; foldl: (X Y -> Y) Y Listof<X> -> Y
(define (foldl fn initial lst)
  (cond
   [(empty? lst) ....]
   [else .... (first lst) .... (foldl fn initial (rest lst)) ....]))
```

Need List Function #2b: **`foldl`** (start from left)

Y = Result Type

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

Expressions with needed "result" type:
- `initial`
- `fn` call
- recursive call itself

(look at signature to help)

```
;; foldl: (X Y -> Y) Y Listof<X> -> Y
(define (foldl fn initial lst)
  (cond
   [(empty? lst) ....]
   [else (foldl .... (first lst) .... (rest lst))]))
```

Now fill in args to recursive call

Need List Function #2b: **foldl** (start from left)

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

```
;; foldl: (X Y -> Y) Y Listof<X> -> Y
(define (foldl fn initial lst)
  (cond
   [(empty? lst) ....]
   [else (foldl fn .... (first lst) .... (rest lst))]))
```

only argument with type of first arg is first arg itself

Need List Function #2b: `foldl` (start from left)

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

Expressions with needed "result" Y type:

- initial
- fn call
- recursive call itself

Now just need middle arg (and need to use the "`first`" piece)

```
;; foldl: (X Y -> Y) Y Listof<X> -> Y
(define (foldl fn initial lst)
  (cond
   [(empty? lst) ....]
   [else (foldl fn .... (first lst) .... (rest lst))]))
```

"rest" of list has proper "list" type

Need List Function #2b: **foldl** (start from left)

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

Expressions with needed “result” Y type:

- initial
- fn call
- recursive call itself

Now just need middle arg (and need to use the “first” piece)

```
;; foldl: (X Y -> Y) Y Listof<X> -> Y
(define (foldl fn initial lst)
  (cond
   [(empty? lst) ....]
   [else (foldl fn (fn (first lst) ....) (rest lst))]))
```

$(((1 + 0) + 2) + 3)$

What goes here? (look at signature)

(and examples)

Need List Function #2b: **`foldl`** (start from left)

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
    [(empty? lst) initial]
    [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

Expressions with needed "result" Y type:
- `initial`
- fn call
- recursive call itself

```
;; foldl: (X Y -> Y) Y Listof<X> -> Y
(define (foldl fn initial lst)
  (cond
    [(empty? lst) ....]
    [else (foldl fn (fn (first lst) initial) (rest lst))]))
```

```
(((1 + 0) + 2) + 3)
```

Need List Function #2b: **foldl** (start from left)

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

Expressions with needed “result” Y type:
- initial
- fn call
- recursive call itself

```
;; foldl: (X Y -> Y) Y Listof<X> -> Y
(define (foldl fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (foldl fn (fn (first lst) initial) (rest lst))]))
```

“initial”???

(((1 + 0) + 2) + 3)

Need List Function #2b: `foldl` (start from left)

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

Expressions with needed “result” Y type:

- ~~initial~~ **result-so-far**
- fn call
- recursive call itself

```
;; foldl: (X Y -> Y) Y Listof<X> -> Y
(define (foldl fn result-so-far lst)
  (cond
   [(empty? lst) result-so-far]
   [else (foldl fn (fn (first lst) result-so-far) (rest lst))]))
```

“result so far”

(((1 + 0) + 2) + 3)

Common list function #3

Your tasks

Follow the design recipe!

Write the following functions:

```
;; smaller-than: ListofInt Int -> ListofInt
;; Returns a list containing elements of given list
;; that are less than the given int
```

```
(check-equal?
  (smaller-than (list 1 3 4 5 9) 4)
                (list 1 3))
```

```
;; larger-than: ListofInt Int -> ListofInt
;; Returns a list containing elements of given list
;; that are greater than the given int
```

```
(check-equal?
  (greater-than (list 1 3 4 5 9) 4)
                (list 5 9))
```

```
;; quicksort: ListofInt -> ListofInt
;; sorts a given list (with no dups) in ascending order
(define (quicksort lst)
  (define pivot (random lst))
  (append (quicksort (smaller-than lst pivot)) pivot (quicksort (greater-than lst pivot))))
```

Your tasks

```
(define (smaller-than lst x)
  (cond
   [(empty? lst) empty]
   [else (if (< (first lst) x)
             (cons (first lst) (smaller-than (rest lst) x))
             (smaller-than (rest lst) x))]))
```

(**Repeated here is ok-ish**, because it will only get run once)

```
(define (larger-than lst x)
  (cond
   [(empty? lst) empty]
   [else (if (> (first lst) x)
             (cons (first lst) (larger-than (rest lst) x))
             (larger-than (rest lst) x))]))
```

Abstraction Recipe

1. Find similar patterns in a program
   - Minimum: 2
   - Ideally: 3+
2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
   - The abstraction must have a short, clear name and “be logical”
4. Use the abstraction by giving concrete “arguments” parameters

Your tasks

```
(define (smaller-than lst x)
  (cond
   [(empty? lst) empty]
   [else (if (< (first lst) x)
             (cons (first lst) (smaller-than (rest lst) x))
             (smaller-than (rest lst) x))]))
```

```
(define (larger-than lst x)
  (cond
   [(empty? lst) empty]
   [else (if (> (first lst) x)
             (cons (first lst) (larger-than (rest lst) x))
             (larger-than (rest lst) x))]))
```

Common list function #3?

Is this a “good” abstraction?

```
;; lst-fn3: ListofInt Int (Int Int -> Boolean) -> ListofInt
;; Returns a list containing elements of given list
;; that are ??? than the given int
```

```
(define (lst-fn3 lst other-int fn?)
  (cond
   [(empty? lst) empty]
   [else (if (fn? (first lst) other-int )
             (cons (first lst) (lst-fn3 (rest lst) other-int))
             (lst-fn3 (rest lst) other-int))]))
```

Abstraction Recipe

1. Find similar patterns in a program
   - Minimum: 2
   - Ideally: 3+
2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
   - The abstraction must have a short, clear name and "be logical"
4. Use the abstraction by giving concrete "arguments" parameters

Abstraction Recipe

1. Find similar patterns in a program
   - Minimum: 2
   - Ideally: 3+
2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
   - The abstraction must have a short, clear name and “be logical”
4. Use the abstraction by giving concrete “arguments” parameters

Common list function #3?

Is this a “good” abstraction?

What are possible use cases?

Should be **more than just the two examples we are abstracting!**

```
;; lst-fn3: ListofInt Int (Int Int -> Boolean) -> ListofInt
;; Returns a list containing elements of given list
;; that are ??? than the given int
```

```
(define (lst-fn3 lst other-int fn?)
  (cond
   [(empty? lst) empty]
   [else (if (fn? (first lst) other-int )
             (cons (first lst) (lst-fn3 (rest lst) other-int))
             (lst-fn3 (rest lst) other-int))]))
```

More tasks

Write the following functions:

```
(check-equal?
  (shorter-than (list “a” “bc “abc”) 2)
                (list “a”))
```

```
;; shorter-than: ListofString Int -> ListofString
;; Returns a list containing elements of given list
;; that have length less than the given int
```

```
(check-equal?
  (shorter-than-str (list “a” “bc” “abc”) “xy”)
                    (list “a”))
```

```
;; shorter-than-str: ListofString String -> ListofString
;; Returns a list containing elements of given list
;; that have length less than the given string
```

```
;; lst-fn3: ListofInt Int (Int Int -> Boolean) -> ListofInt
;; Returns a list containing elements of given list
;; that are ??? than the given int
```

Write the following functions:

```
;; shorter-than: ListofString Int -> ListofString
;; Returns a list containing elements of given list
;; that have length less than the given int
```

Could these be implemented with our new abstraction?

Should we be able to?

```
;; shorter-than-str: ListofString String -> ListofString
;; Returns a list containing elements of given list
;; that have length less than the given string
```

Abstraction Recipe

1. Find similar patterns in a program
   - Minimum: 2
   - Ideally: 3+
2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
   - The abstraction must have a short, clear name and "be logical"
4. Use the abstraction by giving concrete "arguments" parameters

Abstraction Recipe

Remember:
The Design Recipe (like good software development) **is iterative!**

1. Find **similar patterns** in a program
   - Minimum: **2**
   - Ideally: **3+**
2. Identify **differences** and **make them parameters**
3. Create a **reusable abstraction** with the **discovered parameters**
   - E.g., **a function(al) abstraction**
   - E.g., **a data abstraction**
   - The **abstraction** must have a **short, clear name** and **"be logical"**
4. Use the **abstraction** by **giving concrete "arguments" parameters**

Common list function #3?

Is this a "good" abstraction?

```
;; lst-fn3: ListofInt Int (Int Int -> Boolean) -> ListofInt
;; Returns a list containing elements of given list
;; that are ??? than the given int
```

```
(define (lst-fn3 lst other-int fn?)
  (cond
   [(empty? lst) empty]
   [else (if (fn? (first lst) other-int)
             (cons (first lst) (lst-fn3 (rest lst) other-int))
             (lst-fn3 (rest lst) other-int))]))
```

A Better common list function #3?

```
;; lst-fn3: Listof<X> (X -> Boolean) -> Listof<X>
;; Returns a list containing elements of given list
;; for which the given predicate returns true
```

```
(define (lst-fn3 lst other-int general-pred?)
  (cond
   [(empty? lst) empty]
   [else (if (general-pred? (first lst))
             (cons (first lst) (lst-fn3 (rest lst)))
             (lst-fn3 (rest lst)))]))
```

Common list function #3: `filter`

```
;; smaller-than: Listof<Int> Int -> Listof<Int>
;; Returns a list containing elements of given list less than the given int
```

```
(define (smaller-than lst thresh)
  (filter (lambda (x) (< x thresh)) lst))
```

lambda creates an anonymous "inline" function (expression)

```
;; filter: Listof<X> (X -> Boolean) -> Listof<X>
;; Returns a list containing elements of given list
;; for which the given predicate returns true
```

```
(define (filter lst pred?)
  (cond
   [(empty? lst) empty]
   [else (if (pred? (first lst))
             (cons (first lst) (filter (rest lst)))
             (filter (rest lst)))]))
```

Common list function #3: filter

```
;; smaller-than: Listof<Int> Int -> Listof<Int>
;; Returns a list containing elements of given list less than the given int
```

```
(define (smaller-than lst thresh)
  (filter (lambda (x) (< x thresh)) lst)
```

lambda creates an anonymous "inline" function (expression)

```
;; filter: Listof<X> (X -> Boolean) -> L
;; Returns a list containing elements of
;; for which the given predicate returns
```

```
(define (filter lst pred?)
  (cond
   [(empty? lst) empty]
   [else (if (pred? (first lst))
             (cons (first lst) (filter (
             (filter (rest lst)))]))
```

lambda rules:

- Can skip the **design recipe** steps, BUT
- **name**, **description**, and **signature** must be "obvious"
- **code** is arithmetic only
- otherwise, **create standalone function define**

Your Remaining tasks

Submitting

1. File: `in-class-10-07-<Last>-<First>.rkt`
2. Join the in-class team: `cs450f24/teams/in-class`
3. Commit to repo: `cs450f24/in-class-10-07`
   - (May need to `merge/pull + rebase`)

Implement with `filter`

```
;; smaller-than: ListofInt Int -> ListofInt
;; Returns list containing elements of given list less than the given int
```

```
;; larger-than: ListofInt Int -> ListofInt
;; Returns list containing elements of given list greater than the given int
```

```
;; shorter-than: ListofString Int -> ListofString
;; Returns list containing elements of given list with length less than given int
```

```
;; shorter-than-str: ListofString String -> ListofString
;; Returns list containing elements of given list with length less than given string
```