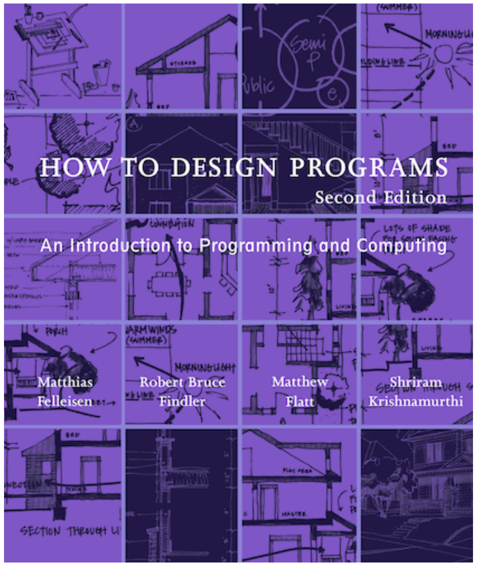
How to Design Programs

using Plait



`http://www.htdp.org`

How to Design Programs

- Determine the **representation**
  - `define-type`, if needed
- Write **examples**
  - `test`
- Create a **template** for the implementation
  - `type-case`, if variants
  - extract field values, if any
  - cross- and self-calls, if data references
- Finish **body** implementation case-by-case
- Run **tests**

Representation

- Keep track of the number of cookies in a cookie jar

```
Number

eat-cookie : (Number -> Number)
```

Examples

```
Number

eat-cookie :  (Number -> Number)

(test
       )
```

Examples

```
Number

eat-cookie : (Number -> Number)

 (test (eat-cookie    )
       )
```

Examples

```
Number

eat-cookie :  (Number -> Number)

(test (eat-cookie 10)
      )
```

Examples

```
Number

eat-cookie :  (Number -> Number)

 (test  (eat-cookie 10)
        9)
```

Examples

```
Number

eat-cookie : (Number -> Number)

(test (eat-cookie 10)
      9)
(test (eat-cookie 1)
      0)
(test (eat-cookie 0)
      0)
```

Template

```
Number

eat-cookie : (Number -> Number)

(define (eat-cookie [n : Number])
  ... n ...)
```

Body

```
Number

eat-cookie : (Number -> Number)

(define (eat-cookie [n : Number])
  ... n ...)



(test (eat-cookie 10)
      9)
(test (eat-cookie 1)
      0)
(test (eat-cookie 0)
      0)
```

Body

```
Number

eat-cookie : (Number -> Number)

(define (eat-cookie [n : Number])
  (if (> n 0)
      (- n 1)
      0))

(test (eat-cookie 10)
      9)
(test (eat-cookie 1)
      0)
(test (eat-cookie 0)
      0)
```

Test

```
Number

eat-cookie : (Number -> Number)

(define (eat-cookie [n : Number])
  (if (> n 0)
      (- n 1)
      0))

(test (eat-cookie 10)
      9)
(test (eat-cookie 1)
      0)
(test (eat-cookie 0)
      0)
```

Representation

- Track a position on the screen

```
(define-type Posn
  (posn [x : Number]
        [y : Number]))

flip : (Posn -> Posn)
```

Examples

```
(define-type Posn
  (posn [x : Number]
        [y : Number]))

flip : (Posn -> Posn)

(test (flip                 )
                  )
```

Examples

```
(define-type Posn
  (posn [x : Number]
        [y : Number]))

flip : (Posn -> Posn)

(test (flip (posn 1 17))
                 )
```

Examples

```
(define-type Posn
  (posn [x : Number]
        [y : Number]))

flip : (Posn -> Posn)

(test (flip (posn 1 17))
      (posn 17 1))
```

Examples

```
(define-type Posn
  (posn [x : Number]
        [y : Number]))

flip : (Posn -> Posn)

(test (flip (posn 1 17))
      (posn 17 1))
(test (flip (posn -3 4))
      (posn 4 -3))
```

Template

```
(define-type Posn
  (posn [x : Number]
        [y : Number]))

flip : (Posn -> Posn)

(define (flip [p : Posn])
  ... (posn-x p)
  ... (posn-y p) ...)
```

or

```
(define (flip [p : Posn])
  (type-case Posn p
    [(posn x y) ... x ... y ...]))
```

Body

```
(define-type Posn
  (posn [x : Number]
        [y : Number]))

flip : (Posn -> Posn)

(define (flip [p : Posn])
  (type-case Posn p
    [(posn x y)  ... x ... y ...]))

(test (flip (posn 1 17))
      (posn 17 1))
(test (flip (posn -3 4))
      (posn 4 -3))
```

Body

```
(define-type Posn
  (posn [x : Number]
        [y : Number]))

flip : (Posn -> Posn)

(define (flip [p : Posn])
  (type-case Posn p
    [(posn x y) (posn y x)]))

(test (flip (posn 1 17))
      (posn 17 1))
(test (flip (posn -3 4))
      (posn 4 -3))
```

Representation

- Track an ant, which has a location and a weight

```
(define-type Ant
  (ant [location : Posn]
       [weight : Number]))

ant-at-home? : (Ant -> Boolean)
```

Examples

```
   (define-type Ant
     (ant [location : Posn]
          [weight : Number]))

   ant-at-home? : (Ant -> Boolean)

(test (ant-at-home? (ant (posn 0 0) 0.0001))
      #t)
(test (ant-at-home? (ant (posn 5 10) 0.0001))
      #f)
```

Template

```
(define-type Ant
  (ant [location : Posn]
       [weight : Number]))

ant-at-home? : (Ant -> Boolean)

(define (ant-at-home? [a : Ant])
  (type-case Ant a
    [(ant loc wgt)
     ... loc ...
     ... wgt ...]))
```

Template

```
(define-type Ant
  (ant [location : Posn]
       [weight : Number]))

ant-at-home? : (Ant -> Boolean)

(define (ant-at-home? [a : Ant])
  (type-case Ant a
    [(ant loc wgt)
     ... (is-home? loc) ...
     ... wgt ...]))

(define (is-home? [p : Posn])
  (type-case Posn p
    [(posn x y) ... x ... y ...]))
```

Body

```
(define-type Ant
  (ant [location : Posn]
       [weight : Number]))

ant-at-home? : (Ant -> Boolean)

(define (ant-at-home? [a : Ant])
  (type-case Ant a
    [(ant loc wgt)
     ... (is-home? loc) ...
     ... wgt ...]))

(define (is-home? [p : Posn])
  (type-case Posn p
    [(posn x y) ... x ... y ...]))
```

Body

```
(define-type Ant
  (ant [location : Posn]
       [weight : Number]))

ant-at-home? : (Ant -> Boolean)

(define (ant-at-home? [a : Ant])
  (type-case Ant a
    [(ant loc wgt) (is-home? loc)]))

(define (is-home? [p : Posn])
  (type-case Posn p
    [(posn x y) ... x ... y ...]))
```

Body

```
(define-type Ant
  (ant [location : Posn]
       [weight : Number]))

ant-at-home? : (Ant -> Boolean)

(define (ant-at-home? [a : Ant])
  (type-case Ant a
    [(ant loc wgt) (is-home? loc)]))

(define (is-home? [p : Posn])
  (type-case Posn p
    [(posn x y) (and (zero? x)
                     (zero? y))]))
```

Representation

- Track an animal, which is a tiger or a snake

```
(define-type Animal
  (tiger [color : Symbol]
         [stripe-count : Number])
  (snake [color : Symbol]
         [weight : Number]
         [food : String]))
```

```
heavy-animal? : (Animal -> Boolean)
```

Examples

```
    (define-type Animal
      (tiger [color : Symbol]
             [stripe-count : Number])
      (snake [color : Symbol]
             [weight : Number]
             [food : String]))

    heavy-animal? : (Animal -> Boolean)

(test (heavy-animal?                    )
        )
```

Examples

```
    (define-type Animal
      (tiger [color : Symbol]
             [stripe-count : Number])
      (snake [color : Symbol]
             [weight : Number]
             [food : String]))

    heavy-animal? : (Animal -> Boolean)

(test (heavy-animal? (tiger 'orange 14))
        )
```

Examples

```
(define-type Animal
  (tiger [color : Symbol]
         [stripe-count : Number])
  (snake [color : Symbol]
         [weight : Number]
         [food : String]))
```

```
heavy-animal? : (Animal -> Boolean)
```

```
(test (heavy-animal? (tiger 'orange 14))
      #t)
```

Examples

```
    (define-type Animal
      (tiger [color : Symbol]
             [stripe-count : Number])
      (snake [color : Symbol]
             [weight : Number]
             [food : String]))

    heavy-animal? : (Animal -> Boolean)

(test (heavy-animal? (tiger 'orange 14))
      #t)
(test (heavy-animal? (snake 'green 10 "rats"))
        )
```

Examples

```
    (define-type Animal
      (tiger [color : Symbol]
             [stripe-count : Number])
      (snake [color : Symbol]
             [weight : Number]
             [food : String]))

    heavy-animal? : (Animal -> Boolean)

(test (heavy-animal? (tiger 'orange 14))
      #t)
(test (heavy-animal? (snake 'green 10 "rats"))
      #t)
```

Examples

```
    (define-type Animal
      (tiger [color : Symbol]
             [stripe-count : Number])
      (snake [color : Symbol]
             [weight : Number]
             [food : String]))

    heavy-animal? : (Animal -> Boolean)

(test (heavy-animal? (tiger 'orange 14))
      #t)
(test (heavy-animal? (snake 'green 10 "rats"))
      #t)
(test (heavy-animal? (snake 'yellow 8 "cake"))
      #f)
```

Template

```
(define-type Animal
  (tiger [color : Symbol]
         [stripe-count : Number])
  (snake [color : Symbol]
         [weight : Number]
         [food : String]))

heavy-animal? : (Animal -> Boolean)

(define (heavy-animal? [a : Animal])
  (type-case Animal a
    [(tiger c sc)
     ... c ... sc ...]
    [(snake c w f)
     ... c ... w ...
     ... f ...]))
```

Body

```
(define-type Animal
  (tiger [color : Symbol]
         [stripe-count : Number])
  (snake [color : Symbol]
         [weight : Number]
         [food : String]))

heavy-animal? : (Animal -> Boolean)

(define (heavy-animal? [a : Animal])
  (type-case Animal a
    [(tiger c sc)
     ... c ... sc ...]
    [(snake n w f)
     ... c ... w ...
     ... f ...]))
```

Body

```
(define-type Animal
  (tiger [color : Symbol]
         [stripe-count : Number])
  (snake [color : Symbol]
         [weight : Number]
         [food : String]))

heavy-animal? : (Animal -> Boolean)

(define (heavy-animal? [a : Animal])
  (type-case Animal a
    [(tiger c sc) #t]
    [(snake c w f)
     ... c ... w ...
     ... f ...]))
```

Body

```
(define-type Animal
  (tiger [color : Symbol]
         [stripe-count : Number])
  (snake [color : Symbol]
         [weight : Number]
         [food : String]))

heavy-animal? : (Animal -> Boolean)

(define (heavy-animal? [a : Animal])
  (type-case Animal a
    [(tiger c sc) #t]
    [(snake c w f) (>= w 10)]))
```

Representation

- Track an aquarium, which has any number of fish, each with a weight

```
(define-type Listof-Number
  (emptyL)
  (biggerL [n : Number]
           [rst : Listof-Number]))

feed-fish : (Listof-Number -> Listof-Number)
```

Representation

- Track an aquarium, which has any number of fish, each with a weight

```
(define-type (Listof Number)
  empty
  (cons [n : Number]
        [rst : (Listof Number)]))

feed-fish : ((Listof Number) -> (Listof Number))
```

Examples

```
 (define-type (Listof Number)
   empty
   (cons [n : Number]
         [rst : (Listof Number)]))

 feed-fish : ((Listof Number) -> (Listof Number))

(test (feed-fish empty)
      empty)
(test (feed-fish (cons 1 (cons 2 (cons 3 empty))))
      (cons 2 (cons 3 (cons 4 empty))))
```

Examples

```
(define-type (Listof Number)
  empty
  (cons [n : Number]
        [rst : (Listof Number)]))

feed-fish : ((Listof Number) -> (Listof Number))

     (test (feed-fish (list))
           (list))
     (test (feed-fish (list 1 2 3))
           (list 2 3 4))
```

Template

```
(define-type (Listof Number)
  empty
  (cons [n : Number]
        [rst : (Listof Number)]))

feed-fish : ((Listof Number) -> (Listof Number))

   (define (feed-fish [lon : (Listof Number)])
     (type-case (Listof Number) lon
      [empty ...]
      [(cons n rst-lon) ...]))
```

Template

```
(define-type (Listof Number)
  empty
  (cons [n : Number]
        [rst : (Listof Number)]))

feed-fish : ((Listof Number) -> (Listof Number))

   (define (feed-fish [lon : (Listof Number)])
     (type-case (Listof Number) lon
      [empty ...]
      [(cons n rst-lon)
       ... n ...
       ... rst-lon ...]))
```

Template

```
(define-type (Listof Number)
  empty
  (cons [n : Number]
        [rst : (Listof Number)]))

feed-fish : ((Listof Number) -> (Listof Number))

  (define (feed-fish [lon : (Listof Number)])
    (type-case (Listof Number) lon
     [empty ...]
     [(cons n rst-lon)
      ... n ...
      ... (feed-fish rst-lon) ...]))
```

Body

```
(define-type (Listof Number)
  empty
  (cons [n : Number]
        [rst : (Listof Number)]))

feed-fish : ((Listof Number) -> (Listof Number))

   (define (feed-fish [lon : (Listof Number)])
     (type-case (Listof Number) lon
      [empty ...]
      [(cons n rst-lon)
       ... n ...
       ... (feed-fish rst-lon) ...]))
```

Body

```
(define-type (Listof Number)
  empty
  (cons [n : Number]
        [rst : (Listof Number)]))

feed-fish : ((Listof Number) -> (Listof Number))

  (define (feed-fish [lon : (Listof Number)])
    (type-case (Listof Number) lon
     [empty ...]
     [(cons n rst-lon)
      ... n ...
      ... (feed-fish rst-lon) ...]))
    (test (feed-fish empty)
          empty)
    (test (feed-fish (cons 1 (cons 2 (cons 3 empty))))
          (cons 2 (cons 3 (cons 4 empty))))
```

Body

```
(define-type (Listof Number)
  empty
  (cons [n : Number]
        [rst : (Listof Number)]))

feed-fish : ((Listof Number) -> (Listof Number))

  (define (feed-fish [lon : (Listof Number)])
    (type-case (Listof Number) lon
     [empty empty]
     [(cons n rst-lon)
      ... n ...
      ... (feed-fish rst-lon) ...]))
    (test (feed-fish empty)
          empty)
    (test (feed-fish (cons 1 (cons 2 (cons 3 empty))))
          (cons 2 (cons 3 (cons 4 empty))))
```

Body

```
(define-type (Listof Number)
  empty
  (cons [n : Number]
        [rst : (Listof Number)]))

feed-fish : ((Listof Number) -> (Listof Number))

  (define (feed-fish [lon : (Listof Number)])
    (type-case (Listof Number) lon
     [empty empty]
     [(cons n rst-lon)
      ... (+ 1 n) ...
      ... (feed-fish rst-lon) ...]))
  (test (feed-fish empty)
        empty)
  (test (feed-fish (cons 1 (cons 2 (cons 3 empty))))
        (cons 2 (cons 3 (cons 4 empty))))
```

Body

```
(define-type (Listof Number)
  empty
  (cons [n : Number]
        [rst : (Listof Number)]))

feed-fish : ((Listof Number) -> (Listof Number))

   (define (feed-fish [lon : (Listof Number)])
     (type-case (Listof Number) lon
      [empty empty]
      [(cons n rst-lon)
       ... (+ 1 n) ...
       ... (feed-fish rst-lon) ...]))
      (test (feed-fish empty)
            empty)
      (test (feed-fish (cons 1 (cons 2 (cons 3 empty))))
            (cons 2 (cons 3 (cons 4 empty))))
```

rst-lon

Body

```
(define-type (Listof Number)
  empty
  (cons [n : Number]
        [rst : (Listof Number)]))

feed-fish : ((Listof Number) -> (Listof Number))

  (define (feed-fish [lon : (Listof Number)])
    (type-case (Listof Number) lon
     [empty empty]
     [(cons n rst-lon)
      ... (+ 1 n) ...
      ... (feed-fish rst-lon) ...]))
```

```
(test (feed-fish empty)
      empty)
(test (feed-fish (cons 1 (cons 2 (cons 3 empty))))
      (cons 2 (cons 3 (cons 4 empty))))
```

`rst-lon`

`(feed-fish rst-lon)`

Body

```
(define-type (Listof Number)
  empty
  (cons [n : Number]
        [rst : (Listof Number)]))

feed-fish : ((Listof Number) -> (Listof Number))

  (define (feed-fish [lon : (Listof Number)])
    (type-case (Listof Number) lon
     [empty empty]
     [(cons n rst-lon)
      (cons (+ 1 n)
            (feed-fish rst-lon))]))
```

```
(test (feed-fish empty)
      empty)
(test (feed-fish (cons 1 (cons 2 (cons 3 empty))))
      (cons 2 (cons 3 (cons 4 empty))))
```

rst-lon

(feed-fish rst-lon)

Implementation Matches Data

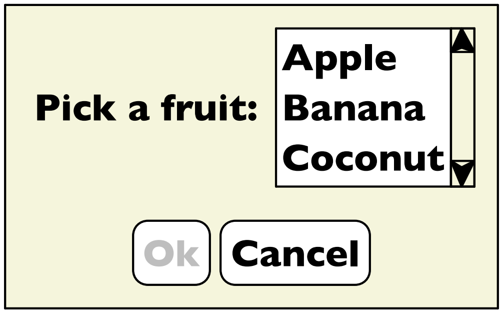
```
(define-type (Listof Number)
  empty
  (cons [n : Number]
        [rst : (Listof Number)]))
```

```
(define (feed-fish [lon : (Listof Number)])
  (type-case (Listof Number) lon
   [empty ...]
   [(cons n rst-lon) ... n
    ... (feed-fish rst-lon) ...]))
```

How to Design Programs

More Examples

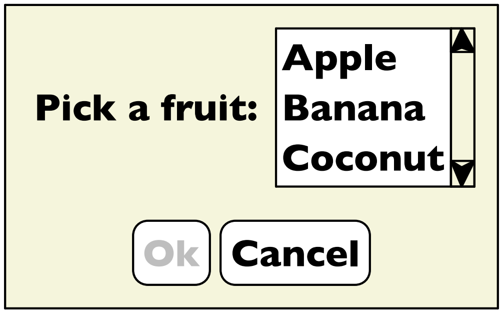
GUIs



Possible programs:

- Can click?
- Find a label
- Read screen

Representing GUIs



- labels
  - a label string
- buttons
  - a label string
  - enabled state
- lists
  - a list of choice strings
  - selected item

```
(define-type GUI
  (label [text : String])
  (button [text : String]
          [enabled? : Boolean])
  (choice [items : (Listof String)]
          [selected : Number]))
```

Read Screen

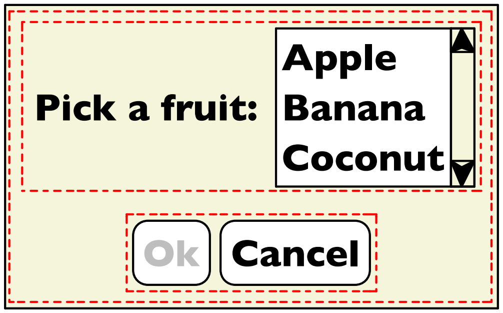
- Implement **`read-screen`**, which takes a GUI and returns a list of strings for all the GUI element labels

Read Screen

```
(define (read-screen [g : GUI]) : (Listof String)
  (type-case GUI g
    [(label t) (list t)]
    [(button t e?) (list t)]
    [(choice i s) i]))

(test (read-screen (label "Hi"))
      (list "Hi"))
(test (read-screen (button "Ok" #t))
      (list "Ok"))
(test (read-screen (choice (list "Apple" "Banana")
                           0))
      (list "Apple" "Banana"))
```

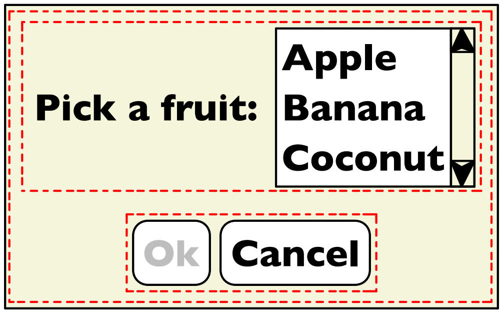
Assembling GUIs



- label
- buttons
- lists
- vertical stacking
  - two sub-GUIs
- horizontal stacking
  - two sub-GUIs

```
(define-type GUI
  (label [text : String])
  (button [text : String]
          [enabled? : Boolean])
  (choice [items : (Listof String)]
          [selected : Number])
  (vertical [top : GUI]
            [bottom : GUI])
  (horizontal [left : GUI]
              [right : GUI]))
```

Assembling GUIs



- label
- buttons
- lists
- vertical stacking
  - two sub-GUIs
- horizontal stacking
  - two sub-GUIs

```
(define gui1
  (vertical
   (horizontal
    (label "Pick a fruit:")
    (choice
     (list "Apple" "Banana" "Coconut")
     0))
   (horizontal
    (button "Ok" #f)
    (button "Cancel" #t))))
```

Read Screen

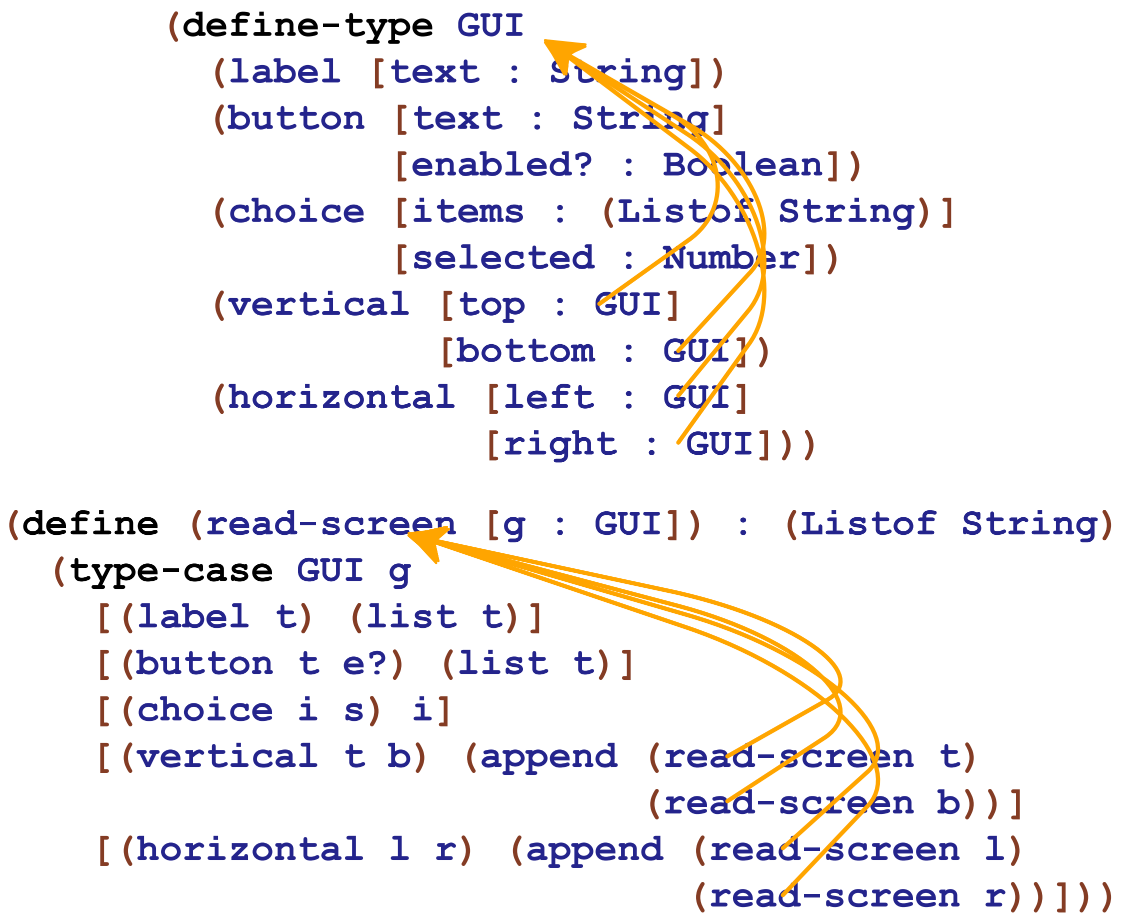
- Implement **`read-screen`**, which takes a GUI and returns a list of strings for all the GUI element labels

Read Screen

```
(define (read-screen [g : GUI]) : (Listof String)
  (type-case GUI g
    [(label t) (list t)]
    [(button t e?) (list t)]
    [(choice i s) i]
    [(vertical t b) (append (read-screen t)
                            (read-screen b))]
    [(horizontal l r) (append (read-screen l)
                              (read-screen r))]))

...
(test (read-screen gui1)
      (list "Pick a fruit:"
            "Apple" "Banana" "Coconut"
            "Ok" "Cancel"))
```

Function and Data Shapes Match



```
    (define-type GUI
      (label [text : String])
      (button [text : String]
              [enabled? : Boolean])
      (choice [items : (Listof String)]
              [selected : Number])
      (vertical [top : GUI]
                [bottom : GUI])
      (horizontal [left : GUI]
                  [right : GUI]))

(define (read-screen [g : GUI]) : (Listof String)
  (type-case GUI g
    [(label t) (list t)]
    [(button t e?) (list t)]
    [(choice i s) i]
    [(vertical t b) (append (read-screen t)
                            (read-screen b))]
    [(horizontal l r) (append (read-screen l)
                              (read-screen r))]))
```

Design Steps

- Determine the representation
  - `define-type`, maybe
- Write examples
  - `test`
- Create a template for the implementation
  - `type-case` plus natural recursion, **check shape!**
- Finish body implementation case-by-case
  - *usually the interesting part*
- Run tests

Enable Button

- Implement **`enable-button`**, which takes a GUI and a string and enables the button whose name matches the string

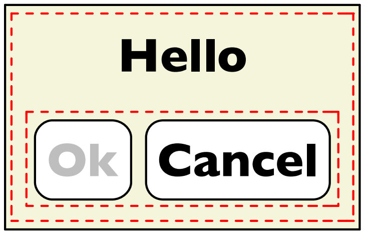
Enable Button

The **name** argument is “along for the ride”:

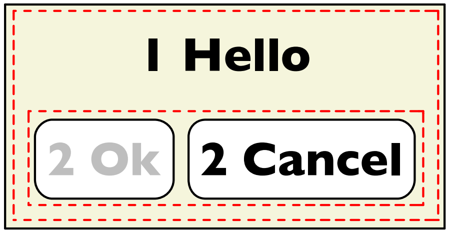
```
(define (enable-button [g : GUI] [name : String]) : GUI
  (type-case GUI g
    [(label t) g]
    [(button t e?) (cond
                    [(equal? t name) (button t #t)]
                    [else g])]
    [(choice i s) g]
    [(vertical t b) (vertical (enable-button t name)
                              (enable-button b name))]
    [(horizontal l r) (horizontal (enable-button l name)
                                  (enable-button r name))]))
...
(test (enable-button gui1 "Ok")
      (vertical
       (horizontal (label "Pick a fruit:")
                   (choice (list "Apple" "Banana" "Coconut") 0))
       (horizontal (button "Ok" #t)
                   (button "Cancel" #t))))
```

Show Depth

(test (show-depth



)



)

Show Depth

Template:

```
(define (show-depth [g : GUI]) : GUI
  (type-case GUI g
    [(label t) ... t ...]
    [(button t e?) ... t ... e? ...]
    [(choice i s) ... i ... s ...]
    [(vertical t b) ... (show-depth t)
                    ... (show-depth b) ...]
    [(horizontal l r) ... (show-depth l)
                      ... (show-depth r) ...]))
```

`(show-depth` $\boxed{\textbf{Ok}}$`)` → $\boxed{\textbf{0 Ok}}$

Show Depth

Template:

```
(define (show-depth [g : GUI]) : GUI
  (type-case GUI g
    [(label t) ... t ...]
    [(button t e?) ... t ... e? ...]
    [(choice i s) ... i ... s ...]
    [(vertical t b) ... (show-depth t)
                    ... (show-depth b) ...]
    [(horizontal l r) ... (show-depth l)
                      ... (show-depth r) ...]))
```

`(show-depth` Ok Cancel `)` → ... 0 Ok ... 0 Cancel ...

Show Depth

Template:

```
(define (show-depth [g : GUI]) : GUI
  (type-case GUI g
    [(label t) ... t ...]
    [(button t e?) ... t ... e? ...]
    [(choice i s) ... i ... s ...]
    [(vertical t b) ... (show-depth t)
                    ... (show-depth b) ...]
    [(horizontal l r) ... (show-depth l)
                      ... (show-depth r) ...]))
```

recursion results don't have the right labels...

Show Depth

The `n` argument is an *accumulator*:

```
(define (show-depth-at [g : GUI] [n : Number]) : GUI
  (type-case GUI g
    [(label t) (label (prefix n t))]
    [(button t e?) (button (prefix n t) e?)]
    [(choice i s) g]
    [(vertical t b) (vertical (show-depth-at t (+ n 1))
                              (show-depth-at b (+ n 1)))]
    [(horizontal l r) (horizontal (show-depth-at l (+ n 1))
                                  (show-depth-at r (+ n 1)))]))

(define (show-depth [g : GUI]) : GUI
  (show-depth-at g 0))
```

How to Design Programs

- Follow the design steps
- Use accumulators when necessary
- Reuse functions and/or "wish" for helpers