

Good programming practices

- Why good design matters
- Code design
 - Behavioral spec
 - Data structures
 - Procedures
- Documentation
- Debugging
- Evaluation and verification

9/28/2005

1

The screenshot shows the top portion of a Wall Street Journal article. The page is dated Friday, September 23, 2005. The article title is "Code Red: Battling Google, Microsoft Changes How It Builds Software" by Robert A. Guth. The sub-headline reads "Delay in New Windows Version Drove Giant to Develop Simpler, Flexible Product". A quote from Jim Allchin is highlighted with a bracket: "It's not going to work," Mr. Allchin says he told the Microsoft chairman. The new version, code-named Longhorn, was so complex its writers would never be able to make it run properly. Another quote follows: "The news got even worse: Longhorn was irredeemable because Microsoft engineers were building it just as they had always built software. Throughout its history, Microsoft had let thousands of programmers each produce their own piece of computer code, then stitched it together into one sprawling program. Now, Mr. Allchin argued, the jig was up. Microsoft needed to start over."

2

Read the entire article:

libraries.mit.edu

VERA

Wall Street Journal

Sept 23 2005

"Code Red: Battling Google..."

9/28/2005

5

Good Design Matters

- Because you'll never get big projects to work.
- Because they'll become moribund: unfixable, unmodifiable.

9/28/2005

6

Code Design: Behavior

- Figure out what it's supposed to *do*.
- Example: Basic personal calendar manager
 - Capabilities?

9/28/2005

7

Code Design: Data

- What basic objects are in this world?
 -
- What are the relations among them?

9/28/2005

8

Code Design: Data

- Data structure design
 - appointment?

9/28/2005

10

Code Design: Procedures

- Computation to be reused
 - What computations appear to be specific to this problem?
 - What computations are likely to be used elsewhere?

9/28/2005

12

Code Design: Test Cases

- Write the test cases *first*
 - Helps you anticipate the tricky parts
 - Encourages you to write a general solution

9/28/2005

13

Code Design: Test Cases

- Choosing good test cases
 - Pick values at limits of legal range
 - Base case of recursive procedure
 - Pick values that span legal range
 - Pick values that reflect different kinds of input
 - Odd versus even integers
 - Empty list, versus single element list, versus many element list

9/28/2005

14

Code Design: Test Cases

- Example: set-union

```
(define (set-union s1 s2) . . . )
```

9/28/2005

15

Coding Style

- Write so it's clear first, fast second
- Write so it's clear first and second, fast third...
- Why
 - Code reading vs. code running
 - Moore's Law
 - **The computer, the desk lamp, and the speed of light**

9/28/2005

16

Documenting code

- Supporting code maintenance
 - Can you read your code a year after writing it and understand what it is supposed to do?
 - Can you read your code a year after writing it and still understand why you made particular design decisions?
- Identifying input/output behaviors
 - Specify expectations on input and the associated contract on output of a procedure

9/28/2005

18

Documenting code

- Description of input/output behavior
- Expected or required types of arguments
- Type of returned value
- List of constraints that must be satisfied by arguments or stages of computation
- Expected state of computation at key points in code

9/28/2005

19

An example of code documentation

```
(define sqrt-helper
  (lambda (X guess)
    ;; compute approximate square root by
    ;; successive refinement, guess is current
    ;; approximation, X is number whose square
    ;; root we are seeking.
    ;; Type: (number, number) → number
    ;; constraint: guess^2 == X
    (if (good-enuf? X guess) ; can we stop?
        guess                ; if yes, then return
        (sqrt-helper X
                     (improve X guess)
                     ; if not, then get better guess
                     ; and repeat process
                     )))
```

9/28/2005

20

Debugging errors

- Common sources of errors
- Common tools to debug

9/28/2005

21

Common errors

- Unbound variable
 - Cause: typo
 - Solution: search for instance

9/28/2005

22

Common errors

- Unbound variable
 - Cause: reference outside scope of binding
 - Solution:
 - Search for instance
 - Use debugging tools to isolate instance

```
(define sqrt (lambda (x)
  (define good-enuf?
    (lambda (guess)
      (< (abs (- (square guess) x))
         0.001)))
  (define try (lambda (n)
    (if (good-enuf? guess)
        guess
        (try (improve n)))))
  (try 1)))
```

9/28/2005

23

Syntax errors

- Wrong number of arguments
 - Source: programming error
 - Solution: use debugger to isolate instance
- Type errors
 - As procedure (5 * 3)
 - As arguments
 - Source: calling error
 - Solution: trace back through chain of calls

9/28/2005

24

Conceptual errors

- Wrong initialization of parameters
- Wrong base case
- Wrong end test
- ... and so on

9/28/2005

25

Evaluation and verification

- Test individual modules
- Retest prior cases after making code changes (regression testing)

9/28/2005

26

Debugging tools

- The **ubiquitous** print/display expression
- Stepping
 - Show the state of computation at each stage of substitution model
- Tracing
 - Print out values of parameters on input to a procedure(s)
 - Print out value return on exit of procedure(s)

9/28/2005

27

Stepping In Dr. Scheme

- Change language level to “Intermediate Student with Lambda”, press RUN.
- Input expression, press Stepper button.

9/28/2005

28

Tracing in Dr. Scheme

- Change language level to “Essentials of Programming Languages”, press RUN.
- Add to top of definition window:

```
(require (lib "trace.ss"))
```
- Indicate which function(s) to trace:

```
(trace fact)
```

9/28/2005

29

Debugging

- We want to compute sines, using the mathematical approximation

$$\sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

9/28/2005

31

Initial code example

```
(define (sine x)
  (define (aux x n current)
    (let ((next (/ (expt x n) (fact n))))
      ;; compute next term
      (if (small-enuf? next) ;; if small
          current ;; just return current guess
          (aux x (+ n 1) (+ current next))
          ;; otherwise, create new guess
          )))
  (aux x 1 0))
```

9/28/2005

32

Test cases

```
(sine 0) ; should be 0
;Value: 0
```

```
(sine 3.1415927) ; should be 0
;Value: 22.140666527138016
```

```
(sine (/ 3.1415927 2.0)) ; should be 1
;Value: 3.8104481565660486
```

9/28/2005

33

Chasing down the error

```
(define (sine x)
  (define (aux x n current)
    (newline)
    (display "n is ")
    (display n)
    (display " current is ")
    (display current)
    (let ((next (/ (expt x n) (fact n))))
      (if (small-enuf? next)
          current
          (aux x (+ n 1) (+ current next)))))
  (aux x 1 0))
```

9/28/2005

34

Test cases

```
(sine 3.1415927)
n is 1 current is 0
n is 2 current is 3.1415927
n is 3 current is 8.076395046346645
n is 4 current is 13.244108055421808
n is 5 current is 17.3028204216732
n is 6 current is 19.85298464991622
n is 7 current is 21.188247537124454
n is 8 current is 21.78751212841507
n is 9 current is 22.022842786585954
n is 10 current is 22.104988684118826
n is 11 current is 22.130795579321248
n is 12 current is 22.138166011464666
n is 13 current is 22.140095586116132
n is 14 current is 22.14056188901145
n is 15 current is 22.140666527138016
;Value: 22.140666527138016
```

9/28/2005

35

$$\sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

Fixing the increments

```
(define (sine x)
  (define (aux x n current)
    (newline)
    (display "n is ")
    (display n)
    (display " current is ")
    (display current)
    (let ((next (/ (expt x n) (fact n))))
      (if (small-enuf? next)
          current
          (aux x (+ n 2) (+ current next)))))
  (aux x 1 0))
```

9/28/2005

36

Test cases

```
(sine 3.1415927)
n is 1 current is 0
n is 3 current is 3.1415927
n is 5 current is 8.309305709075163
n is 7 current is 10.859469937318183
n is 9 current is 11.4587345286088
n is 11 current is 11.54088042614167
n is 13 current is 11.548250858285089
n is 15 current is 11.548717161180408
;Value: 11.548717161180408
```

$$\sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

9/28/2005

37

We need to alternate terms

```
(define (sine x)
  (define (aux x n current addit)
    (newline)
    (display "n is ") (display n)
    (display " current is ") (display current)
    (let ((next (/ (expt x n) (fact n))))
      (if (small-enuf? next)
          current
          (aux x
                (+ n 2)
                (+ current (* addit next))
                (* addit -1))))))
  (aux x 1 0))
```

9/28/2005

38

Test cases

```
(sine 3.1415927)
; procedure aux: expects 4 arguments; given 3
```

9/28/2005

39

Make sure procedure calls change

```
(define (sine x)
  (define (aux x n current addit)
    (newline)
    (display "n is ") (display n)
    (display " current is ") (display current)
    (let ((next (/ (expt x n) (fact n))))
      (if (small-enuf? next)
          current
          (aux x
                (+ n 2)
                (+ current (* addit next))
                (* addit -1))))))
  (aux x 1 0))
```

9/28/2005

40

```
(sine 3.1415927) ; should be 0
n is 1 current is 0
n is 3 current is -3.1415927
n is 5 current is 2.026120309075164
n is 7 current is -.5240439191678563
n is 9 current is .07522067212275974
n is 11 current is -6.925225410112354e-3
n is 13 current is 4.452067333052508e-4
;Value: 4.452067333052508e-4
```

```
(sine (/ 3.1415927 2.0)) ; should be 1
n is 1 current is 0
n is 3 current is -1.57079635
n is 5 current is -.924832238656045
n is 7 current is -1.004524855998199
n is 9 current is -.999843101378741
;Value: -.999843101378741
```

$$\sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

9/28/2005

41

Make sure to start off right

```
(define (sine x)
  (define (aux x n current addit)
    (newline)
    (display "n is ") (display n)
    (display " current is ") (display current)
    (let ((next (/ (expt x n) (fact n))))
      (if (small-enuf? next)
          current
          (aux x
                (+ n 2)
                (+ current (* addit next))
                (* addit -1))))))
  (aux x 1 0 -1))
```

9/28/2005

42

Test cases

```
(sine (/ 3.1415927 2.0)) ; should be 1
n is 1 current is 0
n is 3 current is 1.57079635
n is 5 current is .9248322238656045
n is 7 current is 1.004524855998199
n is 9 current is .999843101378741
;Value: .999843101378741
(sine 3.1415927) ;; go back and check test cases - should be 0
n is 1 current is 0
n is 3 current is 3.1415927
n is 5 current is -2.026120309075164
n is 7 current is .5240439191678563
n is 9 current is -.07522067212275974
n is 11 current is 6.925225410112354e-3
n is 13 current is -4.452067333052508e-4
;Value: -4.452067333052508e-4
(sine 0) ;; go back and check test cases - should be 0
n is 1 current is 0
;Value: 0
```

9/28/2005

43

Summary

- Display parameters to isolate errors
- Test cases to highlight errors
- Check range of test cases
- Be sure to retry test cases after corrections to ensure still are correct
- **Use these tricks and tools!**

9/28/2005

44

Using types as a reasoning tool

- Types can help:
 - Planning code
 - As entry checks for debugging

9/28/2005

45

smallexpt(n): for n between 1 and 4, return a function that raises its argument to that power

```
(define (smallexpt n)
  (cond ((= n 1) x)
        ((= n 2) (* x x))
        ((= n 3) (* x x x))
        ((= n 4) (* x x x x))
        (else (error "invalid input"))))
```

Intended *type* of smallexpt?

9/28/2005

46

Types as a debugging tool

- Check types of arguments on entry to ensure meet specifications
- Check types of values returned to ensure meet specifications
- (possibly) check constraints on values

9/28/2005

49

An example of type checking

```
(define sqrt-helper
  (lambda (X guess)
    ;; compute approximate square root by
    ;; successive refinement, guess is current
    ;; approximation, X is number whose square
    ;; root we are seeking.
    ;; Type: (number, number) → number
    ;; constraint: guess^2 == X
    (if (or (not (number? X))
            (not (number? guess)))
        (error "report this somehow")
        (if (good-enuf? X guess)
            guess
            (sqrt-helper X (improve X guess))))))
```

9/28/2005

50

An example of type checking

```
(define sqrt-helper
  (lambda (X guess)
    ;; compute approximate square root by
    ;; successive refinement, guess is current
    ;; approximation, X is number whose square
    ;; root we are seeking.
    ;; Type: (number, number) → number
    (if (or (not (number? X))
            (not (number? Guess)))
        (error "report this somehow")
        (if (not (>= x 0))
            (error "Not a positive number")
            (if (good-enuf? X guess)
                guess
                (sqrt-helper X
                             (improve X guess)))))))
```

9/28/2005

51

Good programming practices

- Code design
- Documentation
- Debugging
- Evaluation and verification

9/28/2005

52