Good programming practices

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

Read the entire article:
libraries.mit.edu
VERA
Wall Street Journal
Sept 23 2005
“Code Red: Battling Google…”

Good Design Matters

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

Code Design: Behavior

• Figure out what it’s supposed to do.
• Example: Basic personal calendar manager
  – Capabilities?

Code Design: Data

• What basic objects are in this world?
• What are the relations among them?
Code Design: Data
• Data structure design
  – appointment?

Code Design: Procedures
• Computation to be reused
  – What computations appear to be specific to this problem?
  – What computations are likely to be used elsewhere?

Code Design: Test Cases
• Write the test cases first
  – Helps you anticipate the tricky parts
  – Encourages you to write a general solution

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

Code Design: Test Cases
• Example: set-union
  (define (set-union s1 s2) . . . )

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
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

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) \rightarrow 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 ))))

Debugging errors

• Common sources of errors
• Common tools to debug

Common errors

• Unbound variable
  – Cause: typo
  – Solution: search for 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))
Syntax errors

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

Conceptual errors

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

Evaluation and verification

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

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)

Stepping In Dr. Scheme

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

Tracing in Dr. Scheme

- Change language level to “Essentials of Programming Languages”, press RUN.
- Add to top of definition window:
  \(\text{(require (lib "trace.ss"))}\)
- Indicate which function(s) to trace:
  \(\text{(trace fact)}\)
Debugging

- We want to compute sines, using the mathematical approximation

\[ \sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots \]

Initial code example

```scheme
(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))))
    ))
  (aux x 1 0))
```

Test cases

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

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

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

Chasing down the error

```scheme
(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 ;; just return current guess
        (aux x (+ n 1) (+ current next))))
    ))
  (aux x 1 0))
```

Test cases

```scheme
(sine 3.1415927)
```

<table>
<thead>
<tr>
<th>n is</th>
<th>current</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0.076395046346445</td>
</tr>
<tr>
<td>3</td>
<td>8.06298464991622</td>
</tr>
<tr>
<td>4</td>
<td>13.184247537124454</td>
</tr>
<tr>
<td>5</td>
<td>17.91228294215732</td>
</tr>
<tr>
<td>6</td>
<td>21.78951212841507</td>
</tr>
<tr>
<td>7</td>
<td>25.6826786585954</td>
</tr>
<tr>
<td>8</td>
<td>29.5800611598826</td>
</tr>
<tr>
<td>9</td>
<td>33.4863659321248</td>
</tr>
<tr>
<td>10</td>
<td>37.3986900146666</td>
</tr>
<tr>
<td>11</td>
<td>41.315156527138016</td>
</tr>
</tbody>
</table>

;Value: 22.140666527138016

Fixing the increments

```scheme
(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 ;; just return current guess
        (aux x (+ n 2) (+ current next))))
    ))
  (aux x 1 0))
```
We need to alternate terms

We need to alternate terms

Make sure procedure calls change

Make sure to start off right
Test cases

```lisp
(sine (3.1415927 2.0)) ; should be 1
n is 1 current is 0
n is 3 current is 1.5707963
n is 5 current is .999843101
n is 9 current is .999843101
(Value: .999843101)
(n is 1 current is 0)
(n is 3 current is 3.1415927)
(n is 5 current is -2.0261203)
(n is 7 current is .524043919)
(n is 9 current is -.075220672)

(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 -2.0261203
n is 7 current is .524043919
n is 9 current is -.075220672
n is 11 current is 6.925267333052508e-4
n is 13 current is -4.452067333052508e-4
(Value: -4.452067333052508e-4)
```

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!

Using types as a reasoning tool

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

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

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

An example of type checking

```lisp
(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))))))
```

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
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)))))))