

Reading programs is a form of problem solving

- ▶ Programs are long
- ▶ \therefore read selectively
- ▶ Programs have lots of cross-links
- ▶ \therefore use tools to follow them
- ▶ Programs are complicated
- ▶ \therefore read with a friend

Reading is goal-driven

- ▶ Remember what your goal is!
- ▶ Reading for **debugging** is not like
- ▶ reading for **extending** is not like
- ▶ reading for **quality review**.

What is reading?

- ▶ **Navigation**—finding stuff
- ▶ **Comprehension**—understanding what you find
- ▶ **Integration**—into your model of the program
- ▶ Make your own notes and diagrams as you go.

Road map analogy

- ▶ A single sheet road map of New Zealand would be unusable
- ▶ So we use a hierarchy of maps at different scales
- ▶ In the same way we need a hierarchy of views of a program
 - ▶ Architecture
 - ▶ packages (UML)
 - ▶ modules
 - ▶ methods/procedures
- ▶ road maps need **indices**, so do we.

What about Javadoc I

- ▶ Extracts semi-formal comments and makes HTML
- ▶ Describes constructors, fields, and methods
- ▶ for public & protected & nested classes.
- ▶ Encourages you to write “stubs” and comments *first*
- ▶ Outline should be a useful abstraction
- ▶ Now supports “package comment files” for view of package
- ▶ and “overview comment file” for view of application
- ▶ So *allows* layers-of-maps.

What about Javadoc II

- ▶ Application/package/class summary comment at start good
- ▶ Tends to result in bulky low-value comments
- ▶ Hypertext links to other files very valuable
- ▶ But it **doesn't** link to or from the code!
- ▶ Has links **out** of class, but no links to clients
- ▶ ... contrast with Smalltalk and OO-Browser.
- ▶ <http://www.cs.otago.ac.nz/cosc345/xt/docs/index.html>
- ▶ Does not encourage examples.

Look outside the code

- ▶ Look for examples
- ▶ Look for other documentation
- ▶ Look for change logs (from version control)
- ▶ Look for other code that uses this code
- ▶ Code says “what it *does*” not “what it *means*”

Use traces as a guide

- ▶ Run a test case with profiling or coverage
- ▶ Only code that was executed is relevant!
- ▶ Run two cases, one using X and a similar one not
- ▶ Look at code executed in the first case but not the other.
- ▶ The trivial “start; stop” test case is a good foil.

Top-down vs Bottom-Up

- ▶ Top-down strategy tries to read like a book and understand everything in program/module/...
- ▶ ∴ Works for 10 kSLOC programs, not for 100 kSLOC ones.
- ▶ Bottom-up understands a small piece at a time
- ▶ ∴ always applicable
- ▶ **but** top-down leads to better understanding
- ▶ ∴ read medium size coherent units completely
- ▶ **and** MAKE NOTES as you go!

Reading is expectation driven

- ▶ You cannot understand a statement in isolation
- ▶ You need context to tell you what the words mean
- ▶ Context (especially names) tells you what to expect
- ▶ Form hypotheses and **test** them by searching the code
- ▶ Surprises imply hypotheses wrong/incomplete
- ▶ MAKE NOTES as you go!

Self-Documenting Code

- ▶ There's a lot of stuff on the web about self-documenting code and intention-revealing names.
- ▶ Some code can be very good.
- ▶ Some code depends on conventions you don't know.

```
next: numberOfElements
```

```
  |sequence|
```

```
  sequence := self collectionClass new.
```

```
  numberOfElements timesRepeat: [sequence add
```

```
  ↑sequence
```

Self-Documenting Code II

- ▶ You need to know the language syntax, plus
- ▶ “self next” reads one item from this stream
- ▶ “self collectionClass new” makes a new ArrayList thingy suitable for this stream
- ▶ “*n* timesRepeat: [stmts]” does *stmts* *n* times
- ▶ “sequence addLast: *x*” adds *x* at the end of a stretchy sequence
- ▶ It’s obvious once you know.
- ▶ If you don’t know, you have to *find out*.
- ▶ Projects may have their own conventions!

Test cases (example)

```
1 to: 12 do: [:month |  
  1 to: 7 do: [:dow | |n|  
    n := Date year: 2008 month: month first: dow.  
    [n year = 2008] assert.  
    [n month = month] assert.  
    [n dayOfWeek = dow] assert.  
    [n dayOfMonth < 8] assert.  
    ...
```

Test cases (comment)

- ▶ It's a test case, but you see
- ▶ one way to construct a Date
- ▶ some ways to extract information
- ▶ what those methods return
- ▶ Every method should have at least one test.

Tool support

- ▶ Typographic clues (layout, colouring/typeface) help
- ▶ Colouring (XCode, my m2h, UNIX vgrind/lgrind, Emacs)
 - shows where comments really end.
- ▶ Some languages (Occam, Haskell, Clean, Python)
 - enforce layout, so you can trust it.
- ▶ Mostly, layout shows programmer's idea of structure,
- ▶ **not** the real structure.
- ▶ indent, astyle, *etc* tell you the real structure.
- ▶ These are so-so, but better than most programmers.

Literate Programming

- ▶ Knuth introduced Literate Programming in 1984
- ▶ Explain your program in a documentation tool that can produce beautiful books with tables, graphs, formulas, *etc.*
- ▶ A “tangler” extracts the code.
- ▶ tangle, weave, ctangle, cweave, SpiderWeb, FunnelWeb, noweb, nuweb.
- ▶ Even Word has been used (but never again!).
- ▶ nuweb demonstration if time permits.

CASE tools

- ▶ Often seen as glorified drawing tools for bubble diagrams
- ▶ If repository is kept up to date,
 - ▶ Provides layers of maps
 - ▶ Provides navigation services
 - ▶ Links code with tests (*alias* examples)
- ▶ If not, at least tells you original ideas.
- ▶ **Should** tell you “X **here** means . . .”
- ▶ cscope (now at SourceForge) is for C.

Slicing

- ▶ Choose a variable,
- ▶ throw away everything that doesn't affect it.
- ▶ That's a slice.
- ▶ There are tools; it's also a manual technique.
- ▶ Aim is thorough understanding of one aspect.