Code Complete Ch. 4
Building a routine

1. Begin
2. Design the routine
3. Check the design
4. Repeat if necessary
5. Check the code
6. Code the routine
7. Done
Program Design Language (PDL) (a.k.a. Pseudo-code)

• Use English-like statements to precisely describe specific operations
• Avoid syntactic elements from the target programming language
  – Design at a higher level than the code itself
• Write PDL at the level of intent
  – *What* more than *how*, the code shows how
• Write PDL at a low enough level that generating code from it is straightforward
  – If PDL is too high-level, it will gloss over important details
Example of bad PDL

increment resource number by 1
allocate a dlg struct using malloc
if malloc() returns NULL then return 1
invoke OSrsrc_init to initialize a resource
    for the operating system
*hRsrcPtr = resource number
return 0

• Intent is hard to understand
• Focuses on implementation rather than intent
• Includes too many coding details
• Might as well just write the code
Example of good PDL

Keep track of current number of resources in use
If another resource is available
    Allocate a dialog box structure
    If a dialog box structure could be allocated
        Note that one more resource is in use
        Initialize the resource
        Store the resource number at the location provided by the caller
    EndIf
EndIf
EndIf
Return TRUE if a new resource was created
else return FALSE

• Written entirely in English
• Not programming language specific
• Written at level of intent
• Low-level enough to write code from
Design the routine, I

• Prerequisites
  – Is the purpose of the routine well-defined?
  – Does it fit cleanly into the overall design?

• Define the problem to be solved
  – State the problem to be solved in detail
  – Make sure that all inputs, outputs, and possible error conditions have been defined

• Name the routine
  – Clear, unambiguous names
  – If you can’t come up with a good name, then the routine is not well-defined
Design the routine, II

• Decide how to test the routine
  – Write down test ideas as you design

• Think about efficiency
  – Design to performance and behavioral specifications
  – Routine must be organized and readable
    • Allows tweaking for efficiency later

• Research algorithms and data structures
  – Reuse good code where possible (e.g., libraries)
  – Study and use other algorithms and data structures

• Design the routine's data structures
Design the routine, III

• Write a header comment for the routine
  – Purpose, parameters, return value, error conditions
• Write PDL for the routine
• Check the PDL
  – At the very least, review your PDL yourself
  – Formal inspection in certain situations
• Iterate
  – Cycle through all of these as needed
Code the routine, I

• Write the declaration
  – Includes routine name, inputs, outputs

• Turn the PDL into high-level comments
  – Create a code framework for the routine

• Fill in code below each comment
  – Code should be bite-sized and obvious
Code the routine, II

• Check the code informally
  – Do small spontaneous code review
• Clean up the leftovers
  – Are all inputs and outputs accounted for?
  – Sanity check the design again
  – Check data, control structures, layout, documentation
• Repeat steps as needed
Check the code, I

• Mentally check the routine for errors
  – Visually inspect and sanity check the routine

• Compile the routine
  – Set warnings to the pickiest level possible
  – Eliminate all compiler errors and warnings
    • Warnings often suggest low-quality code!
Check the code, II

- Use tools for debugging and checking
  - Watch the code execute in a debugger
  - Write and run automated unit tests for the new routine
- Remove errors from routine
  - If it’s full of bugs, you probably have a lousy design
- You should now have a crystal clear understanding that it works
  - You cannot operate in the realm of superstition at this point
  - If you don’t know why it works, you’re not done!
    - Even if it appears to work