How to code

- Coding/Implementation goals:
  - Make it run
  - Make it right
  - Make it fast
  - Make it small/general

- Spiral design (or RAD or !waterfall or ...) see *DOOM*
Make it run

● **What most of you can do at this point**
  ➤ find a way to model the problem
  ➤ after many hours, turn in something that works reasonably often

● **Good start, but leaves unanswered questions**
  ➤ how can you reduce time spent programming?
  ➤ how can you trust your code is correct?
  ➤ how can you reduce time spent debugging?
  ➤ how to guarantee a good start for future changes?
Make it right

- Once it runs, focus on getting it right
  - testing code actually works (on all cases)
  - add more complex code to handle general cases
  - handle erroneous conditions
  - factor code into small bundles that can be (re)used separately

- Leads to more questions
  - is code efficient enough?
  - is code factored enough to permit variety of implementations?
Make it fast

- Only after it works correctly should you optimize it
  - determine what needs to be optimized
    - profiling
    - cache and heap utilization
  - determine how best to optimize it
    - redesign
    - rewrite or re-factor

- Make sure it still works when you are done!
Make it small/general/simple

- Finally, make it “beautiful”
  - so far, every programmer has common goals
  - now, “beauty is in the eye of the beholder”

- Design with goals:
  - ease of use
  - portability
  - ease of re-use
  - efficiency
  - first to market
  - ??????
How to determine what is a good program

- **Language independent principles of design and programming**
  - design heuristics
    - coupling, cohesion, small functions, small interfaces ...
  - design patterns
    - factories, MVC a.k.a. observer/observable, ...

- **Language specific:**
  - idioms
    - smart pointers, vectors/arrays, overloaded operators ...
  - idiosyncrasies, idiocies
    - must define virtual constructor, comma operator ...
Design Heuristics: class and program rules-of-thumb

(see text by Arthur Riel)

● **Coupling**
  ➤ classes/modules are independent of each other
  ➤ goal: minimal, loose coupling
  ➤ do classes collaborate and/or communicate?

● **Cohesion**
  ➤ classes/modules capture one abstraction/model
  ➤ keep things as simple as possible, but no simpler
  ➤ goal: strong cohesion (avoid kitchen sink)

● **The open/closed principle**
  ➤ classes/programs: open to extensibility, closed to modification
Design patterns

Good design comes from experience, experience comes from bad design
Fred Brooks (or Henry Petroski)

- **Example: iterator**
  - sequentially access elements of aggregate without exposing representation/implementation
  - see DirEntry, List class, Map class
  - heavily used in STL, with pointer like syntax

- **Factory**
  - virtual constructor, have system that doesn’t depend on how items are created
Pattern Essentials

- **Name**
  - good name provides a handle for the pattern, makes it easy to remember and use: vocabulary

- **Problem**
  - when the pattern is applicable, context, criteria to be met, design goals

- **Solution**
  - design, collaborations, responsibilities, and relationships of the classes/design elements

- **Consequences**
  - trade-offs, problems, results from applying pattern: help in evaluating applicability
Goals for the course

- **Learn how to break problems down**
  - learn how to make a model of the problem

- **Learn how to debug**
  - must be able to find and fix semantic errors
  - must be able to understand compiler errors

- **Learn how to care about program**
  - code is a “work of art”
  - all pieces work together to make it good