The UNIVERSITY of VERMONT

Benjamin Fenster
CS-205 (Damon)
20 December 2004

Coding

I. Introduction
   a. Will "finish" code design at some point and be ready to start coding
   b. Will find missing classes / missing methods
   c. May have less clear UI in code than other elements
   d. May want to restructure code – that's okay
   e. Foundation First
      i. Build a kernel of functionality that works first.
      ii. Maybe display a Unit.
      iii. Add new features progressively
   f. Representation
      i. Distinct from abstract value
      ii. Something you've never considered in the design
      iii. think about your own code in terms of representation; everything else in terms of its abstract value.
   g. Goals
      i. Correct, maintainable, portable
      ii. Ease of debugging, testing
      iii. Clarity, consistency
      iv. Performance (at the bottom of the list!)
      v. Correctness
         1. "Test Before Coding" – Write the test before even writing any code.
         2. Makes it easier to get the cases right.
      vi. Maintainable
         1. Isolation of Change!
         2. Accessor (get / set) Functions
         3. Have a place to do other actions on set!
         4. Define Iterators for all aggregates! One place encapsulation is traditionally broken.
      vii. Portable
         1. Number of viable platforms is smaller than in the past
         2. Isolate system dependencies
         3. Define wrappers for system calls
         4. Watch out for system specific values (like end of line, path names)
         5. GUI Issues: Much of work handled by swing in Java, but where menus are, mouse buttons, et cetera still need to be handled.
         6. Internationalization
            a. Keep all messages in a separate file from the code
            b. Be careful with formats (money, dates, …)
            c. Be aware that messages may be longer in other languages, so make sure the box will still correctly display a longer message.
         7. For C / C++
            a. Use preprocessor!
            b. #ifdef SOLARIS #define BIG_ENDIAN
         8. For Java, it's mostly done for you.
         9. Use property lists ("Quit" vs. "Exit", hotkeys, …)
      viii. Debugging
         1. Put in stuff to support debugging.
         2. With trace: Have program say what it's doing over time. Be able to turn that on and off dynamically
         3. Be able to dump data structures so that they're easy to read (nice format for a Tree, for example)
      ix. Testing
         1. Have a standard test routine in each class

2. .repOk()

x. Clarity
   1. Standard abbreviations
   2. Positive names only! if (!(noNegatives == false))
   3. Comments: On class (first thing to write), on each method, on tricky code
   4. Maintain the comments!

xi. Performance
   1. Code for speed last
   2. Fix only what's noticeably slow

xii. Pair Programming: Second pair of eyes; quick load balancing

xiii. Invariants
   1. Pre/Post Conditions
   2. Loop invariants: Getting this right massively increases your chances of getting the loop itself right.

II. Code Reviews

a. Need to know if the code works or not

b. The Idea
   i. Read and discuss the code
   ii. Point out what's wrong
   iii. Seems too simple, but it can be very helpful.

c. The Goal: Identify errors (*not* evaluate alternatives)

d. Walkthroughs
   i. Groups of 3 to 5
   ii. Developer plus others
   iii. Ask questions at detailed level ("shouldn't you increment once more?")
   iv. Play Devil's advocate
   v. Developer leads discussion (describes flow function by function)
   vi. Can get uncomfortable. Ask questions politely.

e. Fagan Inspections
   i. Four Mandatory Roles
      1. Moderator
      2. Designer
      3. Programmer
      4. Tester
   ii. First Meeting
      1. Designer describes the overall design
      2. Everybody gets a copy of the code
   iii. Second Meeting
      1. Programmer reads code aloud
      2. Moderator records bugs
   iv. Moderator files report
   v. Programmer fixes the bugs; moderator confirms they're fixed
   vi. Can iterate (suggest another iteration if more than 5% of lines reviewed changed)
   vii. Can be painful! Do this only with code that's particularly complicated, bug-prone, by a problematic programmer, sometimes just at random

f. Variations
   i. Usually range from full Fagan to unstructured walkthrough
   ii. Example: Before submitting code to build, get the senior person to do a review

g. Time
   i. Varies on amount of code, approach
   ii. Two hours absolute upper bound
   iii. Probably no more than one hour
   iv. Caspers Jones experience:
      1. 150 lines per hour (non-comment) for preparation
      2. 75 lines per hour in meeting / inspection

h. Effectiveness

i. Fagan inspections are pretty consistent (walkthroughs less so)
ii. Finds 67 – 90% of bugs (many of which would never have been found in testing)
iii. Size of the group seems irrelevant (3 is as good as 5)
iv. More time spent preparing = More productivity (diminishing returns though, so can't spend infinite time preparing)
v. Just like any proofreading
vi. Potentially extremely helpful, even if you're just having another developer review your code and mark it up with comments

i. Results
i. May find that the structure of the code isn't quite right
ii. Could live with it. No immediate cost; huge long term cost
iii. Tempting to start from scratch. Expensive, and each time you start over you introduce brand new bugs anyway.
iv. Could do Refactoring

III. Refactoring
a. Restructure the code without changing behavior
b. Want to change how it's put together without changing the result
c. Examples
i. Wrap get / set methods around fields
ii. Make value in a method a parameter
iii. Move a method to sub- or superclass
iv. Take a common chunk of code and make it a method
d. All small steps that leave the same behavior
e. eclipse does many of these things automatically
f. Example
i. Want to introduce the State pattern
ii. Initially have ShapeSet, with data[0..n] member
iii. Want to have ShapeSet → ShapeRep, with SmallRep and BigRep subclasses
iv. First move rep to ShapeRep. Small change.
v. No make ShapeRep abstract; move rep down to a new subclass.
vi. Then add a new subclass for the *other* representation – *not* refactoring now. Did refactoring to prepare code for making this change.

IV. Extreme Programming
a. Cost
i. Old Assumption: Changes are exponentially more expensive when made later
ii. New Thought: Changes in cost are moderate
iii. Comes from options pricing. Building a product gives an option to sell later.
b. Core Values
i. Communication. Everyone is *actively* aware of what everyone else is doing.
ii. Simplicity. Simplest solution that satisfies needs.
iii. Feedback. Want feedback "early and often"
iv. Courage. Not just trying to avoid failure. Really striving for success.
c. Think Small. Do the easy stuff first. May turn out you don't need to do the hard stuff
d. Pair Programming
i. Share a computer, or sit close (on nearby machines, as in Votey 369)
ii. Switch partners frequently (daily, hourly)
e. Release Often
i. Build constantly. Leave no change in isolation for more than few hours.
ii. Faster hardware allows this
iii. Requires small groups again
iv. Disadvantage: Some tests cannot be run
f. Collective Ownership
i. Everyone owns all code. Can make any change you think is appropriate
ii. Requires trust and faith in the team
iii. Coding standards are more important
g. Refactoring

    i. No non-trivial code should be repeated.
    ii. Requires collective ownership.
h. Test Before Coding
    i. Enables continuous integration
    ii. Can't check in code without a test suite
i. Environment
    i. Developers choose their own environment
    ii. Cubicle size, bullpen
    iii. Furniture (hard floor, wheeley chairs in the bullpen)
    iv. Food!
j. On-Site Customer
    i. Through the whole development cycle
    ii. At *least* meet every two weeks
    iii. Not always feasible
k. Metaphor for System. Everybody understands the same goal.
l. Coding Standards
m. Planning Game
    i. One to three weeks *only*
    ii. You're *never* "90% done." You're either 0% or 100%
    iii. Customer sees progress; sets direction
n. 40 Hour Workweeks
    i. Never work back-to-back overtime weeks
    ii. Overtime is a red flag that something's wrong
o. Morning Meetings
    i. Short. Just make sure everyone's connected
    ii. No chairs! Standing people don't meet for long
p. Rearrange People
    i. Change pairs. Move to another part of the project
    ii. Fresh perspectives, shared vision
q. These are just a set of rules. Use the ones that work with your culture and project. Nothing says you have to use every single rule!