Collaborative Programming:
Pair Programming and Reviews
and
Maintenance and Refactoring

Schedule
• Friday final lecture
• Monday holiday
• Wednesday review with sample questions
• Friday final (second midterm)
• Monday (8:30-10:30) project presentations — in this order
  1. c 0.064272
  2. a 0.994975
  3. h 0.366314
  4. d 0.723629
  5. e 0.055863
  6. f 0.749214
  7. b 0.827284
  8. g 0.604225

Pair programming
• pair programming: 2 people, 1 computer
  – take turns “driving”
  – rotate pairs often
  – pair people of different experience levels
• pros:
  – Can produce better code
  – An inexperienced coder can learn from an experienced one
• cons:
  – Some people don’t like it

Reviews
• Review: Other team member(s) read an artifact (design, specification, code) and suggest improvements
  – documentation
  – defects in program logic
  – program structure
  – coding standards & uniformity with codebase
  – enforce subjective rules
  – ... everything might be fair game
• Feedback leads to changes, followed by additional reviews and eventually approval

Motivation for reviews
• Can catch many bugs, design flaws early.
• > 1 person has seen every piece of code.
  – Prospect of someone reviewing your code raises quality threshold.
• Forces code authors to articulate their decisions and to participate in the discovery of flaws
• Allows junior personnel to get early hands-on experience without hurting code quality
  – Pairing them up with experienced developers
  – Can learn by being a reviewer as well
• Accountability. Both author and reviewers are accountable for the code
• Explicit non-purpose:  
  – Assessment of individuals for promotion, pay, ranking, etc.
  – Management is usually not permitted at reviews
Types of code review

• What is reviewed:
  – A specification
  – A coherent module (sometimes called an “inspection”)
  – A single checkin or code commit (incremental review)

• Who participates:
  – One other developer
  – A group of developers

• Where:
  – In-person meeting
    • Best to prepare beforehand: artifact is distributed in advance
    • Preparation usually identifies more defects than the meeting
  – Email/electronic

Review technique and goals

• Specific focus?
  – Sometimes, a specific list of defects or code characteristics
    • Error-prone code
    • Previously-discovered problem types
    • Security
      • Checklist (coding standards)
        – Automated tools (type checkers, lint) can be better
  – Technique
    – Does developer present the artifact to a group?
    – Only identify defects, or also brainstorm fixes?
    – Sometimes, a specific methodology
      • “Walkthrough” = playing computer, trace values of sample data

Code reviews in industry

• Code reviews are a **very** common industry practice.
• Made easier by advanced tools that:
  – Integrate with configuration management systems
  – Highlight changes (i.e., diff function)
  – Allow traversing back into history
  – E.g.: Eclipse, SVN tools

Ernst’s approach

• Distribute code (or other artifacts) ahead of time
  – Common pagination
  – Documentation is required (as is good style)
  – No extra overview from developer
• Each reviewer focuses where he/she sees fit
• Mark up with lots of comments
• Identify 5 most important issues
• At meeting, go around the table raising one issue
  – Discuss the reasons for the current design, and possible improvements
• Author takes all printouts and addresses all issues
  – Not just those raised in the meeting
Refactoring: due to “bit rot”

- After several months and new versions, many codebases reach one of the following states:
  - rewritten: Nothing remains from the original code.
  - abandoned: The original code is thrown out and rewritten from scratch.
- Why is this?
  - Systems evolve to meet new needs and add new features.
  - Market pressures are real.
  - If the code's structure does not also evolve, it will "rot".
  - This does happen even if the code was initially reviewed and well-designed at the time of checkin, and even when checkins are reviewed.

Code maintenance

- maintenance: modification of a software product after it has been delivered.
  - Purposes:
    - fix bugs
    - improve performance
    - improve design
    - add features
  - ~80% of maintenance is for non-bug-fix-related activities such as adding functionality (Pigosky 1997).
    - There is lots of supporting data for this.
    - [I will say that I don't really know what this means].

Maintenance is hard

- It’s harder to maintain (someone else’s?) code than write your own new code.
  - “House of cards” phenomenon (don’t touch it!).
  - Must understand code written by another developer, or code you wrote at a different time with a different mindset.
  - Most developers hate code maintenance.
  - Why?
- Maintenance is how devs spend most of their time.
  - See the statistics on the previous slide.
- It pays to design software well and plan ahead so that later maintenance will be less painful.
  - Capacity for future change must be anticipated.

Refactoring

- refactoring: improving a piece of software’s internal structure without altering its external behavior.
  - Not the same as code rewriting.
  - Incurs a short-term time/work cost to reap long-term benefits.
  - A long-term investment in the overall quality of your system.
Why refactor?

• Why fix a part of your system that isn’t broken?
  – Each part of your system’s code has three purposes:
    • to execute its functionality,
    • to allow change,
    • to communicate well to developers who read it.
• If the code does not do one or more of these, it is broken.

Low-level refactoring

Names:
• Renaming (methods, variables)
• Naming (extracting) “magic” constants

Procedures:
• Extracting code into a method
• Extracting common functionality (including duplicate code) into a module/method/etc.
• Inlining a method/procedure
• Changing method signatures

Reordering:
• Splitting one method into several to improve cohesion and readability (by reducing its size)
• Putting statements that semantically belong together near each other

IDE support for refactoring

• Eclipse / Visual Studio support:
  – variable / method / class renaming
  – method or constant extraction
  – extraction of redundant code snippets
  – method signature change
  – extraction of an interface from a type
  – method inlining
  – providing warnings about method invocations with inconsistent parameters
  – help with self-documenting code through auto-completion

Higher-level refactoring

• Refactoring to design patterns
• Exchanging risky language idioms with safer alternatives
• Performance optimization
• Clarifying a statement that has evolved over time or is unclear

• Compared to low-level refactoring, high-level is:
  – Not as well-supported by tools
  – Much more important!
Refactoring plan?

- When you identify an area of your system that:
  - isn't especially well designed
  - isn't especially thoroughly tested, but seems to work so far
  - now needs new features to be added

- What should you do?
  - Assume that you have adequate time to "do things right." (Not always a valid assumption in software...)

Recommended refactoring plan

- When you identify an area of your system that:
  - isn't especially well designed
  - isn't especially thoroughly tested, but seems to work so far
  - now needs new features to be added

- What should you do?
  - Write unit tests that verify the code's external correctness.
    - (They should pass on the current, badly designed code.)
  - Refactor the code.
    - (Some unit tests may break. Fix the bugs.)
  - Add the new features.

"I don't have time to refactor!"

- Refactoring incurs an up-front cost.
  - many developers don't want to do it
  - most management don't like it, because they lose time and gain "nothing" (no new features)

- However...
  - well-written code is much more conducive to rapid development (some estimates put ROI at 500% or more for well-done code)
  - finishing refactoring increases programmer morale – developers prefer working in a "clean house"

- When to refactor?
  - best done continuously (like testing) as part of the SWE process
  - hard to do well late in a project (like testing) ... Why?

Should startups refactor?

- Many small companies and startups skip refactoring.
  - "We're too small to need it!"
  - "We can't afford it!"

- Reality:
  - Refactoring is an investment in quality of the company's product and code base, often their prime assets
  - Many web startups are using the most cutting-edge technologies, which evolve rapidly. So should the code

- If a key team member leaves (common in startups), ...
- If a new team member joins (also common), ...
Questions?