Updated Schedule of Remaining Class-Related Deliverables

- **Fri, Aug 11 @ 10pm:** indiv assignment #2 due
- **Sun, Aug 13 @ 10pm:** final project release due
- **Mon, Aug 14, in class:** final project demos / presentations
- **Mon, Aug 14 @ 10pm:** peer review #2 due
- **Wed, Aug 16 @ 10pm:**
  indiv assignment #2 responses due
  peer review #2 viewing and usefulness feedback due
- **Thu, Aug 17 @ 10pm:** final take-home exam due (online submission)
- **Fri, Aug 18, before class:** final take-home exam due (on paper)
- **Fri, Aug 18 @ 10pm:** final questionnaire due

Lecture 20: Refactoring (Part II)

"If bug rates are to be reduced, each function needs to have one well-defined purpose, to have explicit single-purpose inputs and outputs, to be readable at the point where it is called, and ideally never return an error condition."

-- Steve Maguire, from "Writing Solid Code"

Outline

- Motivation and definition of refactoring
- Playing with real code examples
- Main refactoring strategies
- Practical suggestions
- When refactoring works and when it does not

References

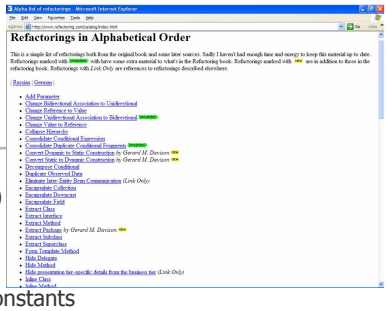
Recommended:

- Refactoring resources online, by Martin Fowler, http://www.refactoring.com/catalog/

Other relevant resources:

- *Applied Software Project Management*, by Andrew Stellman and Jennifer Greene, 2006.
- *Writing Solid Code*, by Steve Maguire, 1994.
- *Agile Software Development: Principles, Patterns, and Practices*, by Robert Martin, 2003.
- *Professional Software Development*, by Steve McConnell, 2004.
- *Sustainable Software Development – An Agile Perspective*, by Kevin Tate, 2006.
- *Freakonomics: A Rogue Economist Explores the Hidden Side of Everything*, by Steven Levitt and Stephen Dubner, 2005.
- *Design Patterns Explained*, by Alan Shalloway and James Trott, 2002.

Types of Refactoring



- Refactoring to patterns
- Renaming (methods, variables)
- Extracting code into a method
- Changing method signatures
- Performance optimization
- Naming (extracting) "magic" constants
- Extracting common functionality (including duplicate code) into a service / module / class / method
- Splitting one method into several to improve cohesion and readability (by reducing its size)
- Putting statements that semantically belong together near each other
- Exchanging risky language idioms with safer alternatives
- Clarifying a statement (that has evolved over time and/or that is hard to "decipher")

Language and Tool Support for Refactoring

- **Modern IDEs (e.g., 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
- **Older development environments (e.g., vi, Emacs, etc.) have little or no support for these.**
  - Discourages programmers from refactoring their code

Student Submission

When Making Code Changes…

- **In what order would you do the following?** (Please, number them 1-3.)

Make the planned code changes

Refactor the code

Write unit tests to ensure that any conditions that need to be met are indeed met

Recommended Actions When Making Code Changes

1. Write unit tests to ensure that any conditions that need to be met are indeed met
   - Both before and after any refactoring or other changes you do
2. Refactor the existing code
   - To accommodate any necessary code changes and to make sure that the tests still pass
3. Make the planned code changes

A Bit of Practical Advice

- Prioritize what needs to be refactored
  - Not all parts of your code are equally important at all times.
  - This way it won't feel like a useless, time-consuming exercise – but like something that helps you to more effectively do your job.

Student Submission

Refactoring in Context: Small Startup Companies

- List one reason why refactoring should (or should not) be done in *small startups*.

Should:

Should not:

Refactoring in Context: Small Startups, Pros

- How refactoring may help in small startups:
  - It's an investment in quality, regardless of the size of the company
  - Ideas and technologies are typically cutting edge and evolving quickly over time, so the code needs to also evolve at the same pace, to make it easier (not harder) to do the next change when it becomes necessary.
  - Even with a small team, if a team member suddenly quits, it will be easier to take over his/her code and be able to maintain and extend it.

Refactoring in Context: Small Startups, Cons

- How refactoring may not help in small startups:
  - The company may never need to do another version (if the product is unsuccessful).
  - The company wants to get to market as fast as possible, even at the expense of quality.
    - The typical customers of version 1.0 products want *something* working, not solid products.
    - "[They're] so busy sawing, there's simply no time to sharpen the saw." J

Refactoring in Context: Larger Companies

Student Submission

- List one reason why refactoring should (or should not) be done in *larger companies*.

Should:

Should not:

Refactoring in Context: Larger Companies, Pros

- How refactoring may help in larger companies:
  - The users demand quality or else will turn to the competition.
  - The company aims the product for the long haul, so long-term investments are justified.
  - More people work on the development of the product over larger periods of time
    - The original code writer(s) may not be around to explain what they intended with a piece of code.
      - They'll have saved themselves 5 minutes (by not clarifying) at the expense of 5 days for those who follow.

Refactoring in Context: Larger Companies, Cons

- How refactoring may not help in larger companies:
  - There's typically less sense of ownership of the code or the product than in smaller companies
    - You don't know the "poor" people who will have to maintain your code, so you care less about them.
    - … in contrast to a startup where the maintainer will be either you, or the person sitting next to you.
  - Large companies are sometimes just former small companies that never realized they had grown
  - Company culture may not reward programmers for doing it
    - E.g.: if performance evaluations are mostly based upon delivering immediate results on competing projects…

Refactoring – When to Do It?

- **Refactoring is necessary from a business standpoint too**
  - Helps to increase schedule predictability and achieve higher outputs at lower costs
  - In general, ROI for improved software practices is 500% (!) or better
  - By doing refactoring a team saves on unplanned defect-correction work
- **When is refactoring necessary?**
  - Best done continuously, along with coding and testing
  - Very hard to do late, much like testing
    - Often forced before plunging into version 2

Food for Thought: Analyzing the Incentives

Student Submission

Who is supposed to do the refactoring?

(A) programmer
(B) management
(C) maintainer
(D) user

Food for Thought: Analyzing the Incentives (cont.)

Student Submission

Who is supposed to do the refactoring?

(A) programmer
(B) management
(C) maintainer
(D) user

Who benefits from the refactoring?

(A) programmer
(B) management
(C) maintainer
(D) user

Analysis of the Incentives Shows…

- Those who can do the job often do not have the incentive to do so.
- Those who need the job done can not do it themselves.

Result:

Classic case of misalignment of incentives that often leads to situations where great ideas get stalled indefinitely.

Conclusion: Top Reasons for Refactoring

- Improving maintainability
  - … and hence productivity!
- Responding to changes in the spec / design by improving the code structure
  - Or proactively preparing for such changes

Lecture 21: Software Maintenance and Code Reviews

Outline

- Software maintenance
- Code reviews

Resources

- *Rapid Development*, by Steve McConnell
  - Ch. 4.3, 23
- *Test Driven Development: By Example*, by Kent Beck
- *Code Complete*, by Steve McConnell

What is Software Maintenance?

- Producing new (versions of) software under the constraints of existing software
  - Backwards compatibility is often assumed / required
- Comprises all phases of the lifecycle, starting with requirements gathering
  - Another turn of the spiral

Conflicting Developer Hopes about Software Maintenance

- The hope is that you (and your company) will get to the maintenance stage.
  - Condition: the company has been selling the products successfully and there is demand for future versions
- Most developers hope that they won't have to deal with maintenance.
  - It's harder to maintain (someone else's) code than it is to write a new one.

In Reality, Maintenance...

- **... is what you do most of the time.**
- **... is often done as an afterthought, turning it into a nightmare.**
  - You have to think ahead of time how you (or someone else) are going to maintain the code later.
  - Refactoring is crucial
    - ... but it should be done regularly and must start early
- **... is often given to junior developers.**
  - "This way they'll learn the guts of the system better."
  - Shhht!!! – senior developers don't want to work on maintenance themselves.
  - Result: brittle code with little conceptual or design integrity; even more maintenance headaches to come.

The Maintenance Spiral: Viewed through an Influence Diagram

- Junior devs in maintenance => lower code quality
- Lower code quality => code is harder to maintain
- Hard to maintain code => those with a choice (senior devs) prefer to avoid doing maintenance
- Fewer senior devs in maintenance => more junior devs in maintenance

- How do we break out of the positive feedback loop?

Code Reviews – What and Why?

- **What:** A practice whereby one needs to get a sign-off before committing changes / new code
- **Why:** ...

Code Reviews – Motivations Behind the Practice

- **Ensures that more than one person has seen every piece of code**
  - The prospect of someone else reviewing your code raises the quality threshold.
  - Even if someone is absent, another person has a clue.
- **Forces code writers to articulate 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 who can answer subtle questions and provide valuable feedback

Mechanics of Code Reviews

- Done before committing code to a repository and incorporating it into a new build
- Review includes suggestions for improvement on a logical and/or structural level, to conform to a previously agreed upon set of quality standards.
  - Code review feedback often leads to a refactoring step, followed by a second code review.
- Reviewer attests that code is maintainable and meets the established quality standards.
- Both code writer and reviewer are accountable for allowing the code to be committed.

Code Review Tool Support

- Made easy by advanced tools that:
  - integrate with configuration management systems
  - highlight changes (i.e., *diff* function)
  - allow traversing back into history
- E.g.: Eclipse offers such support

In Reality...

- Code reviews are a common practice.
- Code reviews can become perfunctory if:
  - ... not part of the (organization / company) culture
    - "Let's just quickly get it over with, because we've got important work to do."
  - ... done without proper tool support
    - "This is going to be a pain; let's pray it'll soon be over."

Parting Thought...

"Don't do anything that you don't want to appear on the front page of the newspaper."

- Sound advice beyond the domain of writing software!