

Refactoring

CSE 403

Problem: "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.

...even if the code was initially reviewed and well-designed at the time of checkin, and even if checkins are reviewed
- Why is this?
 - Systems evolve to meet new needs and add new features
 - If the code's structure does not also evolve, it will "rot"

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)

Maintenance is hard

- It's harder to maintain code than write new code.
 - must understand code written by another developer, or code you wrote at a different time with a different mindset
 - danger of errors in fragile, poorly-understood code (don't touch it!)
- Maintenance is how devs spend **most of their time**
 - Many developers *hate* code maintenance. Why?
- With good design and advance planning, maintenance is less painful
 - Capacity for future change must be anticipated

Refactoring

- **refactoring:** Improving a piece of software's internal structure without altering its external behavior.
 - Incurs a short-term time/work cost to reap long-term benefits
 - A long-term investment in the overall quality of your system.
- refactoring is not the same thing as:
 - rewriting code
 - adding features
 - debugging code

Refactoring examples

Why refactor?

Why fix a part of your system that isn't broken?

- Each part of your system's code has 3 purposes:
 1. to execute its functionality,
 2. to allow change,
 3. to communicate well to developers who read it.If the code does not do one or more of these, it *is* broken.
- Refactoring improves software's design
 - more extensible, flexible, understandable, performant, ...
 - Every design improvement has costs (and risks)

Code “smells”: Signs you should refactor

- Duplicated code
- Poor abstraction (change one place → must change others)
- Large loop, method, class, parameter list; deeply nested loop
- Module has too little cohesion
- Modules have too much coupling
- Module has poor encapsulation
- A "middle man" object doesn't do much
- a “weak subclass” doesn’t use inherited functionality
- a “data class” has little functionality
- Dead code
- Design is unnecessarily general
- Design is too specific

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

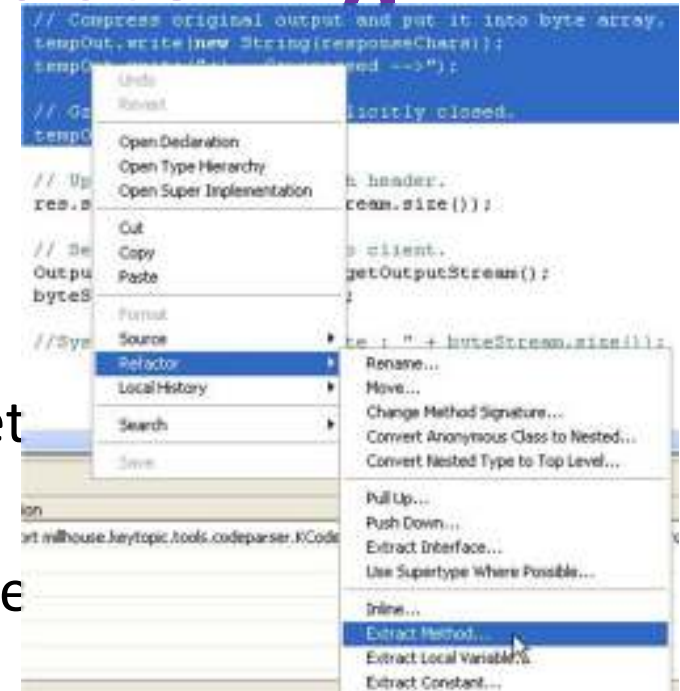
Reordering:

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

– See also <http://www.refactoring.org/catalog/>

IDE support for refactoring

- Eclipse / Visual Studio support:
 - variable / method / class renaming
 - method or constant extraction
 - extraction of redundant code snippet
 - 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:
 - is poorly designed
 - is poorly tested, but seems to work so far
 - now needs new features
- What should you do?
 - Let's assume that you have adequate time to "do things right."
(Not always a valid assumption in software...)

Recommended refactor plan

- When you identify an area of your system that:
 - is poorly designed
 - is poorly tested, but seems to work so far
 - now needs new features
- 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.
 - As always, keep changes small, do code reviews, etc.

"I don't have time to refactor!"

- Refactoring incurs an up-front cost.
 - some 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 SE 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), ...