Refactoring

Emina Torlak
dima@cs.washington.edu
Outline

• Problem: code maintenance
• Refactoring: when, why, and how
• Refactoring in the real world
intro

code maintenance is hard
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, and even if later 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 ...

- **Code maintenance**: modification of a software product after it has been delivered.

- Purposes:
  - fixing bugs
  - improving performance
  - improving design
  - adding features

- ~80% of maintenance is for non-bug-fix-related activities such as adding functionality (Pigosky 1997)
Code maintenance is hard

- It's harder to maintain code than write new code.
  - You must understand code written by another developer, or code you wrote at a different time with a different mindset
  - Danger of errors in fragile, hard-to-understand code
- Maintenance is how developers spend most of their time
  - Many developers hate code maintenance. Why?
- 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: what, when, why, and how
What is refactoring?

• **Refactoring**: improving a piece of software's internal structure without altering its external behavior.
  • Incurs a short-term overhead to reap long-term benefits
  • A long-term investment in overall system quality.

• Refactoring is not the same thing as:
  • rewriting code
  • adding features
  • debugging code
Why refactor?

• Why fix a part of your system that isn't broken?
• Each part of your system's code has 3 purposes:
  • to execute its functionality,
  • to allow change,
  • to communicate well to developers who read it.
• If the code does not do these, it is broken.
• Refactoring improves software's design
  • to make it more extensible, flexible, understandable, performant, …
  • but every improvement has costs (and risks)
When to refactor?

- When is it best for a team to refactor their code?
  - Best done **continuously** (like testing) as part of the process
  - Hard to do well late in a project (like testing)

- Refactor when you identify an area of your system that:
  - isn't well designed
  - isn't thoroughly tested, but seems to work so far
  - now needs new features to be added
Code “smells”: signs you should refactor

• Duplicated code; dead code
• Poor abstraction
• Large loop, method, class, parameter list
• 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
• Design is unnecessarily general or 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 refactoring.com/catalog/
IDE support for low-level 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
High-level refactoring

• Deep implementation and design changes
  • 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!
How to refactor?

• 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?
How to refactor? Have a plan!

I refactored once

It was horrible
Refactoring plan (1/2)

• Write **unit tests** that verify the code's external correctness.
  • They should pass on the current poorly designed code.
  • Having unit tests helps make sure any refactor doesn't break existing behavior (regressions).

• Analyze the code to decide the **risk** and benefit of refactoring.
  • If it is too risky, not enough time remains, or the refactor will not produce enough benefit to the project, don't do it.

---

Code refactoring — it won't be long.
Refactoring plan (2/2)

- **Refactor** the code.
  - Some tests may break. Fix the bugs.
- **Code review** the changes.
- **Check in** your refactored code.
  - Keep each refactoring **small**; refactor one unit at a time.
    - Helps isolate new bugs and regressions.
  - Your checkin should contain only your refactor.
  - Your checkin should **not** contain other changes such as new features, fixes to unrelated bugs, and other tweaks.
refactoring in the real world
Barriers to refactoring: “I don’t have time!”

• Refactoring incurs an up-front cost.
  • Some developers don't want to do it
  • Most managers don't like it, because they lose time and gain “nothing” (no new features).

• However …
  • Clean code is more conducive to rapid development
    • Estimates put ROI at >500% for well-done code
  • Finishing refactoring increases programmer morale
    • Developers prefer working in a “clean house”
Barriers to refactoring: company/team culture

• 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) …
Refactoring and teamwork: communicate!

• Amount of overhead/communication needed depends on size of refactoring.
  • Small: just do it, check it in, get it code reviewed.
  • Medium: possibly loop in tech lead or another dev.
  • Large: meet with team, flush out ideas, do a design doc or design review, get approval before beginning, and do a phased refactoring.

• Avoids possible bad scenarios:
  • Two devs refactor same code simultaneously.
  • Refactor breaks another dev's new feature they are adding.
  • Refactor actually is not a very good design; doesn't help.
  • Refactor ignores future use cases, needs of code/app.
  • Tons of merge conflicts and pain for other devs.
Summary

• Refactoring improves internal software structure without altering its external behavior.
  • Short-term overhead …
  • But many long-term benefits
• Have a refactoring plan.
• Communicate the plan to your team.