Legacy Code Matters

- Since maintenance consumes ~60% of software costs, *it is probably the most important life cycle phase of software…*

  "Old hardware becomes obsolete; old software goes into production every night."

Robert Glass, *Facts & Fallacies of Software Engineering (fact #41)*

*How do we understand and **safely** modify legacy code?*

Maintenance ≠ Bug Fixes

- Enhancements: 60% of maintenance costs
- Bug fixes: 17% of maintenance costs

Hence the "60/60 rule":

- 60% of software cost is maintenance
- 60% of maintenance cost is enhancements

Glass, R. *Software Conflict.* Englewood Cliffs, NJ: Yourdon Press, 1991

Code maintenance ...

Code maintenance ...

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

Code maintenance ...

- **Code maintenance**: modification of a software product after it has been delivered.
- Purposes:
  - fixing bugs
  - improving performance
  - improving design
  - adding features

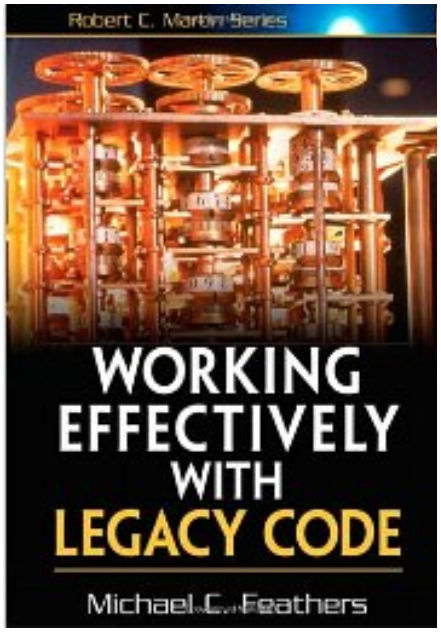
Code maintenance ...

- **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)

What Makes Code "Legacy"?

- Still meets customer need, **AND:**
- You didn't write it, and it's poorly documented
- You did write it, but a long time ago (and it's poorly documented)
- *It lacks good tests (regardless of who wrote it)* - Feathers 2004



Two Ways to Think About Modifying Legacy Code

- Edit & Pray
  - "I kind of think I probably didn't break anything"

- Cover & Modify
  - Let *test coverage* be your safety blanket

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

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 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

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?

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

How Agile Can Help

1. **Exploration:** determine where you need to make changes (*change points)*
2. **Refactoring**: is the code around change points (a) tested? (b) testable?
   - (a) is true: good to go
   - !(a) && (b): apply BDD+TDD cycles to improve test coverage
   - !(a) && !(b): **refactor**

How Agile Can Help, cont.

3. Add tests to **improve coverage** as needed
4. **Make changes**, using tests as ***ground truth***
5. **Refactor** further, to leave codebase better than you found it

- This is "embracing change" on long time scales

"Try to leave this world a little better than you found it."

Lord Robert Baden-Powell, founder of the Boy Scouts

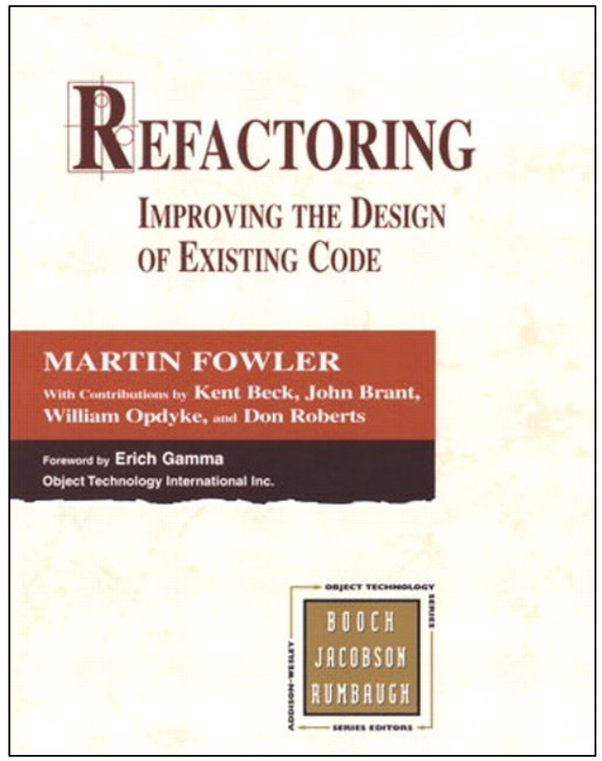
Exploration

- “Size up” the overall code base
- Identify key classes and relationships
- Identify most important data structures
- Ideally, identify place(s) where change(s) will be needed
- Keep design docs as you go
  - diagrams
  - GitHub wiki
  - comments you insert using RDoc

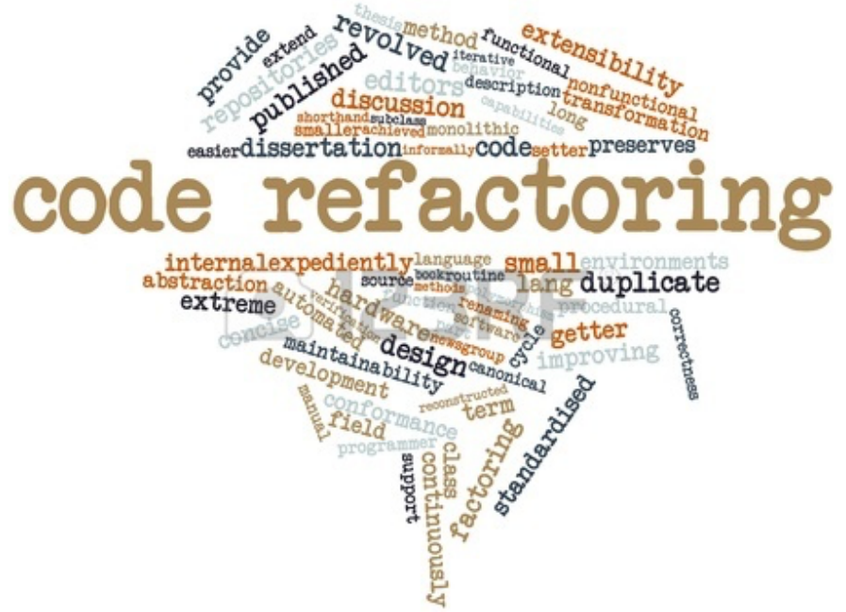
refactoring: what, when, why, and how

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

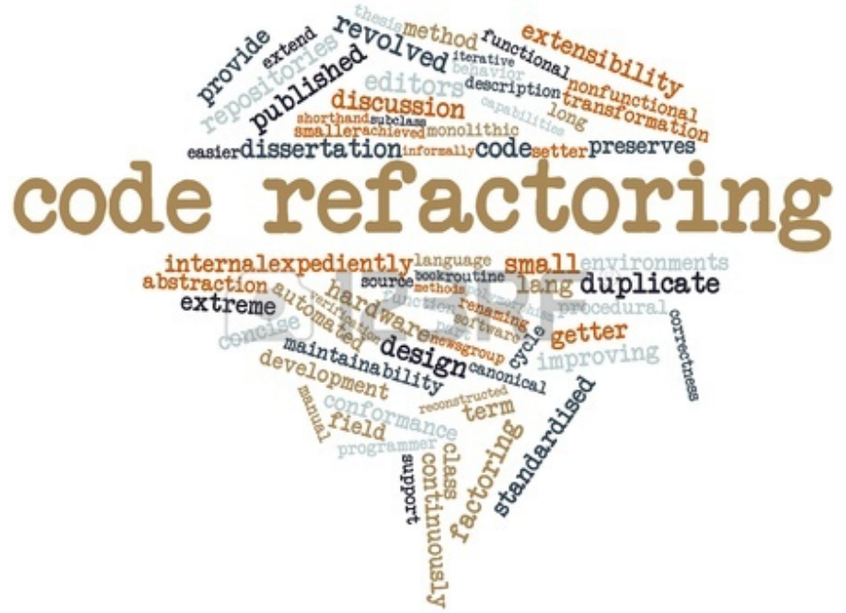


Why refactor?



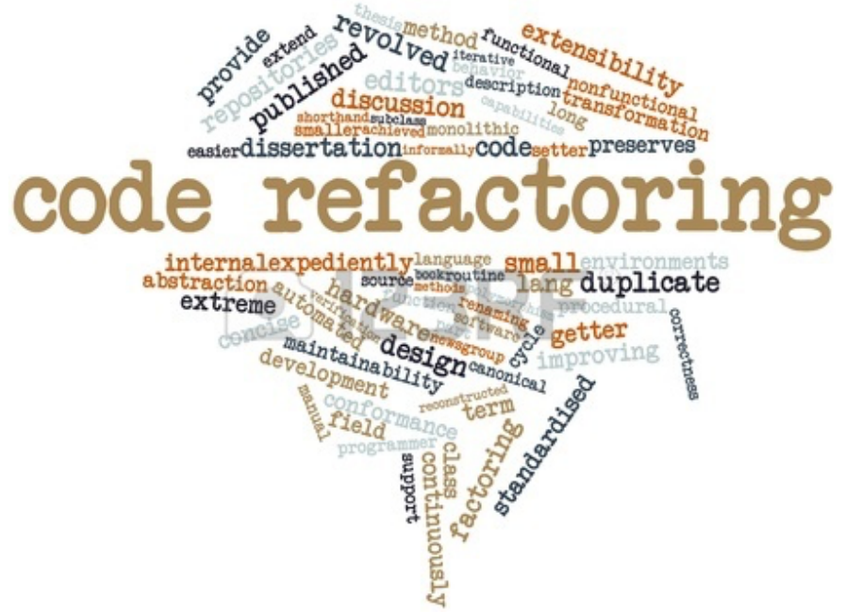
Why refactor?

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



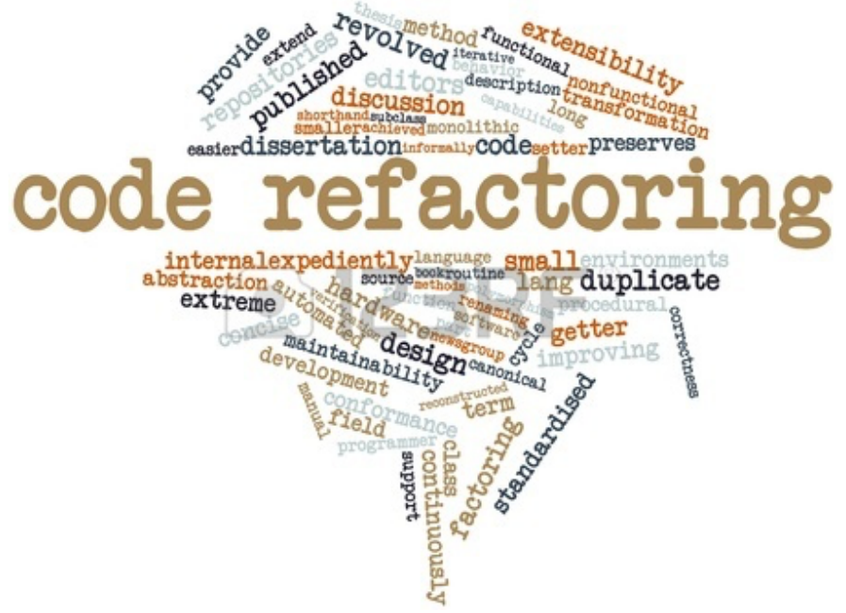
Why refactor?

- 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.



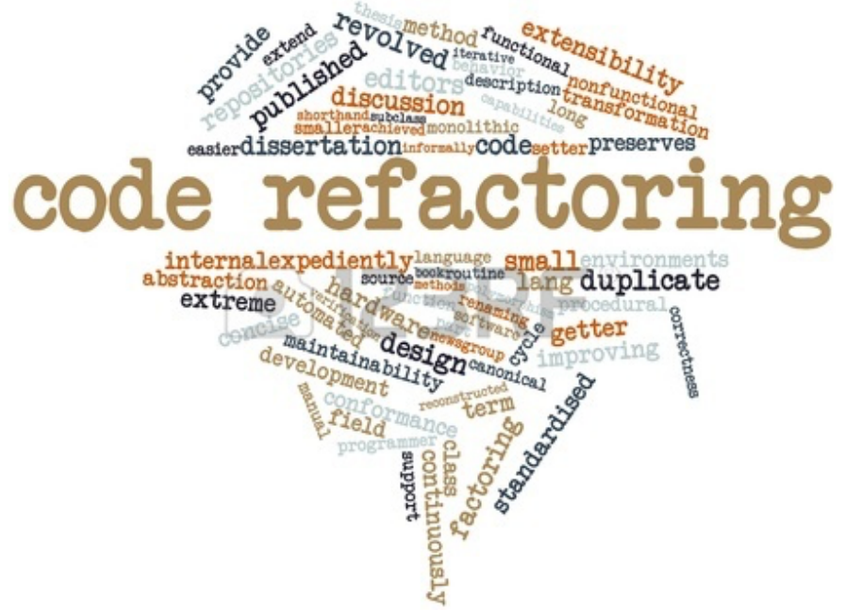
Why refactor?

- 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.



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 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)

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

Low-level refactoring

- Names:
    - Renaming (methods, variables)
    - Naming (extracting) "magic" constants

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

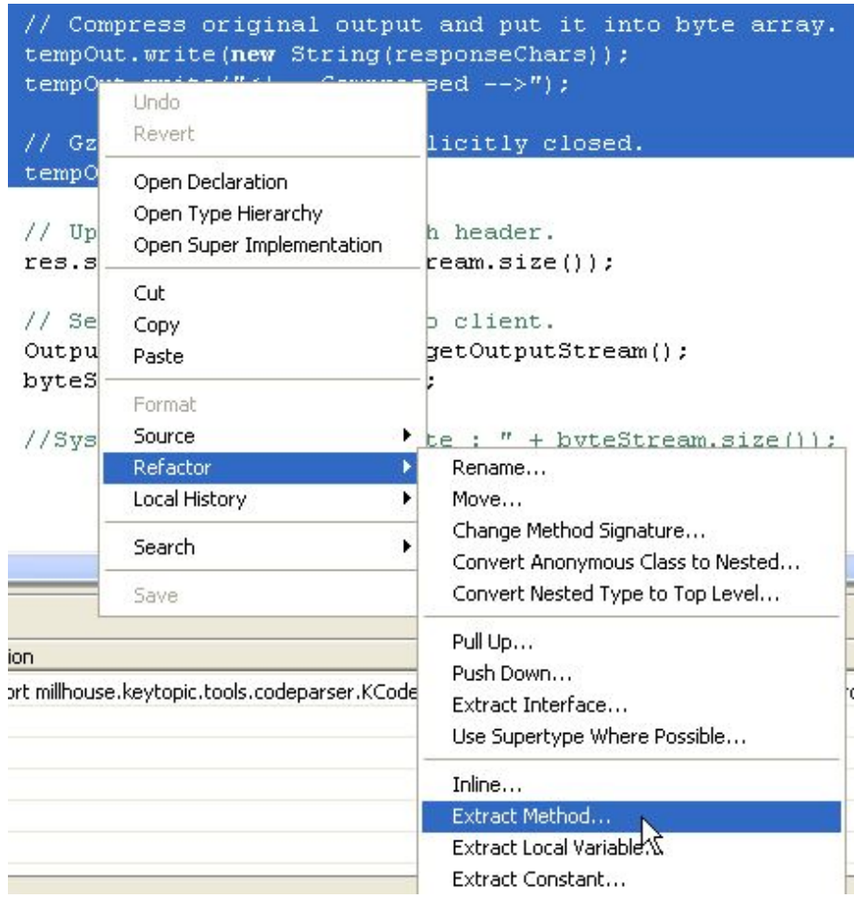
Low-level refactoring

- 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

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

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?

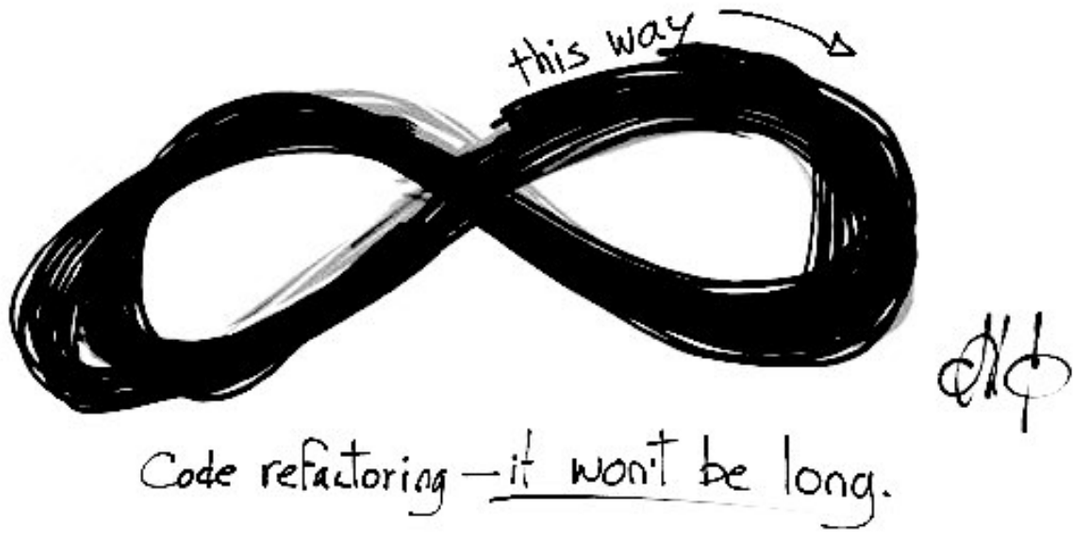
How to refactor? Have a plan!

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).

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.



Refactoring plan (2/2)

- **Refactor** the code.
    - Some tests may break. Fix the bugs.

Refactoring plan (2/2)

- **Refactor** the code.
    - Some tests may break. Fix the bugs.
- **Code review** the changes.

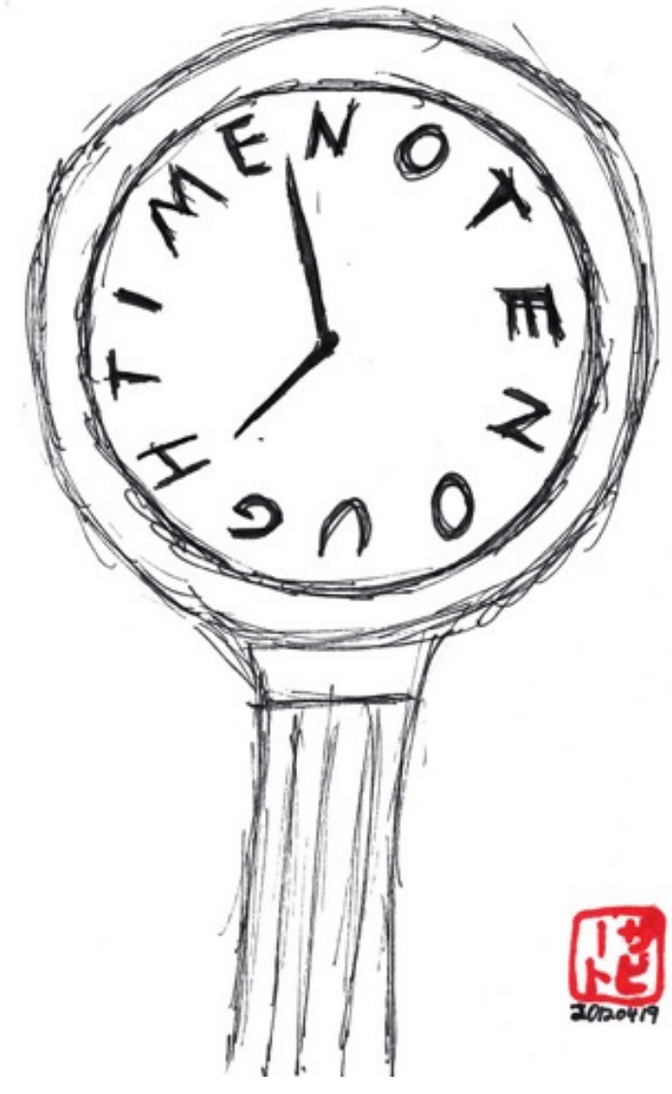
Refactoring plan (2/2)

- **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”



Refactoring: Idea

- Start with code that has 1 or more problems/ smells
- Through a series of *small steps,* transform to code without those smells
  - But be careful not to introduce new smells
- Protect each step with tests
- *Minimize time during which tests are red*

Quantitative: Metrics

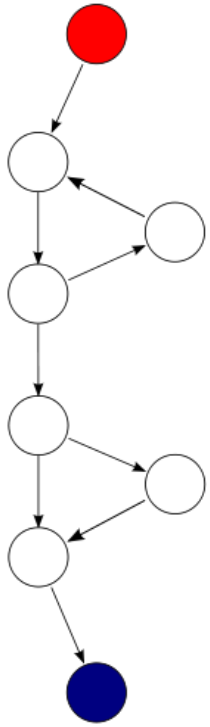
Metric	Tool	Target score
Code-to-test ratio	rake stats	≤ 1:2
C0 (statement) coverage	SimpleCov	90%+
Assignment-Branch-Condition score	flog	< 20 per method
Cyclomatic complexity	saikuro	< 10 per method (NIST)

- "Hotspots": places where *multiple metrics* raise red flags
  - add `require 'metric_fu'` to **`Rakefile`**
  - **`rake metrics:all`**
- Take metrics with a grain of salt
  - Like coverage, better for *identifying where improvement is needed* than for *signing off*

Cyclomatic Complexity (McCabe, 1976)

- # of linearly-independent paths thru code = E– N+2P (edges, nodes, connected components)

```
def mymeth
  while(...)
    ....
  end
  if (...)
    do_something
  end
end
```



- Here, E=9, N=8, P=1, so CC=3
- NIST (Natl. Inst. Stds. & Tech.): ≤10 /module

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 refactor.
    - 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.