What is refactoring?

- Refactoring is the process of improving the design or internal structure (i.e., quality) of existing code without negatively impacting correctness.
- To refactor is “to restructure software by applying a series of [changes to its internal structure] without changing its observable behavior.”

“Refactoring is risky. It requires changes to working code that can introduce subtle bugs.”
- Erich Gamma
Who’s Martin Fowler?

Martin Fowler is an author and international speaker on software development, specializing in object-oriented analysis and design, UML, patterns, and agile software development methodologies, including extreme programming.

Fowler is a member of the Agile Alliance and helped create the Manifesto for Agile Software Development in 2001, along with more than 15 co-authors. Martin Fowler was born in Walsall England, and lived in London a decade before moving to United States in 1994.

Why refactor?

- Refactoring improves the design of existing software
  - “Poorly designed code usually takes more code to do the same things”
- Refactoring improves the readability of code
- Refactoring helps to find more bugs, thereby improving quality
- Refactoring improves (i.e., decreases) development time
  - “Without a good design, you can progress quickly for a while, but soon the poor design starts to slow you down”
- Refactoring implies that we are aware of our inability to be perfect (i.e., helps us practice egoless programming)
**Refactor! But when?**

- “When you refactor you are improving the design of the code after it has been written”
- In general, “you refactor because you want to do something else, and refactoring helps you do that other thing”
- Refactor when you add new functionality
- Refactor when you find and fix a bug
- Refactor as a result of a code review
- Do not refactor unless you have a full test suite to help ensure whatever you change does not break existing functionality (that much)

**Problems with code**

- Most of a software system’s lifetime is spent in maintenance, where we modify and/or extend existing code
- Modifying code is inherently difficult when:
  - The code is difficult to read (i.e., poor naming conventions, long and murky blocks of code, spaghetti code, lack of clear and useful comments, code bloat, dead code, etc.)
  - The code is obfuscated or has overly complex logic
  - The code has duplicated functionality or logic
  - To make a single change or addition to the system, the code must be modified in multiple places
Refactor or rebuild?

- In some instances, you should not refactor your code or design; instead, start over!
- In short, “code has to work mostly correctly before you refactor”
- For large-scale systems, first refactor into large components using strong encapsulation and clear architectural divisions
  - Architectural divisions define specific boundaries between major system components
  - Well-designed and well-reviewed APIs are critical
  - Next, make the “refactor-versus-rebuild decision” for each of these components

AntiPatterns

- “Proper AntiPatterns define a migration (or refactoring) from negative solutions to positive solutions”