COMP2511

Refactoring

Prepared by

Ashesh Mahidadia

Refactoring: Motivation

- Code refactoring is the process of restructuring existing computer code without changing its external behavior.
- Originally Martin Fowler and Kent Beck defined refactoring as,

  "A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior… It is a disciplined way to clean up code that minimizes the chances of introducing bugs."
- **Advantages**: improved code readability, reduced complexity; improved maintenance and extensibility
- If done well, helps to identify *hidden* or *dormant* bugs or vulnerabilities, by simplifying code logic.
- If done poorly, may change external behavior, and/or introduce new bugs!
- Refactoring is different to adding features and debugging.

Refactoring: Motivation

- Refactoring is usually motivated by noticing a *code smell (possible bad design/coding practices)*.
- Code Smell is a hint that something might be wrong, **not** a certainty.
- Identifying a Code Smell allows us to re-check the implementation details and consider possible *better* alternatives.
- Automatic unit tests should be set up before refactoring to ensure routines still behave as expected.
- Refactoring is an iterative cycle of making a small program transformation, testing it to ensure correctness, and making another small transformation.

Software Maintenance

- Software Systems evolve over time to meet new requirements and features.
- Software maintenance involve:
  - Fix bugs
  - Improve performance
  - Improve design
  - Add features
- Majority of software maintenance is for the last three points!
- Harder to maintain code than write from scratch!
- Most of the development time is spent in maintenance!
- Good design, coding and planning can reduce maintenance pain and time!
- Avoid code smells to reduce maintenance pain and time!

Code Smells: Possible Indicators

- Duplicated code
- Poor abstraction (change one place → must change others)
- Large loop, method, class, parameter list; deeply nested loop
- Class has too little cohesion
- Modules have too much coupling
- Class has poor encapsulation
- A subclass doesn’t use majority of inherited functionalities
- 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 class/method/etc.
  - 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

- For more, see **http://www.refactoring.com/catalog/**

IDEs support low-level refactoring

- Renaming:
  - Variable, method, class.
- Extraction:
  - Method, constant
  - Repetitive code snippets
  - Interface from a type
- Inlining: method, etc.
- Change method signature.
- Warnings about inconsistent code.

Higher-level refactoring

- Refactoring to design patterns.
- Changing language idioms (safety, brevity).
- Performance optimization.
- Generally high-level refactoring is much more important, but unfortunately not well-supported by tools.