Software Refactoring: Investing in Software Quality

Rob Cameron

Computing Science 275
School of Computing Science
Simon Fraser University

October 18, 2021

Refactoring: Improving Software Maintainability

Software Refactoring

Refactoring is the process of modifying a software code base to improve quality while maintaining functionality intact.

Refactoring: Improving Software Maintainability

Software Refactoring

Refactoring is the process of modifying a software code base to improve quality while maintaining functionality intact.

Refactoring Goals

- Readability: making software easier to understand.
- Maintainability: making software easier to change.
- Reusability: making software modules suitable for different purposes.
- Adaptability: making software amenable to changing use cases.

Refactoring Needed: The Problem of Bit Rot

Bit Rot

Software systems that go through many versions over time often see a gradual detioration of code quality: bit rot.

Refactoring Needed: The Problem of Bit Rot

Bit Rot

Software systems that go through many versions over time often see a gradual detioration of code quality: bit rot.

Example: Code Bloat

- New feature required similar to existing feature.
- Existing feature implementation is cloned and altered.
- Result: code duplication, bit rot.

Refactoring Needed: The Problem of Bit Rot

Bit Rot

Software systems that go through many versions over time often see a gradual detioration of code quality: bit rot.

Example: Code Bloat

- New feature required similar to existing feature.
- Existing feature implementation is cloned and altered.
- Result: code duplication, bit rot.

Example: Shotgun Maintenance

- New feature requires new logic in many places.
- Shotgun pellets throughout the code

```
if (NewFeature) { ... }
```

- Result: increasing complexity, bit rot.

Refactoring Needed: Maintenance is Hard

Maintenance Activities

- Bug fixes.
- New features: adds complexity.
- Adapting to new systems: adds complexity.

Refactoring Needed: Maintenance is Hard

Maintenance Activities

- Bug fixes.
- New features: adds complexity.
- Adapting to new systems: adds complexity.

Maintenance Complexity

- Maintainers not original designers.
- Maintainers may be junior programmers.
- Code understanding is hard.
- Time pressure for fixes, new features.
- Result: bit rot.

Refactoring as Investment

Refactoring in Maintenace

- Focus is improving quality.
- Reduce code bloat and complexity without adding features.
- Bug fixes are incidental.
  - No particular bugs are the target of refactoring.
  - Refactoring may identify and remove bugs as part of code clean-up.

Refactoring as Investment

Refactoring in Maintenace

- Focus is improving quality.
- Reduce code bloat and complexity without adding features.
- Bug fixes are incidental.
    - No particular bugs are the target of refactoring.
    - Refactoring may identify and remove bugs as part of code clean-up.

Refactoring Cost-Benefit Tradeoff

- Refactoring is an investment of time for future benefit.
- Reduced future cost of bug fixes.
- Reduced future cost of new feature additions.
- Ability to reuse refactored components in future systems.
- Increased ability to adapt systems for new applications.

Test-Driven Refactoring

Test Cases First!

- Test cases validate current functionality.
- Refactoring steps should continue to pass all tests.
- New test cases should be written if testing is inadequate.
- Test failures: revert the refactoring and try again.

Test-Driven Refactoring

Test Cases First!

- Test cases validate current functionality.
- Refactoring steps should continue to pass all tests.
- New test cases should be written if testing is inadequate.
- Test failures: revert the refactoring and try again.

Small Steps

- Refactoring can be done in multiple small steps.
- Each step should preserve correctness: pass all tests.
- Each step should be a separate commit.
- Good commit history makes it easy to use git-bisect to locate bugs.

A Note About Git-Bisect

Searching through Commits

- A bug may be introduced by a particular commit.
- Binary search can locate the responsible commit.

A Note About Git-Bisect

Searching through Commits

- A bug may be introduced by a particular commit.
- Binary search can locate the responsible commit.

Git-Bisect Process

- Find a good commit `XXXXXX` before the bug occurred.
- `git bisect good XXXXXX` identifies the starting commit.
- Now check out a later faulty commit and issue the `git bisect bad` command.
- Git then checks out a commit half-way between.
- Test this commit and issue `git bisect good` or `git bisect bad` based on test success or failure.
- Git checks out the next commit to test using binary search.
- Repeat until the faulty commit is identified!

Incremental Refactoring

Catalogs of Refactorings are Known

- Global renaming: variable/field/method names.
- Method extraction: encapsulate a common code snippet.
- Moving features between objects.
- Simplifying conditional expressions.
- Many others, see Refactoring: Improving the Design of Existing Code, (Fowler et al)

Incremental Refactoring

Catalogs of Refactorings are Known

- Global renaming: variable/field/method names.
- Method extraction: encapsulate a common code snippet.
- Moving features between objects.
- Simplifying conditional expressions.
- Many others, see Refactoring: Improving the Design of Existing Code, (Fowler et al)

IDE Support

- Many IDEs support incremental refactoring.
- Reduce or eliminate programmer errors in refactoring.
- Allow larger refactorings to be composed of sequences of smaller ones.

Strategic Refactoring

Pattern-Driven Refactoring

- Analysis of a software system may determine that it is should be restructured according to a well-known *design pattern*.
- Example: Model-View-Controller pattern.
    - Separate the model, view and control components of an interactive application.
    - Good advice for interactive app development.
- Improves quality for future maintainance.
- Improves readability by using a well-known pattern.

Strategic Refactoring

Pattern-Driven Refactoring

- Analysis of a software system may determine that it is should be restructured according to a well-known *design pattern*.
- Example: Model-View-Controller pattern.
  - Separate the model, view and control components of an interactive application.
  - Good advice for interactive app development.
- Improves quality for future maintainance.
- Improves readability by using a well-known pattern.

Reducing Coupling

- Analysis of class structure may shown too many dependencies between classes.
- Restructuring to reduce coupling can make code more modular and easier to understand and maintain.