Refactoring

CSE 403 Software Engineering

Autumn 2023

Today's Outline

- What's refactoring
- Why refactor
- When refactor
- How refactor

Here’s the problem

Software can live and evolve for months and years, with new features, new bug [fixes], new algorithms, new developers, new coding practices, new …

- If the code's structure does not also evolve, it will become harder and harder to maintain, no less improve
- This can happen even if the code was initially reviewed and well-designed at the time of check-in

Is there anything wrong with this code?

```
char b[2][10000],*s,*t=b,*d,*e=b+1,**p;main(int c,char**v)
{int n=atoi(v[1]);strcpy(b,v[2]);while(n--){for(s=t,d=e;*s;s++)
{for(p=v+3;*p;p++)if(**p==*s){strcpy(d,*p+2);d+=strlen(d);
goto x;}*d++=*s;x:}s=t;t=e;e=s;*d++=0;}puts(t);}
```

```
while (*a++ = *b--) ;
```

We can maintain code

Code maintenance: modifying or repairing of code generally after it has been delivered/deployed

Purposes:

- Fix bugs
- Adapt to environment changes (e.g., performance, load)
- Add and evolve features

Note that maintenance is hard

- It can be harder to maintain code than write your own new code
  - “House of cards" phenomenon (don't touch it!)
  - Must understand code written by another developer,
    or code you wrote at a different time with a different mindset
- Yet maintenance is how developers spend much of their time
- It pays to design software well and plan ahead so that later maintenance will be less painful (e.g., extensible design)

We can also periodically refactor code

Refactoring: revising the code to improve its internal structure, reduce complexity, or otherwise accommodate change without altering its external behavior

Is adding a feature or a bug fix, refactoring?

Why fix something that isn't broken?

Each part of a system's code has 3 purposes:

1. To execute its functionality
2. To allow for evolution
3. To communicate well to developers who read it

If the code does not do one or more of these, it is "broken" and needs some investment!

Pick up on the need-to-refactor signs

Consider refactoring when:

- Code is **duplicated**
- A routine is **too long**
- A loop is too long or **deeply nested**
- A class has poor **cohesion**
- A class uses too much **coupling**
- Inconsistent level of **abstraction**
- Too many **parameters**
- To **compartmentalize** changes
- To modify an **inheritance hierarchy** in parallel
- To **group related data** into a class
- A **"middle man"** object doesn't do much
- **Spaghetti code**
- **Poor encapsulation** of data that should be private
- A **weak subclass** doesn't use its inherited functionality
- A class contains **unused code**

"I don't have time!"

Refactoring incurs an **up-front cost**.

- Some developers don't want to do it
- Management can have concerns - they lose time and gain "nothing" (no new features)

But...

- Well-written code is more conducive to **rapid development** (some estimates put ROI at 500% or more for well-done code)
- Refactoring is good for **programmer morale**
  - Developers prefer working in a "clean house"

So when should we refactor?

Let’s do some refactoring!

```
function base(aReading) {...}
function taxableCharge(aReading) {...}
function calculateBaseCharge(aReading) {...}
```

Example 1:
What aspects should be refactored and how?

```
class Reading {
	base() {...}
	taxableCharge() {...}
	calculateBaseCharge() {...}
}
```

```
function foundPerson(people) {
  for (let i = 0; i < people.length; i++) {
      if (people[i] === "Don") {
          return "Don";
      }
      if (people[i] === "John") {
          return "John";
      }
      if (people[i] === "Kent") {
          return "Kent";
      }
  }
  return "";
}
```

Example 2:
What aspects should be refactored and how?

Is this an improvement?

```
function foundPerson(people) {
        const candidates = ["Don", "John", "Kent"];
        return people.find( p=>candidates.includes(p) )  || ‘’;
 }
```

```
Class Animal {
  static final int TYPE_DOG = 1;
  static final int TYPE_CAT = 2;
  int type;

  void makeSound() {
   switch (type) {
      case TYPE_DOG:
              System.out.println(“woof”);
              break;
      case TYPE_CAT:
            System.out.println(“meow”);
            break;
   }
 }
}
```

Example 3:
What aspects should be refactored and how?

```
Interface Animal {
  void makeSound();
}

Class Dog implements Animal {
  @Override
  void makeSound() {
    System.out.println(“woof”);
}}

Class Cat implements Animal {
  @Override
  void makeSound() {
    System.out.println (“meow”);
}}
```


https://www.baeldung.com/cs/refactoring

Great resource by Martin Fowler

Let's look at a few!

This is the online catalog of refactorings, to support my book Refactoring 2nd Edition.

This catalog of refactorings includes those refactorings described in my original book on Refactoring, together with the Ruby Edition.

Using the Catalog ▶

Tags

- basic
- encapsulation
- moving-features
- organizing-data
- simplify-conditional-logic
- refactoring-apis
- dealing-with-inheritance
- collections
- delegation
- errors
- extract
- parameters
- fragments
- grouping-function
- immutability
- inline
- remove
- rename
- split-phase
- variables

#

Change Function Declaration
Add Parameter • Change Signature • Remove Parameter • Rename Function • Rename Method

Change Reference to Value

Change Value to Reference

Collapse Hierarchy

Combine Functions into Class

Combine Functions into Transform

Consolidate Conditional Expression

Decompose Conditional

Encapsulate Collection

Remove Dead Code

Remove Flag Argument
Replace Parameter with Explicit Methods

Remove Middle Man

Remove Setting Method

Remove Subclass
Replace Subclass with Fields

Rename Field

Rename Variable

Replace Command with Function

Replace Conditional with

There are MANY forms of refactoring

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

There are MANY forms of refactoring

High level refactoring

- Refactoring design or even architecture

Compared to low-level refactoring, **high-level** is:

- Not as well-supported by tools
- But can be even more important and valuable

Tools, did you say IDE tools?

Visual Studio Code Docs Updates Blog API Extensions FAQ Learn

OVERVIEW
SETUP
GET STARTED
USER GUIDE
- Basic Editing
- Extension Marketplace
- IntelliSense
- Code Navigation
- Refactoring
- GitHub Copilot
- Debugging
- VS Code for the Web
- Tasks
- Profiles
- Settings Sync
- Snippets
- Emmet
- Command Line Interface
- Workspace Trust
- Multi-root Workspaces
- Accessibility
- Custom Layout
- Port Forwarding

Refactoring

Edit

Source code refactoring can improve the quality and maintainability of your project by restructuring your code while not modifying the runtime behavior. Visual Studio Code supports refactoring operations (refactorings) such as Extract Method and Extract Variable to improve your code base from within your editor.

JS app.js ×

```
24
25  app.use('/', index);
26  app.use('/users', users);
27  app.us
28  // catch 404 and forward to error handler
29  app.use(function(req, res, next) {
30    var err = new Error('Not Found');
31    err.status = 404;
32    next(err);
33
34
35  // error handler
36  app.use(function(err, req, res, next) {
```

Extract to function in global scope

For example, a common refactoring used to avoid duplicating code (a maintenance headache) is the Extract Method refactoring, where you select source code that you'd like to reuse elsewhere and pull it out into its own shared method.

JET BRAINS Developer Tools Team Tools

AppCode

Refactorings & Code Generat

```
view.addSubview(tableView)

let label = SmartLabel()
label.text = UIConstants.strings.autocompleteEmptyState
label.font = UIConstants.fonts.settingsDescriptionText
```

Refactor This
1. Rename... ⇧F6
2. Copy File... F5
Extract/Introduce
3. Introduce Variable... ⌥⌘V
4. Closure...
5. Extract Method... ⌥⌘M

Refactorings

To help you easily improve code design as it evolves over time, AppCode provides a solid set of reliable code refactorings.

To see all refactorings available at the current location, use the **Refactor This...** menu (press `^T`).

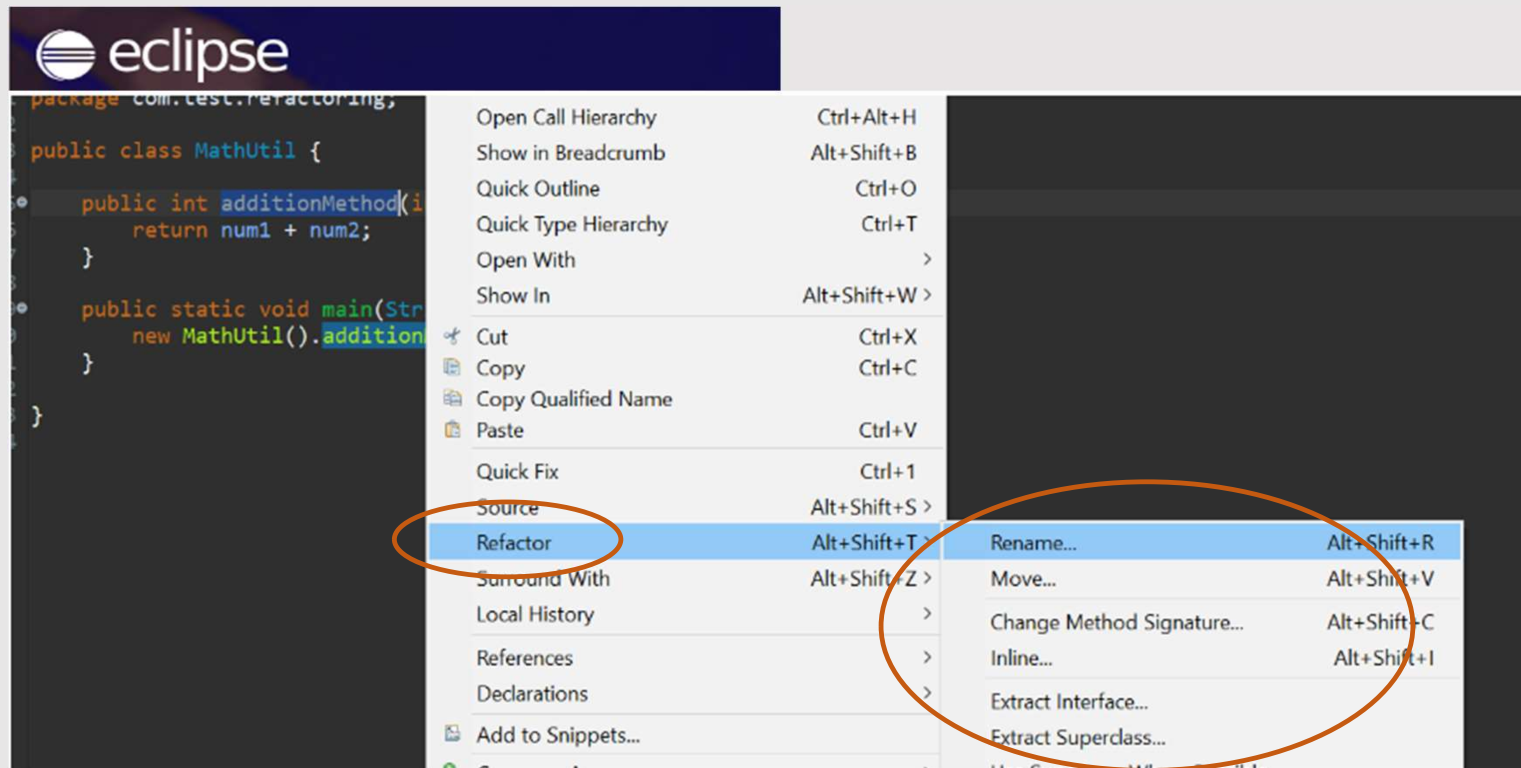
Tools, did you say tools?

JET BRAINS

Developer Tools Team Tools

Visual St

OVERVIEW
SETUP
GET STARTED
USER GUIDE
Basic Editing
Extension Market
IntelliSense
Code Navigation
Refactoring
GitHub Copilot
Debugging
VS Code for the W
Tasks
Profiles
Settings Sync
Snippets
Emmet
Command Line Interface
Workspace Trust
Multi-root Works
Accessibility
Custom Layout
Port Forwarding



Generat

lves over
ode

headache) is the Extract Method refactoring, where you select source code that you'd like to reuse elsewhere and pull it out into its own shared method.

To see all refactorings available at the current location, use the **Refactor This...** menu (press ^T).

There are many others!

Modern IDEs support low level refactoring patterns:

- 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
- Warnings about method invocations with inconsistent parameters
- Help with self-documenting code through auto-completion

Sadly, older development “environments” (e.g., vi, emacs, etc.)

- Have little or no support for refactoring, and thus offer little encouragement for the developer

Back to basics

pollev.com/cse403au

When adding some new functionality, in what order would you do the following?

Make the necessary code changes

Refactor the code

Write unit tests to ensure that the important conditions that need to be met are indeed met

Respond at **pollev.com/cse403au**

Refactoring: When adding new functionality

0 done

0 underway

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

When poll is active, respond at **pollev.com/cse403au**

First

- Make the necessary code changes
- Refactor the code
- Write unit tests

Total Results: 0

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

When poll is active, respond at **pollev.com/cse403au**

Second

Make the necessary code changes

Refactor the code

Write unit tests

Total Results: 0

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

When poll is active, respond at **pollev.com/cse403au**

Third

Make the necessary code changes

Refactor the code

Write unit tests

Total Results: 0

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Back to basics

When adding some new functionality, in what order would you do the following?

Write unit tests to ensure that the important **(existing)** conditions that need to be met are indeed met

Refactor the code

Make the necessary code changes

It can depend on the development process you're using

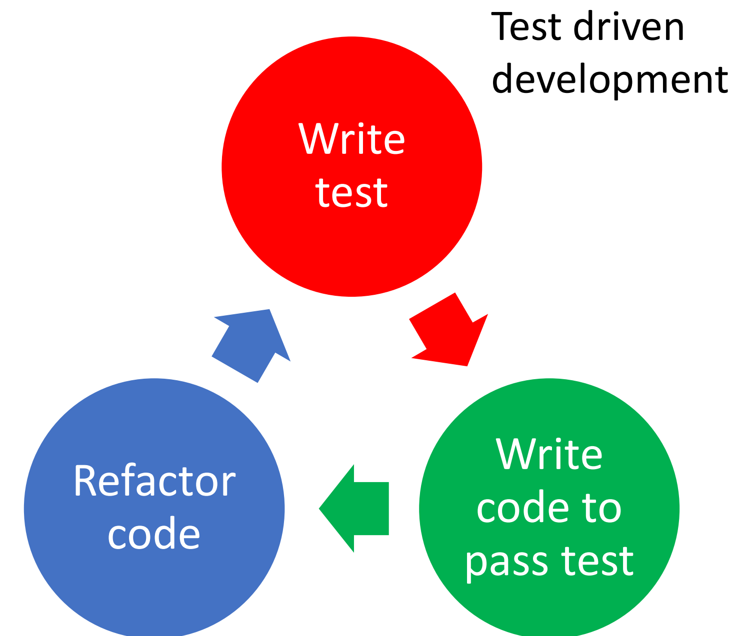
Back to basics

When adding some new functionality, in what order would you do the following?

Write unit tests to ensure that the important **(new)** conditions that need to be met are indeed met

Make the necessary code changes

Refactor the code



Refactoring in six steps

1. Analyze the code to decide the risk/reward of refactoring
2. Check in the code before you change it
3. Write unit tests that verify the code's external correctness
4. Refactor the code and ensure the tests still pass!
5. Code review the changes
6. Check in the refactored code (and only the refactor)

To summarize - top reasons for refactoring

Improve maintainability, which is the ability to

- Fix bugs
- Adapt to environment changes (e.g., performance, load)
- Add and evolve features

and hence, **improve productivity**!

It’s [almost] a wrap!

What’s left:

- Final release milestone and demo
  - Don’t forget to signup for a presentation slot (see Ed for link)!
- Individual retrospective milestone
- Team member survey