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?

```c
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.

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!
Example 1:
What aspects should be refactored and how?

function base(aReading) {...}
function taxableCharge(aReading) {...}
function calculateBaseCharge(aReading) {...}
class Reading {
    base() {...}
    taxableCharge() {...}
    calculateBaseCharge() {...}
}

https://refactoring.com
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 "";
}
function foundPerson(people) {
    const candidates = ["Don", "John", "Kent"];
    return people.find(p => candidates.includes(p)) || "";
}

Is this an improvement?
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;
        }
    }
}
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");
    }
}
Great resource by Martin Fowler

Let’s look at a few!

Using the Catalog

<table>
<thead>
<tr>
<th>Tags</th>
<th>Change Function Declaration</th>
<th>Remove Dead Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic</td>
<td>Add Parameter · Change Signature · Remove Parameter · Rename Function · Rename Method</td>
<td></td>
</tr>
<tr>
<td>encapsulation</td>
<td>Change Reference to Value</td>
<td>Remove Flag Argument</td>
</tr>
<tr>
<td>moving-features</td>
<td>Change Value to Reference</td>
<td>Replace Parameter with Explicit Methods</td>
</tr>
<tr>
<td>organizing-data</td>
<td>Collapse Hierarchy</td>
<td>Remove Middle Man</td>
</tr>
<tr>
<td>simplify-conditional-logic</td>
<td>Combine Functions into Class</td>
<td>Remove Setting Method</td>
</tr>
<tr>
<td>refactoring-apis</td>
<td>Combine Functions into Transform</td>
<td></td>
</tr>
<tr>
<td>dealing-with-inheritance</td>
<td>Consolidate Conditional Expression</td>
<td></td>
</tr>
<tr>
<td>collections</td>
<td>Decompose Conditional</td>
<td>Replace Subclass with Fields</td>
</tr>
<tr>
<td>delegation</td>
<td>Encapsulate Collection</td>
<td>Rename Field</td>
</tr>
<tr>
<td>errors</td>
<td></td>
<td>Rename Variable</td>
</tr>
<tr>
<td>extract</td>
<td></td>
<td>Replace Command with Function</td>
</tr>
<tr>
<td>parameters</td>
<td></td>
<td>Replace Conditional with</td>
</tr>
<tr>
<td>fragments</td>
<td></td>
<td></td>
</tr>
<tr>
<td>grouping-function</td>
<td></td>
<td></td>
</tr>
<tr>
<td>immutability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inline</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rename</td>
<td></td>
<td></td>
</tr>
<tr>
<td>split-phase</td>
<td></td>
<td></td>
</tr>
<tr>
<td>variables</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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?

Refactoring

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.

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.
Tools, did you say tools?

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

1. Refactor the code
2. Make the necessary code changes
3. Write unit tests to ensure that the important conditions that need to be met are indeed met
Refactoring: When adding new functionality
First

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

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

Test driven development

Write test

Write code to pass test

Refactor 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