

CS3300 Introduction to Software Engineering

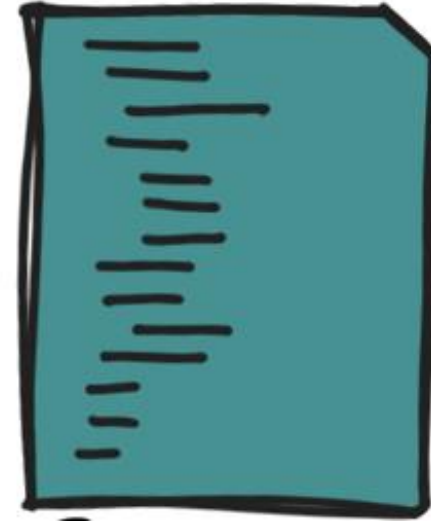
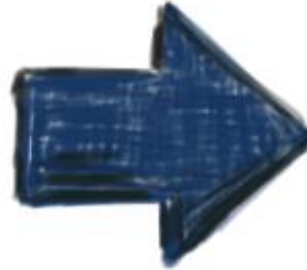
# Lecture 20: Software Refactoring

Nimisha Roy ▶ [nroy9@gatech.edu](mailto:nroy9@gatech.edu)

# What is Refactoring?



Program



Refactored Program

Applying transformations to a program, with the goal of improving its design without changing its functionality

**Goal:** Keep program readable, understandable, and maintainable. Avoid small problems soon.

**Key Feature:** Behavior Preserving- make sure the program works after each step; typically small steps

# Behavior Preserving

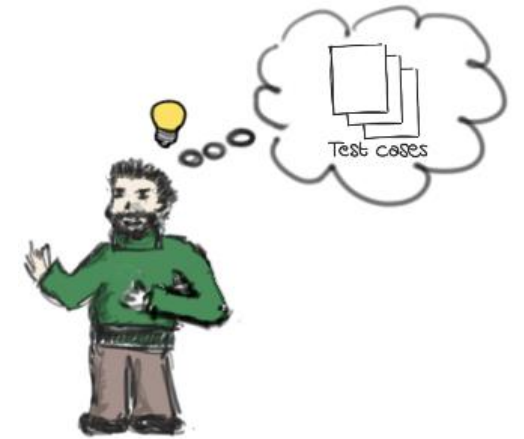


How can we ~~guarantee~~ it?

Test the code

In agile we already have lot of test cases, rerun before and after refactoring)

But beware: No guarantees!



# Behavior Preserving Quiz



Why can't testing guarantee that a refactoring is behavior preserving?

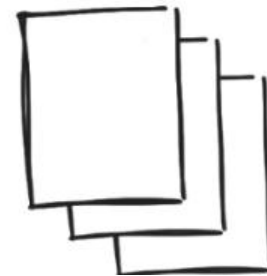
Because testing and refactoring are different activities

Because testing is inherently incomplete

Because testers are often inexperienced



Program Domain



Test Cases

# Why Refactoring?



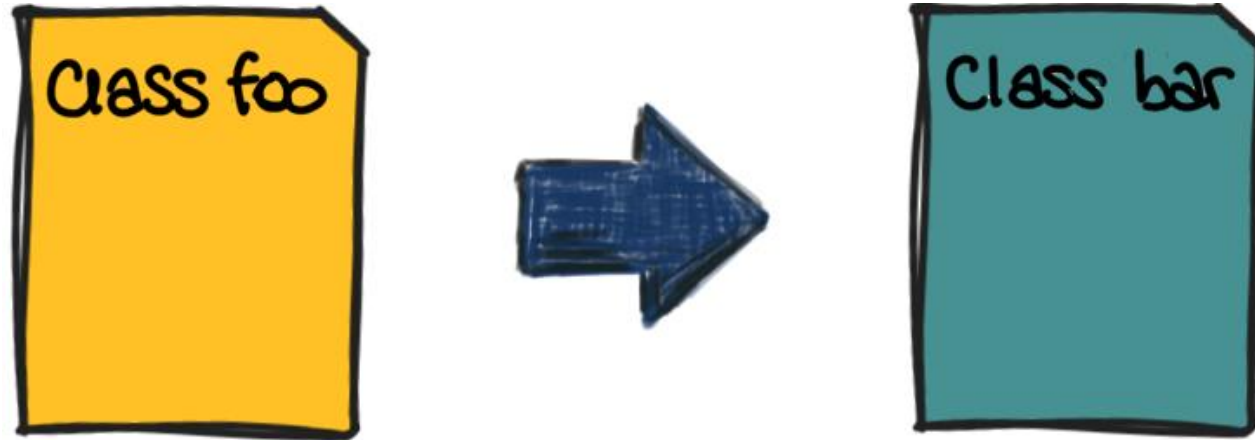
Requirements Change – different design needed

Design needs to be improved – so that new features can be added; design patterns are often a target

Sloppiness by programmers – copy & paste for a new method

Refactoring often has the effect of making a design more flexible

# Have you used Refactoring Before?



Even renaming a class is a refactoring!  
(albeit a trivial one)

# A little bit of history

- Refactoring is something programmers have always done
- Especially important for object-oriented languages
- Opdyke's PhD Thesis (1990) discusses refactoring for SMALLTALK
- Increasingly popular due to agile development (makes the changes less expensive)

# Fowler's Book – Improving the Design of Existing Code



- Catalogue of Refactorings
- Lot of Bad Smells
- Guidelines on when to apply refactoring
- Example of code before and after



# Many Refactorings in Fowler's Book

- Add parameter
- Change Association
- Reference to Value
- Value to Reference
- Collapse Hierarchy
- Consolidate Conditionals
- Procedures to Objects
- Decompose Conditionals
- Encapsulate Collection
- Encapsulate DOWNCast
- Encapsulate Field
- Extract Method
- Extract Class
- Inline Class
- Form Template Method
- Hide delegate
- Hide method
- Inline temp
- ...

# Collapse Hierarchy

If a superclass and a subclass are too similar

=> Merge Them



# Many Refactorings in Fowler's Book

- Add parameter
- Change Association
- Reference to Value
- Value to Reference
- Collapse Hierarchy
- Consolidate Conditionals
- Procedures to Objects
- Decompose Conditionals
- Encapsulate Collection
- Encapsulate DOWNCast
- Encapsulate Field
- Extract Method
- Extract Class
- Inline Class
- Form Template Method
- Hide delegate
- Hide method
- Inline temp
- ...

# Consolidate Conditional Expression

If there are a set of conditionals with the same results

=> Combine and extract them

```
double disabilityAmount(){
    if (seniority < 2)
        return 0;
    if (monthsDisabled > 12)
        return 0;
    if (isPartTime)
        return 0;
    // compute disability amount
}
```



```
double disabilityAmount(){
    if (notEligibleForDisability())
        return 0;
    // compute disability amount
}
```

# Many Refactorings in Fowler's Book

- Add parameter
- Change Association
- Reference to Value
- Value to Reference
- Collapse Hierarchy
- Consolidate Conditionals
- Procedures to Objects
- Decompose Conditionals
- Encapsulate Collection
- Encapsulate DOWNCast
- Encapsulate Field
- Extract Method
- Extract Class
- Inline Class
- Form Template Method
- Hide delegate
- Hide method
- Inline temp
- ...

# Decompose Conditionals

If a conditional statement is particularly complex (can tell what but obscures why)

⇒ Extract methods from conditions, give the right name to the extracted method


⇒ Modify THEN and ELSE part of the conditional

```
if (date.before (SUMMER_START) || date.after (SUMMER_END))  
    charge = quantity * winterRate + winterServiceCharge;  
else  
    charge = quantity * summerRate;
```



```
if (not Summer(date))  
    charge = winterCharge(quantity)  
else  
    charge = summerCharge(quantity)
```

# Many Refactorings in Fowler's Book

- Add parameter
  - Change Association
  - Reference to Value
  - Value to Reference
  - Collapse Hierarchy
  - Consolidate Conditionals
  - Procedures to Objects
  - Decompose Conditionals
  - Encapsulate Collection
- 
- Encapsulate DOWNCast
  - Encapsulate Field
  - Extract Method
  - Extract Class
  - Inline Class
  - Form Template Method
  - Hide delegate
  - Hide method
  - Inline temp
  - ...

# Extract Class

If a class is doing the work of two classes

⇒ Create a new class and move the relevant fields/methods (high cohesion, low coupling)





# Many Refactorings in Fowler's Book

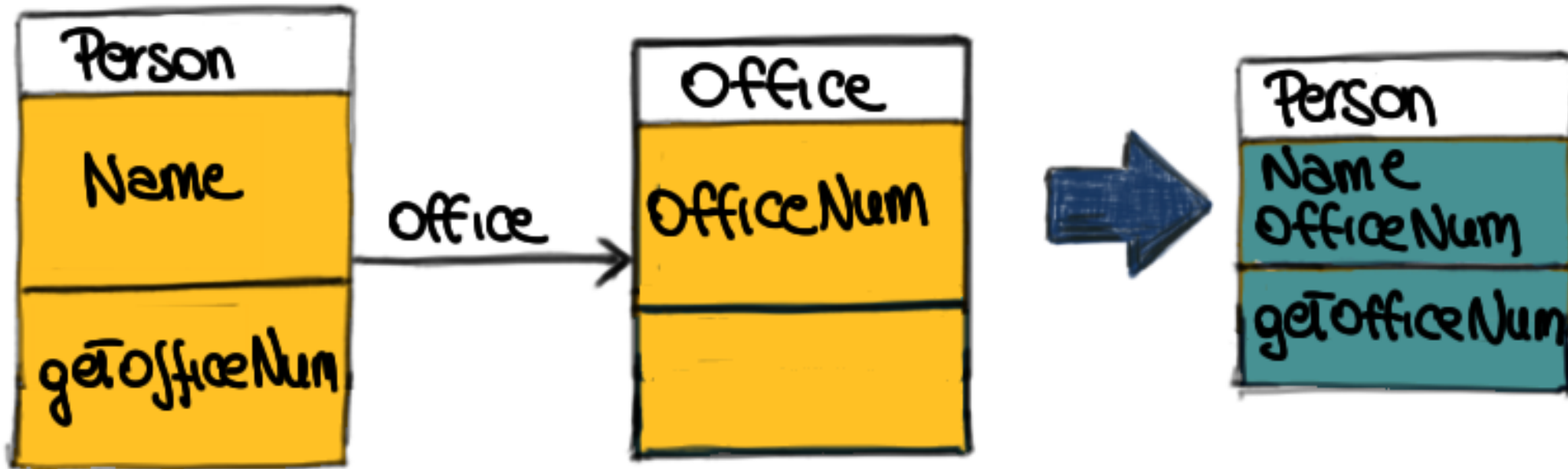
- Add parameter
- Change Association
- Reference to Value
- Value to Reference
- Collapse Hierarchy
- Consolidate Conditionals
- Procedures to Objects
- Decompose Conditionals
- Encapsulate Collection
- Encapsulate DOWNCast
- Encapsulate Field
- Extract Method
- Extract Class
- Inline Class
- Form Template Method
- Hide delegate
- Hide method
- Inline temp
- ...




# Inline Class

If a class is not doing much during system evolution

⇒ Move its features into another class and delete this one



# Many Refactorings in Fowler's Book

- Add parameter
  - Change Association
  - Reference to Value
  - Value to Reference
  - Collapse Hierarchy
  - Consolidate Conditionals
  - Procedures to Objects
  - Decompose Conditionals
  - Encapsulate Collection
- 
- Encapsulate Dwnccast
  - Encapsulate Field
  - Extract Method
  - Extract Class
  - Inline Class
  - Form Template Method
  - Hide delegate
  - Hide method
  - Inline temp
  - ...

# Extract Method

If there is a cohesive code fragment in a large method

=> Create a method using that code fragment, replace code fragment with a call to the method

```
void printOwing() {  
    ...  
    System.out.println("name:" + name +  
                       "address:" + address);  
    ...  
    System.out.println("amount owed " +  
                       amount);  
    ...  
}
```



# How can we actually perform Refactoring?

Manually

Also automated using the right tools

## Demo Time!!

Using Extract Method Refactoring in Eclipse IDE

# Refactoring Techniques Quiz



When is it appropriate to apply refactoring “extract method”?

- When there is duplicated code in two or more methods
- When a class is too large
- When the names of two classes are too similar
- When a method is highly coupled with a class other than the one where it is defined

# Refactoring Risks



Powerful tool, but...

- May introduce subtle faults (regression errors)
- Should not be abused
- Should be used carefully on systems in production (affects users, new version of software to be released soon)

# Cost of Refactoring

Manual Work

Test Development and Maintenance  
(update test cases even in agile  
development)

Documentation Maintenance





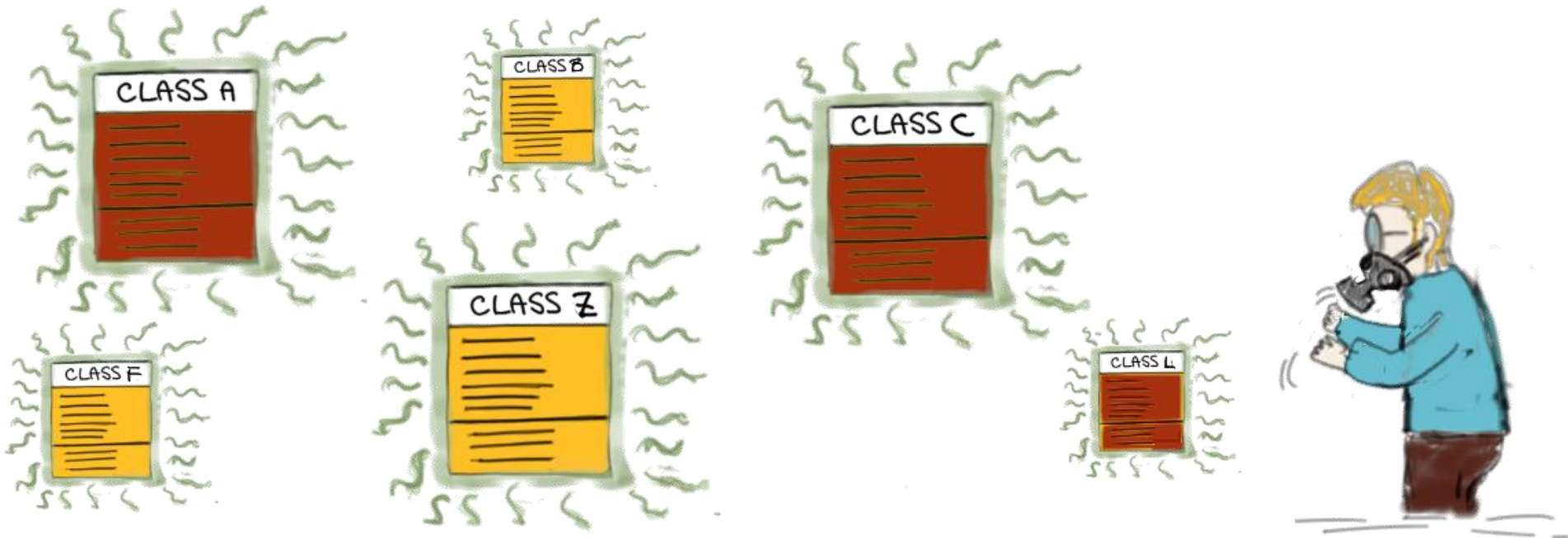
# When not to Refactor?

When code is broken (not a way to fix code)

When a deadline is close

When there is no reason to!

# BAD SMELLS



- Symptoms that indicate deeper problems in the code.
- Should be able to sense/sniff it.
- Not bugs, indicate weakness in design and hence maintenance in code.

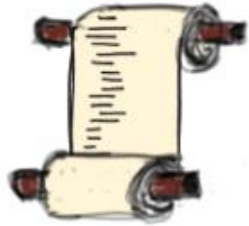
# A catalogue of Bad Smells

- Duplicated Code
- Long Method
- Large Class
- Long parameter list
- Divergent Change
- Shotgun Surgery
- Feature Envy
- Data Clumps
- Primitive Obsession
- Switch Statements
- Parallel Interface Hierarchy
- Lazy Class
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- Inappropriate Intimacy
- Incomplete Library Class
- Data Class
- Refused bequest

# A few Examples



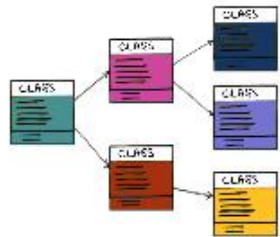
Duplicated code → extract method



Long method → extract method, decompose conditionals, ...



Large class → extract class (or subclass)



Shotgun surgery → move method / field, inline class, ...



Feature envy → extract method, move method

# Bad Smells Quiz



Which of the following can be considered to be “bad smells” in the context of refactoring?

The program takes too long to execute

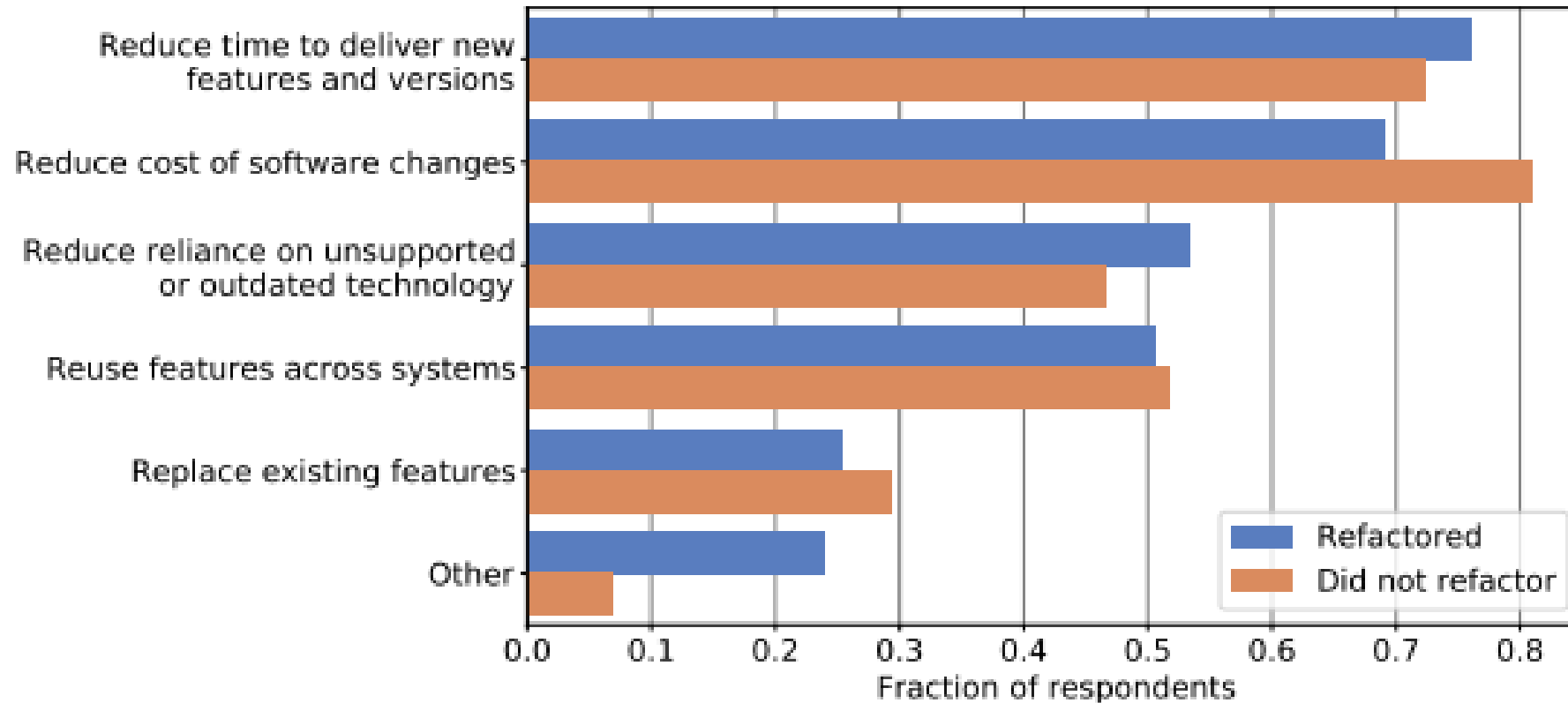
Method `m()` in class `c()` is too long

Classes *Cat* and *Dog* are subclasses of class *Animal*

Every time we modify method `m1()`, we also need to modify method `m2()`

# Refactoring Industry Standards – Industry Survey

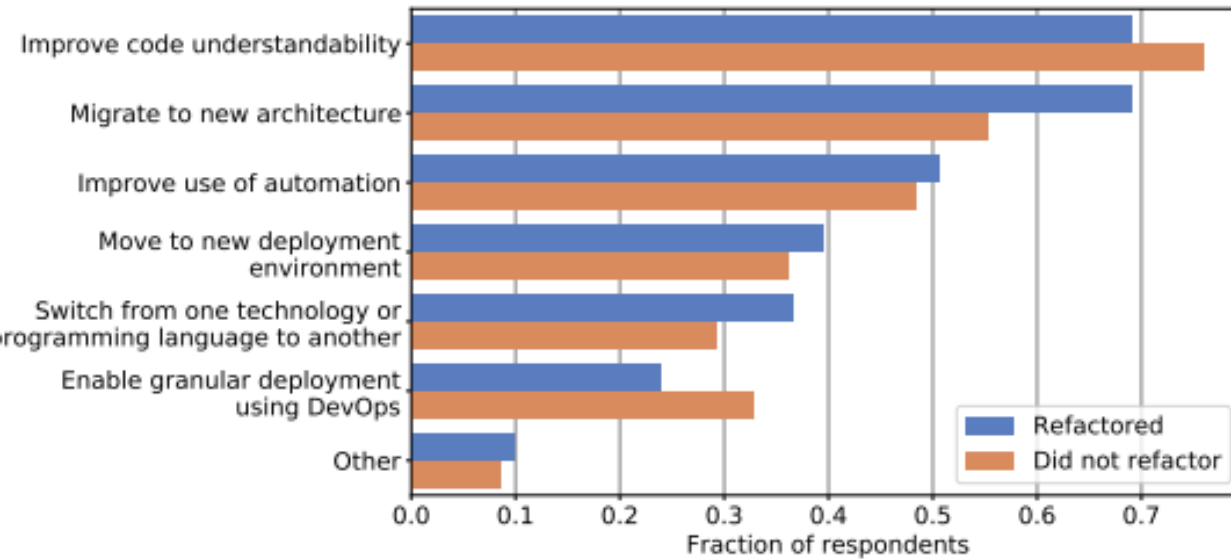
- Small-scale (floss) refactoring is common ; performed by a single developer; manual
- Multiple Large-scale refactoring also common; takes months; sometimes adding new features becomes priority



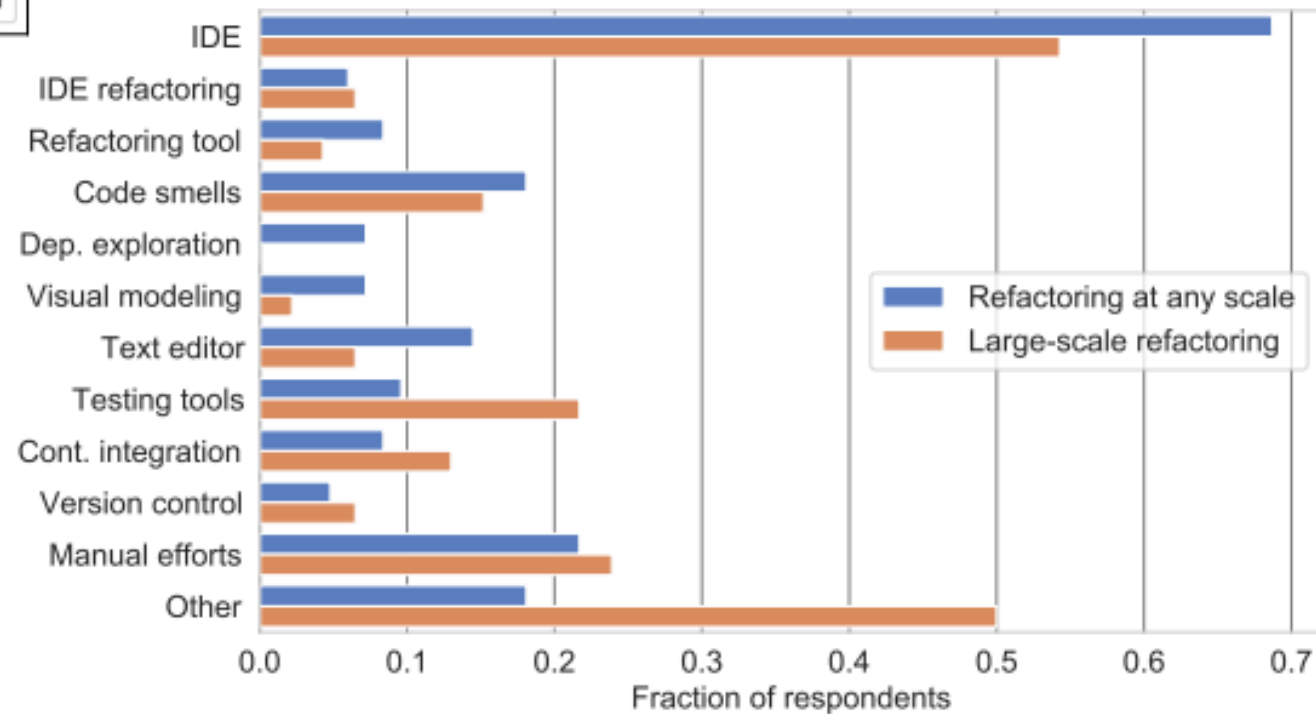
**Figure 4: Business reasons for large-scale refactoring.**

[Reference Article](#)

# Refactoring Industry Standards – Industry Survey



**Top Tools:** ReSharper (.Net), Jdeodrant (Eclipse Plugin), JetBrains Rider (.NET), JetBrains IntelliJ IDEA (Java), Spring Tool Suite, Stepsize



**Figure 5: Technical reasons for large-scale refactoring.**

Clear need for better tools and an opportunity for refactoring researchers to make a difference in industry

**Figure 7: Categories of tools used to support refactoring.**

# Refactoring Industry Standards – My Survey

- “Don’t touch code if it is working”
- **Jetbrains** integrated in Visual Studio, paid tools integrated
- Gives helpful prompts while writing code
- When refactoring?
  - Approving other developers PR- suggest floss refactoring
  - LSR – automated code quality check tools
- **SonarQube** - code quality inspection tool before completing a PR- minimum of **B**
- Based on many different rules for different language (650 for Java) covering code smells, test coverage, code security.
- Final verdict: automated tools are very important since there is no time to make changes manually, without prompt , or compulsory quality checks