CS3300 Introduction to Software Engineering

Lecture 18: Test Driven Development; Software Refactoring

Dr. Nimisha Roy ▸ nroy9@gatech.edu

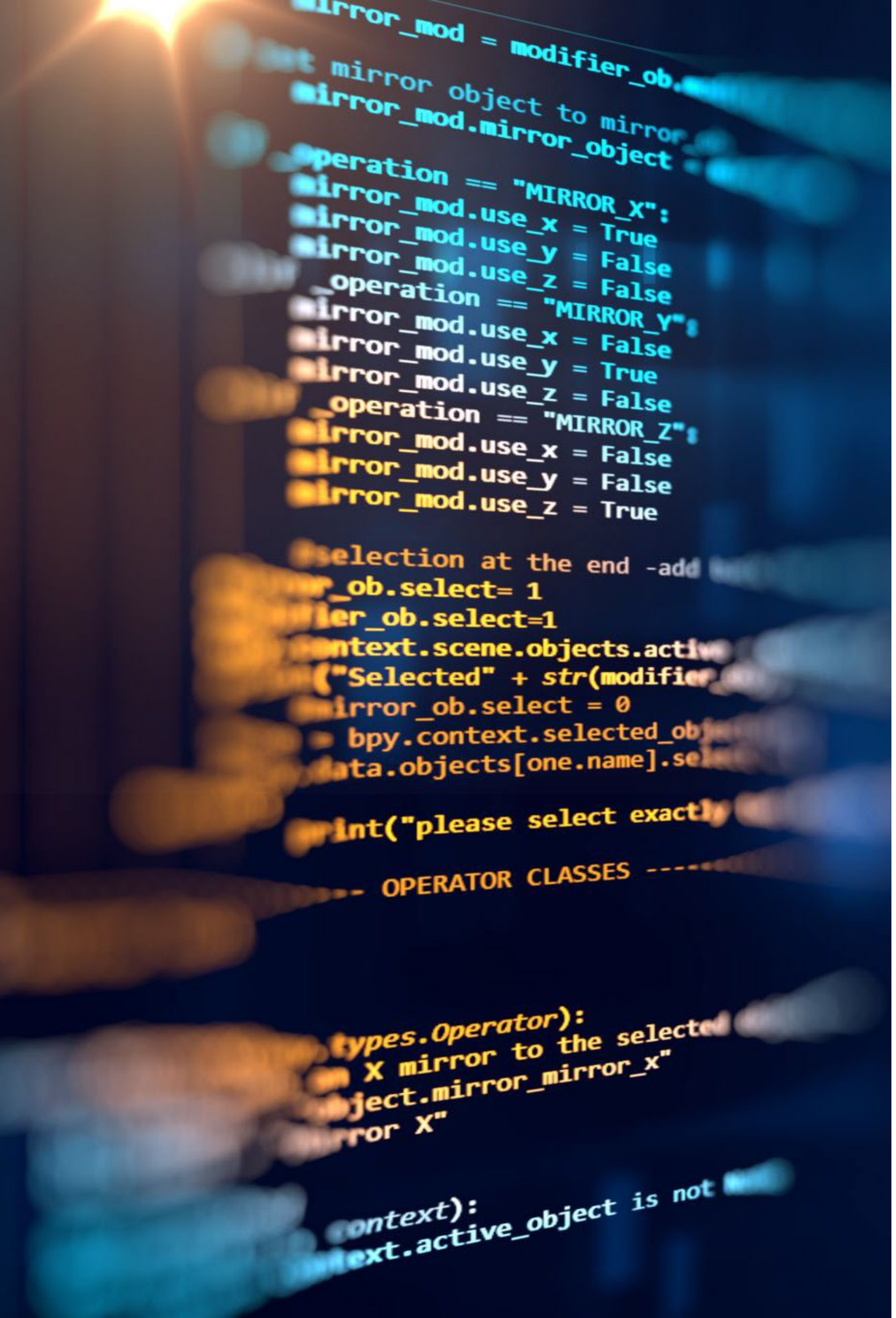
Slides adapted from Alessandro Orso

Transition from Waterfall to Agile has made testing easier and more approachable

From Waterfall….

… To Agile

- Blackbox testing allows test cases to be built before implementation
- Agile (XP specifically) introduces Test Driven Development as a solution to more testing confidence and motivation



What is Test Driven Development (TDD)

- Test is written ***before*** the class to be tested, and the developer writes unit testing code for nearly *all* production code.
- Write test code
    - Code that fulfills requirements
- Write functional code
    - Working code that fulfills requirements
- Refactor
    - Clean working code that fulfills requirements

TDD Basics – Unit Testing

Red, Green, Refactor

1

Make it Fail

- No code without a failing test

2

Make it Work

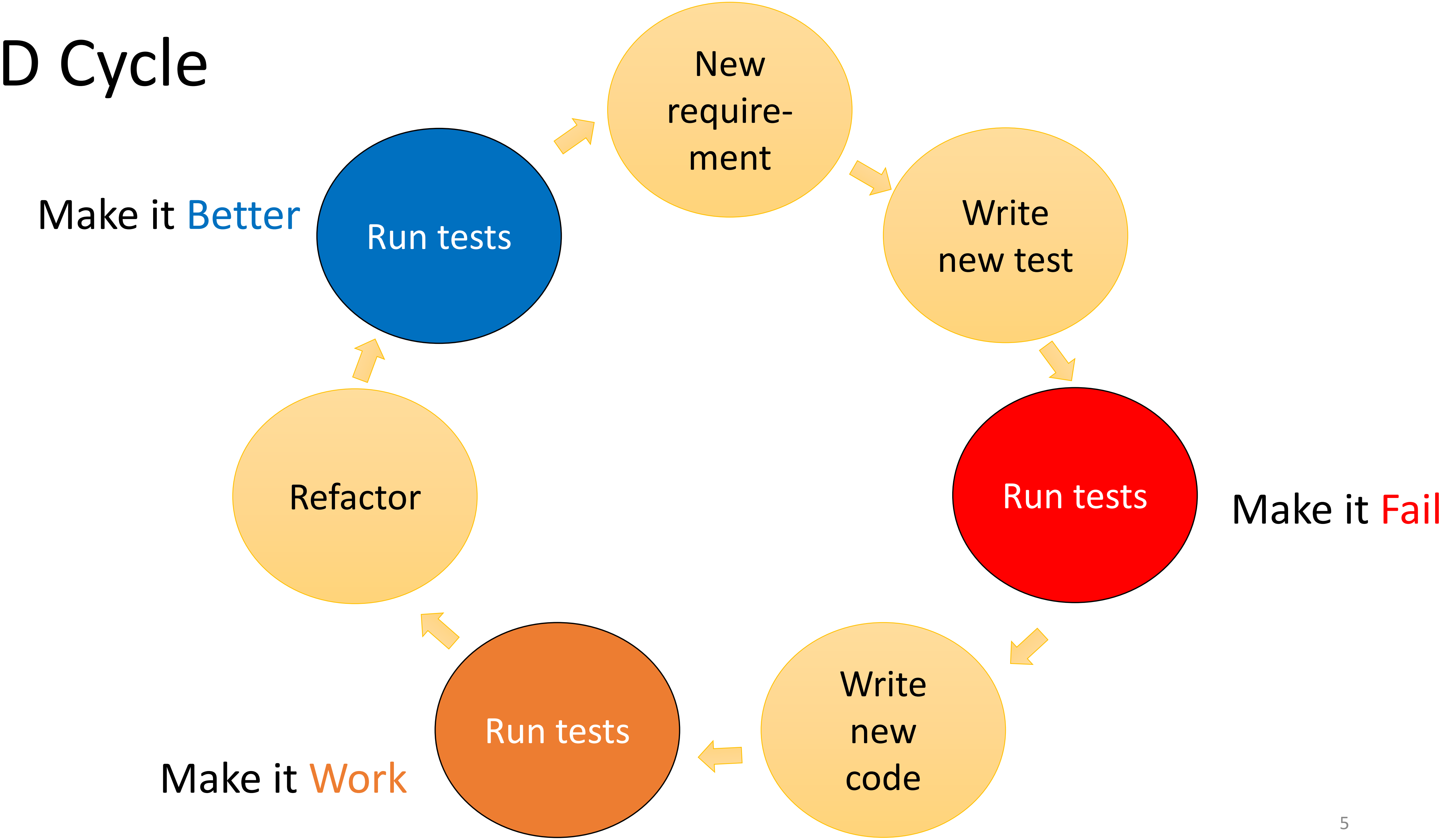
- As simply as possible

3

Make it Better

- Refactor

TDD Cycle


New require-ment
Write new test
Run tests
Make it Fail
Write new code
Run tests
Make it Work
Refactor
Run tests
Make it Better

Why TDD?

- Imposes developers’ discipline
- Provides incremental specification
- Avoid regression errors
- Allows for changing with confidence

TDD Example

Consider writing a program to score the game of bowling

You might start with the following test

```
public class TestGame extends TestCase {
        public void testOneThrow() {
        Game g = new Game();
        g.addThrow(5);
        assertEquals(5, g.getScore());
        }
}
```

When you compile this program, the test “fails” because..

the Game class does not yet exist.

But:

You have defined two methods on the class that you want to use

https://home.cs.colorado.edu/~kena/classes/5828/f16/lectures/22-tdd.pdf

TDD Example

Now you would write the Game class

```
public class Game {
        public void addThrow(int pins) {
        }
        public int getScore() {
                return 0;
        }
}
```

The **code now compiles but the test will still fail**: getScore() returns 0 not 5

- In Test-Driven Design, we take small, simple steps

- So, we get the test case to compile before we get it to pass

TDD Example

Once we confirm that the test still fails, we would then write the simplest code to make the test case pass; that would be

```
public class Game {
        public void addThrow(int pins) {
        }
        public int getScore() {
                return 5;
        }
}
```

The test case now passes

But this test case is not very helpful

TDD Example

Let's add a new test case to enable progress

```
public class TestGame extends TestCase {
        public void testOneThrow() {
                Game g = new Game();
                g.addThrow(5);
                assertEquals(5, g.getScore());
        }
...
```

```
...
    public void testTwoThrows() {
        Game g = new Game();
        g.addThrow(5);
        g.addThrow(4);
        assertEquals(9, g.getScore());
    }
}
```

The first test passes, but the second case fails (since 9 ≠ 5)

TDD Example

- We have duplication of information between the first test and the Game Class
  - In particular, the number 5 appears in both places
- This duplication occurred because we were writing the simplest code to make the test pass
  - Now, in the presence of the second test case, this duplication does more harm than good
- So, we must now **refactor** the code to remove this duplication

TDD Example

```
public class Game {
        private int score = 0;
        public void addThrow(int pins) {
                score += pins;
        }
        public int getScore() {
                return score;
        }
}
```

Both tests pass now.

Progress!

TDD Example

But now, to make additional progress, we add another test case to the TestGame class

```
public void testSimpleSpare() {
        Game g = new Game()
        g.addThrow(3); g.addThrow(7); g.addThrow(3);
        assertEquals(13, g.scoreForFrame(1));
        assertEquals(16, g.getScore());
}
```

We’re back to the code not compiling due to scoreForFrame()

- We’ll need to add a method body for this method and give it the simplest implementation that will make all three of our tests cases pass

Refactoring

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

Slide adapted from Alessandro Orso

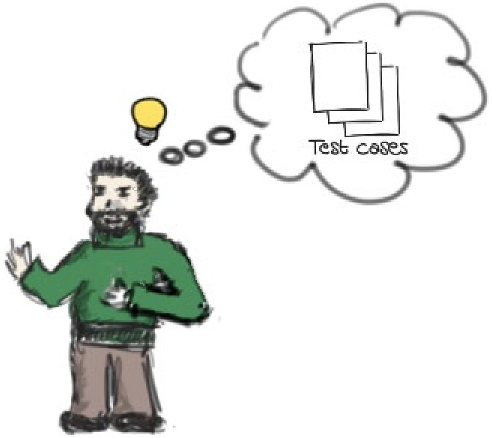
Behavior Preserving

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!



Slide adapted from Alessandro Orso

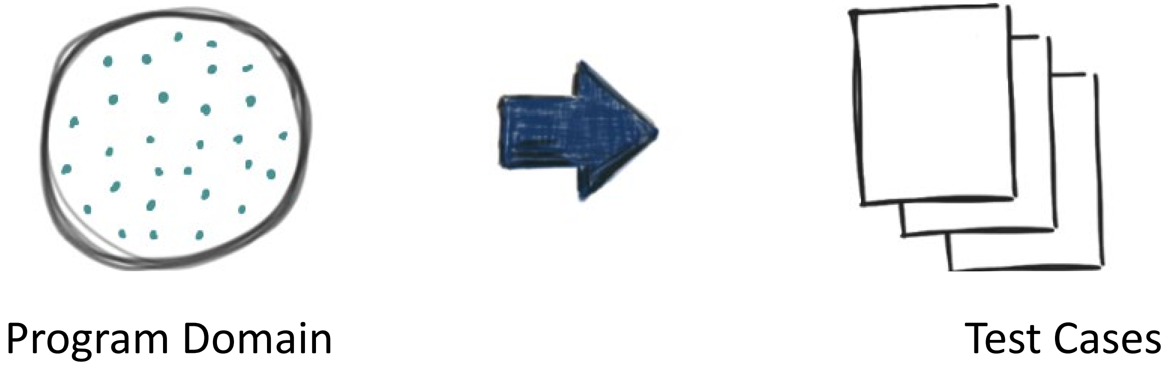
Behavior Preserving Quiz

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



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

Slide adapted from Alessandro Orso

Have you used Refactoring Before?

Class foo
Class bar

Even renaming a class is a refactoring!

(albeit a trivial one)

Many Refactorings in Fowler’s Book

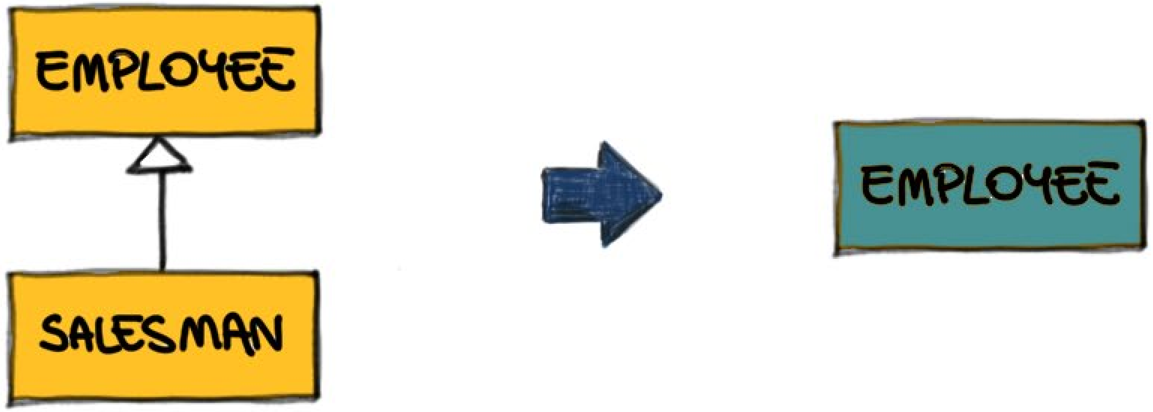
- Add parameter
- Change Association
- Reference to Value
- Value to Reference
- Collapse Hierarchy
- Consolidate Conditionals
- Procedures to Objects
- Decompose Conditionals
- Encapsulate Collection
- Encapsulate Downcast
- Encapsulate Field
- Extract Method
- Extract Class
- Inline Class
- Form Template 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

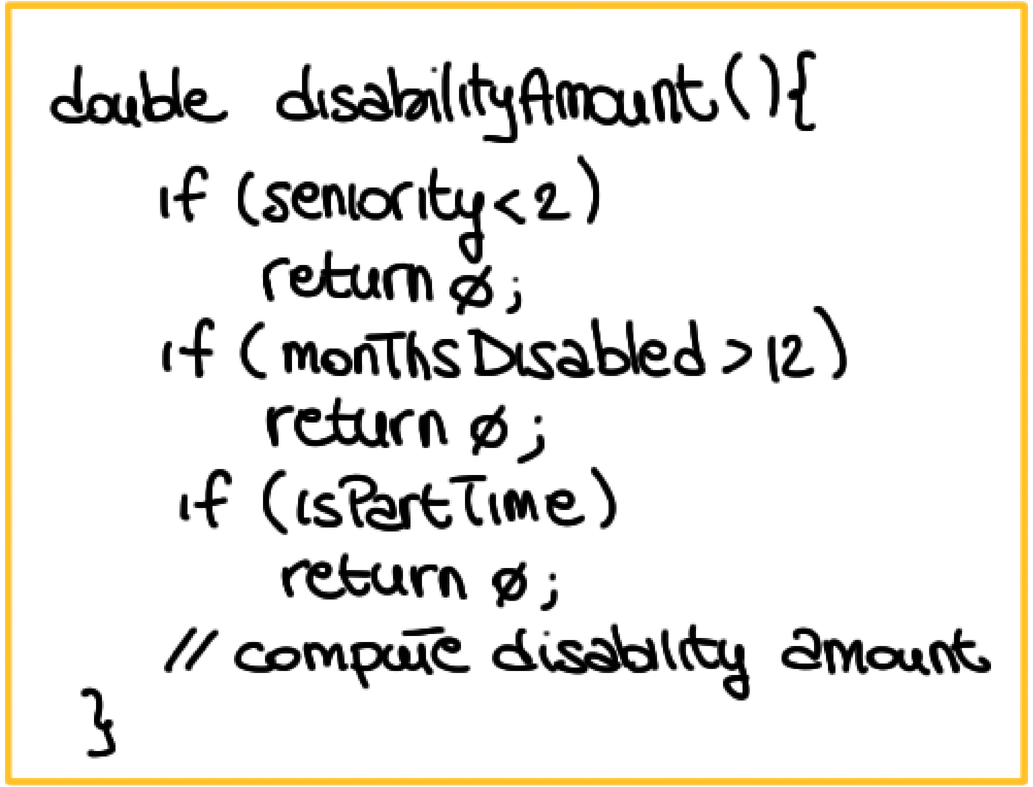
- Encapsulate Downcast
- Encapsulate Field
- Extract Method
- Extract Class
- Inline Class
- Form Template Method
- Hide delegate
- Hide method
- Inline temp

…

Consolidate Conditional Expression

If there are a set of conditionals with the same results

=> Combine and extract them



```
double disabilityAmount(){
    if (seniority<2)
        return ø;
    if (monthsDisabled>12)
        return ø;
    if (isPartTime)
        return ø;
    // 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

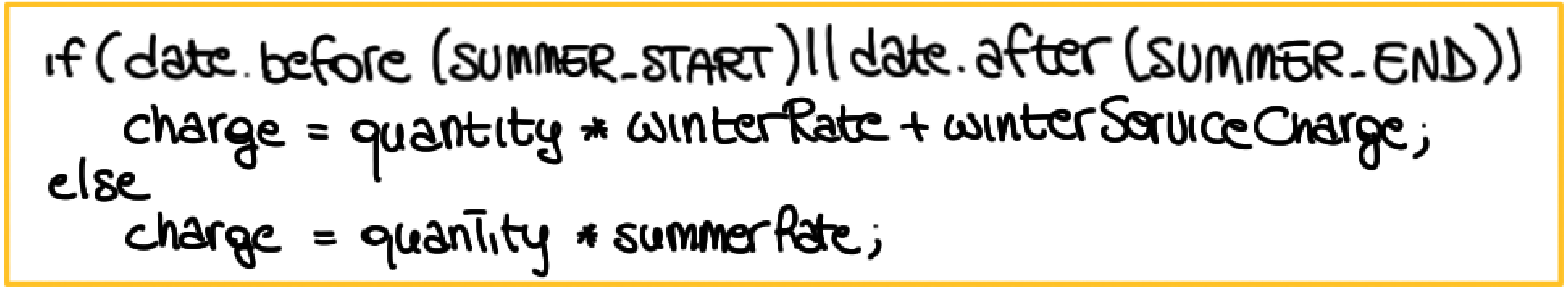
…

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;
```

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

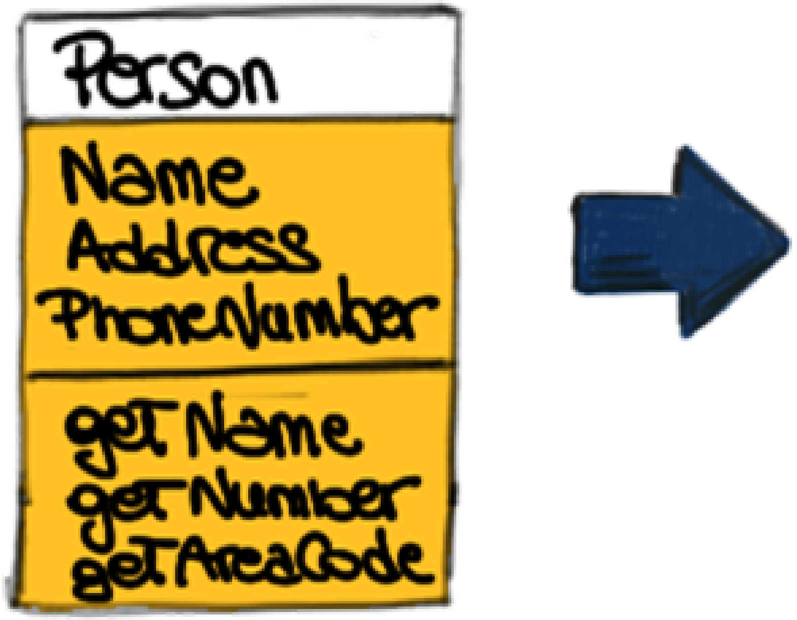
- Encapsulate Downcast
- Encapsulate Field
- Extract Method
- Extract Class
- Inline Class
- Form Template Method
- Hide delegate
- Hide method
- Inline temp

…

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

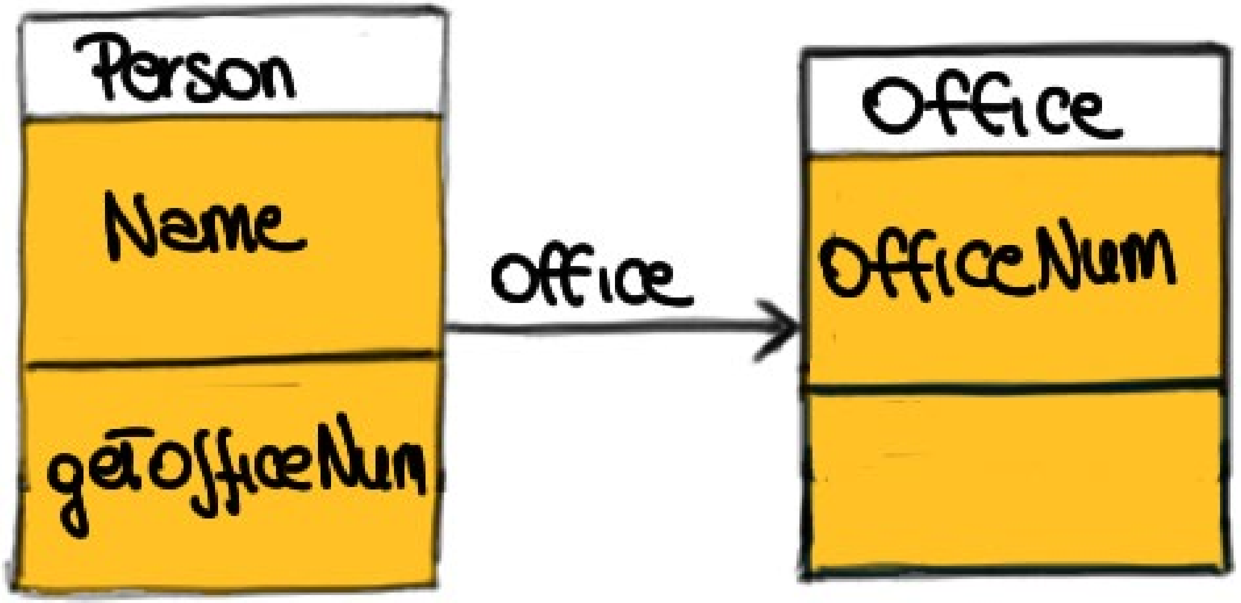
- Add parameter
- Change Association
- Reference to Value
- Value to Reference
- Collapse Hierarchy
- Consolidate Conditionals
- Procedures to Objects
- Decompose Conditionals
- Encapsulate Collection
- Encapsulate Downcast
- Encapsulate Field
- Extract Method
- Extract Class
- Inline Class
- Form Template 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

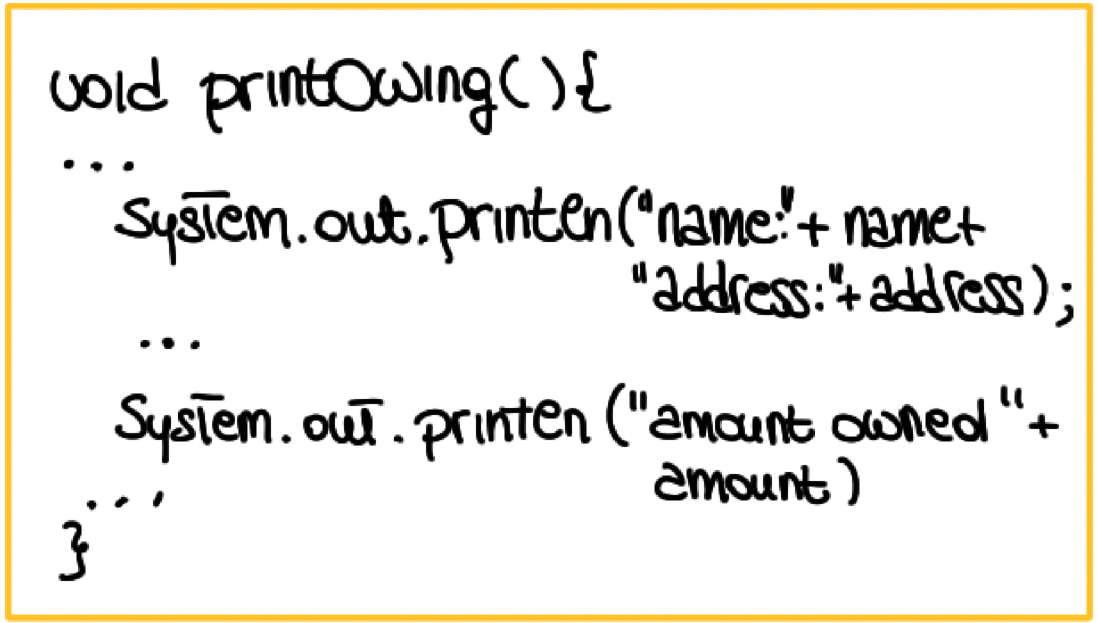
- Encapsulate Downcast
- Encapsulate Field
- Extract Method
- Extract Class
- Inline Class
- Form Template Method
- Hide delegate
- Hide method
- Inline temp

…

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.printen("name:"+ name+
                          "address:"+ address);
    ...
   System.out.printen ("amount owned"+
                          amount)
...
}
```

Refactoring in IDEs

Most IDEs have a set of built-in refactoring tools

The **Refactor** menu includes:

Rename class/method/variable

Change a method signature

Move a class to a new package

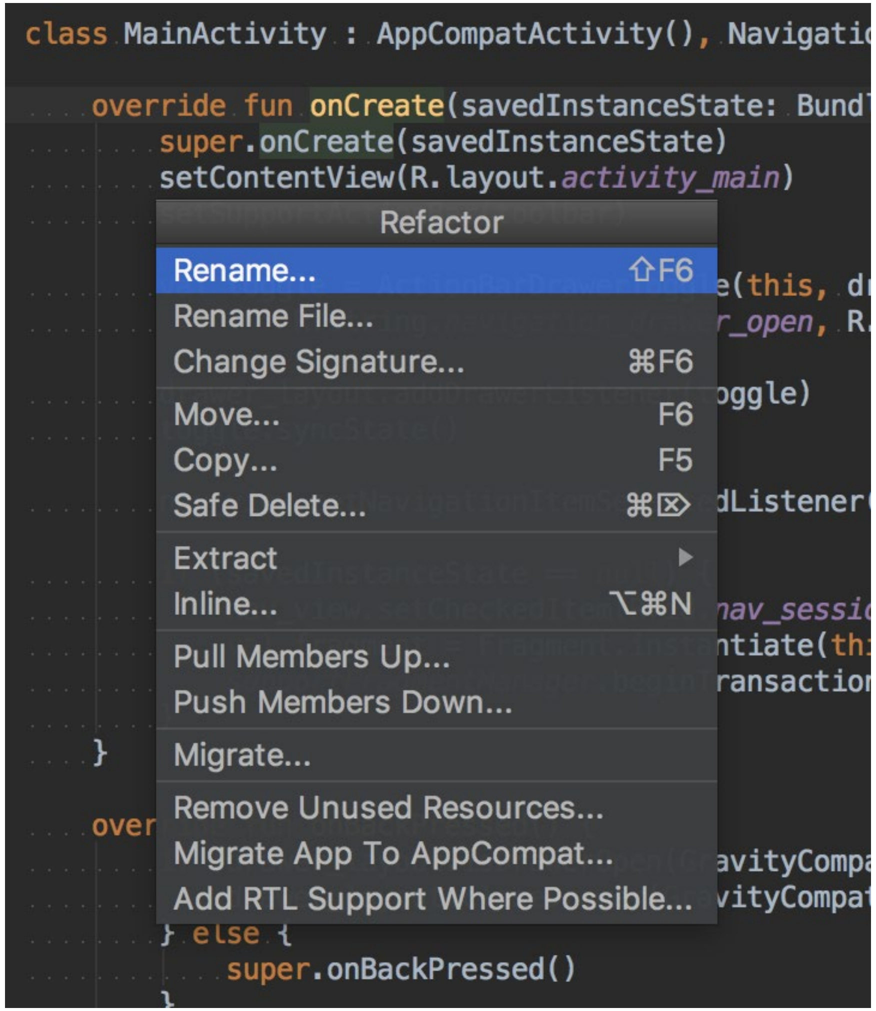
Extract a method or variable

Extract a method parameter

Create a new constant

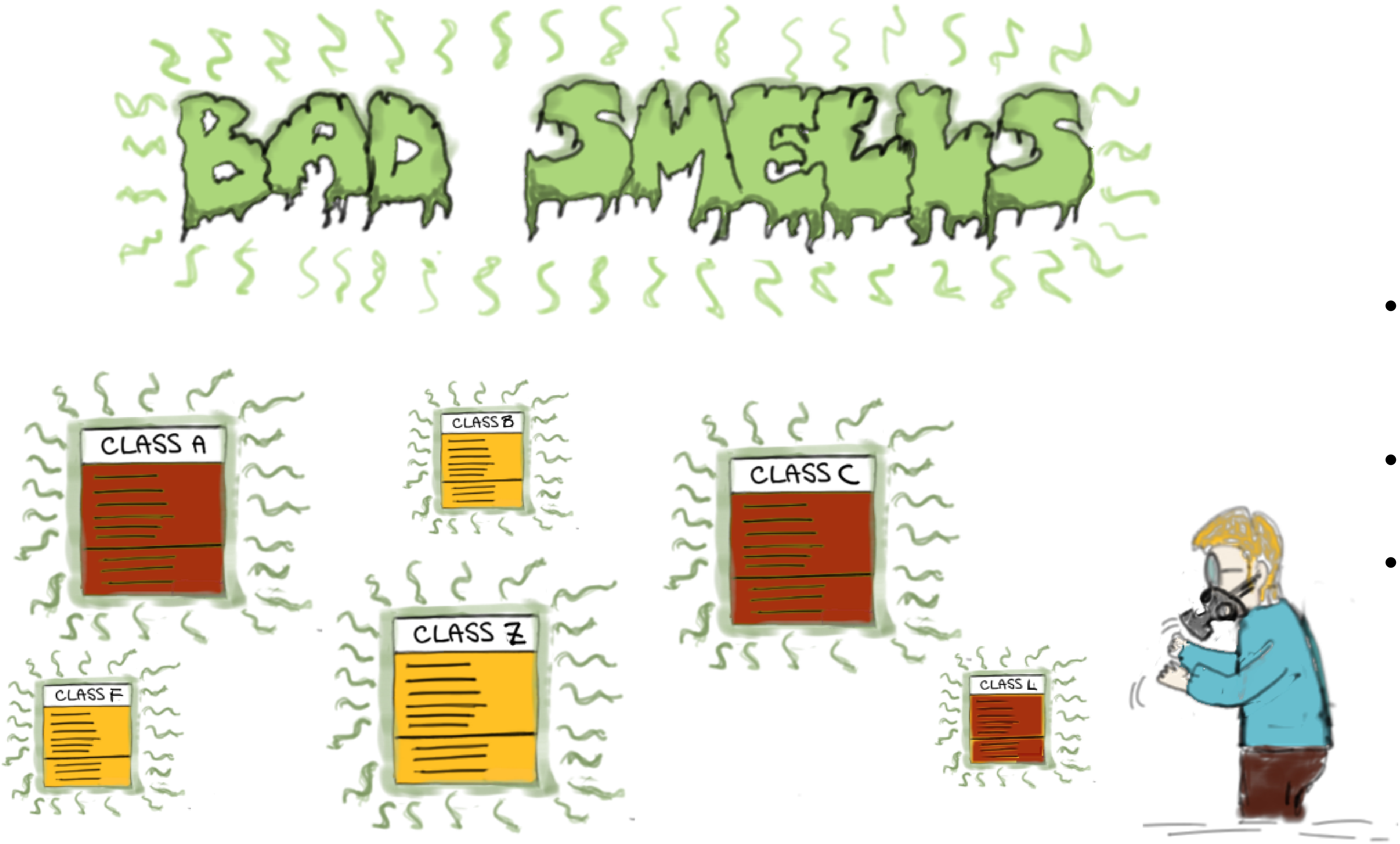
Inline a method

Safe delete



```
class MainActivity : AppCompatActivity(), Navigatio

    override fun onCreate(savedInstanceState: Bund
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
                                                  e(this, d
                                                  r_open, R.

                                                  oggle)

                                                  dListener(


                                                  nav_sessio
                                                  ntiate(th
                                                  ransactio
    }

    over
                                                  avityCompa
                                                  vityCompat
        } else {
            super.onBackPressed()
        }
```

https://medium.com/android-testing-daily/refactoring-28b8a4a07d42



- 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.

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

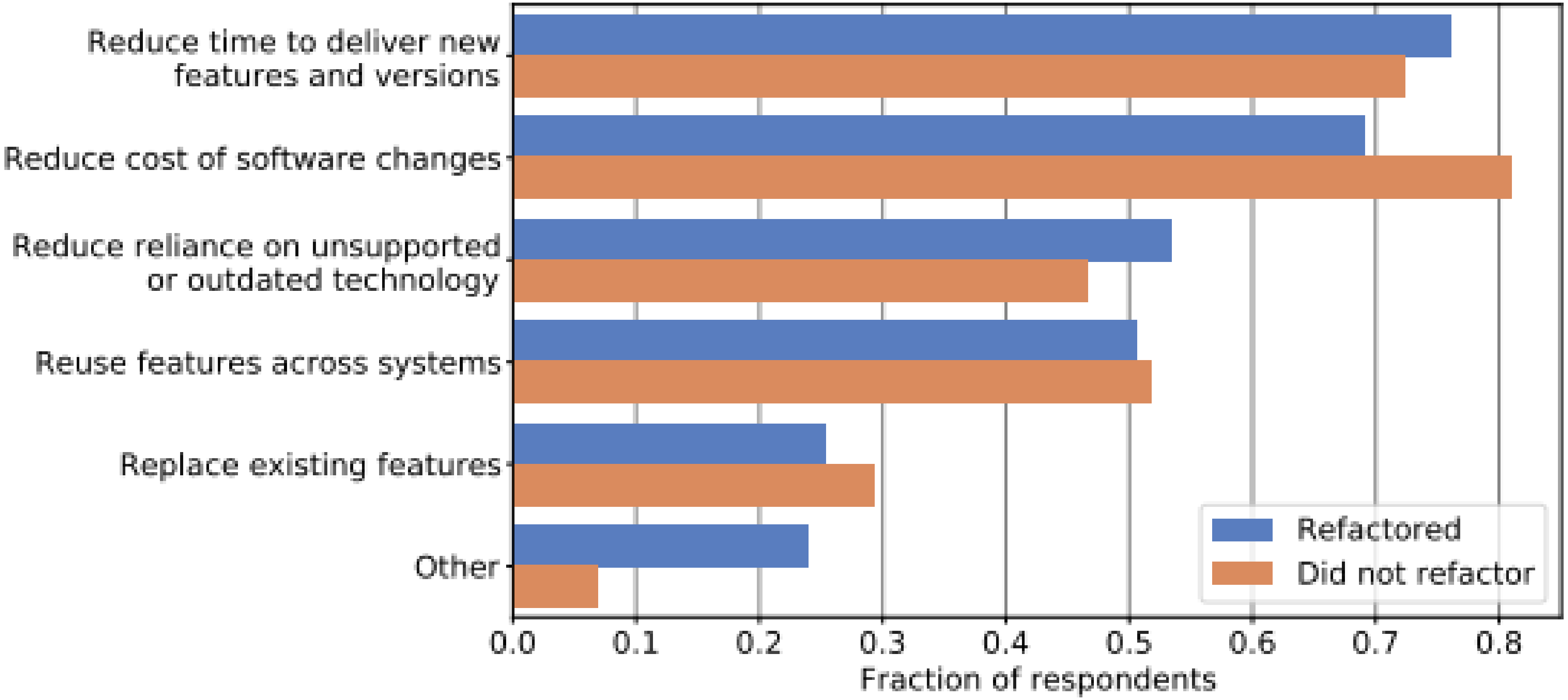


Figure 4: Business reasons for large-scale refactoring.

Reference Article

Refactoring Industry Standards – Industry Survey

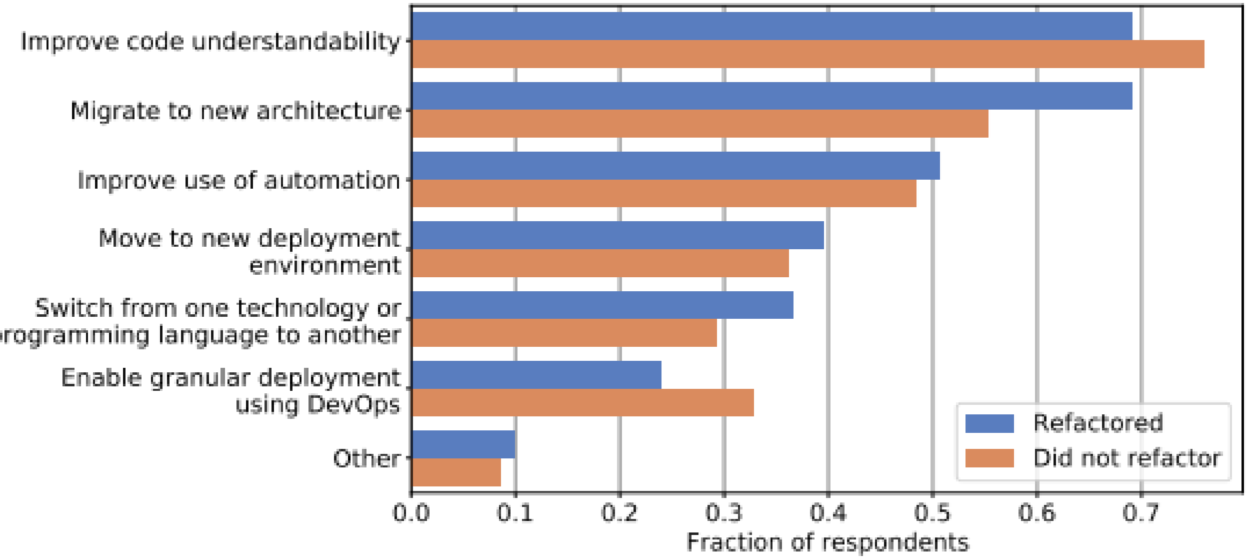


Figure 5: Technical reasons for large-scale refactoring.

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

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

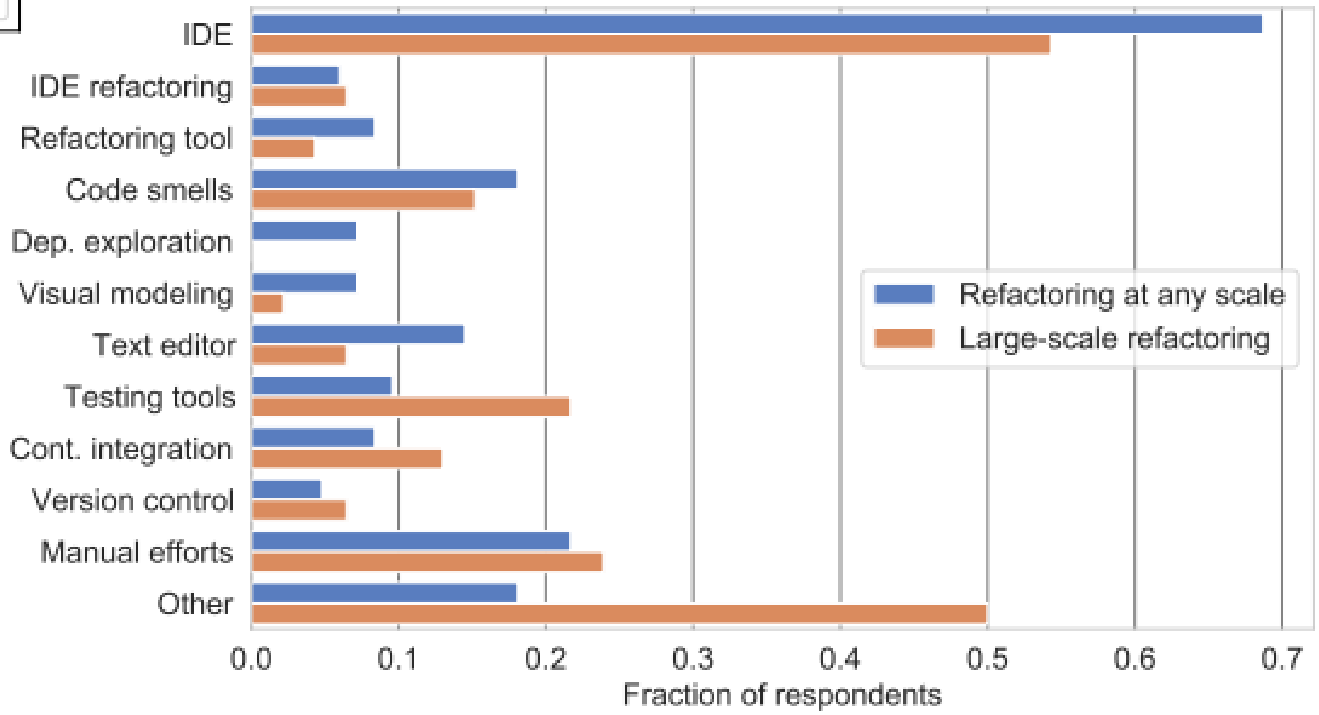


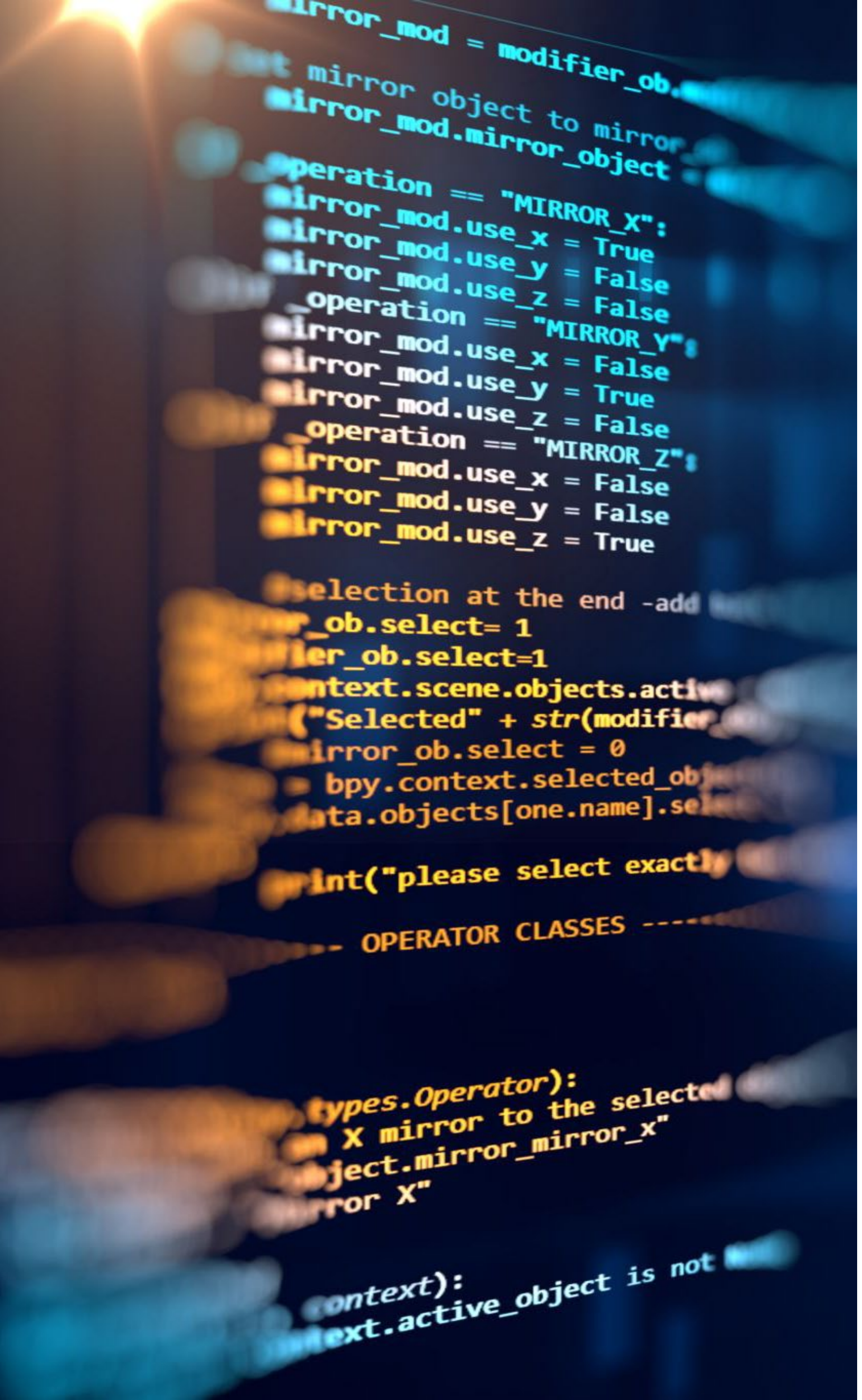
Figure 7: Categories of tools used to support refactoring.

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

Refactoring Industry Tools

- IDEs – IntelliJ/VS Code
- SonarQube
- SonarLint – free IDE plugin for real-time refactoring
- RefactorFirst – Java source code analyzer
- Rope – Python open source library
- Piranha – Open source tool to delete stale code
- Refraction – AI based refactoring.



When to refactor?

- When you find you have to **add a feature** to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature.
- During a **code review**: may be the last chance to tidy up the code before it become
- Every step of TDD

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!