

---

# CPSC 310 – Software Engineering

## Lecture 10 –

# Maintenance/Refactoring

---

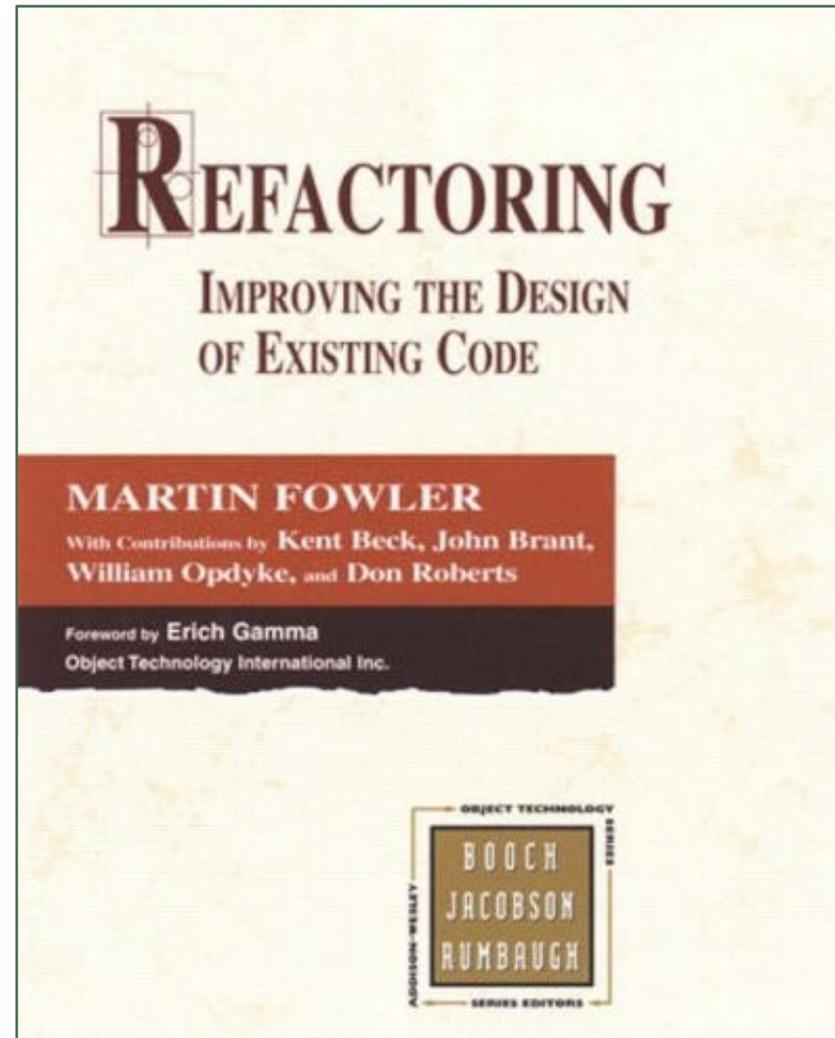
*Any fool can write code that a computer can understand. Good programmers write code that humans can understand.*

Martin Fowler

Refactoring: Improving the design of existing code

# Overview

- Introduction
- Why refactor?
- Refactoring: What, When, How?
- Drawbacks
- How to refactor



---

# Learning Goals

After this unit, you will be able to:

- Describe concrete reasons why code needs to be refactored
  - Explain the benefits of refactoring
  - Describe the textbook process you should follow when refactoring
  - Given code, be able to identify code smells and apply appropriate refactorings
-

---

# Your app relies on components



## The Angular.JS case

<https://jaxenter.com/angular-2-0-announcement-backfires-112127.html>

---

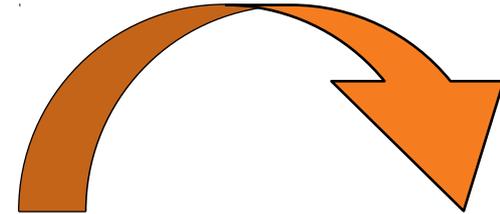
# Change in Software is a constant

```
public String printStatement() {
    double totalAmount = 0;
    Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        //determine amount for each movie
        switch (each.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2)
                    thisAmount += (each.getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented() * 3;
                break;
        }

        totalAmount += thisAmount;
    }

    return "Amount owed: " + totalAmount + "";
}
```



```

public String printStatement() {
    double totalAmount = 0;
    Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements())
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        //determine amount for each movie
        switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            thisAmount += 2;
            if (each.getDaysRented() > 2)
                thisAmount += (each.getDaysRented() - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:
            thisAmount += each.getDaysRented() * 3;
            break;
        }

        totalAmount += thisAmount;
    }

    return "Amount owed: " + totalAmount;
}

```

```

public String printStatement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";

    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        //determine amounts for each line
        switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            thisAmount += 2;
            if (each.getDaysRented() > 2)
                thisAmount += (each.getDaysRented() - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:
            thisAmount += each.getDaysRented() * 3;
            break;
        case Movie.CHILDRENS:
            thisAmount += 1.5;
            if (each.getDaysRented() > 3)
                thisAmount += (each.getDaysRented() - 3) * 1.5;
            break;
        }

        // add frequent renter points
        frequentRenterPoints++;

        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) && each.getDaysRented() > 1)
            frequentRenterPoints++;

        // show figures for this rental
        result += "\t" + each.getMovie().getTitle() + "\t" + String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;
    }

    // add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter points";

    return result;
}

```

```

public String printStatement() {
    double totalAmount = 0;
    Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements())
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        //determine amount for each movie
        switch (each.getMovie().getPriceCode())
        case Movie.REGULAR:
            thisAmount += 2;
            if (each.getDaysRented() > 7)
                thisAmount += (each.getDaysRented() - 7) * 2;
            break;
        case Movie.NEW_RELEASE:
            thisAmount += each.getDaysRented() * 3;
            break;
        }

        totalAmount += thisAmount;
    }

    return "Amount owed: " + totalAmount;
}

```

```

public String printStatement() {
    double totalAmount = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";

    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        //determine amounts for each line
        switch (each.getMovie().getPriceCode())
        case Movie.REGULAR:
            thisAmount += 2;
            if (each.getDaysRented() > 7)
                thisAmount += (each.getDaysRented() - 7) * 2;
            break;
        case Movie.NEW_RELEASE:
            thisAmount += each.getDaysRented() * 3;
            break;
        case Movie.CHILDRENS:
            thisAmount += 1.5;
            if (each.getDaysRented() > 3)
                thisAmount += (each.getDaysRented() - 3) * 1.5;
            break;
        }

        // add frequent renter points
        frequentRenterPoints++;

        if ((each.getMovie().getPriceCode() == Movie.REGULAR) ||
            (each.getMovie().getPriceCode() == Movie.CHILDRENS))
            // show figures for this rental
            result += "\t" + each.getMovie().getTitle() + "\t" + thisAmount + "\n";
        totalAmount += thisAmount;
    }

    // add footer lines
    result += "\nAmount owed is: " + String.valueOf(totalAmount) + "\n";
    result += "\nRenter has " + String.valueOf(frequentRenterPoints) + " frequent renter points\n";

    return result;
}

```

You might then add another method “printLongFormStatement” that reuses a lot of this code. And since you are in a hurry, you might just copy this method, and augment it.

*don't pretend you haven't done this*

---

# Yes But...

- No one would *really* introduce code duplication like that, would they?
  - So what's the problem?
-

---

# What's the problem?

- Facts:
    - “Nobody” writes code like that.
    - Your code is (probably) great.
  - But...
    - Others alter your code
    - You alter other people's code
    - This is good: collaboration = better product!
-

---

# What's the problem?

- Facts:
    - “Nobody” writes code like that.
    - Your code is (probably) great.
    - ***Code like that emerges after a collaborative effort!***
-

---

# What's the problem?

- Facts:
    - “Nobody” writes code like that.
    - Your code is (probably) great.
    - Code like that emerges after a collaborative effort!
  - But...
    - You won't get it right the first time around
    - Defects show up at all points of the lifecycle
    - Code is being constantly revisited
-

---

# What's the problem?

- Facts:
    - “Nobody” writes code like that.
    - Your code is (probably) great.
    - Code like that emerges after a collaborative effort!
    - ***Accumulated modifications lead to this code!***
-

---

# What's the problem?

- Facts:
    - “Nobody” writes code like that.
    - Your code is (probably) great.
    - Code like that emerges after a collaborative effort!
    - Accumulated modifications lead to this code!
  - But...
    - There is no such thing as a *perfect coder*
    - External conditions can affect the quality of your work
      - Approaching deadlines, stress, skunks...
-

# What's the problem?

- Facts:

- “Nobody” writes code like that.
- Your code is (probably) great.
- Code like that emerges after a collaborative effort!
- Accumulated modifications lead to this code!
- ***This code can appear in suboptimal conditions!***

```
q = ((p<=1) ? (p ? 0 : 1) : (p==-4) ? 2 : (p+1));
```

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

```
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);}
```

yes, even code like this!!

---

# What's the problem?

- Facts:
    - “Nobody” writes code like that.
    - Your code is (probably) great.
    - Code like that emerges after a collaborative effort!
    - Accumulated modifications lead to this code!
    - This code can appear in suboptimal conditions!
  - But...
    - Agility means very small upfront design
      - “The simplest thing that could possibly work!”
-

---

# What's the problem?

- Facts:
    - “Nobody” writes code like that.
    - Your code is (probably) great.
    - Code like that emerges after a collaborative effort!
    - Accumulated modifications lead to this code!
    - This code can appear in suboptimal conditions!
    - ***This code is expected in an agile process!***
      - *If it doesn't show up, then you're probably not agile...*
-

# Motivating Thought Experiment

Case: Imagine you've written a piece of code but then accidentally deleted and lost it.

Question:

- How much time would it take you to reconstruct from scratch what you had – the same amount, or more, or less?
- Would the code have a better design the second time you write it?

---

# Code Evolves

- Rule of thumb: It's harder to maintain (someone else's) code than it is to write new code.
  - Most developers hope that they won't have to deal with code maintenance
- Reality: Evolving/maintaining code is what most developers do most of the time.
- Advice: It pays off to keep code simple and easy to understand.

---

# You're not alone



*“Always code as if the person who ends up maintaining your code is a violent psychopath who knows where you live.”*

*John F. Woods*

---

# The problem is: CODE SMELLS!

- Facts:
  - “Nobody” writes code like that.
  - Your code is (probably) great.
  - Code like that emerges after a collaborative effort!
  - Accumulated modifications lead to this code!
  - This code can appear in suboptimal conditions!
  - ***This code is expected in an agile process!***
    - *If it doesn't show up, then you're probably not agile...*

Gradually, code begins to rot in places.

Those places are said to “smell”

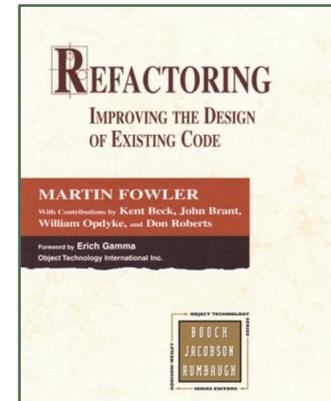
We, as designers/software developers, have to chase down these code smells and fix them.

# What is a Code Smell?

- A **recognizable** indicator that something **may** be wrong in the code
- Can occur in the **product code** as well as in the **test code!**

The smells/refactorings in the following slides are from Martin Fowler, Refactoring, “Improving the design of existing code”.

For test code smells: van Deursen et al. “Refactoring Test Code”.

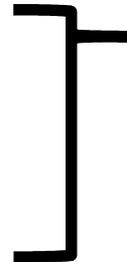


# Some common smells

- Magic Numbers
- Duplicated Code
- Long Method
- Complicated Conditionals
- Switch Statements
- Large class (doing the work of two)
- Shotgun Surgery
- Comments



within-class  
smells



between-class smells

[http://en.wikipedia.org/wiki/Code\\_smell](http://en.wikipedia.org/wiki/Code_smell)

# Magic Numbers?!

```
double potentialEnergy(double mass, double height) {  
    return mass * 9.81 * height;  
}
```

Any use of an actual number right  
in the code

# Duplicate code

```
extern int array1[];
extern int array2[];

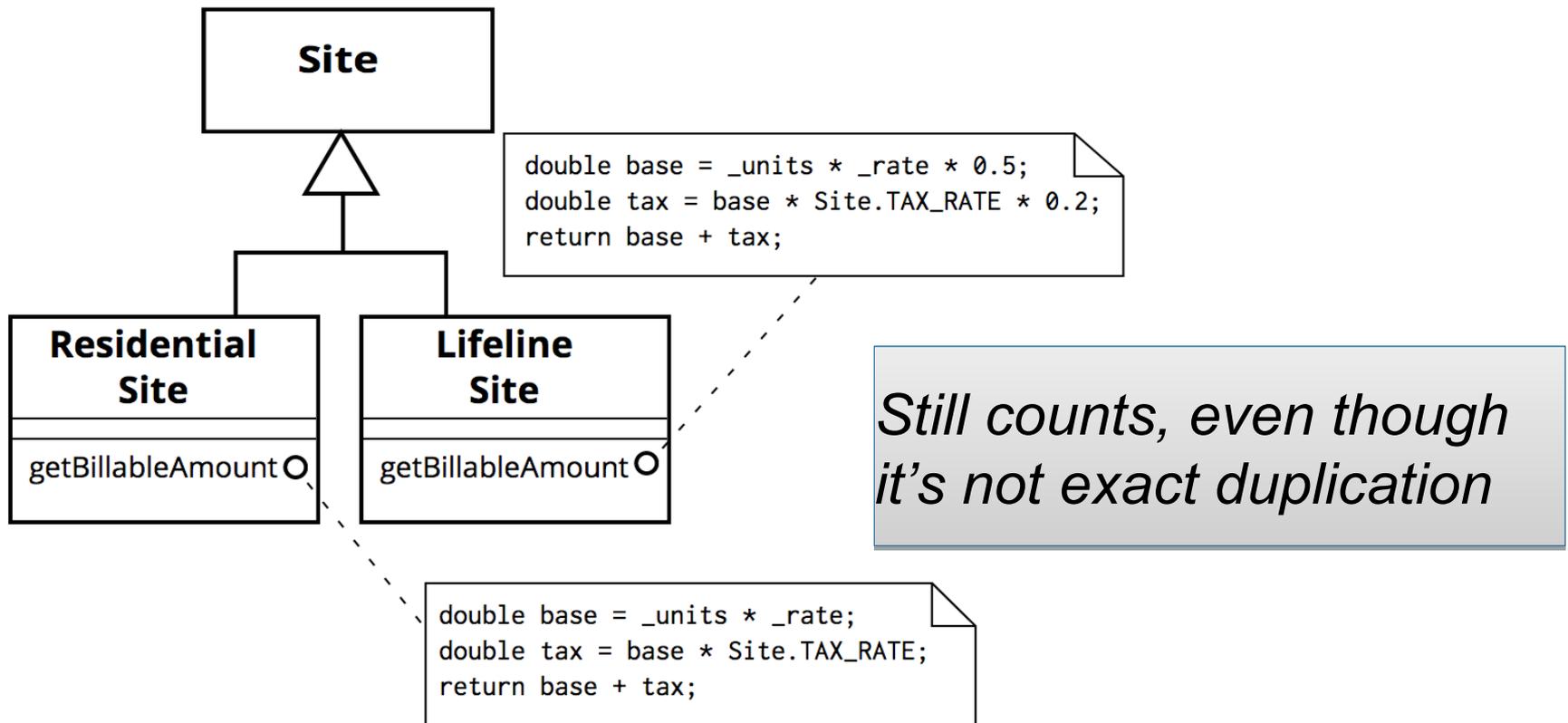
int sum1 = 0;
int sum2 = 0;
int average1 = 0;
int average2 = 0;

for (int i = 0; i < 4; i++)
{
    sum1 += array1[i];
}
average1 = sum1/4;

for (int i = 0; i < 4; i++)
{
    sum2 += array2[i];
}
average2 = sum2/4;
```

These two loops are the same!

# Sometimes the duplication is not as exact



# When is a method too long?

some red flags...



**Deeply nested control structures:** e.g. for-loops 3 levels deep or even just 2 levels deep with nested if-statements that have complex conditions.



**Too many state-defining parameters:** By state-defining parameter, I mean a function parameter that guarantees a particular execution path through the function. Get too many of these type of parameters and you have a combinatorial explosion of execution paths (this usually happens in tandem with #1).



**Logic that is duplicated in other methods:** poor code re-use is a huge contributor to monolithic procedural code. A lot of such logic duplication can be very subtle, but once re-factored, the end result can be a far more elegant design.



**Excessive inter-class coupling:** this lack of proper encapsulation results in functions being concerned with intimate characteristics of other classes, hence lengthening them.



**Unnecessary overhead:** Comments that point out the obvious, deeply nested classes, superfluous getters and setters for private nested class variables, and unusually long function/variable names can all create syntactic noise within related functions that will ultimately increase their length.



**Your massive developer-grade display isn't big enough to display it:** Actually, displays of today are big enough that a function that is anywhere close to its height is probably way too long. But, if it is larger, this is a smoking gun that something is wrong.



**You can't immediately determine the function's purpose:** Furthermore, once you actually do determine its purpose, if you can't summarize this purpose in a single sentence or happen to have a tremendous headache, this should be a clue.



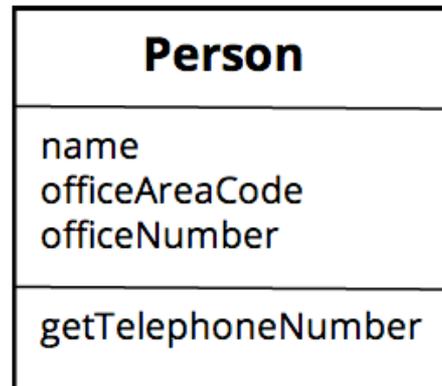
---

# When do you have a complicated conditional.?

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
  charge = quantity * _winterRate + _winterServiceCharge;
else charge = quantity * _summerRate;
```

# Large Class (One class is actually two)

*This reveals a failure of the single-responsibility principle.*



---

# What's shotgun surgery?

*A change that requires to alter many classes*

You whiff this when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes. When the changes are all over the place, they are hard to find, and it's easy to miss an important change.



# Is a comment really a smell?

no

... comments often are used as a deodorant. It's surprising how often you look at thickly commented code and notice that the comments are there because the code is bad.

*but do keep commenting!*

A good time to use a comment is when you don't know what to do. In addition to describing what is going on, comments can indicate areas in which you aren't sure. A comment is a good place to say why you did something. This kind of information helps future modifiers, especially forgetful ones.



---

<http://sourcemaking.com/refactoring/comments>

---

# How to Deal with a Smell?

- First, determine if it is a *bad smell*!
  - Some smells are worse than others so prioritize
- Then apply the appropriate refactoring(s)

<http://www.industriallogic.com/wp-content/uploads/2005/09/smellstorefactorings.pdf>

---

# Code Smells require Refactoring

- When a code smell is detected, you can rework the code to fix it.
  - In our code duplication example from earlier, what could we have done?
-

---

# What is Refactoring?

*“[Refactoring is] the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure” – Martin Fowler*

Changes made to a system that:

- ***Do not change observable behavior***
  - Remove duplication or needless complexity
  - Enhance software quality
  - Make the code easier and simpler to understand
  - Make the code more flexible
  - Make the code easier to change
- Requires Tests!
-

# What is Refactoring?

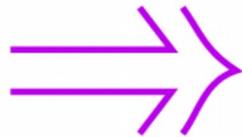
- At its simplest, it's just a ***small, behaviour-preserving, source-to-source*** transformation.
- Example:

```
extern int array1[];
extern int array2[];

int sum1 = 0;
int sum2 = 0;
int average1 = 0;
int average2 = 0;

for (int i = 0; i < 4; i++)
{
    sum1 += array1[i];
}
average1 = sum1/4;

for (int i = 0; i < 4; i++)
{
    sum2 += array2[i];
}
average2 = sum2/4;
```



```
int calcAverage (int* Array_of_4)
{
    int sum = 0;
    for (int i = 0; i < 4; i++)
    {
        sum += Array_of_4[i];
    }
    return sum/4;
}
```

---

# Why Refactor?

- Long-term investment in the quality of the code and its structure
- No refactoring may save costs / time in the short term but incurs a huge penalty in the long run
  - Technical Debt Metaphor

# Why fix it if it ain't broken?

Every module has three functions:

- To execute according to its purpose
- To afford change
- To communicate to its readers

If it does not do one or more of these, it *is* broken.

```
q = ((p<=1) ? (p ? 0 : 1) : (p== -4) ? 2 : (p+1));
```

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

```
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); }
```

---

# When to Refactor?

- **NOT:** 2 weeks every 6 months
- Do it as you develop - Opportunistic Refactoring
- Boy Scout principle: leave it better than you found it.
- If you recognize a warning sign (a *bad smell*)
  - When you add a function
    - Before, to start clean and/or
    - After, to clean-up
  - When you fix a bug
  - When you code review
  - You can use *The Rule of Three*

---

# The Rule of Three

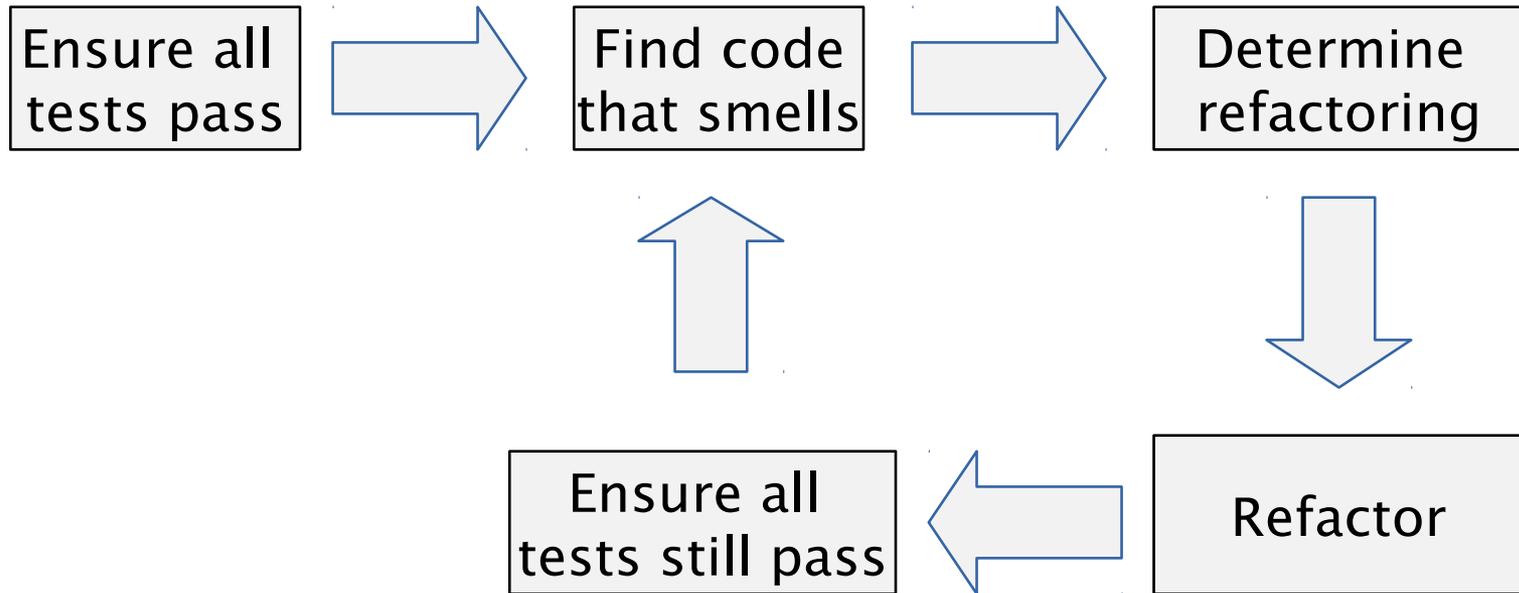
- The first time, just do it!
  - Need it somewhere else? Copy and paste it!
  - The third time, ***refactor!***  
*(if you remember/know that it has been duplicated before)*
  - Often associated with Extreme Programming
-

---

# When Not to Refactor?

- When the tests are failing
  - When you should just rewrite the code
  - When you have impending deadlines
-

# How to Refactor?



---

# One Smell – Multiple Refactorings

## Duplicated Code (Smell):

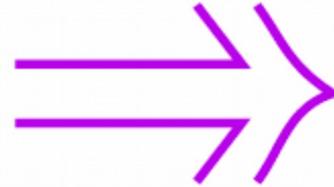
- Code repeated in multiple places
  - Multiple possible refactorings
    - Extract Method
    - Extract Class
    - Pull Up Method
    - Form Template Method
  - Choose appropriate one depending on context
-

# Refactoring: Extract Method Example

*Duplicated Code (Smell)*

```
void printOwing() {
    printBanner();

    //print details
    System.out.println ("name:  " + _name);
    System.out.println ("amount " + getOutstanding());
}
```



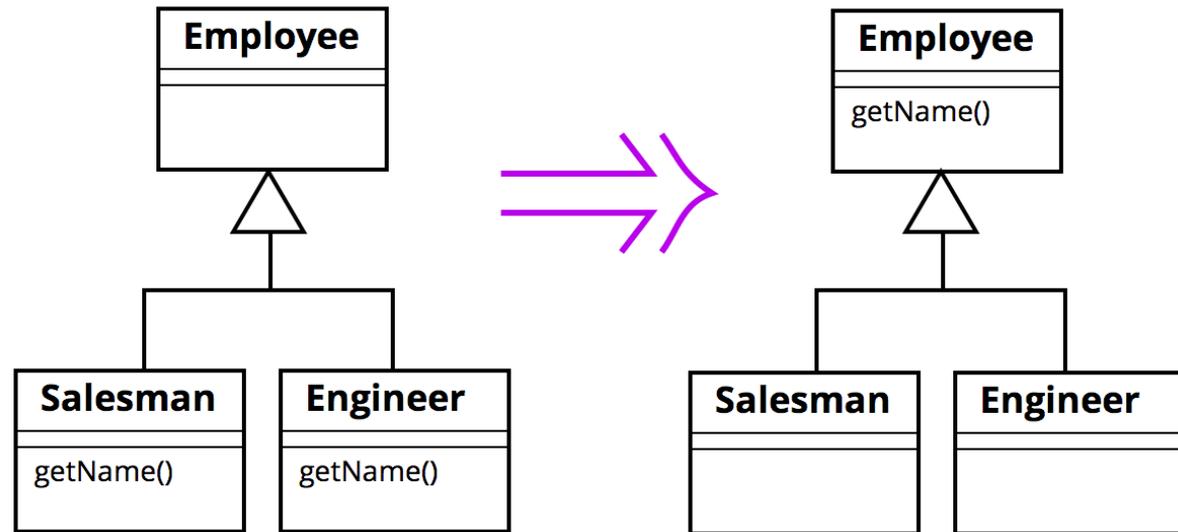
```
void printOwing() {
    printBanner();
    printDetails(getOutstanding());
}

void printDetails (double outstanding) {
    System.out.println ("name:  " + _name);
    System.out.println ("amount " + outstanding);
}
```

# Example Refactoring: Pull Up Method

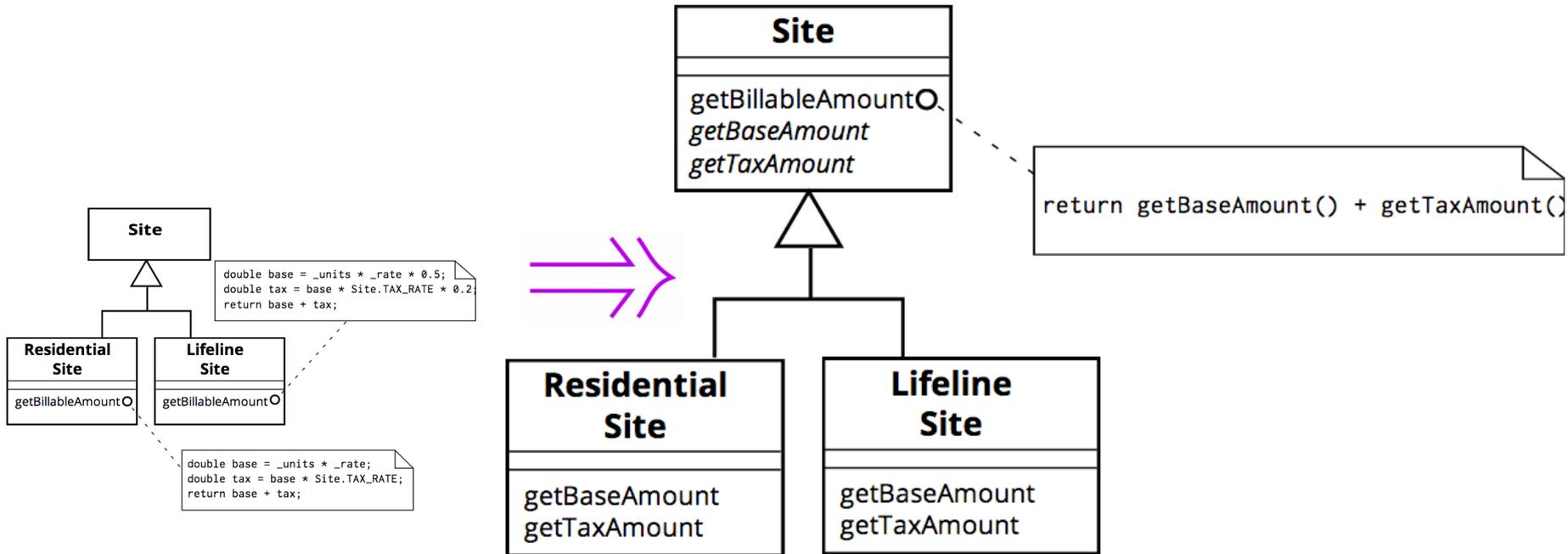
*Duplicated Code (Smell)*

Refactoring: Pull up method - If there are identical methods in more than one subclass, move it to the superclass



# Fixing Not Quite Duplicate Code

- Our early knotty code not quite duplication problem can be solved using refactoring.
- We can take that code, and transform it into a template method:



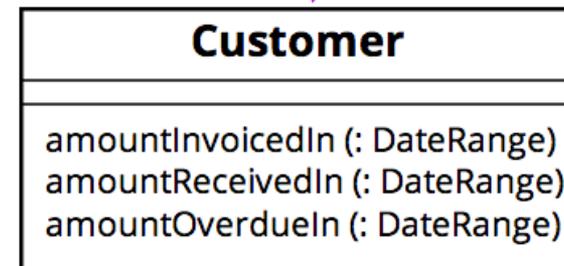
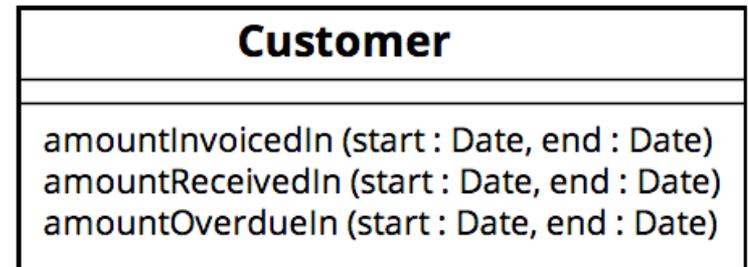
# Example Refactoring: Introduce Parameter Object

Smell: long parameter list /  
data clump

*constantly see the same  
few data items passed  
around together.*

Refactoring:

Introduce parameter  
object - If you have a group  
of parameters that naturally  
go together then you can  
replace them with an object.



---

# Smell: Long Method

- Methods with many statements, loops, or variables
  - Possible refactorings
    - Extract Method
    - Replace Temp with Query
    - Replace Method with Method Object
    - Decompose Conditional
    - Consolidate Conditional Expression
-

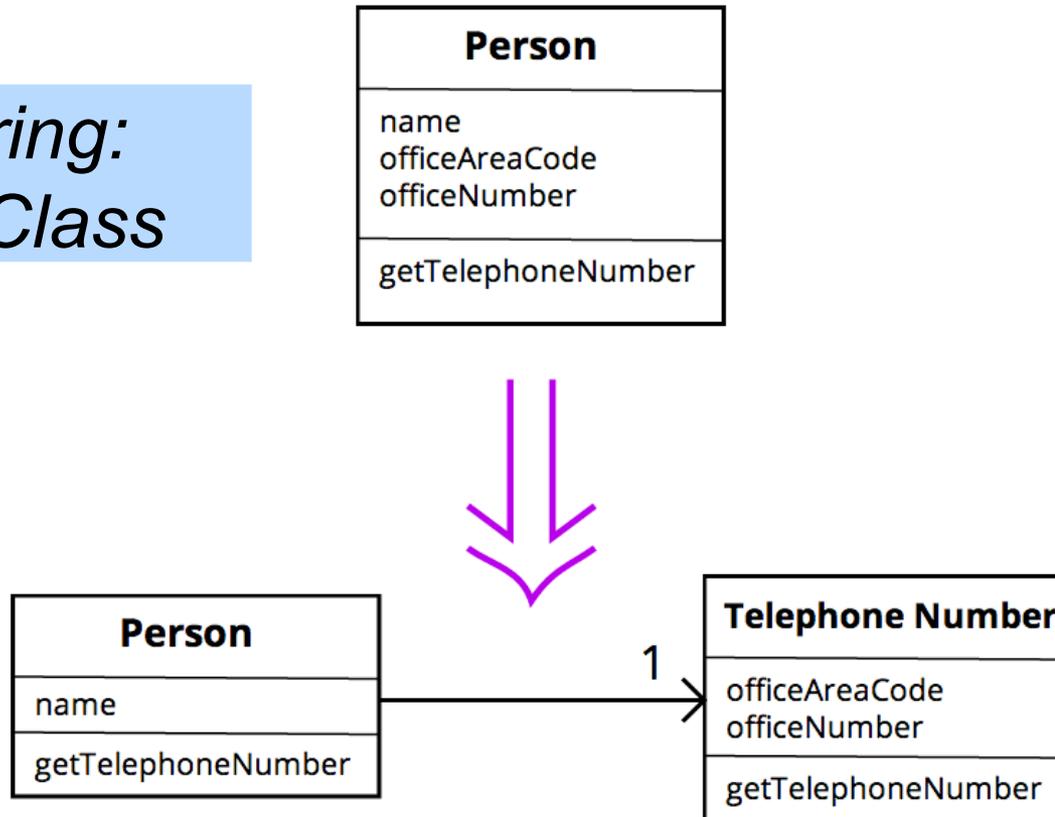
---

# Refactoring: Extract Method (again)

- Pull code out into a separate method when the original method is long or complex
  - Name the new method so as to make the original method clearer
  - Each method should have just one task
-

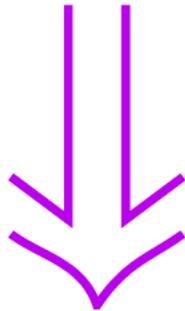
# Smell: One class doing the work of two

*Refactoring:  
Extract Class*



# Smell: Complicated Conditional

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))  
    charge = quantity * _winterRate + _winterServiceCharge;  
else charge = quantity * _summerRate;
```



*Refactoring:*  
***Decompose Conditional***

*extract methods from the condition, the “then” and the “else” parts.*

```
if (notSummer(date))  
    charge = winterCharge(quantity);  
else charge = summerCharge (quantity);
```

---

# How to refactor?

## Options

- Sloppy (manually)
- By the book (manually, but following a specific process)
- Automatic, using IDE support

Demos: <http://xp123.com/xplor/xp0605/index.shtml>

---

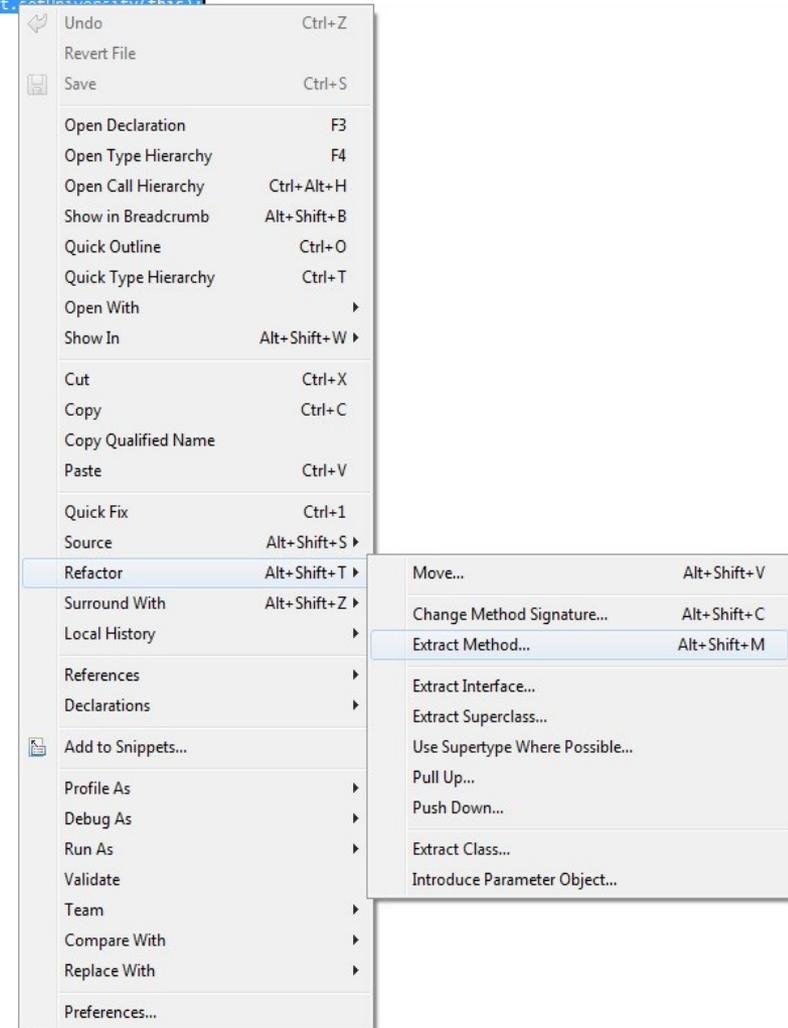
# How to refactor?

Using IDE support is the best option. You are least likely to make mistakes using this approach.

## Learn about Eclipse support:

[http://www.ibm.com/developerworks/opensource/library/os-eclipse-refactoring/?ca=dgr-lnxw97Refactoringdth-OS&S\\_TACT=105AGX59&S\\_CMP=grlnxw97](http://www.ibm.com/developerworks/opensource/library/os-eclipse-refactoring/?ca=dgr-lnxw97Refactoringdth-OS&S_TACT=105AGX59&S_CMP=grlnxw97)

```
public void addStudent(Student newStudent){  
    if(!students.contains(newStudent)){  
        students.add(newStudent);  
        newStudent.enrollInCourse(this);  
    }  
}
```



---

# Refactoring Truths

- Most of the time your intuition is good
- Doing it *by the book* is hard
  - Use IDE tools
- Unit tests are the key
  - Run Unit tests
  - Refactor
  - Run Unit tests



---

# Refactor Every Chance You Get

- Improve the design of existing code without changing functionality
    - Simplify code
    - Improve design
    - Remove duplicate code
  - The ability to refactor is your reward for spending time writing unit tests
-

---

# Remember!

- A potential for refactoring ***is not a smell***
    - Just because you see a potential for refactoring doesn't mean you should apply it. Only refactor if the code suffers from a code smell.
    - Some refactorings are opposites of one another (you could get caught in a loop of refactorings if you do them just for the sake of it! Inline versus Extract method, for instance.)
  - First smell, then refactor
-

---

# Refactoring Drawbacks

- When taken too far
    - Incessant tinkering with code
    - Trying to make it *perfect*
  - Attempting refactoring when the tests don't work – or without tests – can lead to dangerous situations!
  - Refactoring published interfaces propagates to external users relying on these interfaces
-

---

# Why Developers Fear Refactoring?

1. “I don’t understand the code enough to do it”
2. Short-term focus (Adding a new working feature is cooler!)
3. Not paid for overhead tasks such as refactoring?

- Solutions:

1. Test!
  2. Learn to appreciate beauty!
  3. Teach the benefits of better code!
-

---

# Resources

- “The” Book, by Martin Fowler
    - Refactoring: Improving the design of existing code
  - Code Smells
    - <http://sourcemaking.com/refactoring/bad-smells-in-code>
  - Refactorings List
    - <http://www.refactoring.com/catalog>
    - <http://sourcemaking.com/refactoring>
  - A refactoring “cheat sheet”
    - <http://industriallogic.com/papers/smellstorefactorings.pdf>
-

---

# Summary

- Code decays for many reasons
    - Collaboration, rework, external conditions, agility
  - Refactoring improves existing code
    - Does not change existing behaviour
  - Refactoring improves maintainability and hence productivity
  - Refactor continuously
  - Refactoring is an iterative process
    - Tests pass → Find smell → Refactor → Repeat
  - Many smells, even more refactorings!
-