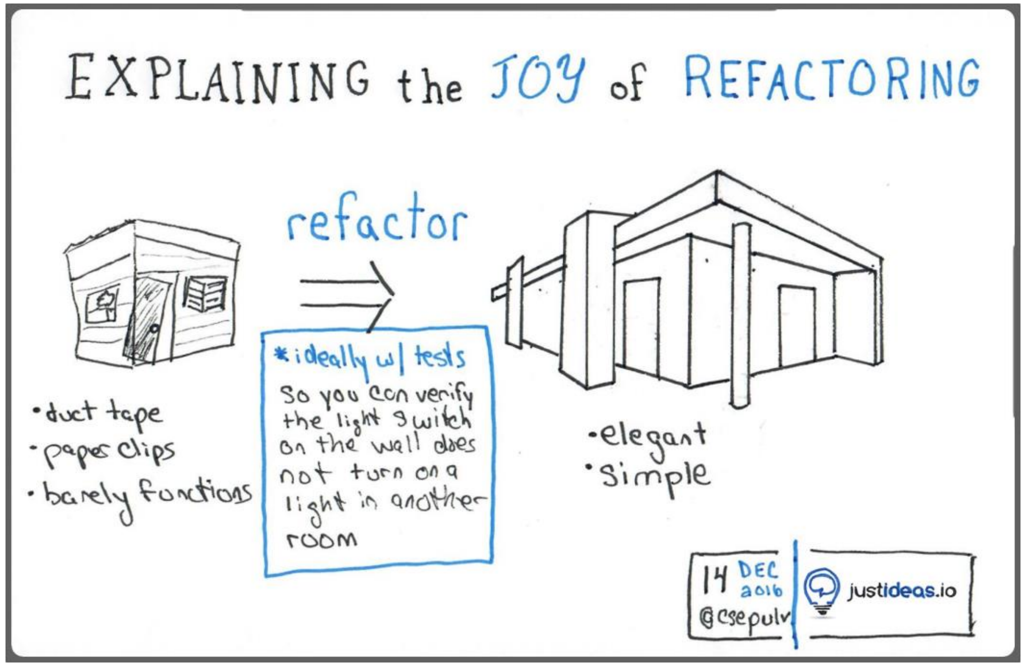
COMP2059 Developing Maintainable Software

LECTURE 03 – REFACTORING SKILLS

Boon Giin Lee (Bryan)

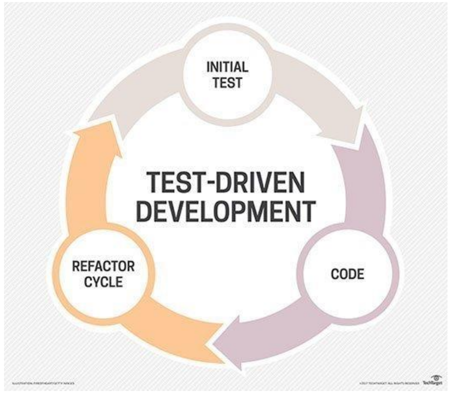
Today Learning Objectives

- To be able to assess when code needs refactoring by identifying Code Smells.
- To understand the concept of refactoring.
  - To refactor small sections of code to increase maintainability.
  - To refactor larger code structures for the application of SOLID/Design Patterns.
- To understand the links between regression testing, TDD, refactoring and legacy code.

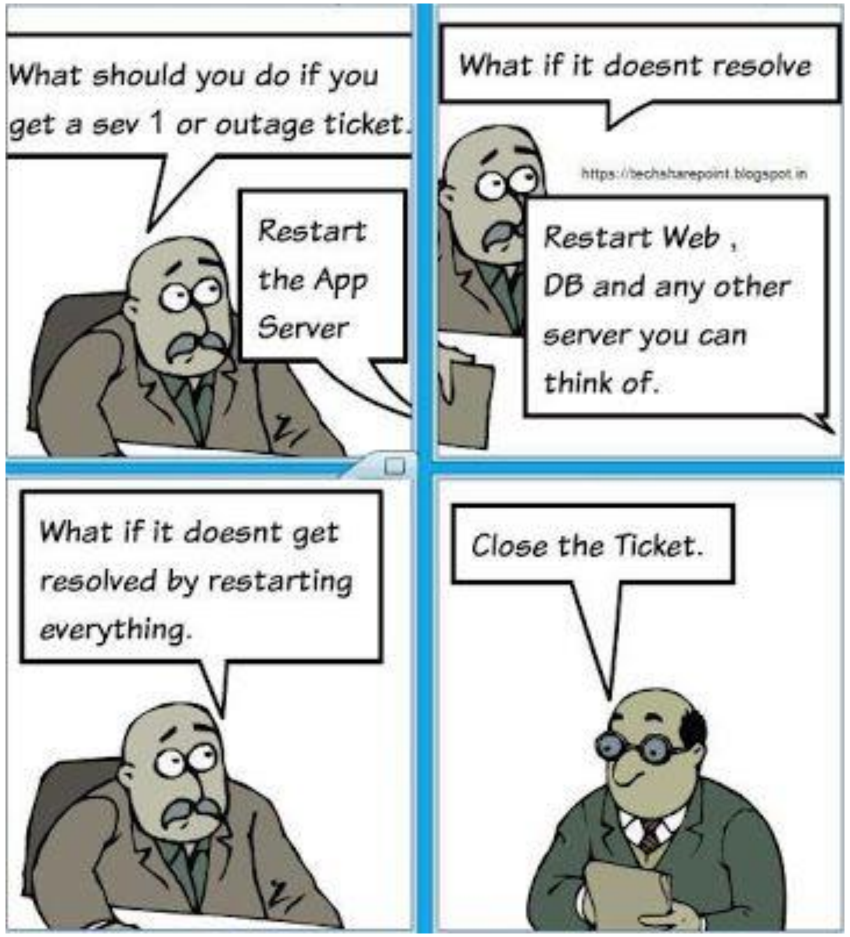

EXPLAINING the JOY of REFACTORING
refactor
*ideally w/ tests
So you can verify the light switch on the wall does not turn on a light in another room
•duct tape
•paper clips
•barely functions
•elegant
•simple
14 DEC 2016 @csepulv
justideas.io

https://medium.com/justideas-io/explaining-the-joy-of-refactoring-to-the-non-developer-72d97223359c

The Software Maintenance Starter Pack



GitHub JUnit Gradle 5


What should you do if you get a sev 1 or outage ticket.
Restart the App Server
What if it doesnt resolve
https://techsharepoint.blogspot.in
Restart Web , DB and any other server you can think of.
What if it doesnt get resolved by restarting everything.
Close the Ticket.

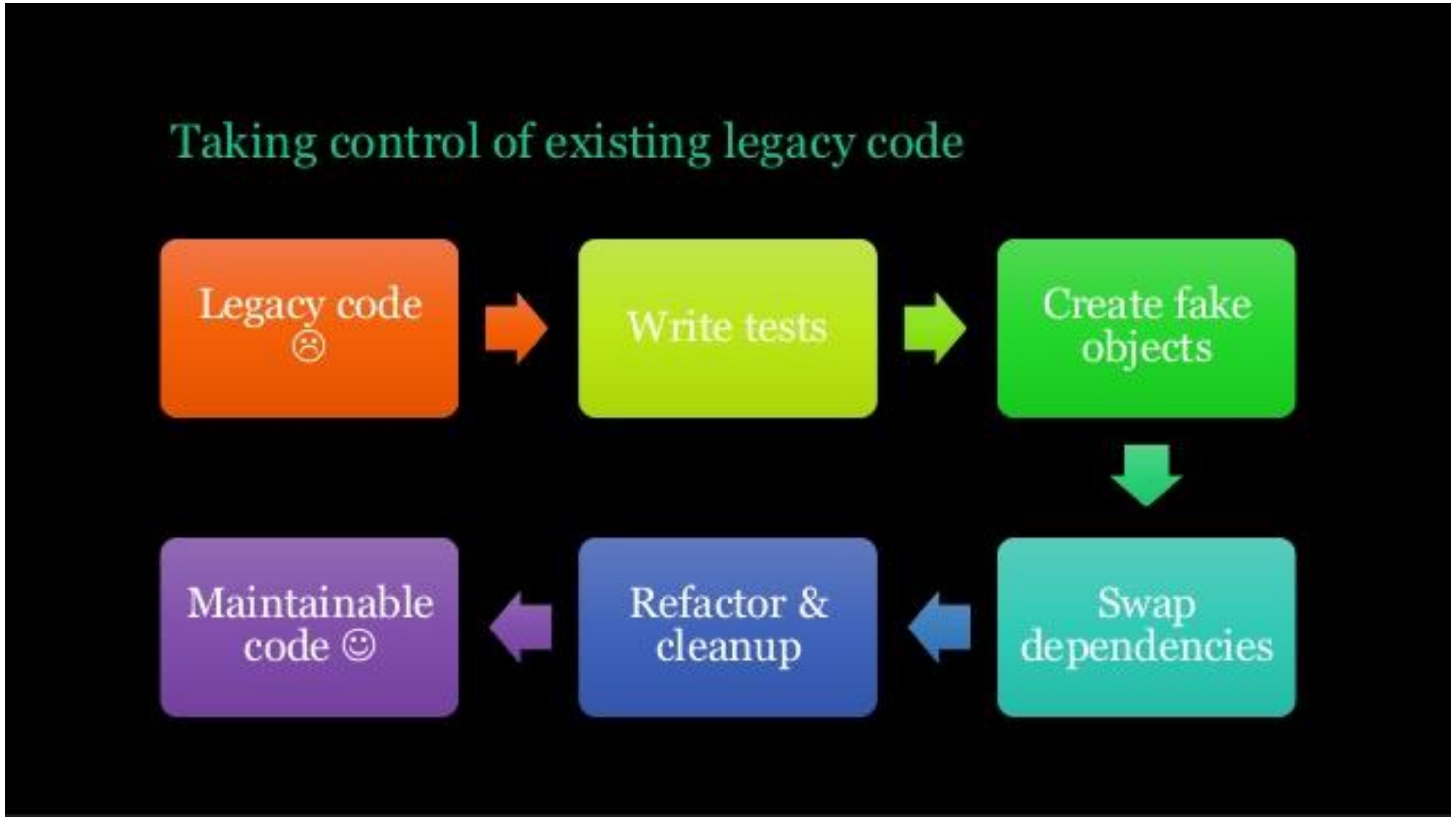
Legacy Code Maintenance
The Core of An SE Role

Java 2, XHTML... we can toss all these out
I think we should keep this one...
When is it from?
2004
Are you kidding? It's 15 years old! We can toss it!
I don't know, it might be one of the only books that gets better with age
I don't know...
WORKING EFFECTIVELY WITH LEGACY CODE
Michael C. Feathe
CommitStrip.com

LEGACY CODE
It Worked For 65 Million Years.
No One Knows How. No One Knows Why.
fb.com/wesoftwareengineers/

CARL QUIT. HE'S THE ONLY ONE WHO KNOWS HOW TO PROGRAM THE LEGACY SYSTEM.
scottadams@aol.com
www.dilbert.com
IT CAN'T BE THAT HARD. GO FIGURE IT OUT.
© 2006 Scott Adams, Inc./Dist. by UFS, Inc.
12-9-06
?
FRACK.

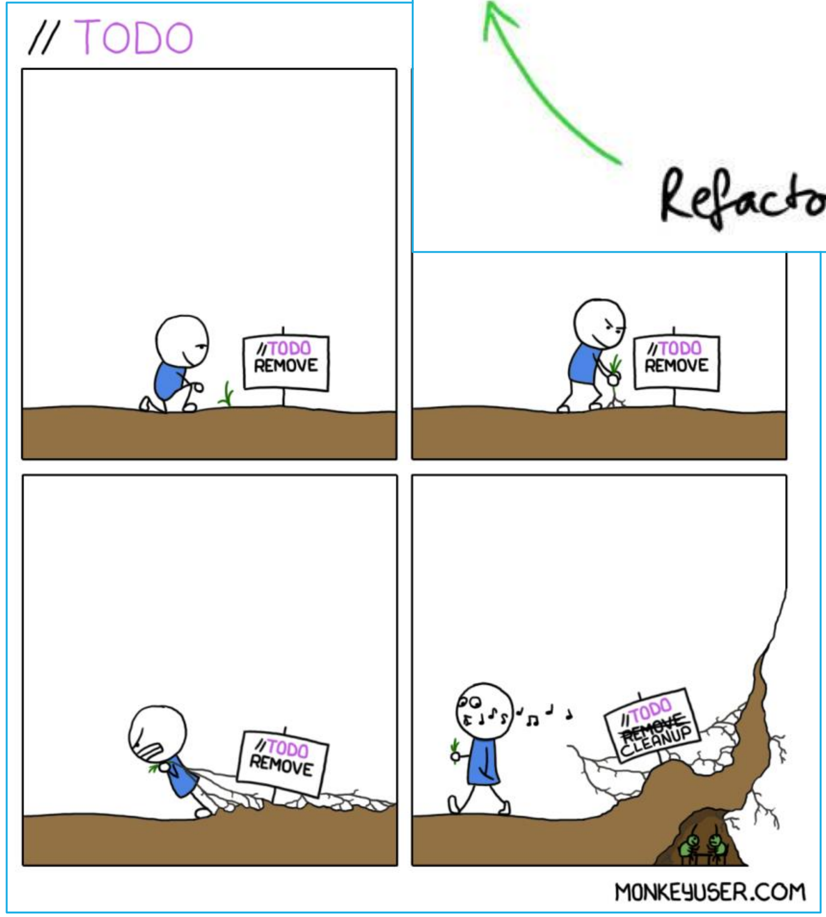
The Process of Working with Legacy Code

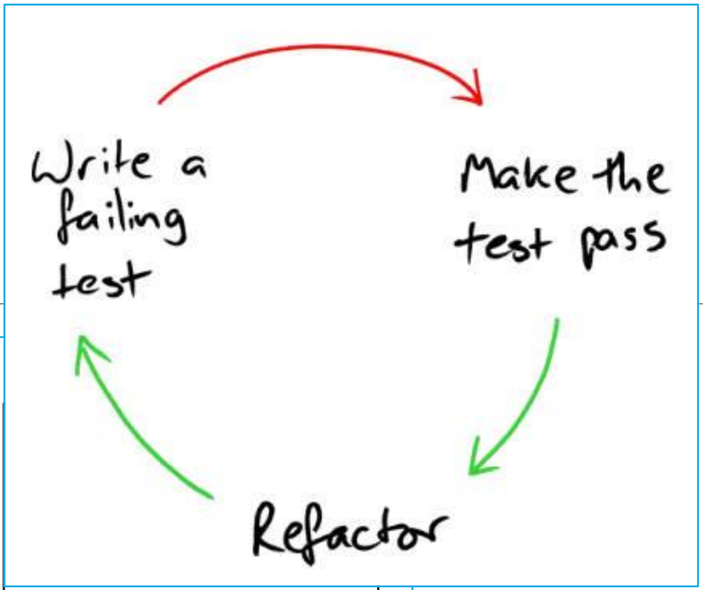


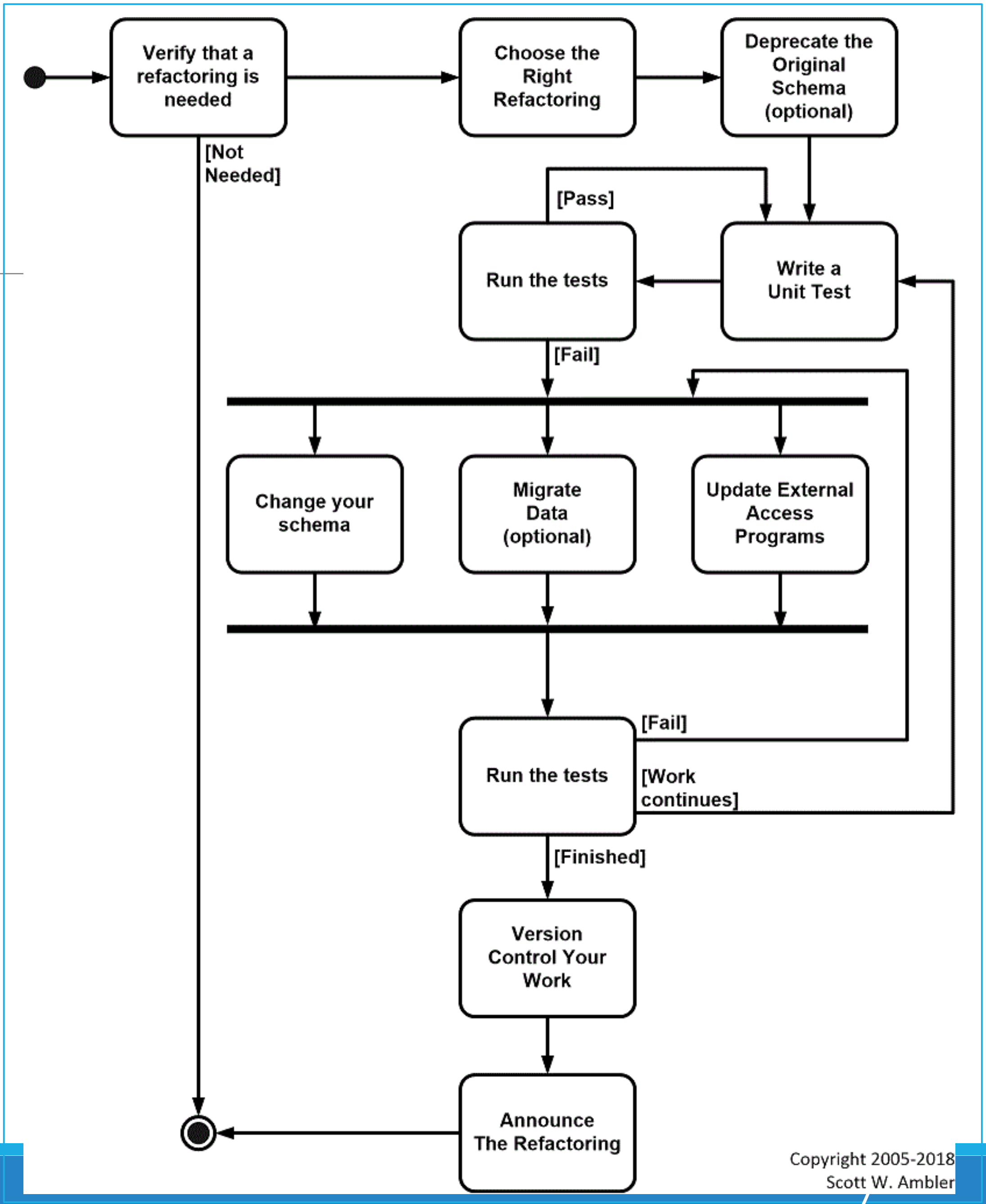
https://youtu.be/qSrhVt0654U

Refactoring

EASY IN THEORY


// TODO
//TODO REMOVE
//TODO REMOVE
//TODO REMOVE
//TODO ~~REMOVE~~ CLEANUP
MONKEYUSER.COM


Write a failing test
Make the test pass
Refactor


Verify that a refactoring is needed
[Not Needed]
Choose the Right Refactoring
Deprecate the Original Schema (optional)
[Pass]
Run the tests
Write a Unit Test
[Fail]
Change your schema
Migrate Data (optional)
Update External Access Programs
Run the tests
[Fail]
[Work continues]
[Finished]
Version Control Your Work
Announce The Refactoring
Copyright 2005-2018 Scott W. Ambler

Quotes About Refactoring

- “*A series of small decisions and actions all made through the filter of a set of values and the desire to make something excellent*” – Chad Fowler, Ruby Central.

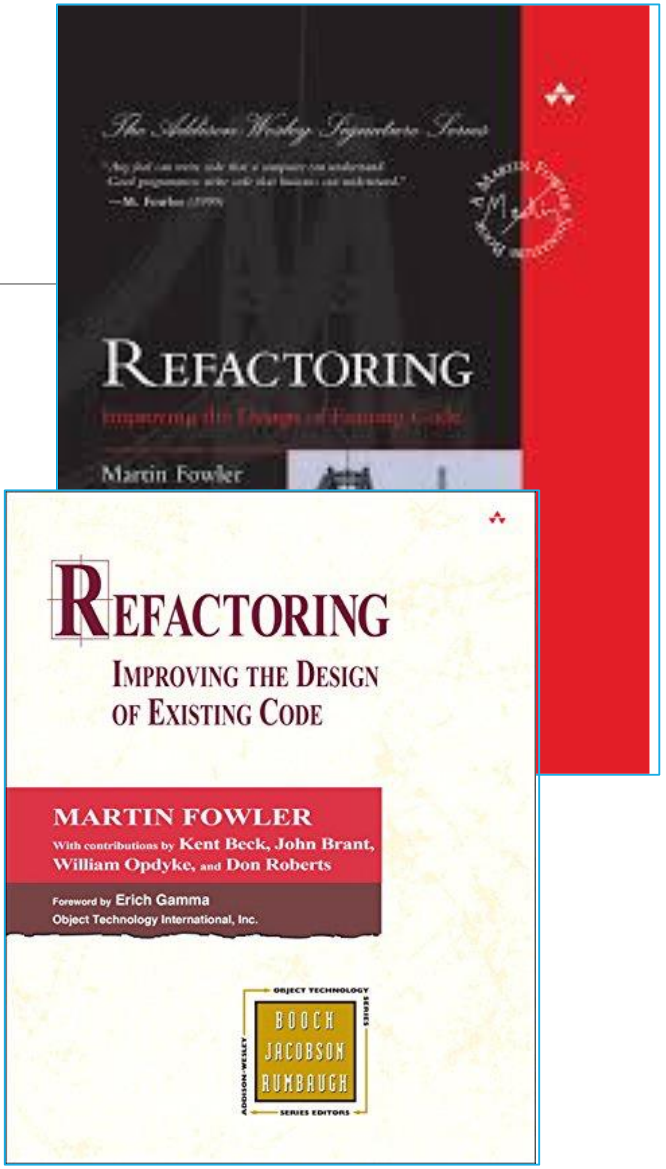
- “*Refactoring is the process of changing a software system in a way that does not alter the external behavior of the code yet improves its internal structure*” – Martin Fowler.

- “*The XP philosophy is to start where you are now and move towards the ideal. From where you are now, could you improve a little bit?*” – Kent Beck.

- “*Any fool can write code that a computer can understand. Good programmers can write code that humans can understand*” – Martin Fowler.

Martin Fowler
- Refactoring Godfather

Kent Beck
- Agile Guru & Code Smeller


The Addison-Wesley Signature Series
REFACTORING
Improving the Design of Existing Code
Martin Fowler
REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International, Inc.
BOOCH JACOBSON RUMBAUGH

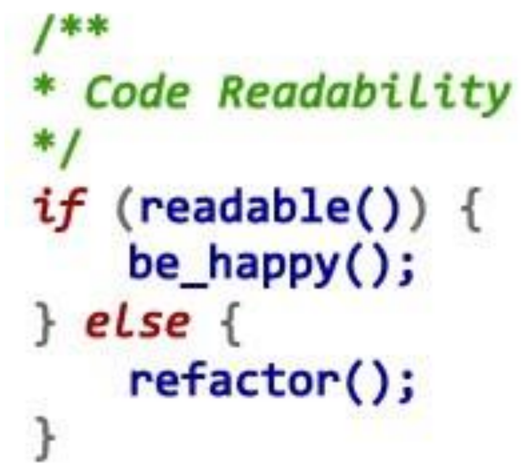
University of Nottingham
UK | CHINA | MALAYSIA

Refactoring Improves Existing Code

- The process of changing a software system to not alter the external behavior of the code yet improves its internal structure in an incremental fashion.
- A disciplined way to clean up code to minimize the chance of introducing bugs.
- Improving the design of code after it has been written.
  - Can be the code written yesterday or code written decades ago.
- Explores the tradeoff space between clarity and performance.
- Refactoring is **NOT** optimizing code to make it run faster.
  - It is about making code to be more sense and increasing its robustness.
  - Making code open for modification.
  - Application of SOLID principles and design pattern.

Kent Beck Says …

- *“The majority of the cost of software is incurred after the software has been first deployed. Thinking about my experience of modifying code, I see that I spend much more time reading the existing code than I do writing new code. If I want to make my code cheap, therefore, I should make it easy to read.”*



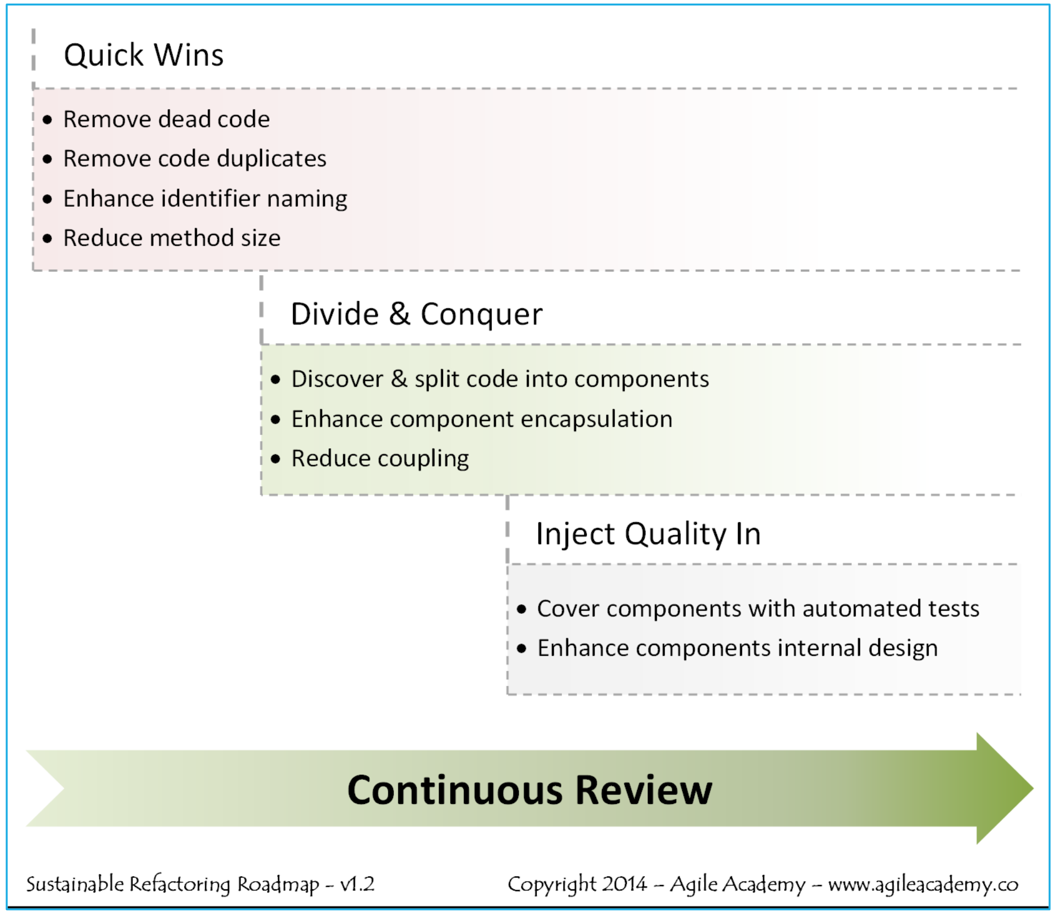
```
/**
* Code Readability
*/
if (readable()) {
    be_happy();
} else {
    refactor();
}
```

University of Nottingham
UK | CHINA | MALAYSIA

Who is Involved in Refactoring?

- Refactoring is a multidisciplinary activity involving:
  - Programmers/developers.
  - Senior designers.
  - Management.
  - System architects.
- Each of those roles will adapt the principles of refactoring to a specific project or workspace.
- Refactoring occurs in different forms across a project.

Refactoring Roadmap



Quick Wins

- Remove dead code
- Remove code duplicates
- Enhance identifier naming
- Reduce method size

Divide & Conquer

- Discover & split code into components
- Enhance component encapsulation
- Reduce coupling

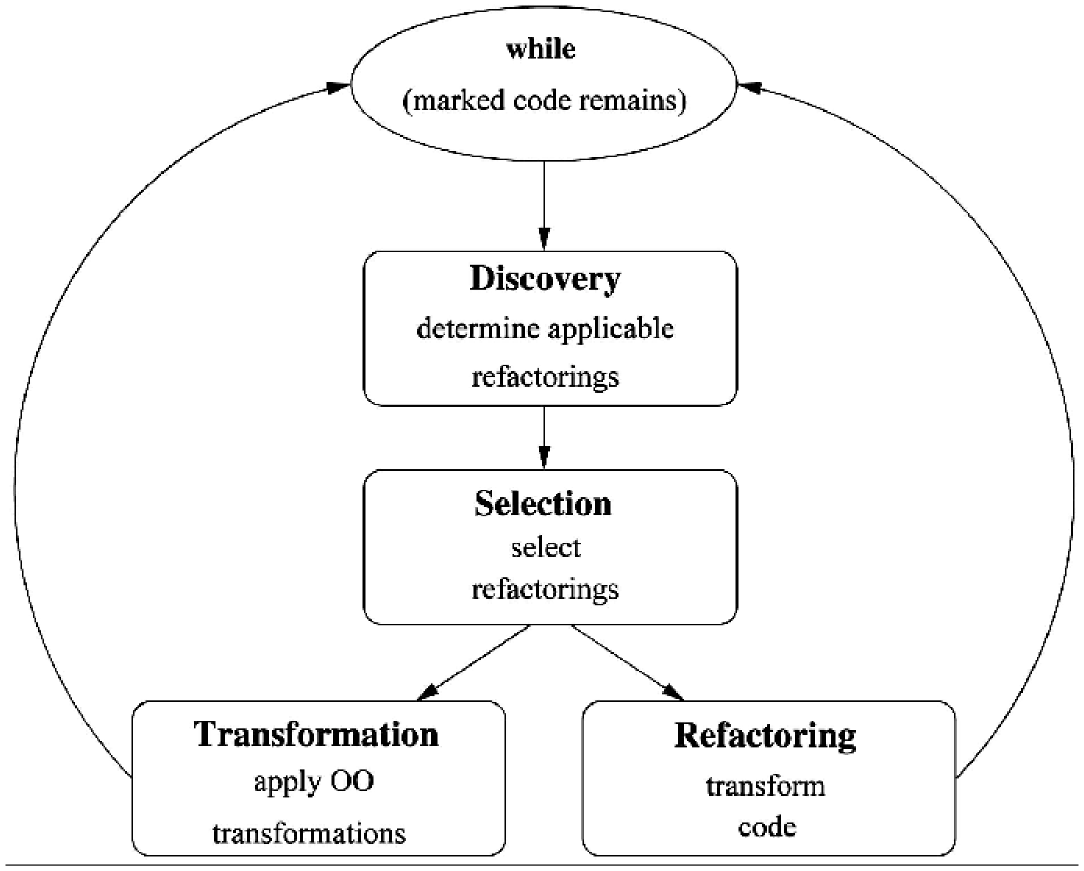
Inject Quality In

- Cover components with automated tests
- Enhance components internal design

Continuous Review

Sustainable Refactoring Roadmap – v1.2

Copyright 2014 – Agile Academy – www.agileacademy.co



University of Nottingham
UK | CHINA | MALAYSIA

Build Test Scaffold Before Touch Anything

- The key to successful refactoring is to **AVOID** breaking anything while make any changes.
- This requires to use the existing or develop new test scaffold.
  - Regression testing?
- Run the test scaffold after every instance of change, not after every feature refactored.
- Legacy code may not include any test cases, in which case needs to add them before starts “tinkering”, i.e., making incremental changes.
- Substitute ‘live’ components with dummy ones, i.e., use ‘Mock Objects’.
- Ready to go if all these are done.

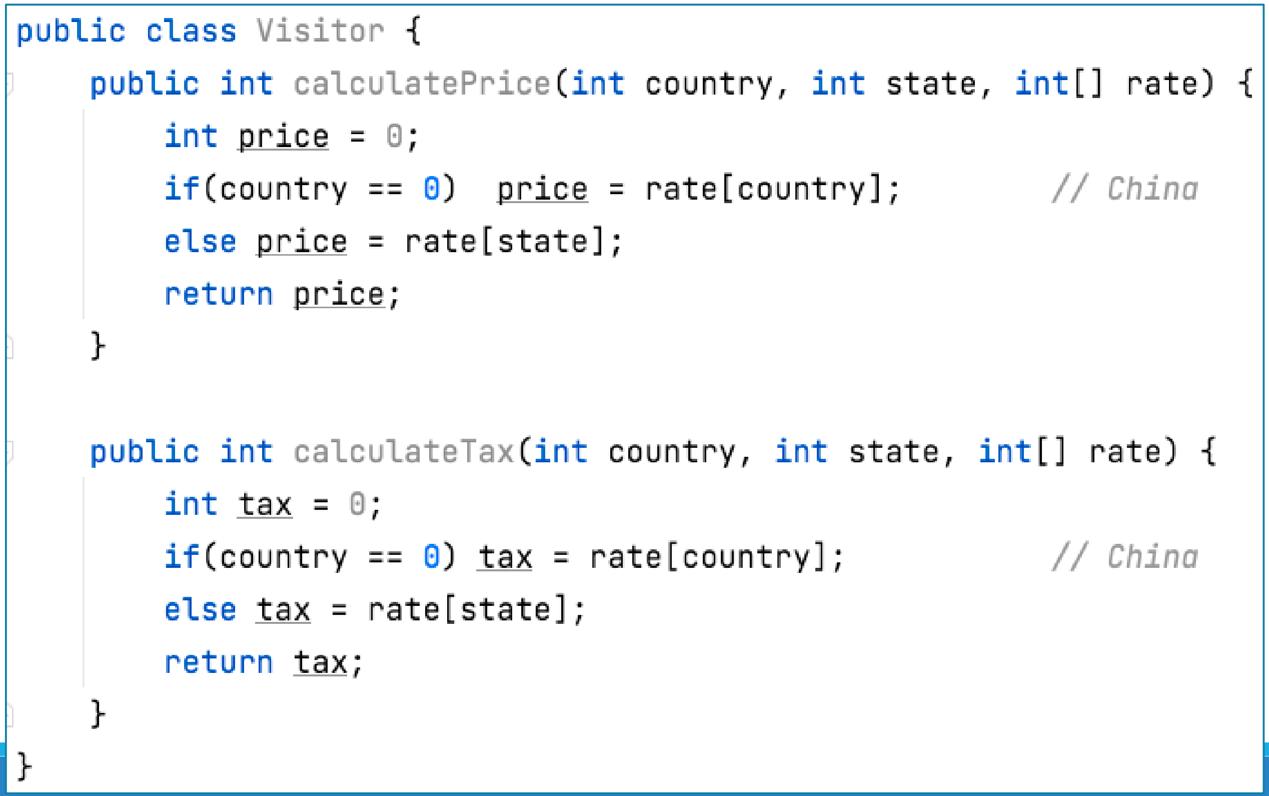
https://martinfowler.com/articles/mocksArentStubs.html

Identifying Code Smells

DECIDE WHAT TO REFACTOR

Code Smells Because It Is Hard to Understand

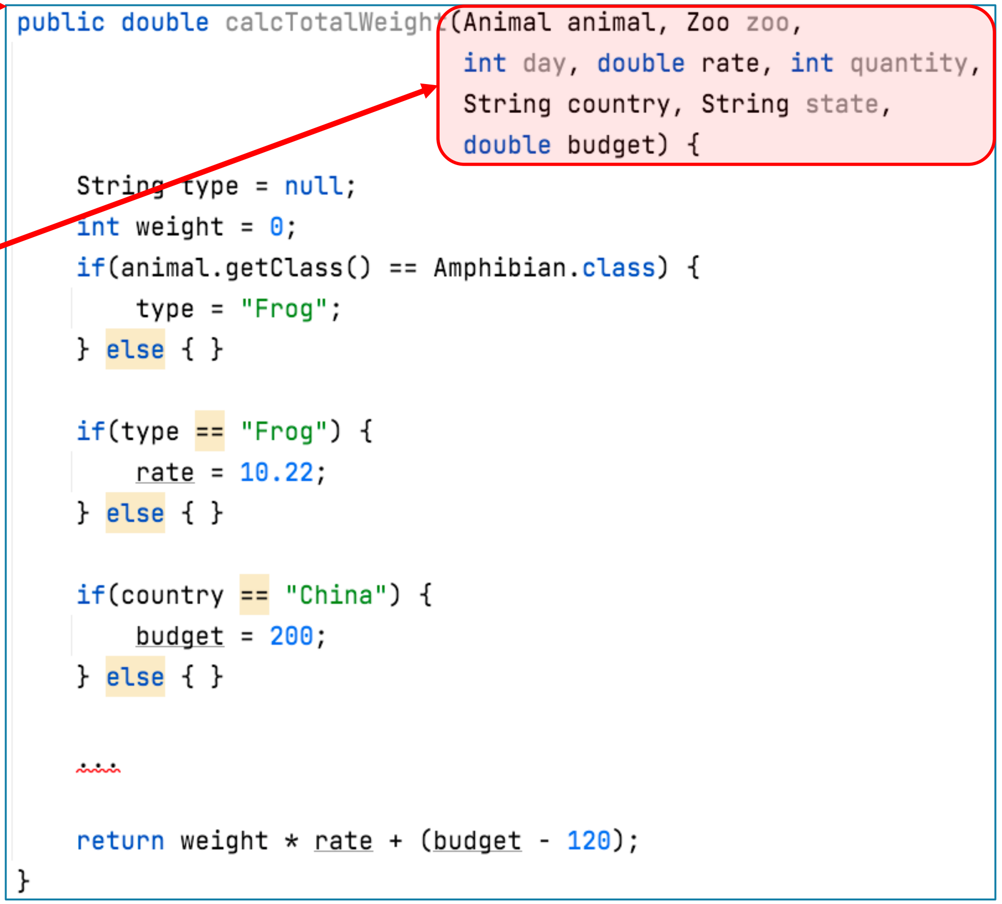
- Duplicate code.
  - Bad because if you modify one instance of duplicated code but not the others, you (may) have introduced a bug!



```
public class Visitor {
    public int calculatePrice(int country, int state, int[] rate) {
        int price = 0;
        if(country == 0)  price = rate[country];        // China
        else price = rate[state];
        return price;
    }

    public int calculateTax(int country, int state, int[] rate) {
        int tax = 0;
        if(country == 0) tax = rate[country];           // China
        else tax = rate[state];
        return tax;
    }
}
```

Code Smells Because It Is Hard to Understand

- Long methods.
  - Long methods are more difficult to understand.
  - Note: performance concerns with respect to lots of short methods are largely unfounded.
- Long parameter lists.
  - Hard to understand, can become inconsistent.



```
public double calcTotalWeight(Animal animal, Zoo zoo,
                              int day, double rate, int quantity,
                              String country, String state,
                              double budget) {

    String type = null;
    int weight = 0;
    if(animal.getClass() == Amphibian.class) {
        type = "Frog";
    } else { }

    if(type == "Frog") {
        rate = 10.22;
    } else { }

    if(country == "China") {
        budget = 200;
    } else { }

    ...

    return weight * rate + (budget - 120);
}
```

Code Smells Because It Is Hard to Understand

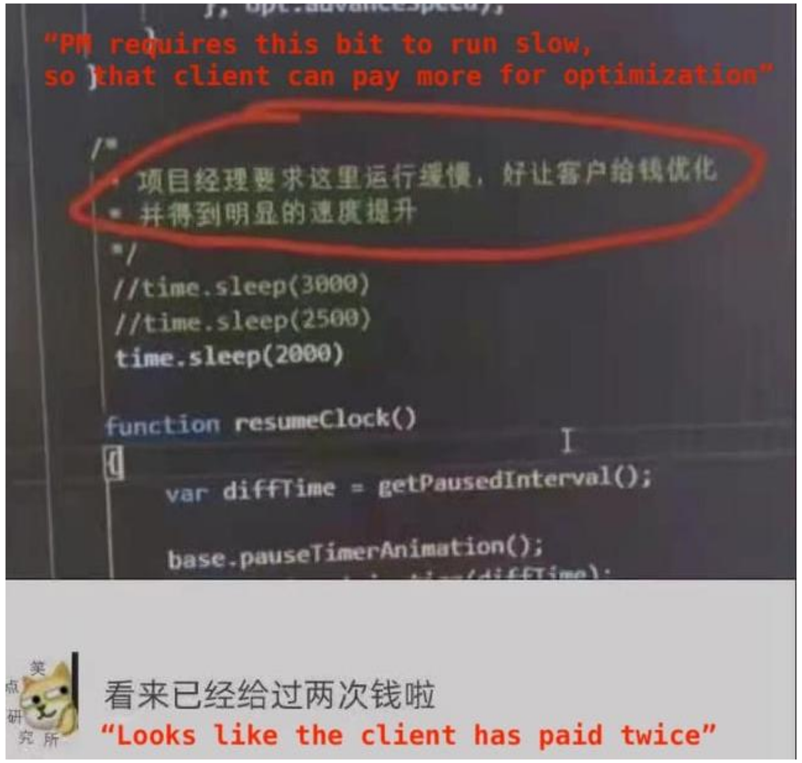
- Large classes.
  - Classes try to do too much, which reduces cohesion (violates single responsibility).
- Divergent changes.
  - Related to cohesion where one type of change requires changing one subset of methods; another type of change requires changing another subset.
- Lazy classes.
  - A class that no longer "pays its way", part formed functionality.
- Speculative generality.
  - "Oh, I think we need the ability to do this kind of thing someday."
- Temporary field.
  - An attribute of an object is only set in certain circumstances; but an object should need all its attributes.

More Code Smells Examples

- Data class.
  - Theses are classes that have fields, getting and setting methods for the fields, and nothing else; they are data holders, but objects should be about data **AND** behaviour.

- Refused bequest.
  - A subclass ignores most of the functionalities provided by its superclass where the subclass may not pass the “is a” test (Liskov Substitution Violation).

- Comment-based cover up.
  - Comments are sometimes used to hide bad code, symptomatic of large blocks of comments, not using automatic comment generation.

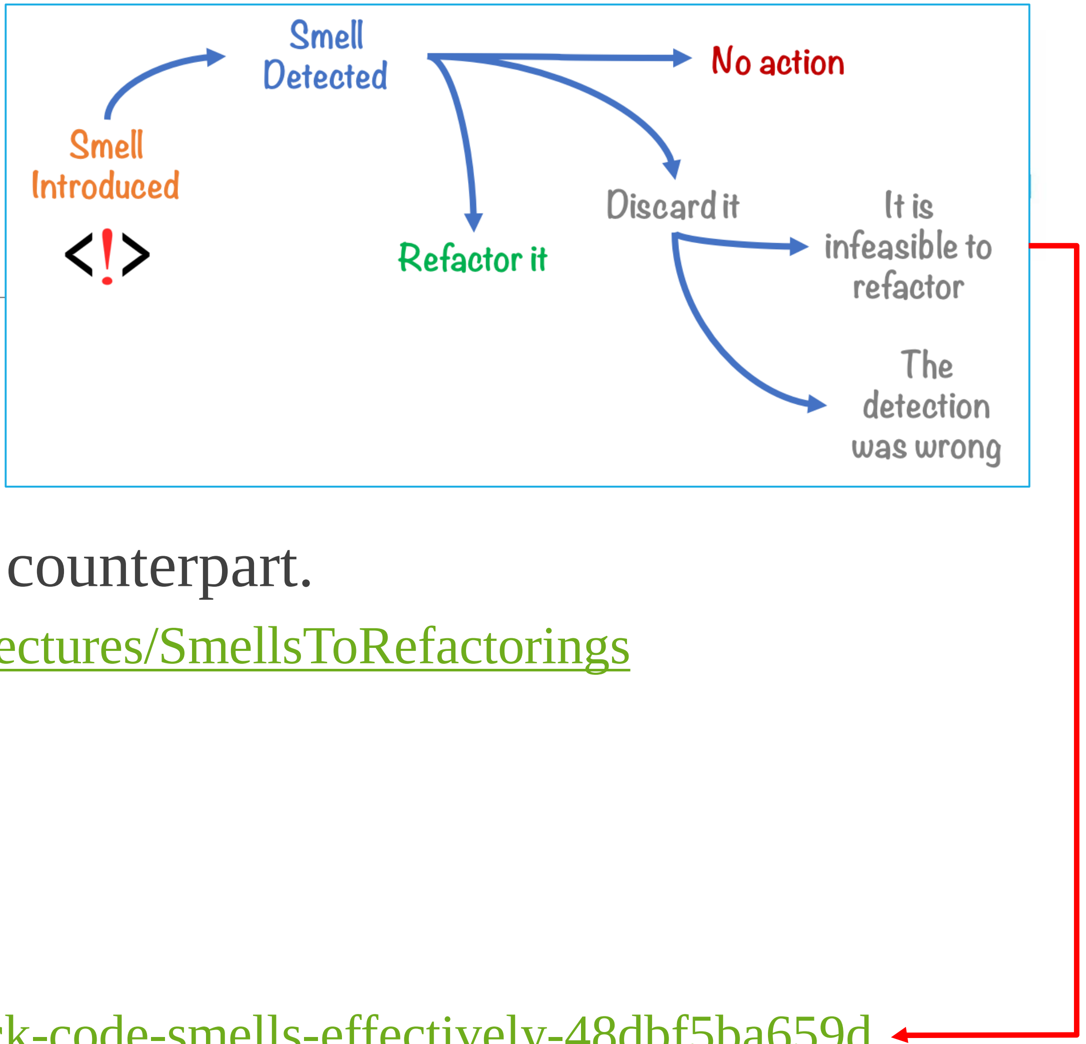


"PM requires this bit to run slow,
so that client can pay more for optimization"

```
/*
 * 项目经理要求这里运行缓慢，好让客户给钱优化
 * 并得到明显的速度提升
 */
//time.sleep(3000)
//time.sleep(2500)
time.sleep(2000)

function resumeClock()
{
    var diffTime = getPausedInterval();

    base.pauseTimerAnimation();
```

看来已经给过两次钱啦
“Looks like the client has paid twice”

More: References



- Many more code smells at
  - http://wiki.c2.com/?CodeSmell
- Matching up a smell to its refactoring counterpart.
  - https://users.csc.calpoly.edu/~jdalbey/305/Lectures/SmellsToRefactorings
- Taxonomy of code smells.
  - http://www.tusharma.in/smells/
- How to track code smells effectively.
  - https://medium.com/@tusharma/how-to-track-code-smells-effectively-48dbf5ba659d

Refactoring Code

SHAMELESSLY INSPIRED BY REFACTORING (FOWLER)

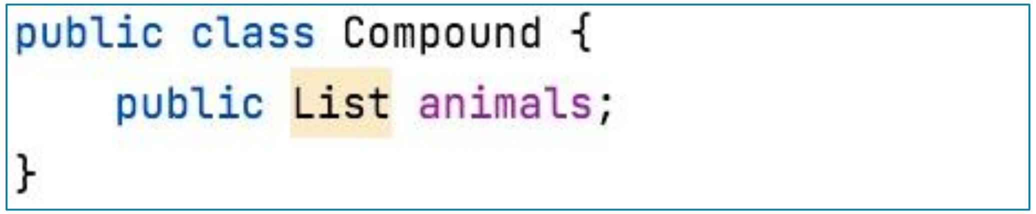
CHAPTERS 6 – 10

Fowler Defined Different Categories of Refactoring

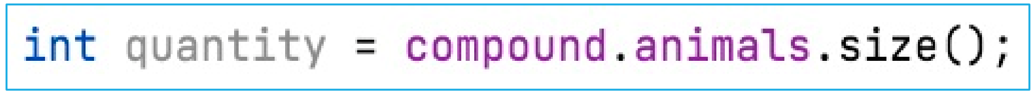
- Composing Methods: Refactoring within a method or within an existing class.
- Moving Features Between Objects: Refactoring changing the responsibility of a class.
- Organizing Data: Refactoring improve data structures and object linking.
- Simplifying Conditional Expressions: Encapsulation of conditions by replacing with polymorphism.
- Making Method Calls Simpler: Refactoring that make interfaces simpler.
- Dealing with Generalization: Moving methods up and down the hierarchy of inheritance by (often) applying the Factory or Template design pattern.
- Big Refactoring: Architectural refactoring to promote loose coupling or to realise a redesign via object orientation (NOTE: this is extremely difficult for most projects).

Encapsulate Fields to Retain Private Variables

- Un-encapsulated data is a violation of key object-oriented principles.
- Use property `get` and `set` procedures to provide public access to private (encapsulated) member variables.



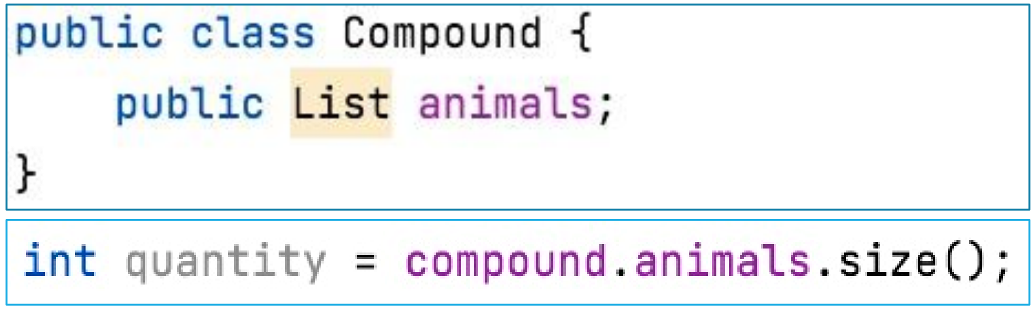
```
public class Compound {
    public List animals;
}
```



```
int quantity = compound.animals.size();
```

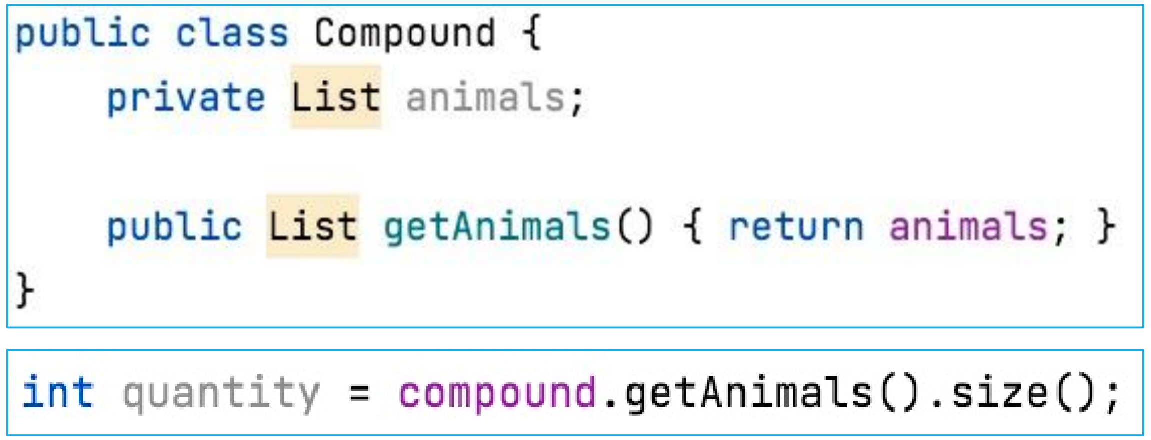
Encapsulate Fields to Retain Private Variables

- Un-encapsulated data is a violation of key object-oriented principles.
- Use property `get` and `set` procedures to provide public access to private (encapsulated) member variables.



```
public class Compound {
    public List animals;
}
```

```
int quantity = compound.animals.size();
```

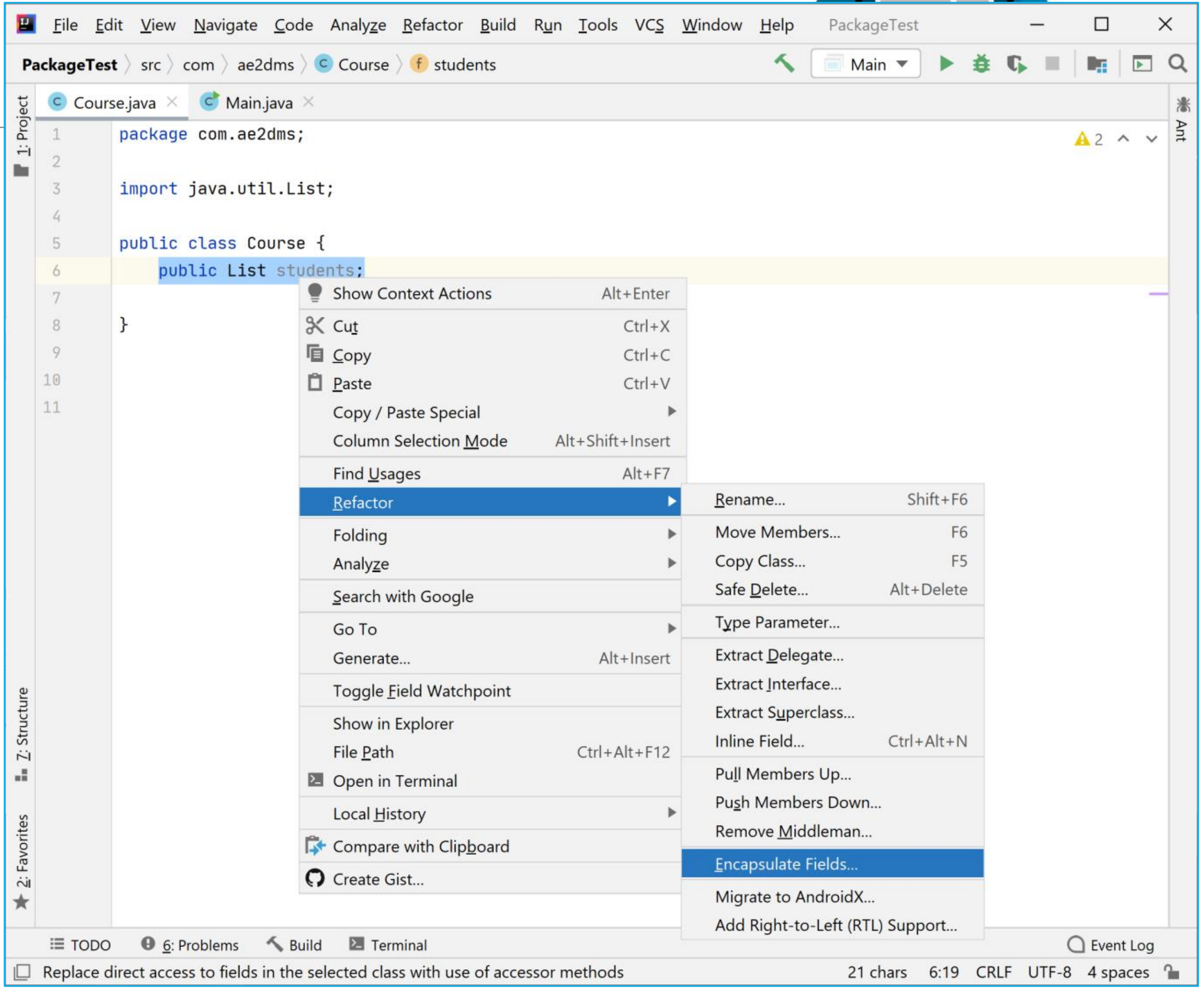


```
public class Compound {
    private List animals;

    public List getAnimals() { return animals; }
}
```

```
int quantity = compound.getAnimals().size();
```

Encapsulating Fields Automatically with IDEs

- Assuming a class has 10 fields; it is quite an overhead to manage and is a pain to have to go through to add `get_()` and `set_()` for each one.
- Refactoring tools.
  - https://www.jetbrains.com/help/idea/refactoring-source-code.html
- Also allows for automating two other refactoring tasks.
  - Rename Method: Cascading the new name of the method.
  - Change Method Parameters: Auto update when parameters are modified.



```java
package com.ae2dms;

import java.util.List;

public class Course {
    public List students;

}
```

Replace direct access to fields in the selected class with use of accessor methods

University of Nottingham
UK | CHINA | MALAYSIA

Extract Method From A Larger Block of Code

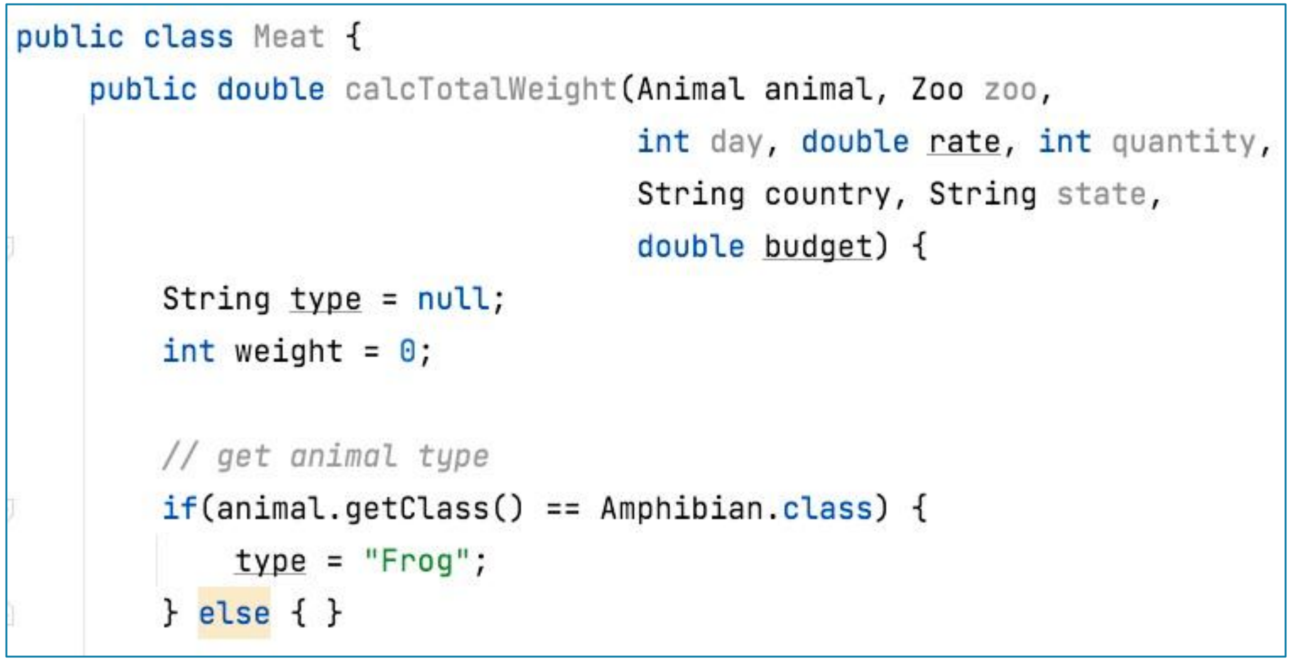
- If a method is too long or needs excessive code to describe its function, then probably extract a new method out of the code.
- Short, well-named methods are easier to maintain.
- Obeys the single responsibility principle.
- How big is too big?
  - Used to be around 125 LOC in Fortran.
  - Now have 80 lines of code (from the size of the UNIX terminal).
- Length is not the issue – it is down to “the semantic distance between a method name and the method body”.

Create A New Method to Perform The Method Extraction

1. Create a new method and name it after the intention of the method.
   - If can't derive this name, probably should not be extracting the method!
2. Copy the extracted code from the source method into the new method.
3. Scan the extracted code for references to local and temporary variables.
4. Check if any of the local variables are modified by the extracted code.
   - See if this can be transformed into a query instead of using the "Replace Temporarily with Query Refactor (more on this later).
5. Pass into new method as any parameters required local scope variables.
6. Replace the extracted code with a method call to the new method, checking the use of temporarily variables.
7. Run the regression tests.

Using “Extract Method” Is A Quick Win!

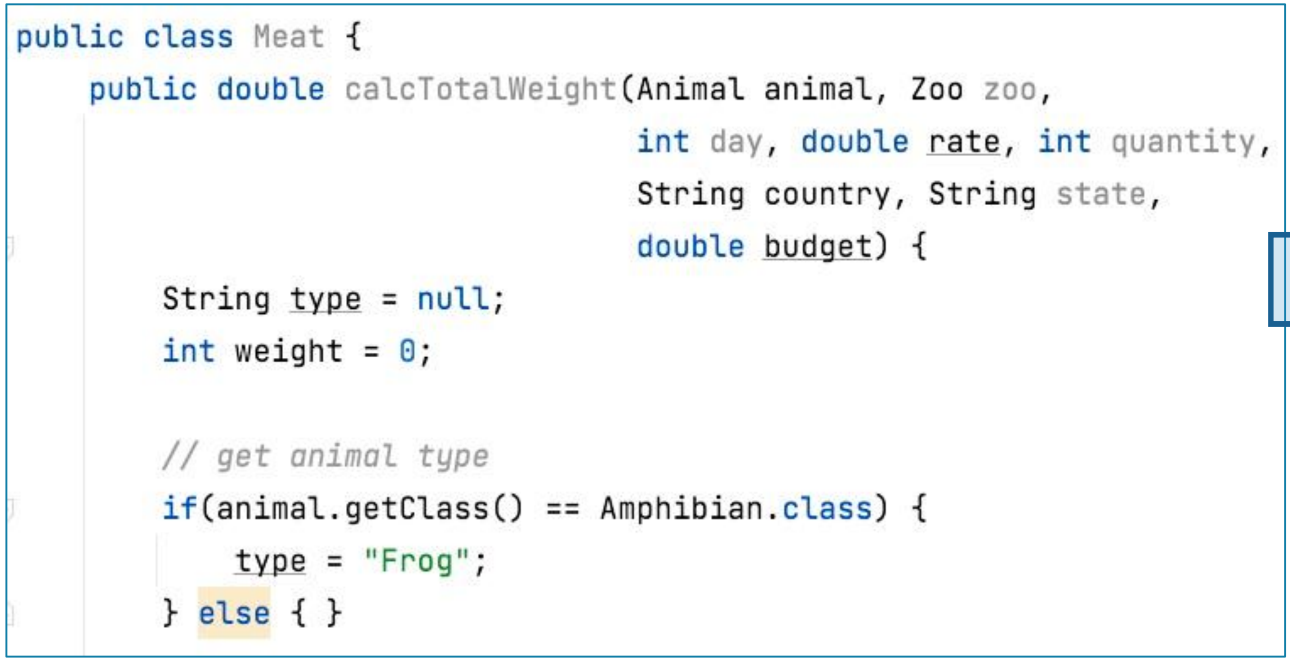
- What are the code smells? See how the comment becomes the method name …



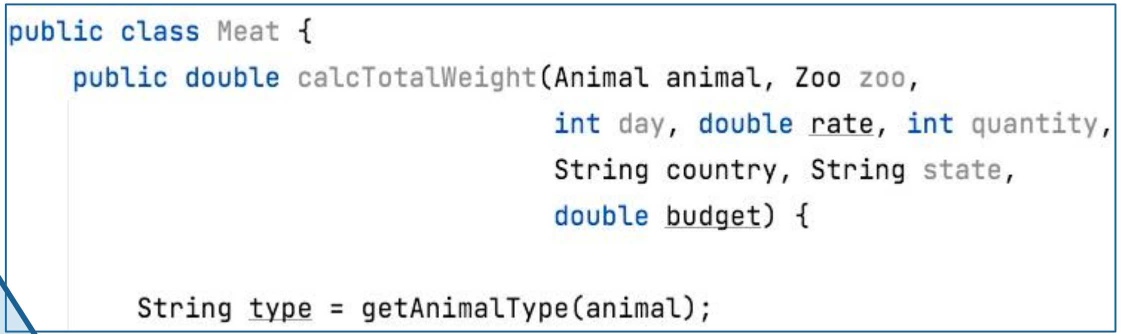
```java
public class Meat {
    public double calcTotalWeight(Animal animal, Zoo zoo,
                                  int day, double rate, int quantity,
                                  String country, String state,
                                  double budget) {
        String type = null;
        int weight = 0;

        // get animal type
        if(animal.getClass() == Amphibian.class) {
            type = "Frog";
        } else { }
```

Using “Extract Method” Is A Quick Win!

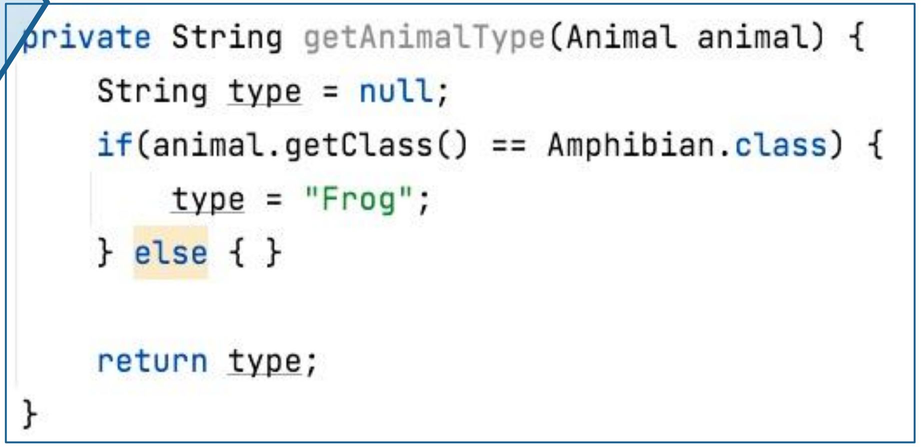
- What are the code smells? See how the comment becomes the method name …



```
public class Meat {
    public double calcTotalWeight(Animal animal, Zoo zoo,
                                  int day, double rate, int quantity,
                                  String country, String state,
                                  double budget) {
        String type = null;
        int weight = 0;

        // get animal type
        if(animal.getClass() == Amphibian.class) {
            type = "Frog";
        } else { }
```



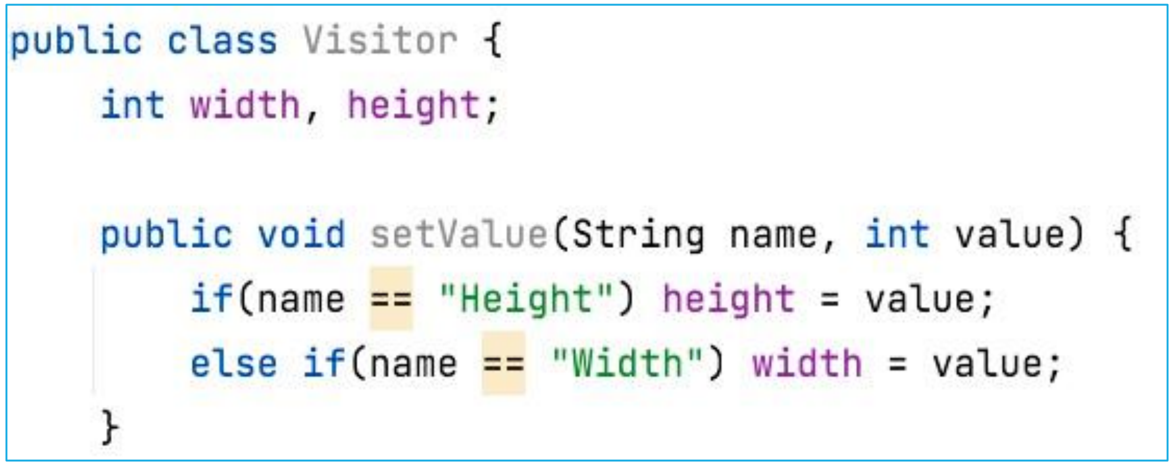
```
public class Meat {
    public double calcTotalWeight(Animal animal, Zoo zoo,
                                  int day, double rate, int quantity,
                                  String country, String state,
                                  double budget) {

        String type = getAnimalType(animal);
```



```
private String getAnimalType(Animal animal) {
    String type = null;
    if(animal.getClass() == Amphibian.class) {
        type = "Frog";
    } else { }

    return type;
}
```

Replace Parameter with Explicit Method

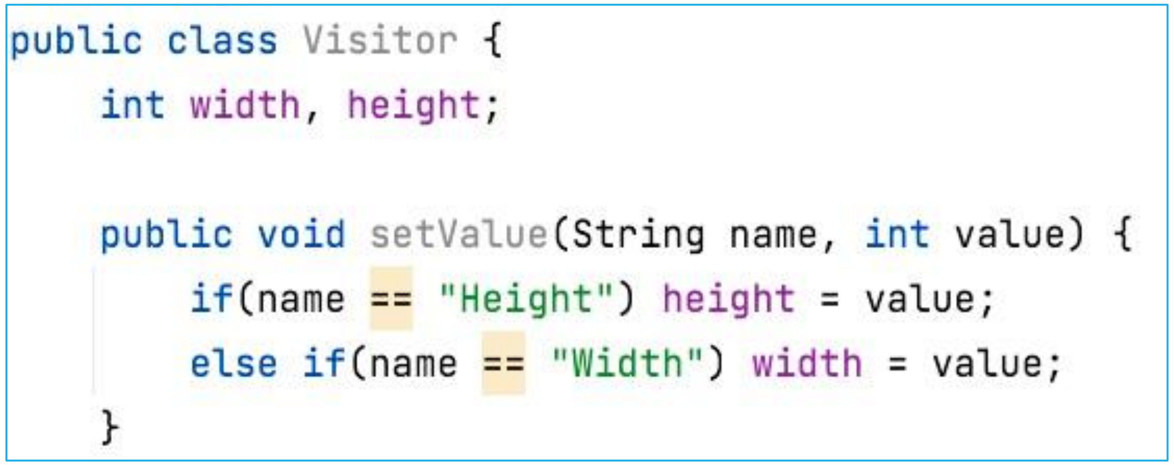
- Have a method that runs different code depending on the values of an enumerated parameter.
- Create a separate method for each value of the parameter.



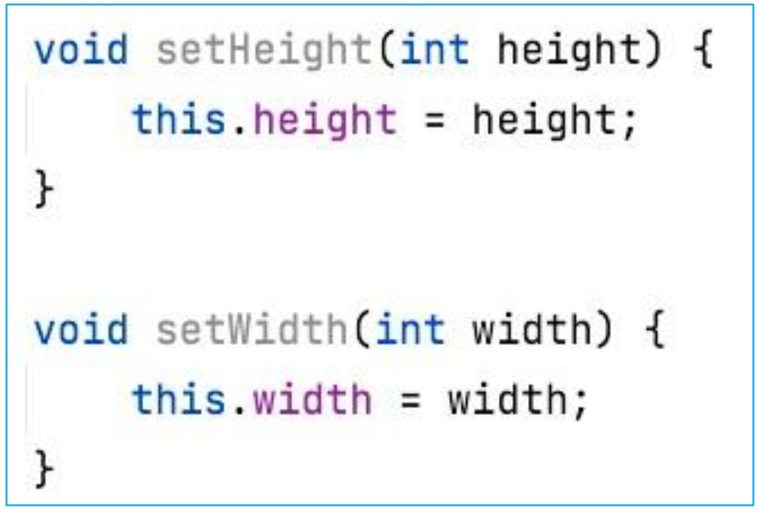
```java
public class Visitor {
    int width, height;

    public void setValue(String name, int value) {
        if(name == "Height") height = value;
        else if(name == "Width") width = value;
    }
```

Replace Parameter with Explicit Method

- Have a method that runs different code depending on the values of an enumerated parameter.
- Create a separate method for each value of the parameter.



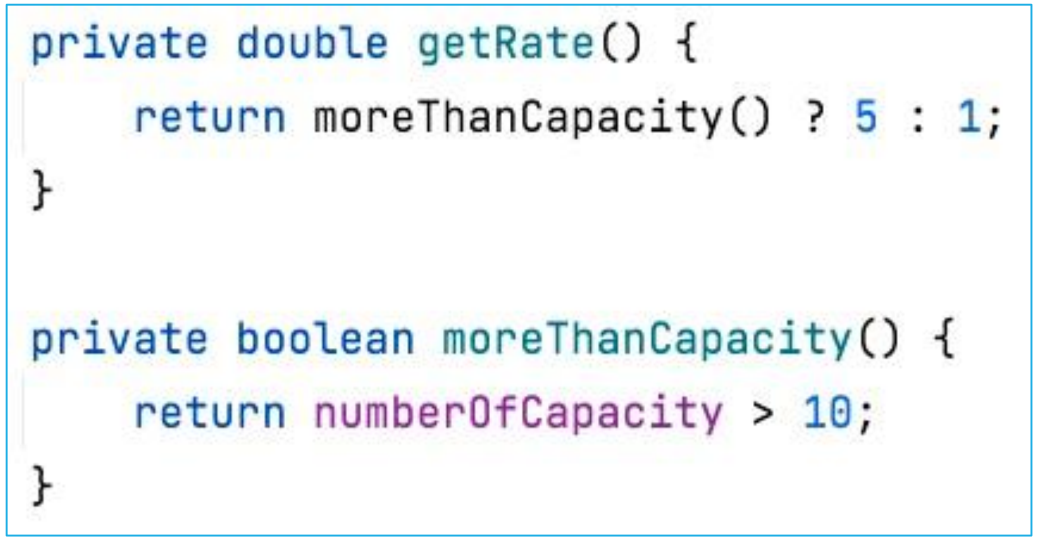
```java
public class Visitor {
    int width, height;

    public void setValue(String name, int value) {
        if(name == "Height") height = value;
        else if(name == "Width") width = value;
    }
```



```java
void setHeight(int height) {
    this.height = height;
}

void setWidth(int width) {
    this.width = width;
}
```

Inline Method to Reduce Indirection

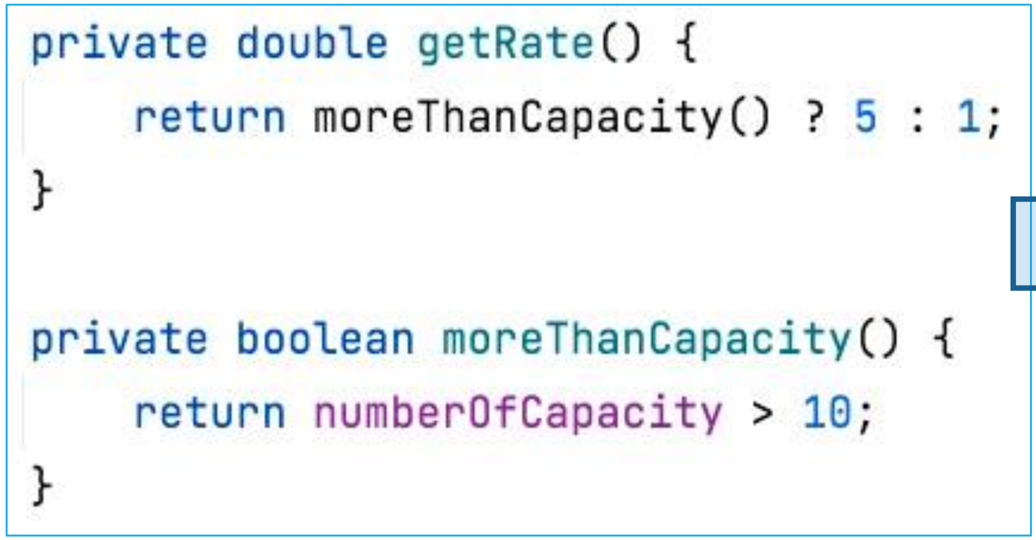
- The method does in the body precisely the method intention.



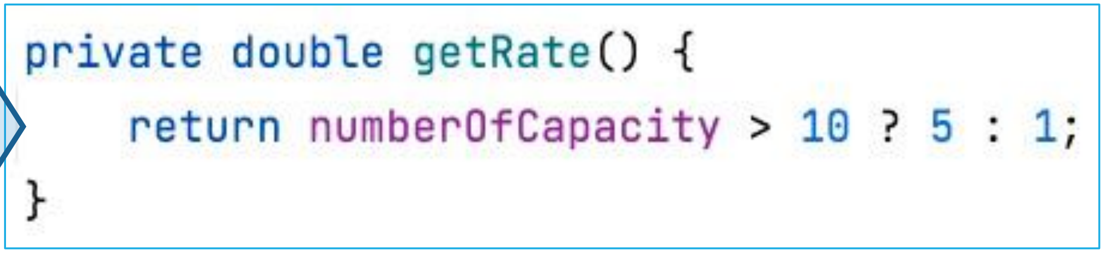
```
private double getRate() {
    return moreThanCapacity() ? 5 : 1;
}

private boolean moreThanCapacity() {
    return numberOfCapacity > 10;
}
```

Inline Method to Reduce Indirection

- The method does in the body precisely the method intention.



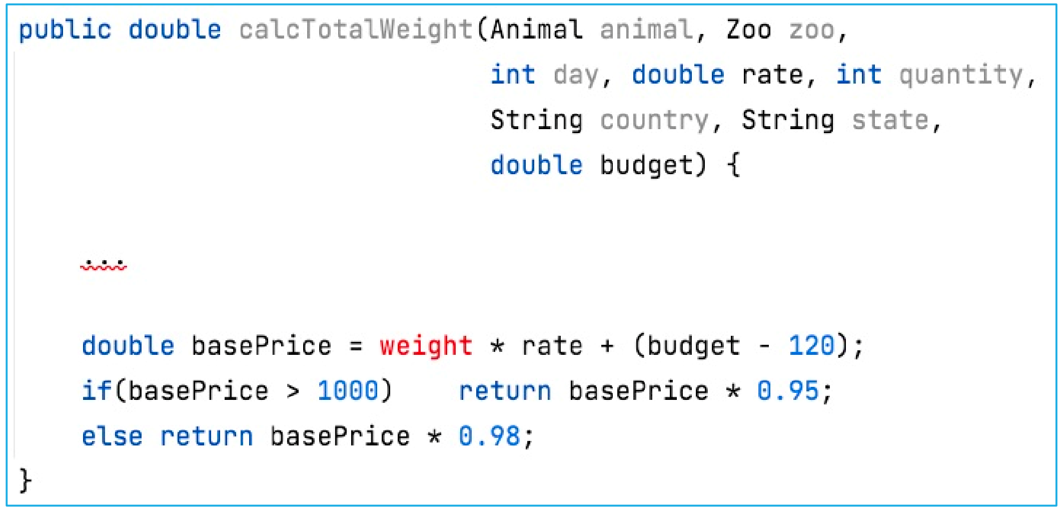
```java
private double getRate() {
    return moreThanCapacity() ? 5 : 1;
}

private boolean moreThanCapacity() {
    return numberOfCapacity > 10;
}
```



```java
private double getRate() {
    return numberOfCapacity > 10 ? 5 : 1;
}
```

Replace Temporary with Query

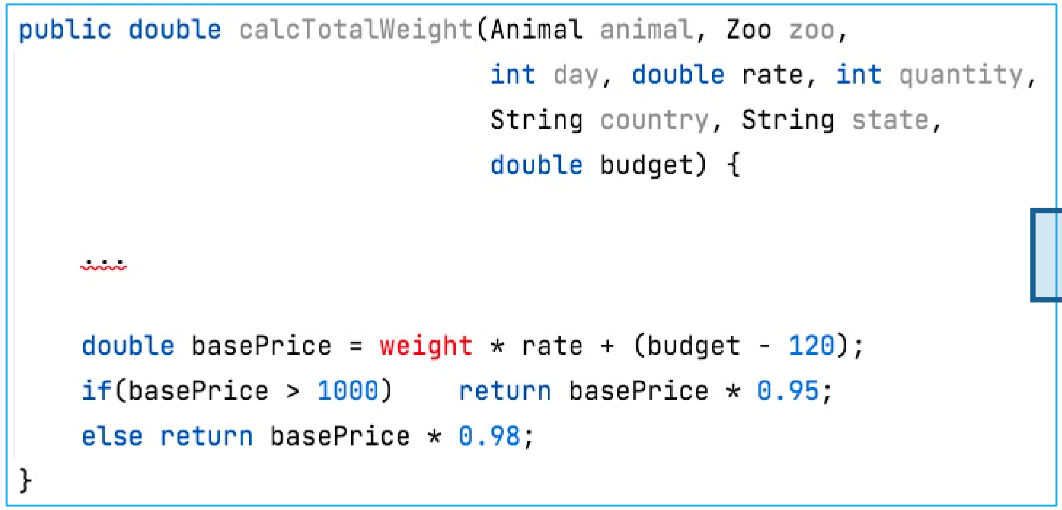
- Place the result of an expression in a local variable for later use in the code??
- Move the entire expression to separate method and return the result from it.
- Query the method instead of using a variable. Incorporate the new method in other methods, if necessary.



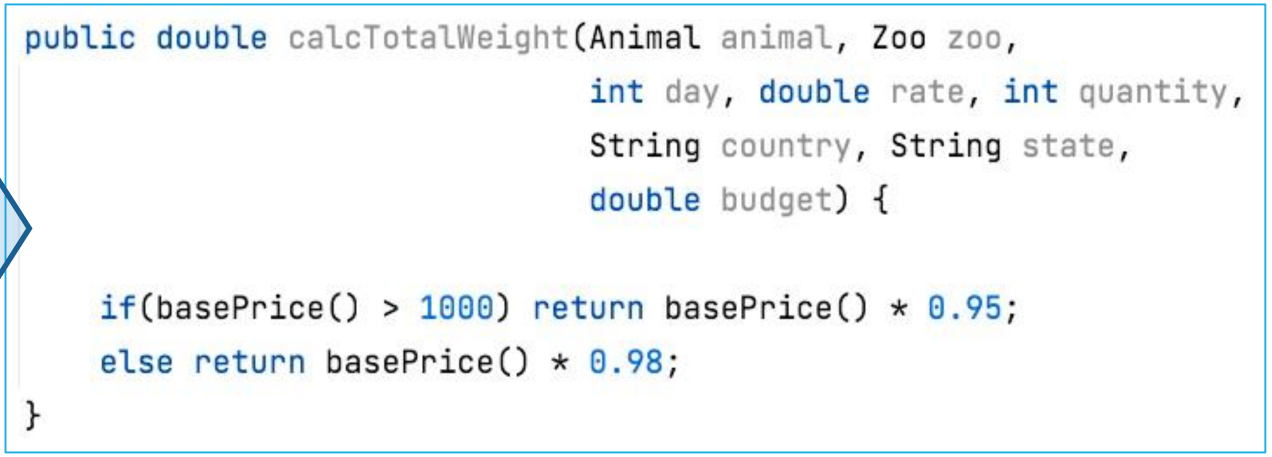
```
public double calcTotalWeight(Animal animal, Zoo zoo,
                              int day, double rate, int quantity,
                              String country, String state,
                              double budget) {


    ...

    double basePrice = weight * rate + (budget - 120);
    if(basePrice > 1000)    return basePrice * 0.95;
    else return basePrice * 0.98;
}
```

Replace Temporary with Query

- Place the result of an expression in a local variable for later use in the code??
- Move the entire expression to separate method and return the result from it.
- Query the method instead of using a variable. Incorporate the new method in other methods, if necessary.



```java
public double calcTotalWeight(Animal animal, Zoo zoo,
                              int day, double rate, int quantity,
                              String country, String state,
                              double budget) {

    ...

    double basePrice = weight * rate + (budget - 120);
    if(basePrice > 1000)    return basePrice * 0.95;
    else return basePrice * 0.98;
}
```



```java
public double calcTotalWeight(Animal animal, Zoo zoo,
                              int day, double rate, int quantity,
                              String country, String state,
                              double budget) {

    if(basePrice() > 1000) return basePrice() * 0.95;
    else return basePrice() * 0.98;
}
```

Rename Variable or Method to Self Document

- Perhaps one of the simplest, but one of the most useful that bears repeating: If the name of a method or variable does not reveal its purpose, then change the name of the method or variable.

```
public class Customer
{
   public double getinvcdtlmt();
}
```

Rename Variable or Method to Self Document

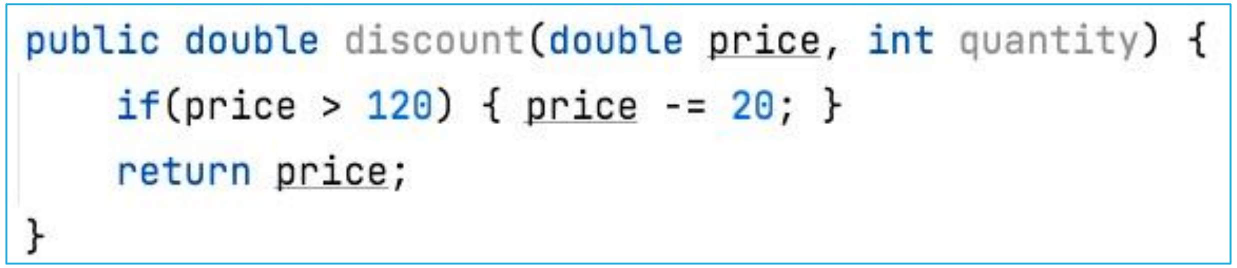
- Perhaps one of the simplest, but one of the most useful that bears repeating: If the name of a method or variable does not reveal its purpose, then change the name of the method or variable.

```
public class Customer
{
   public double getinvcdtlmt();
}
```

```
public class Customer
{
   public double getInvoiceCreditLimit();
}
```

Remove Assignments to Parameters

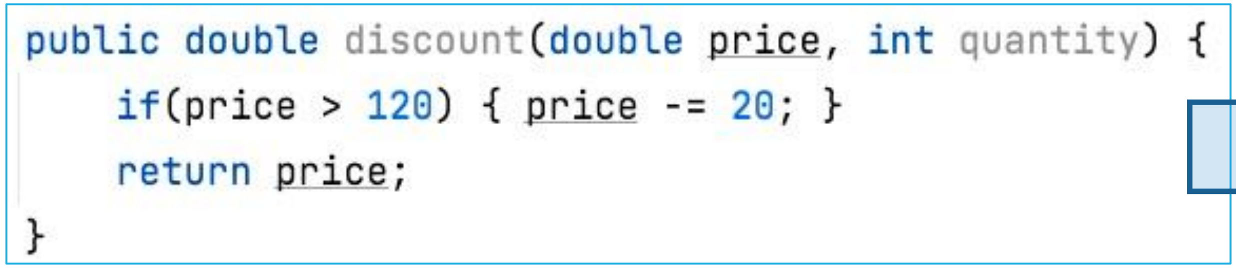
- Create a local variable and assign the initial value of parameter.
- In all method code that follows this line, replace the parameter with new local variable.



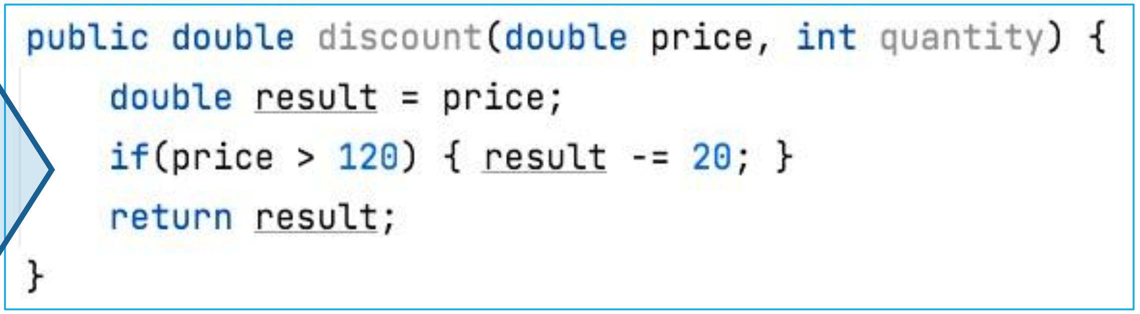
```java
public double discount(double price, int quantity) {
    if(price > 120) { price -= 20; }
    return price;
}
```

Remove Assignments to Parameters

- Create a local variable and assign the initial value of parameter.
- In all method code that follows this line, replace the parameter with new local variable.



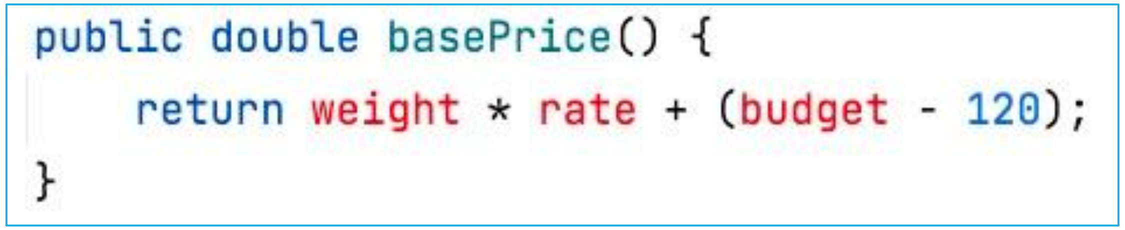
```java
public double discount(double price, int quantity) {
    if(price > 120) { price -= 20; }
    return price;
}
```



```java
public double discount(double price, int quantity) {
    double result = price;
    if(price > 120) { result -= 20; }
    return result;
}
```

Replace Magic Number with Symbolic Content

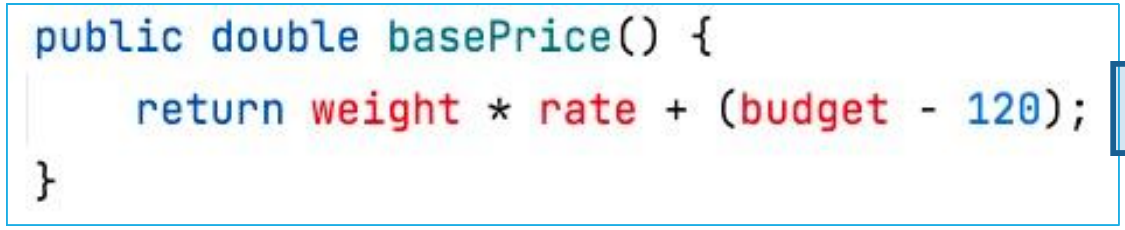
- Used if have a literal number with meaning.
- Create a constant, name it after the meaning and replace the number with it.
- This works if this is a universal constant in the code!
  - (global variables are bad).



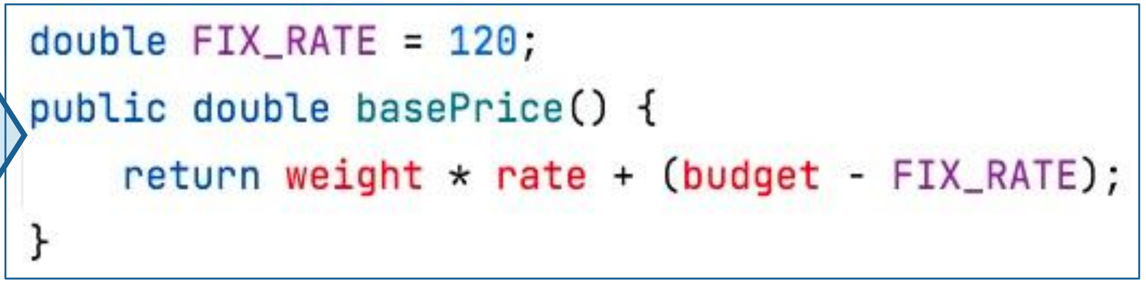
```
public double basePrice() {
    return weight * rate + (budget - 120);
}
```

Replace Magic Number with Symbolic Content

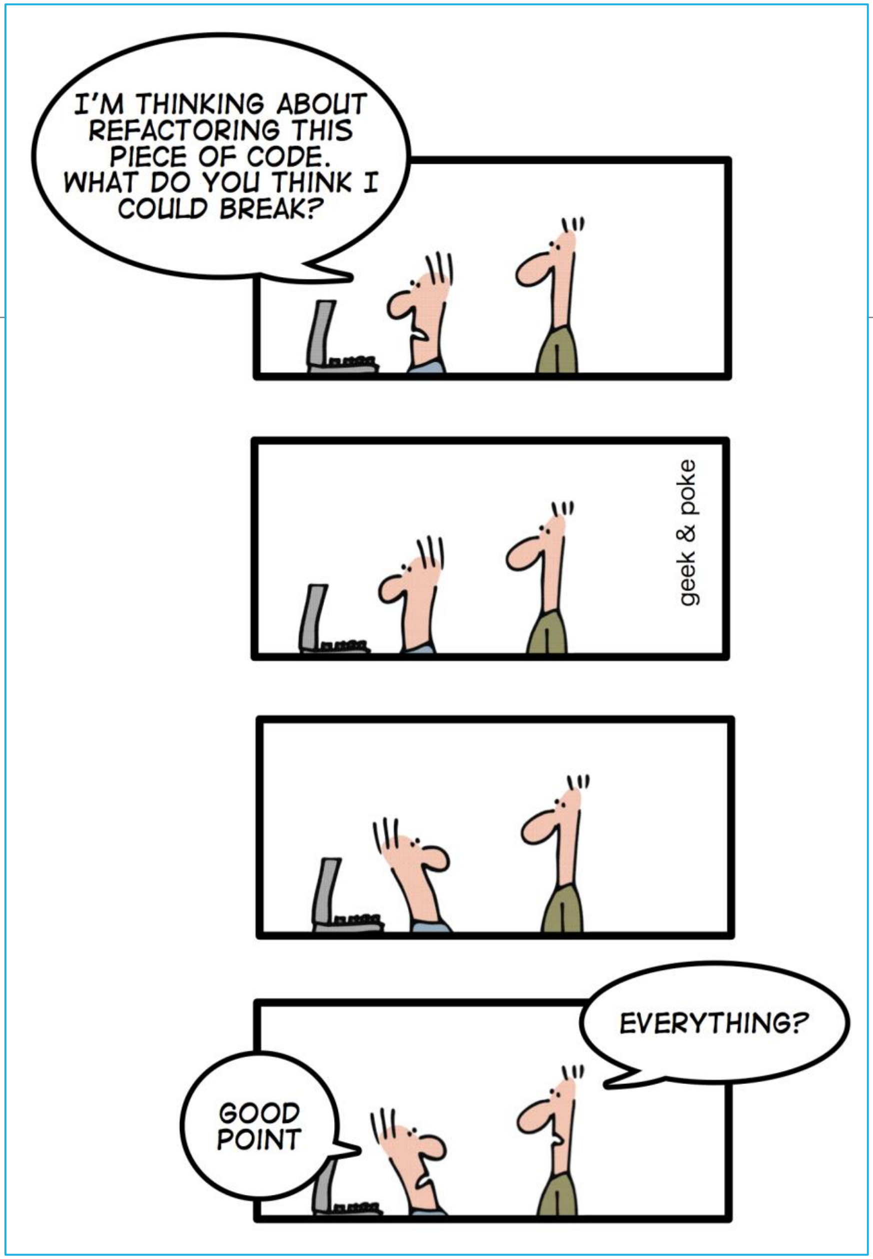
- Used if have a literal number with meaning.
- Create a constant, name it after the meaning and replace the number with it.
- This works if this is a universal constant in the code!
    - (global variables are bad).



```
public double basePrice() {
    return weight * rate + (budget - 120);
}
```



```
double FIX_RATE = 120;
public double basePrice() {
    return weight * rate + (budget - FIX_RATE);
}
```


I'M THINKING ABOUT REFACTORING THIS PIECE OF CODE. WHAT DO YOU THINK I COULD BREAK?
geek & poke
EVERYTHING?
GOOD POINT

THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:
"MY CODE'S COMPILING."
HEY! GET BACK TO WORK!
COMPILING!
OH. CARRY ON.

Refactoring At A Higher Level

DESIGNING CLASSES AND REALLY REFACTORING TO BE CLEAN AND OBJECT-ORIENTED

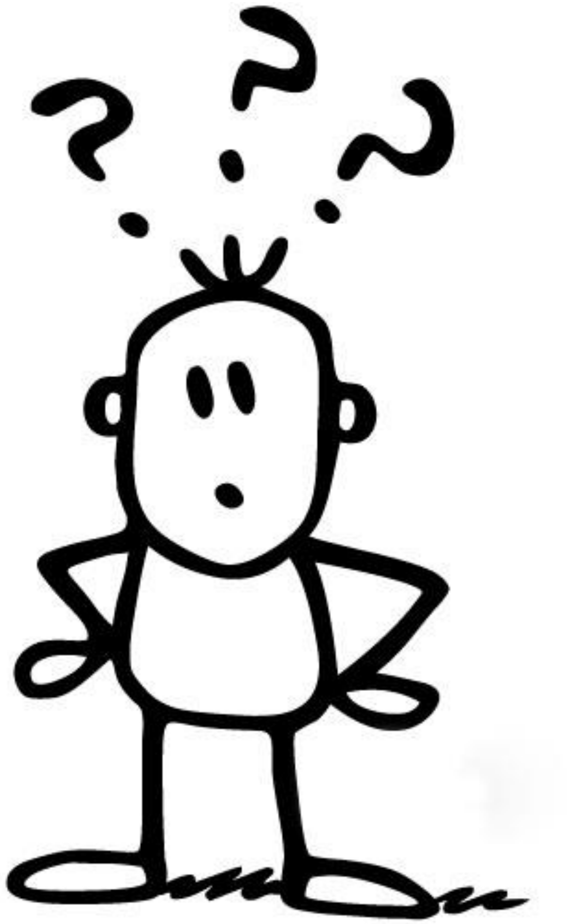
Replace Type Code with Polymorphism

- `switch` statements are very rare in properly designed object-oriented code, much more common in procedural programs.
  - Therefore, a `switch` statement is a simple and easily detected "bad smell".
  - Of course, not all uses of `switch` are bad.
  - A `switch` statement should **NOT** be used to distinguish between various kinds of object.
  - Lengthy to test all of the test cases.
- There are several well-defined refactoring for this case.
  - The simplest is the creation of subclasses.

Replace Type Code with Polymorphism

```
class Animal {
    final int MAMMAL = 0, BIRD = 1, REPTILE = 2;
    int myKind;  // set in constructor
    ...
    String getSkin() {
        switch (myKind) {
            case MAMMAL: return "hair";
            case BIRD: return "feathers";
            case REPTILE: return "scales";
            default: return “skin";
        }
    }
}
```

Improvement Over `switch` using `extends`



Improvement Over `switch` using `extends`

```
class Animal {
      String getSkin() { return “skin"; }
}
class Mammal extends Animal {
      String getSkin() { return "hair"; }
}
class Bird extends Animal {
      String getSkin() { return "feathers"; }
}
class Reptile extends Animal {
      String getSkin() { return "scales"; }
}
```

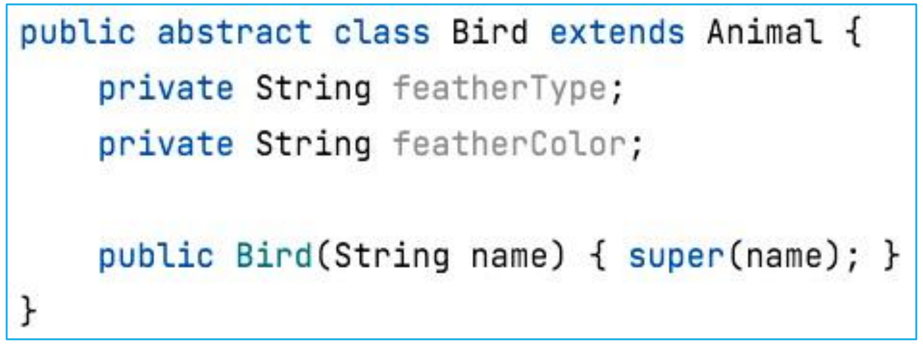
University of Nottingham
UK | CHINA | MALAYSIA

Refactoring Reduces Repetitions and Helps Focus on Objects

- Adding a new animal type, such as Amphibian, does not require revising and recompiling existing code.
  - We have provided good extensibility which is less likely to break future code.
- Mammals, birds, and reptiles are likely to differ in other ways, and we've already separated them out.
  - Therefore, we don't need more `switch` statements in future.
- Eliminated the 'flag' we needed to tell one kind of animal from another, i.e., elegantly remove `MAMMAL = 0` etc.
- We now use Objects the way they were meant to be used and give our code better structure.

Single Responsibility with Extract Class

- When one class does the jobs of two, it promotes tight coupling.
- Break one class into two, e.g., having the skin details as part of the `Bird` class is not a realistic OO model and breaks the Single Responsibility design principle.
  - Can refactor this into two separate classes, each with appropriate responsibility.



```java
public abstract class Bird extends Animal {
    private String featherType;
    private String featherColor;

    public Bird(String name) { super(name); }
}
```

University of Nottingham UK | CHINA | MALAYSIA

Single Responsibility with Extract Class

- When one class does the jobs of two, it promotes tight coupling.
- Break one class into two, e.g., having the feather details as part of the `Bird` class is not a realistic OO model and breaks the Single Responsibility design principle.
  - Can refactor this into two separate classes, each with appropriate responsibility.

```java
public abstract class Bird extends Animal {
    private String featherType;
    private String featherColor;

    public Bird(String name) { super(name); }
}
```

```java
public abstract class Bird extends Animal {
    private Feather feather;

    public Bird(String name) { super(name); }
}
public class Feather {
    private String featherType;
    private String featherColor;
}
```

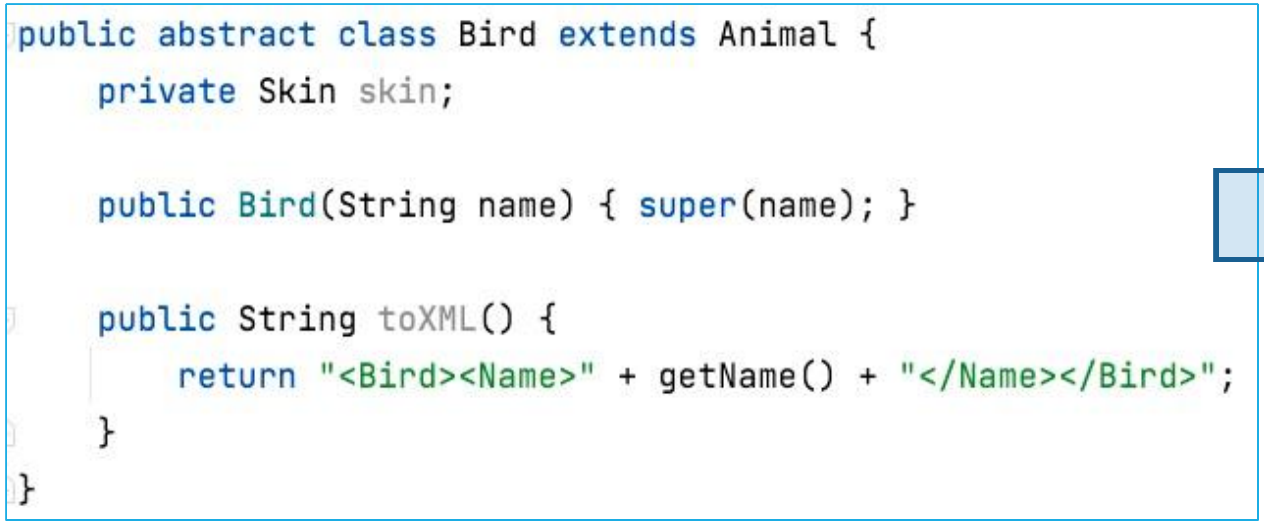
Generalization with Extract Interface

- Extract an interface from a class. Some visitor may need to know a Bird's name, while others may only need to know that certain objects can be serialized to XML. Having `toXml()` as part of the Bird interface breaks the Interface Segregation design principle which tells us that it's better to have more specialized interfaces than to have one multi-purpose interface.

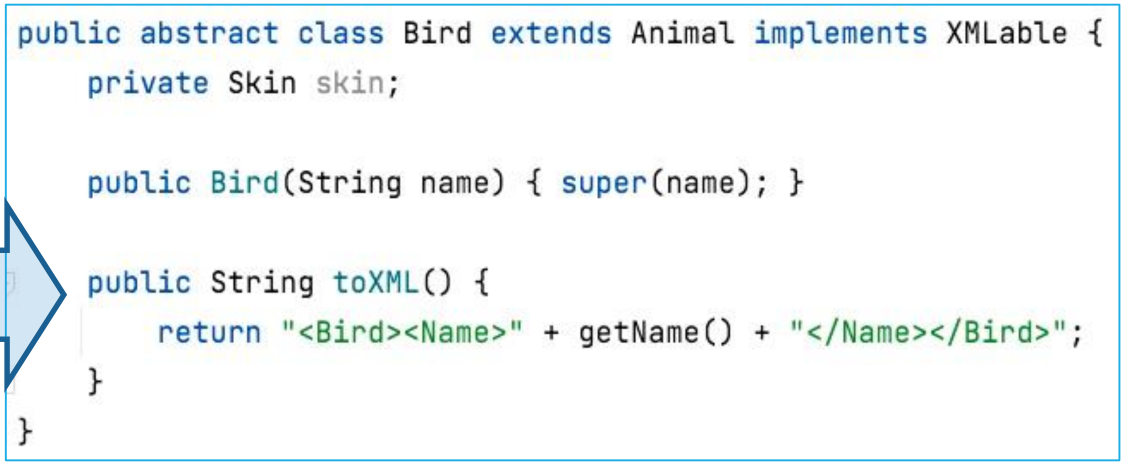
- How to do this:

1. Create an empty interface.
2. Declare common operations in the interface.
3. Declare necessary classes as implementing the interface.
4. Change type declarations in the client code to use the new interface.

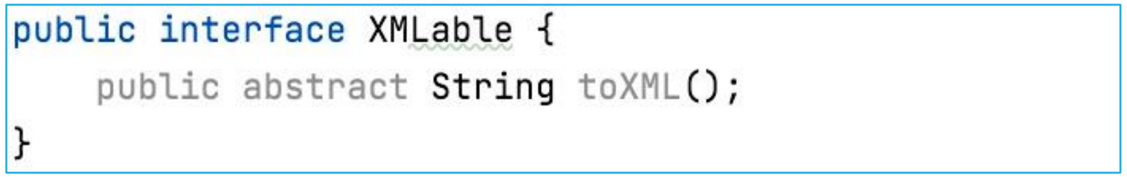
Extract Interface



```java
public abstract class Bird extends Animal {
    private Skin skin;

    public Bird(String name) { super(name); }

    public String toXML() {
        return "<Bird><Name>" + getName() + "</Name></Bird>";
    }
}
```



```java
public abstract class Bird extends Animal implements XMLable {
    private Skin skin;

    public Bird(String name) { super(name); }

    public String toXML() {
        return "<Bird><Name>" + getName() + "</Name></Bird>";
    }
}
```



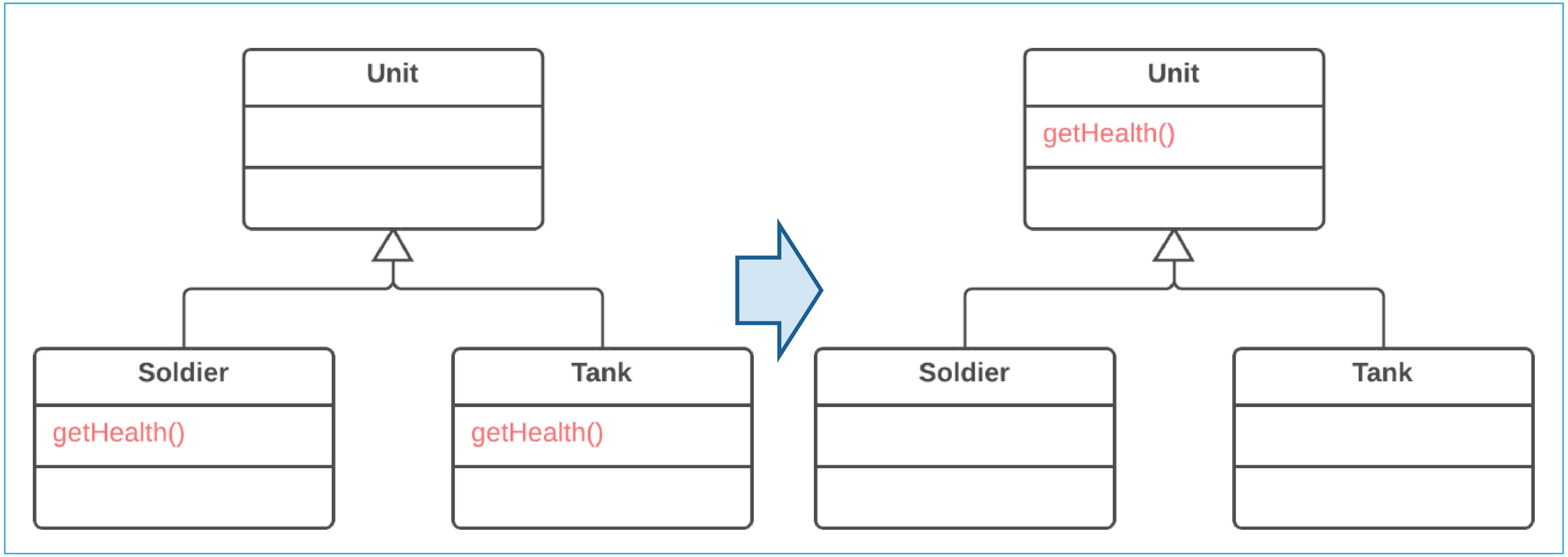
```java
public interface XMLable {
    public abstract String toXML();
}
```

University of Nottingham
UK | CHINA | MALAYSIA

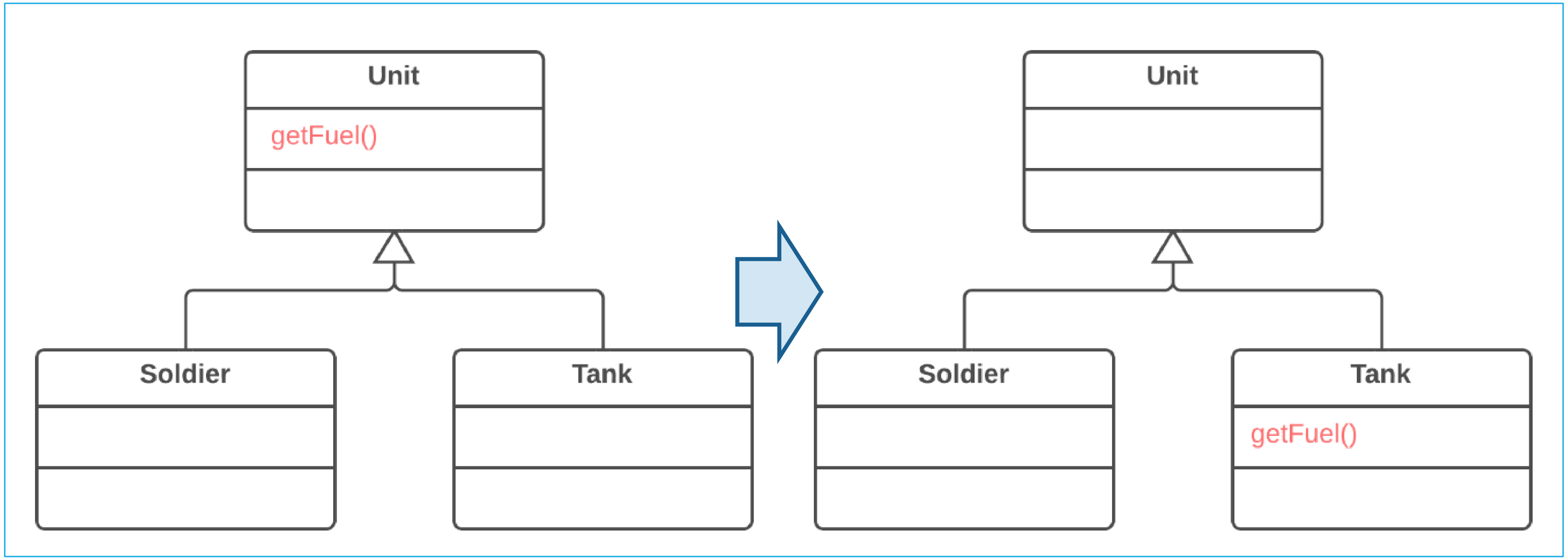
Push Down / Pull Up of Methods

- Eliminating duplicate behavior is a core value of refactoring.
  - "breeding ground" for bugs.
  - Whenever there is duplication, there is the risk that will alter one instance and not the other one.
- 'Pull Up' when have methods of same composition.
  - Often after have refactored a bit to reduce the temporary variables.
  - Use when have a subclass which overrides a superclass method yet does the same thing or after applying Substitute Algorithm to make them identical. https://www.refactoring.com/catalog/formTemplateMethod.html
  - If have two methods which are similar but not identical, use 'Form Template' method.
- Applies to the refactoring of a module hierarchy where a method is duplicated on two or more classes, moving the method up the hierarchy.
- Push down is the opposite, to specialize a class to stop a method implemented in a superclass which is only used by one or small number of subclasses.

Pull Up Methods

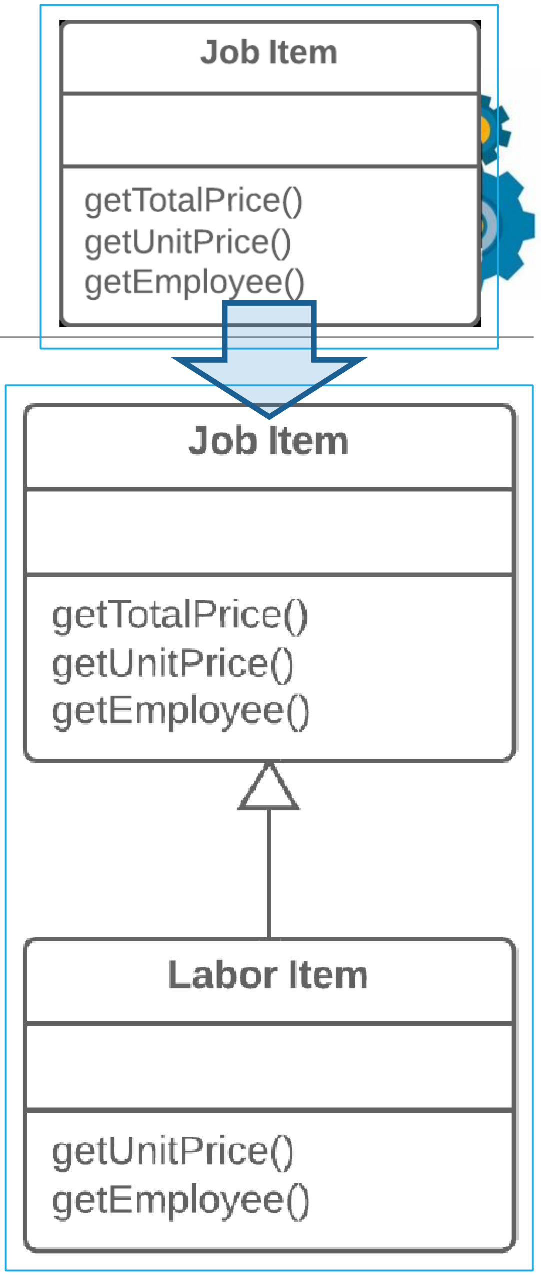


Pull Down Methods

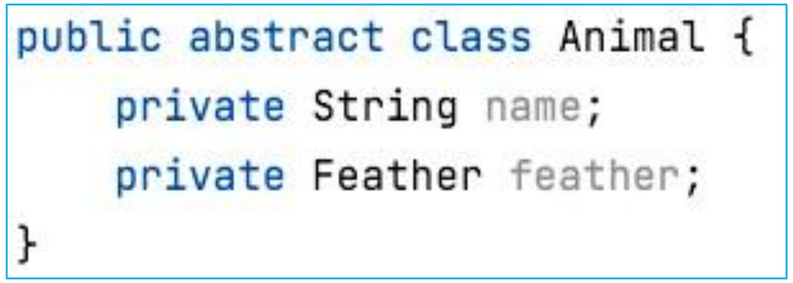


Inherit If Needs To Do with Extract Subclass

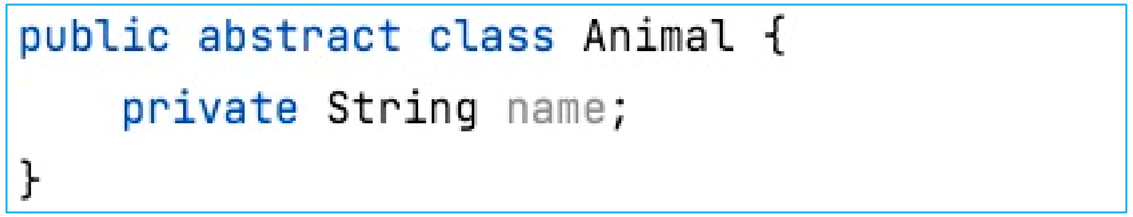
- Have found a class has features (attributes and methods) that would only be useful in specialized instances, can create a specialized of that class and give the class those features.
- This makes the original class less specialized (i.e., more abstract), and good design is about binding to abstractions whenever possible.
- Caution – too much inheritance can promote tight coupling, may want to avoid making a large collection of subclasses.



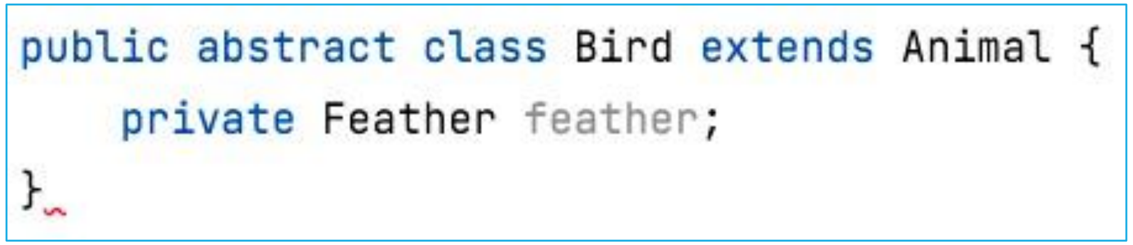
Extract Subclass Example



```java
public abstract class Animal {
    private String name;
    private Feather feather;
}
```



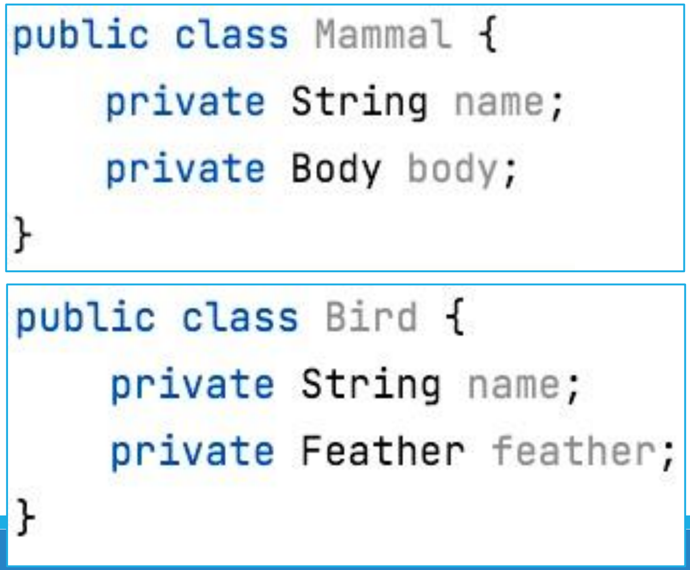
```java
public abstract class Animal {
    private String name;
}
```



```java
public abstract class Bird extends Animal {
    private Feather feather;
}
```

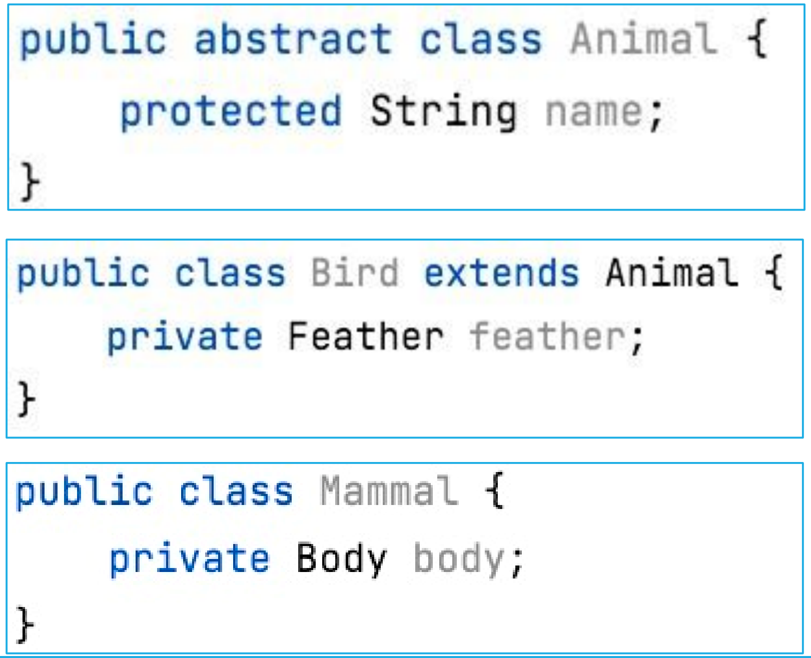
Extract Super Classes

- When there are two or more classes that share common features, consider abstracting those shared features into a superclass.
- This make it easier to produce an abstraction and removes duplicate code from the original classes.
- Not used if superclass already exists.



```
public class Mammal {
    private String name;
    private Body body;
}
```

```
public class Bird {
    private String name;
    private Feather feather;
}
```

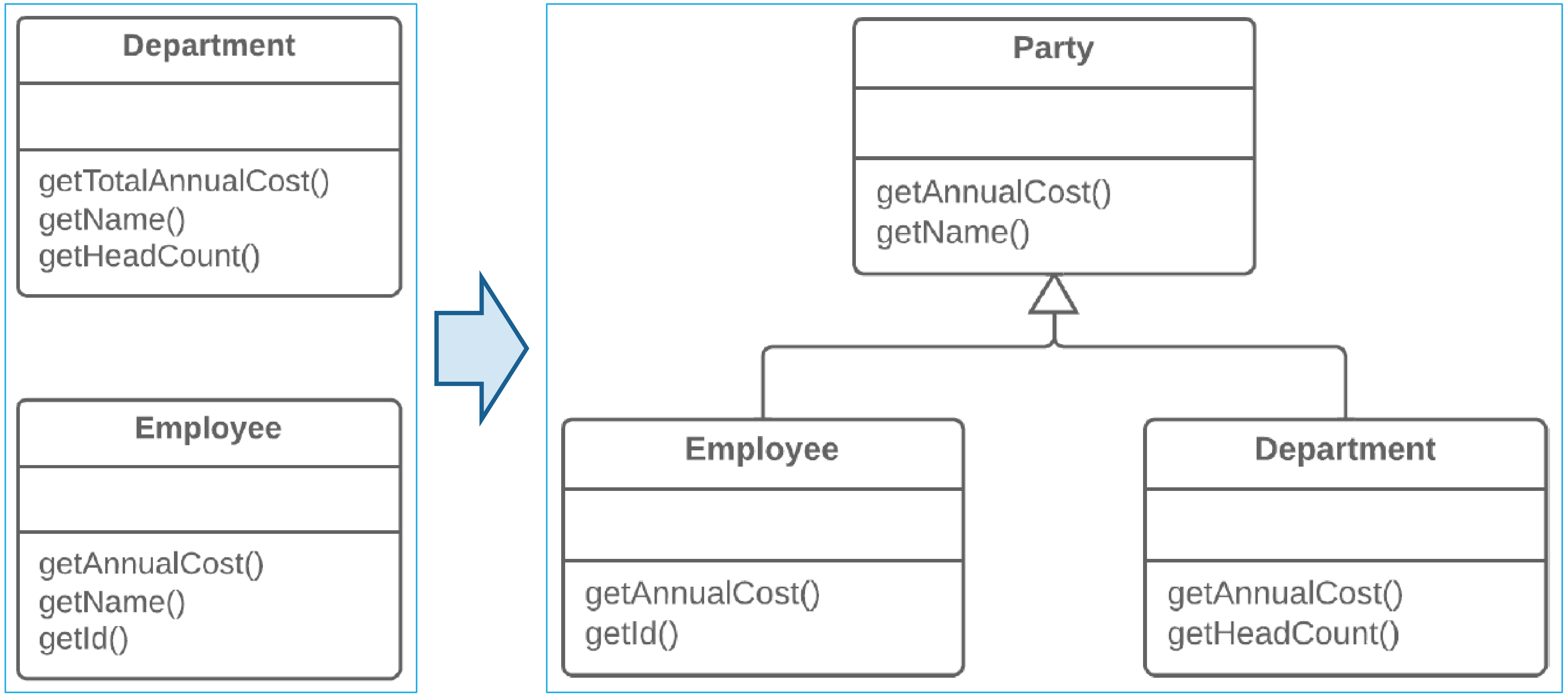


```
public abstract class Animal {
    protected String name;
}
```

```
public class Bird extends Animal {
    private Feather feather;
}
```

```
public class Mammal {
    private Body body;
}
```

Example: Calculates Annual Cost



Apply A Design Pattern In Refactoring

THE FORM TEMPLATE METHOD

University of Nottingham
UK | CHINA | MALAYSIA

Adds Consistencies to Methods to Perform “Pull Up Methods”

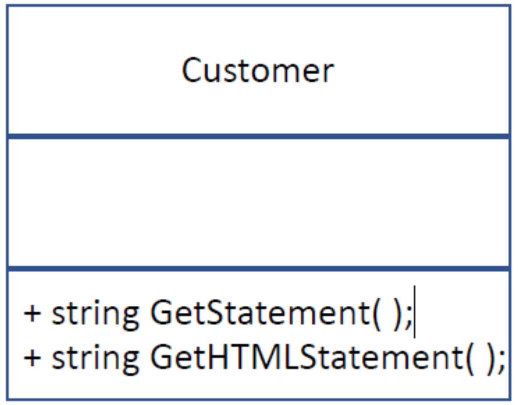
- When there are two methods in subclasses that perform the same steps, but do different things in each step, create methods for those steps with the same signature and move the original method into the base class.
- Subclasses are developed in parallel, sometimes by different people, which leads to code duplication, errors and difficulties in code maintenance.
- Each change must be made in all subclasses – why is this problematic?
- Code duplication doesn’t always refer to cases of simply copy/paste.
  - Often duplication occurs at a higher level, such as when there is a method for sorting numbers and a method for sorting object collections that are differentiated only by the comparison of elements.
- Creating a template method eliminates this duplication by merging the shared algorithm steps in a superclass and leaving just the differences in the subclasses.

Fowler's Famous Video Rental Example

- This example makes use of the classic described in the first chapter of Fowler's refactoring [available in Java, Ruby and Javascript].
- Describes a management system for a video rental store (such place doesn't exist anymore …).
- It contains a `Movie` class, a `Customer` class and a `Rental` class, and prints out statements.
- Performs a series of refactoring tasks on this original codebase.
  - https://martinfowler.com/articles/refactoring-video-store-js/

For more details see https://youtu.be/6wDoopbtEqk

Customer Rental System Before Refactoring



```
public string GetStatement()
{
        var result = new StringBuilder();
        result.AppendLine(”Rental record: “ +Name);
        foreach (var rental in Rentals)
                result.AppendLine(“\t” + rental.Movie.Title);
        result.AppendLine(“Owed:” + TotalCharge.ToString());
        result.AppendLine(“Customer earned:” +
TotalFrequentRenterPoints.ToString()+ “renter points” );
        return result.ToString();
}
```

GetHTMLStatement() Is Surprisingly Similar …

```
public string GetHTMLStatement()
{
        var result = new StringBuilder();
        result.AppendLine("<h1>Rental record:<em> " + Name + "</em></h1>");
        foreach (var rental in Rentals)
                result.AppendLine(rental.Movie.Title + "<br />");
        result.AppendLine("<p> Owed: <em>" + TotalCharge.ToString() + "<em></p>");
        result.AppendLine("<p> Customer earned: <em> " +
                        TotalFrequentRenterPoints.ToString()+ "</em>renter
points</p>");
        return result.ToString();
}
```

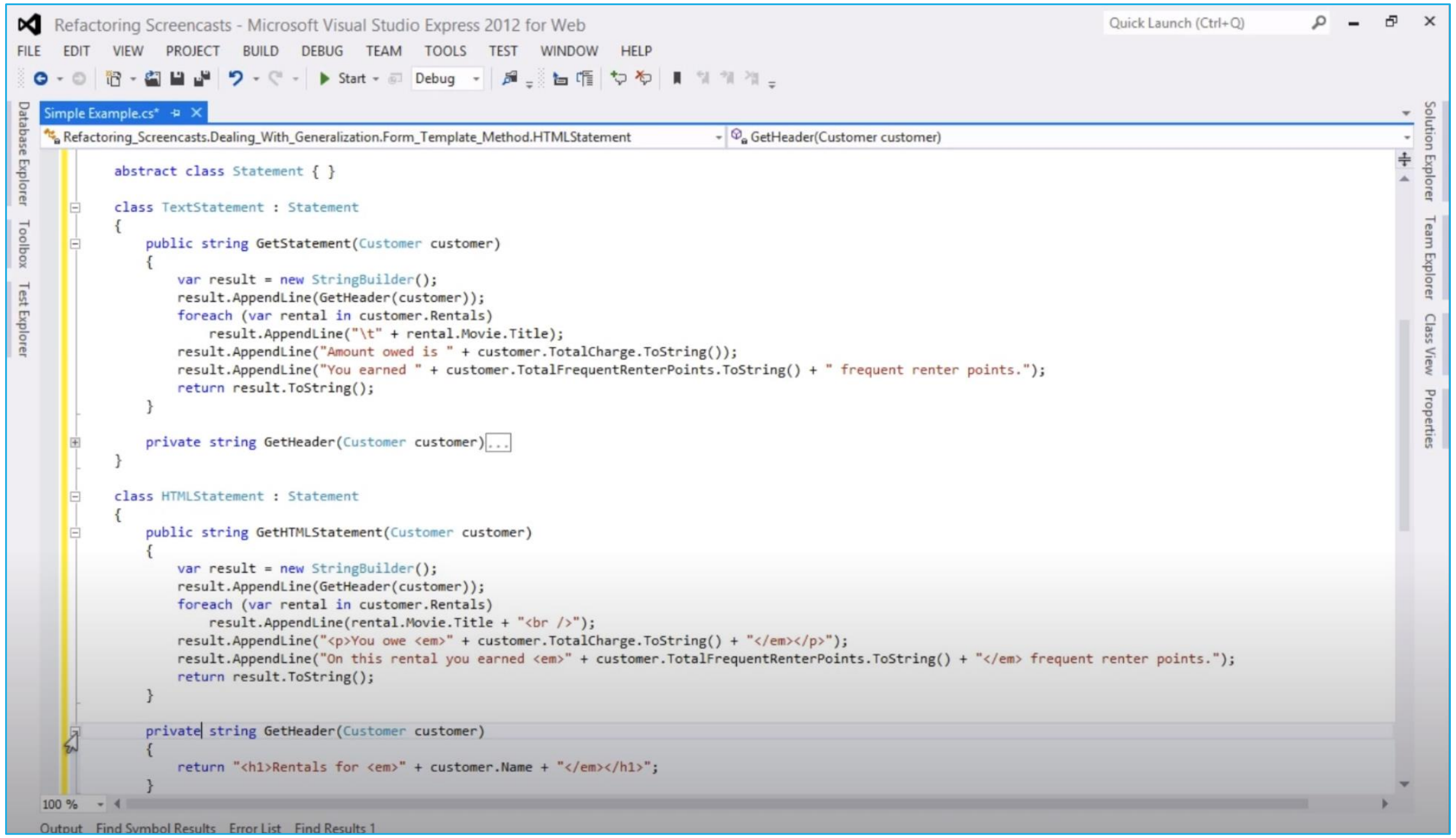
Mechanics of the Form Template Method

1. Decompose the methods so that the extracted methods are identical.
2. Apply Pull Up of the identical methods into the superclass (if using inheritance) or the class that extends the module.
3. Test after each pull up/extraction.
4. Use the same set of method calls, but the subclasses/modules handle the method calls differently.
5. Test again.
6. Apply Pull Up to the original method.
7. Test again, then remove the extraneous methods, test after each removal.

University of Nottingham
UK | CHINA | MALAYSIA

https://youtu.be/Oz4_45DD8qU

Example by David Donahue (C#)



Refactoring Screencasts - Microsoft Visual Studio Express 2012 for Web
Quick Launch (Ctrl+Q)
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST WINDOW HELP
Start
Debug
Simple Example.cs*
Refactoring_Screencasts.Dealing_With_Generalization.Form_Template_Method.HTMLStatement
GetHeader(Customer customer)
Database Explorer
Toolbox
Test Explorer
Solution Explorer
Team Explorer
Class View
Properties
100 %
Output Find Symbol Results Error List Find Results 1

```
abstract class Statement { }

class TextStatement : Statement
{
    public string GetStatement(Customer customer)
    {
        var result = new StringBuilder();
        result.AppendLine(GetHeader(customer));
        foreach (var rental in customer.Rentals)
            result.AppendLine("\t" + rental.Movie.Title);
        result.AppendLine("Amount owed is " + customer.TotalCharge.ToString());
        result.AppendLine("You earned " + customer.TotalFrequentRenterPoints.ToString() + " frequent renter points.");
        return result.ToString();
    }

    private string GetHeader(Customer customer)...
}

class HTMLStatement : Statement
{
    public string GetHTMLStatement(Customer customer)
    {
        var result = new StringBuilder();
        result.AppendLine(GetHeader(customer));
        foreach (var rental in customer.Rentals)
            result.AppendLine(rental.Movie.Title + "<br />");
        result.AppendLine("<p>You owe <em>" + customer.TotalCharge.ToString() + "</em></p>");
        result.AppendLine("On this rental you earned <em>" + customer.TotalFrequentRenterPoints.ToString() + "</em> frequent renter points.");
        return result.ToString();
    }

    private string GetHeader(Customer customer)
    {
        return "<h1>Rentals for <em>" + customer.Name + "</em></h1>";
    }
```

Note on ‘Big Refactoring’

- Little refactoring is about making many almost imperceptible changes to code.
- Big refactoring is a team wide effort involving major redesigns.
- Four Big Refactorings:
  - Tease apart inheritance to promote encapsulation, separating out functionality.
  - Convert procedural design features to object-oriented (beyond variables) by applying design patterns or SOLID principles.
  - Separate domain from presentation by, i.e., using the MVC or MVVM pattern.
  - Extract a hierarchy by creating a hierarchy of classes with each subclass representing a specific case.
- Massive risk of breaking an entire codebase if careful testing is not applied during these challenging processes – but the payoff in reducing future maintenance costs is huge: ***REMEMBER, software engineering is about making money from code***.

Employee Management System

REFACTORING EXAMPLE

Original Code

```
public class Employee {
    private String name;
    private int employeeId;
    private String department;
    private double monthlySalary;
    private double yearlyBonus; // Temporary Field
    private int workHours;
    private double hourlyRate; // Temporary Field
    private String position;

    public Employee(String name, int employeeId, String department, double monthlySalary, String position) {
        this.name = name;
        this.employeeId = employeeId;
        this.department = department;
        this.monthlySalary = monthlySalary;
        this.position = position;
    }
```

Original Code (Continued ...)

```
// Long Method
public double calculateAnnualCompensation(boolean includeBonus) {
    double annualSalary = monthlySalary * 12;
    if (includeBonus) {
        yearlyBonus = monthlySalary * 0.1; // Hardcoded and speculative
        annualSalary += yearlyBonus;
    }
    return annualSalary;
}

// Duplicate Code for calculating hourly wage
public double calculateHourlyWage() {
    hourlyRate = monthlySalary / (4 * 40); // 4 weeks and 40 hours per week
    return hourlyRate;
}
```

Original Code (Continued ...)

```
// Long Method, Comment-Based Cover-Up, Large Parameter List
    public String getEmployeeReport(boolean includeSalary, boolean includeDepartment, boolean
includePosition, boolean includeWorkHours) {
        String report = "Employee Report for " + name + " (" + employeeId + ")\n";
        if (includeSalary) {
            report += "Salary: " + monthlySalary + "\n";
        }
        if (includeDepartment) {
            report += "Department: " + department + "\n";
        }
        if (includePosition) {
            report += "Position: " + position + "\n";
        }
        if (includeWorkHours) {
            report += "Work Hours: " + workHours + "\n";
        }
        // Comment: "More details can be added here later"
        return report;
    }
```

Original Code (Continued ...)

```
    // Speculative Generality
    public void speculativeMethod() {
        // Unused method, anticipating future use
    }

    // Data Class (Refused Bequest)
    public String getName() {
        return name;
    }

    public int getEmployeeId() {
        return employeeId;
    }

    public String getDepartment() {
        return department;
    }
}
```

Issues!

- **Duplicate Code**: `calculateHourlyWage()` and `calculateAnnualCompensation`() both deal with salary-related calculations.
- **Long Methods**: `calculateAnnualCompensation()` and `getEmployeeReport`() have too much going on.
- **Large Class**: The `Employee` class is doing too much, including calculation, reporting, etc.
- **Temporary Fields**: `yearlyBonus` and `hourlyRate` are only used for short calculations.
- **Speculative Generality**: The unused `speculativeMethod()` was added prematurely.
- **Refused Bequest/Data Class**: The `Employee` class is mostly storing data but not using its fields in any meaningful way.
- **Comment-Based Cover-Up**: Comments are hiding the incomplete logic in `getEmployeeReport()`.

Refactored (Employee)

```
public class Employee {
    private String name;
    private int employeeId;
    private String department;
    private double monthlySalary;
    private String position;
    private Compensation compensation; // Delegating compensation calculations to another class

    public Employee(String name, int employeeId, String department, double monthlySalary, String position) {
        this.name = name;
        this.employeeId = employeeId;
        this.department = department;
        this.monthlySalary = monthlySalary;
        this.position = position;
        this.compensation = new Compensation(monthlySalary); // Encapsulation of salary-related fields
    }

    // Simplifying Method Call
    public String getEmployeeReport() {
        ReportGenerator reportGenerator = new ReportGenerator();
        return reportGenerator.generateReport(this);
    }
```

Refactored (Employee) Continued …

```
    // Data Class now only focuses on core attributes
    public String getName() {
        return name;
    }

    public int getEmployeeId() {
        return employeeId;
    }

    public String getDepartment() {
        return department;
    }

    public String getPosition() {
        return position;
    }

    public double getMonthlySalary() {
        return monthlySalary;
    }
}
```

Refactored (Compensation)

```
public class Compensation {
    private double monthlySalary;
    private final int MONTHS = 12;
    private final double BONUS_RATE = 0.1;
    private final int WEEKS = 4;
    private final int HOURS = 40;

    public Compensation(double monthlySalary) {
        this.monthlySalary = monthlySalary;
    }

    // Composed Method: Generalized salary calculation
    public double calculateAnnualCompensation(boolean includeBonus) {
        double annualSalary = monthlySalary * MONTHS;  // Replaced with constant number
        if (includeBonus) {
            double yearlyBonus = calculateBonus(); // Moving bonus calculation to its own method
            annualSalary += yearlyBonus;
        }
        return annualSalary;
    }

    private double calculateBonus() {
        return monthlySalary * BONUS_RATE; // Generalize hardcoded value & replaced with constant number
    }

    public double calculateHourlyWage() {
        return monthlySalary / (WEEKS * HOURS); // Simplified without temporary field & replaced with constant number
    }
}
```

Refactored (ReportGenerator)

```
public class ReportGenerator {
    // Composed Method, removed large parameter list
    public String generateReport(Employee employee) {
        StringBuilder report = new StringBuilder();
        report.append("Employee Report for ").append(employee.getName()).append(" (")
              .append(employee.getEmployeeId()).append(")\n");

        addDetail(report, "Salary", String.valueOf(employee.getMonthlySalary()));
        addDetail(report, "Department", employee.getDepartment());
        addDetail(report, "Position", employee.getPosition());

        return report.toString();
    }

    // Helper Method to add details to the report
    private void addDetail(StringBuilder report, String label, String value) {
        report.append(label).append(": ").append(value).append("\n");
    }
}
```

Refactored Explains

- **Duplicate Code**:
  - **Solution**: Consolidated salary-related logic in the `Compensation` class (`calculateBonus`, `calculateAnnualCompensation`, `calculateHourlyWage`).
- **Long Methods**:
  - **Solution**: Split large methods into smaller, more focused methods. For example, `calculateAnnualCompensation()` now delegates bonus calculation to `calculateBonus()`, and report generation is simplified using `addDetail()`.
- **Large Classes**:
  - **Solution**: Moved compensation and report generation responsibilities out of `Employee` into dedicated classes (`Compensation`, `ReportGenerator`).
- **Temporary Fields**:
  - **Solution**: Removed temporary fields like `yearlyBonus` and `hourlyRate`, avoiding unnecessary field assignments by directly returning calculated values.

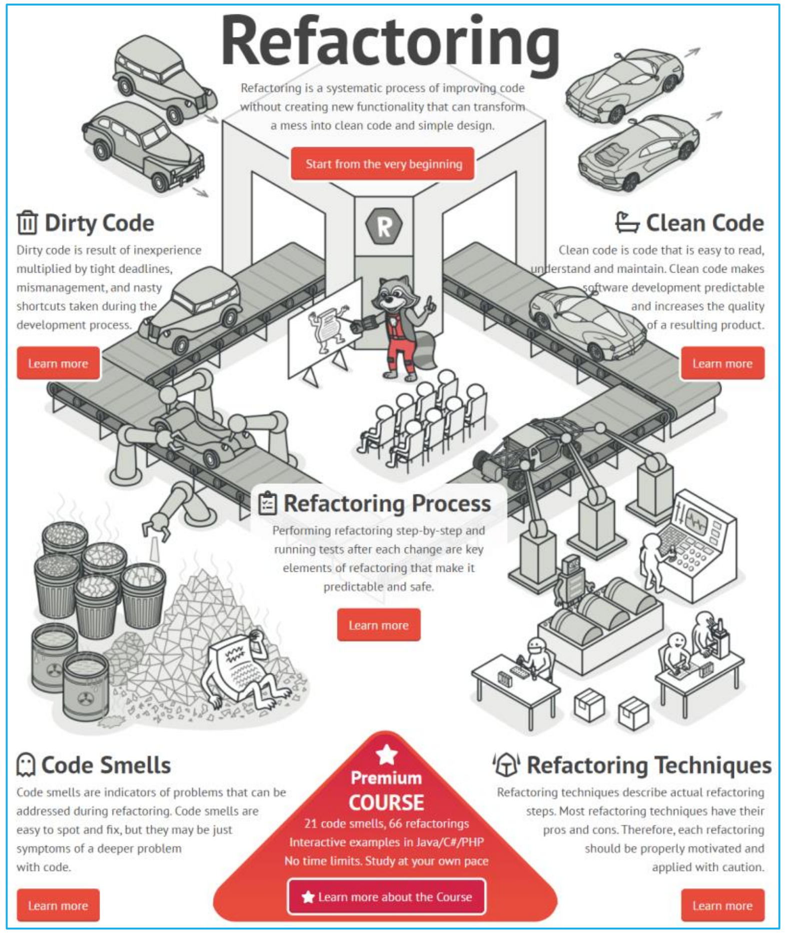
University of Nottingham
UK | CHINA | MALAYSIA

Refactored Explains

- **Speculative Generality**:
  - **Solution**: Removed unused speculative methods like `speculativeMethod()`.
- **Refused Bequest/Data Class**:
  - **Solution**: Focused the Employee class on core responsibilities, making it a well-encapsulated data class. Moved extraneous logic out into specialized classes.
- **Comment-Based Cover-Up**:
  - **Solution**: Removed comments by implementing the necessary logic to generate a clean, detailed employee report.
- **Magic Number**:
  - **Solution**: Replace the fixed number with constant value.

More Examples to Explore

- Refer to Fowler’s book – 1st Edition Java, 2nd Edition Javascript.
- Check out Refactoring Guru where all the techniques in the book are explained one by one.
  - https://refactoring.guru/refactoring
- Make sure you watch the video on the Form Template Method.



Refactoring

Refactoring is a systematic process of improving code without creating new functionality that can transform a mess into clean code and simple design.

Start from the very beginning

Dirty Code

Dirty code is result of inexperience multiplied by tight deadlines, mismanagement, and nasty shortcuts taken during the development process.

Learn more

Clean Code

Clean code is code that is easy to read, understand and maintain. Clean code makes software development predictable and increases the quality of a resulting product.

Learn more

Refactoring Process

Performing refactoring step-by-step and running tests after each change are key elements of refactoring that make it predictable and safe.

Learn more

Code Smells

Code smells are indicators of problems that can be addressed during refactoring. Code smells are easy to spot and fix, but they may be just symptoms of a deeper problem with code.

Learn more

Premium COURSE

21 code smells, 66 refactorings

Interactive examples in Java/C#/PHP

No time limits. Study at your own pace

Learn more about the Course

Refactoring Techniques

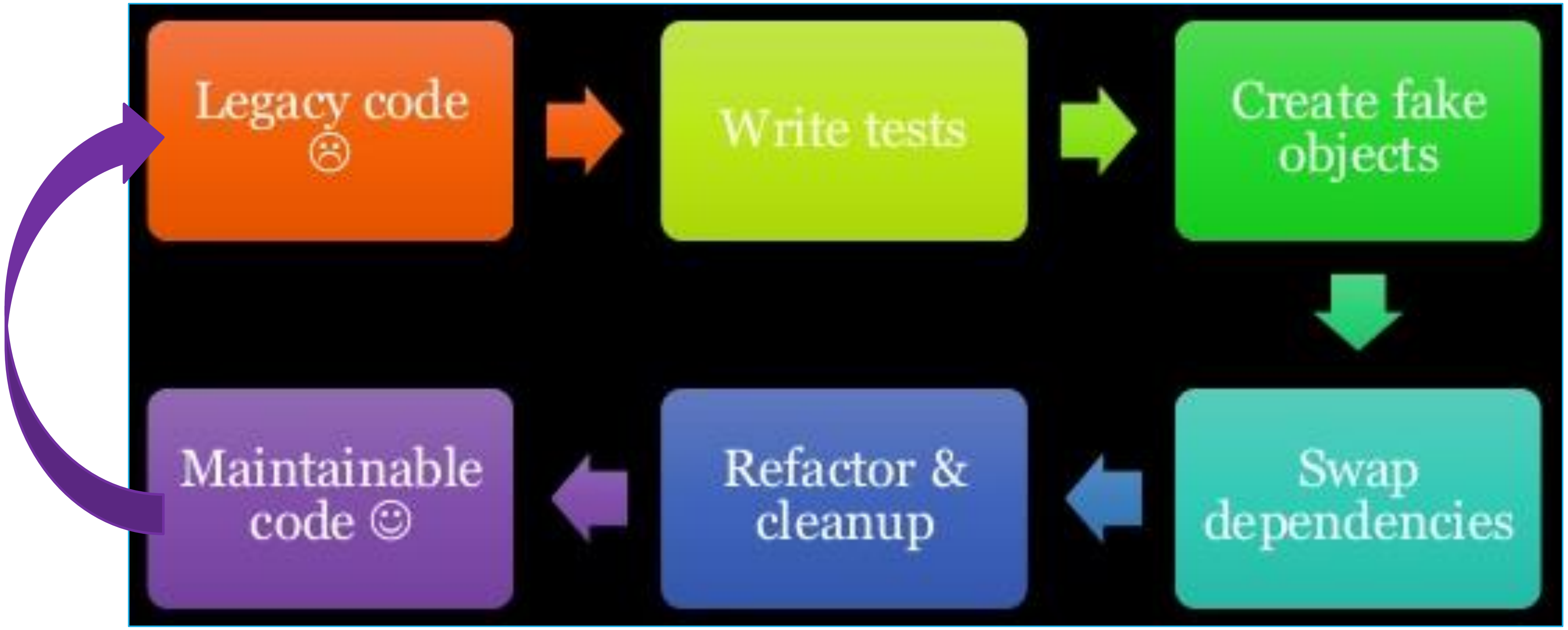
Refactoring techniques describe actual refactoring steps. Most refactoring techniques have their pros and cons. Therefore, each refactoring should be properly motivated and applied with caution.

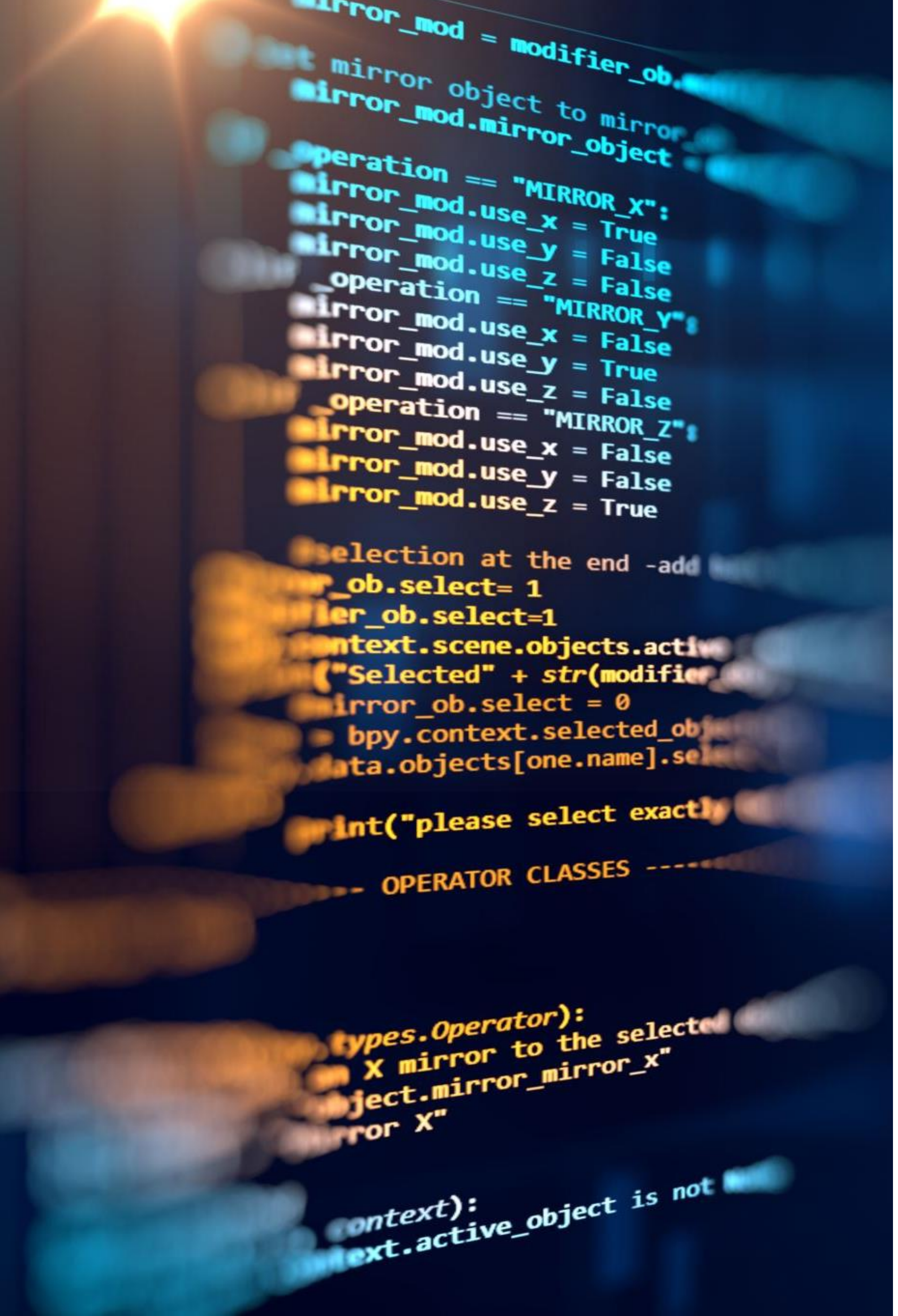
Learn more

Refactoring to Avoid Technical Debt

- No one sets out to write 'bad' code – though the agile principles of minimal design upfront and release early, often teach us to focus on delivery rather than clean code.
- Technical debt is the same as consumer debt, not following the practice of testing and refactoring over time means that the blob and spaghetti code build up.
- Can speed up the development without explicit testing but has consequences.
  - Business pressure to release new features all the time.
  - Monolithic code rather than decomposed modules.
  - Lack of testing and documentation.
  - Lack of interaction between team members.
  - Lack of compliance monitoring.

Recap, This is An Iterative Process





```
irror_mod = modifier_ob.
mirror object to mirror
irror_mod.mirror_object
peration == "MIRROR_X":
irror_mod.use_x = True
irror_mod.use_y = False
irror_mod.use_z = False
 operation == "MIRROR_Y":
irror_mod.use_x = False
irror_mod.use_y = True
irror_mod.use_z = False
 operation == "MIRROR_Z":
irror_mod.use_x = False
irror_mod.use_y = False
irror_mod.use_z = True

election at the end -add
_ob.select= 1
er_ob.select=1
ntext.scene.objects.active
("Selected" + str(modifier
irror_ob.select = 0
 bpy.context.selected_ob
ata.objects[one.name].se

int("please select exactly

 OPERATOR CLASSES ----

types.Operator):
 X mirror to the selected
ject.mirror_mirror_x"
ror X"

context):
ext.active_object is not
```

Self Study Quick Questions

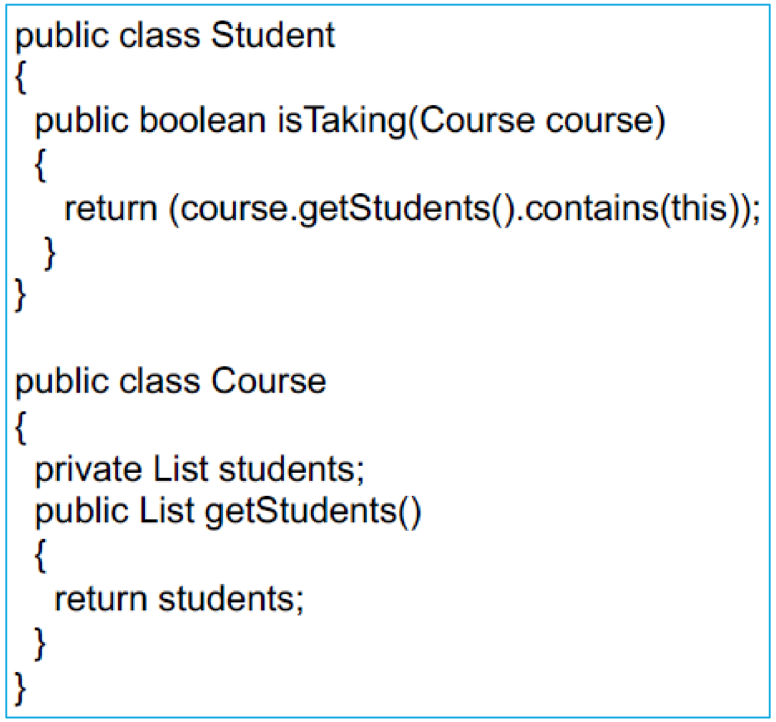
- Legacy code is …
- Code smells are …
- Refactoring is …
- Testing works with refactoring because …
- The extract method works by …
- Replace magic numbers with …
- The Form Template Method uses …

Extras

FOR REFACTORING SKILLS IN MORE DEPTH

Extras: Move Method (Original)

- If a method on one class uses (or is used by) another class more than the class on which it is defined, move it to other class.



```
public class Student
{
  public boolean isTaking(Course course)
  {
     return (course.getStudents().contains(this));
   }
}

public class Course
{
  private List students;
  public List getStudents()
  {
    return students;
  }
}
```

Extras: Move Method (Refactored)

- The `Student` class now no longer needs to know about the `Course` interface, and the `isTaking()` method is closer to the data on which it relies – making the design of `Course` more cohesive and the overall design more loosely coupled.

```
public class Student
{
}

public class Course
{
  private List students;
  public boolean isTaking(Student student)
  {
     return students.contains(student);
  }
}
```

Extras: Introduce `null` Object

- If relying on `null` for default behaviour, use inheritance instead.

```
public class User
{
  Plan getPlan()
  {
      return plan;
  }
}
```

```
if (user == null)
    plan = Plan.basic();
else
   plan = user.getPlan();
```

```
public class User
{
  Plan getPlan()
  {
    return plan;
  }
}

public class NullUser extends User
{
  Plan getPlan()
  {
     return Plan.basic();
  }
}
```

Extras: Replace Error Code with Exception

- A method returns a special code to indicate an error is better accomplished with an `Exception`.

```
int withdraw(int amount)
{
  if (amount > balance)
                return -1;
  else {
                balance -= amount;
                return 0;
         }
}
```

```
void withdraw(int amount)
   throws BalanceException
{
  if (amount > balance)
  {
                throw new BalanceException();
   }
  balance -= amount;
}
```

Self-Test

- Try out some examples HERE, see if you know how to perform refactoring.

DID YOU FINISH THE SOFTWARE YET?
NO, I'M STILL PAYING OFF THE TECHNICAL DEBT FROM THE LAST PROGRAMMER YOU RUSHED.
I DON'T KNOW WHAT THAT MEANS.
WELL, THAT EXPLAINS A LOT.
Dilbert.com @ScottAdamsSays
1-3-17 © 2017 Scott Adams, Inc. /Dist. by Andrews McMeel

University of Nottingham
UK | CHINA | MALAYSIA

Put Your Mind in Maintenance Mode