Smells / Basic Refactorings

Michael Inden

Agenda

- Motivation and Intro (Why Software rots)
- Good / Bad Design & Design Smells
- Typical Smells in Code
- Intro Base Refactorings (Fowler)
- Catalog of Refactoring techniques
- Take Aways & Pitfalls

Motivation and Intro

Why is Software rotting?

- Code rapidly rots in presence of change
- Todays (agile) Software Development processes require many code changes.
- As in your household / flat: if you don’t clean up often it will get messier and messier
- Example: Implement a class „Copier“ for copying keyboard input to printer

Why is Software rotting?

- First, simple (a bit stupid) prototype implementation

```
public class Copier {
      public void copy() {
            int ch = Keyboard.read();
            while (ch != -1) {
                  Printer.write(ch);
                  ch = Keyboard.read();
            }
      }
}
```

Why is Software rotting?

- First, simple (a bit stupid) prototype implementation

```
public class Copier {
      public void copy() {
            int ch = Keyboard.read();
            while (ch != -1) {
                  Printer.write(ch);
                  ch = Keyboard.read();
            }
      }
}
```

- NEW REQUIREMENT: Extend to be able to write alternatively to Console

Why is Software rotting?

- Fast modification

```
public class Copier
{
    public boolean writeToConsole = false;

    public void copy()
    {
        int ch = Keyboard.read();
        while (ch != -1)
        {
            if (writeToConsole)
                Console.write(ch);
            else
                Printer.write(ch);

            ch = Keyboard.read();
        }
    }
}
```

Why is Software rotting?

- NEW REQUIREMENT: Extend to be able to read from file
- NEW REQUIREMENT: Extend to be able to write alternatively to file
- …
- => a lot more special cases and your software gets bigger and messier and unmaintainable with every step

How to avoid rotting Software Design

- What can we do against it?
- How to avoid that software starts to rot?

Why is Software rotting?

- **Iterative MERCILESS Clean up / Refactoring**
  - Not in an extra refactoring iteration
  - Not at the end of each iteration
  - Not every Friday
  - Whenever the Software has to change!
  - Continuously during development!

- **Design is a process, not an initial step!**

Refactoring
What is the definition???

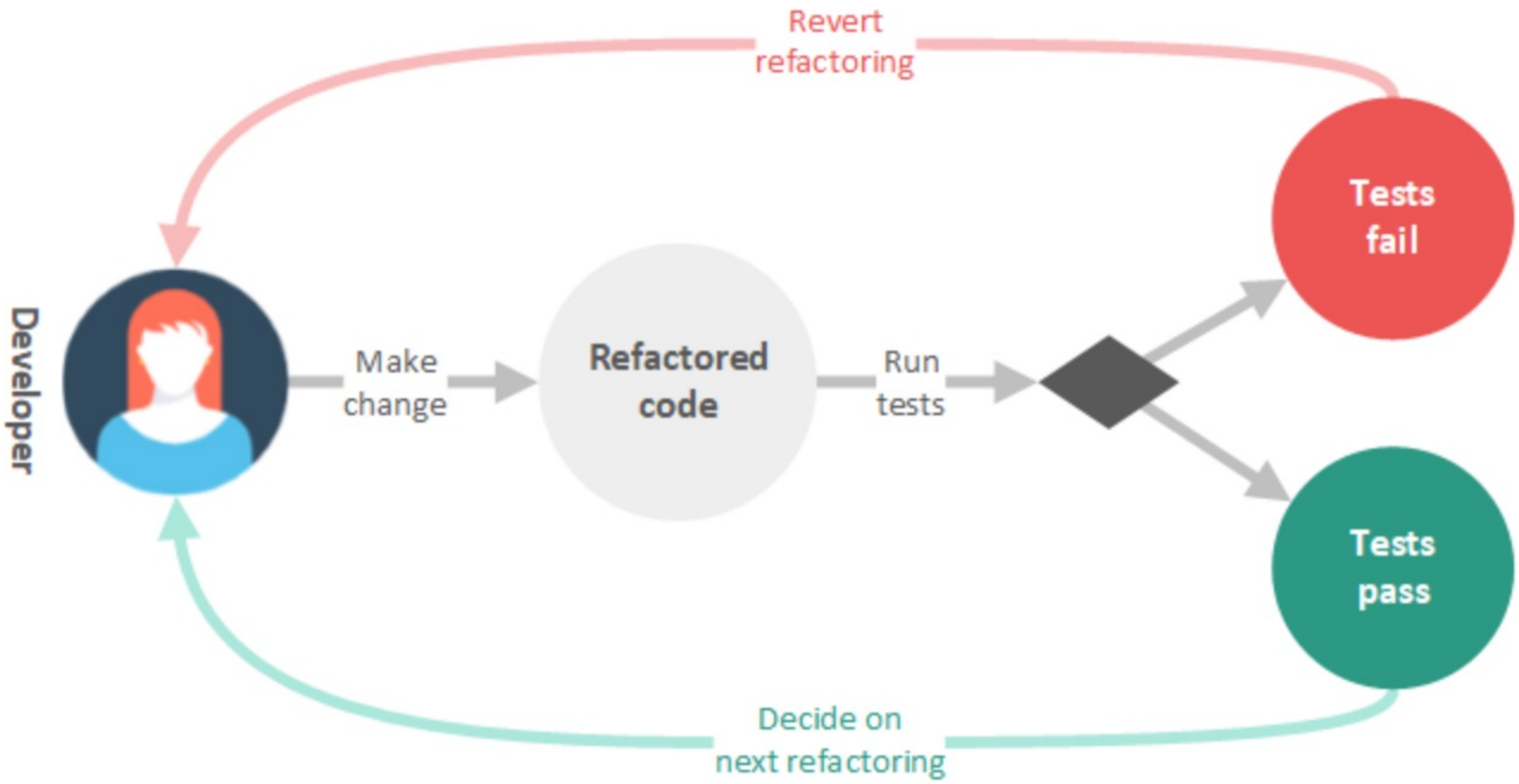
Definition Refactoring

Refactoring is a **systematic process** of **improving code**. This takes places **without changing observable behavior or creating new functionality**. Refactoring transforms a **mess into clean code and simple design**.

Refactorings at first glance

```
private static void badReadability(int b, int h)
{
    double x = 2 * (b + h);
    System.out.println("Umfang: " + x);
    x = b * h;
    System.out.println("Fläche: " + x);
}

private static void goodReadability(int breite, int höhe)
{
    double umfang = 2 * (breite + höhe);
    System.out.println("Umfang: " + umfang);
    double fläche = breite * höhe;
    System.out.println("Fläche: " + fläche);
}

private static void inlined(int breite, int höhe)
{
    System.out.println("Umfang: " + (double) (2 * (breite + höhe)));
    System.out.println("Fläche: " + (double) (breite * höhe));
}
```

Refactoring in Practice



Source: https://dzone.com/articles/what-is-refactoring

Good / Bad Design

What is Good / Bad Design?

How do you recognize bad design?

- the source code is **not very comprehensible**, among other things caused by **unfavorable naming** or **missing documentation** of complicated parts.
- There is **unnecessary complexity** or an **extremely flexible design** with a high number of variants, although these extras are (partly) not needed.
- **Extensions** or **modifications** can be accomplished only **with difficulty** and **effect** partially large (not related) **other parts** of the source code.
- **The problem to be solved cannot be derived from the implementation**, for example because semantic structuring is missing or no clear layout of the source code is present.

How do you recognize bad design?

- there are **only a few tests** and **testability is hardly given**, e.g. because there are nearly no classes that can be tested individually and objects are strongly dependent on each other, so that there are only clusters of objects that are difficult to test.

- **various bugs are already known** and probably many more are hidden. Sometimes the system is so rotten that only a clean-up effort or a complete redesign can help.

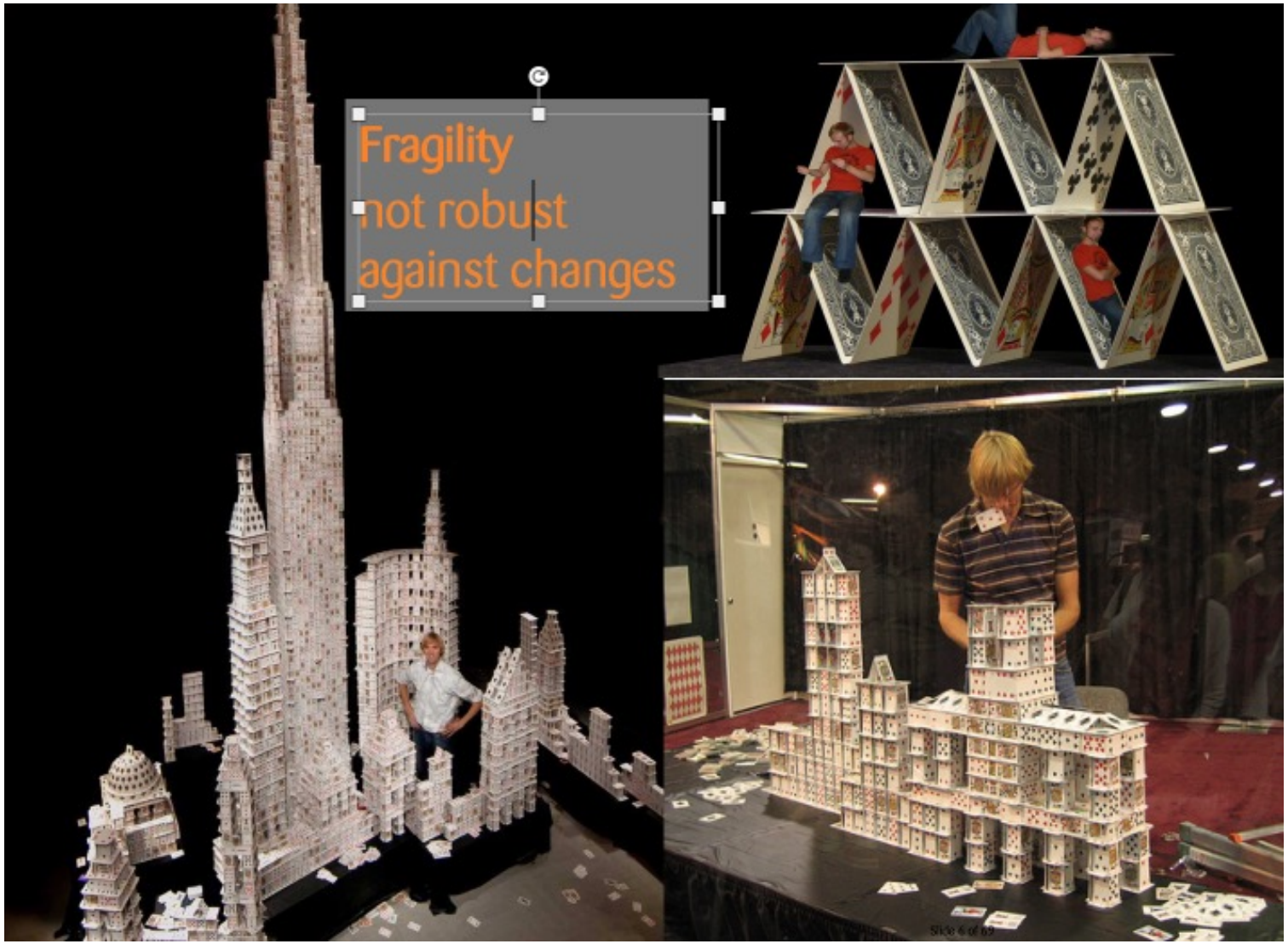
Design Smells (by Robert C. Martin)

- **Rigidity:** difficult to change, chain of changes
- **Fragility:** not robust against changes
- **Immobility:** parts not reusable
- **Viscosity:** easy to do the wrong thing
- **Needless Complexity**: contains unused elements
- **Needless Repetition**: no reuse, copy & paste
- **Opacity**: difficult to understand

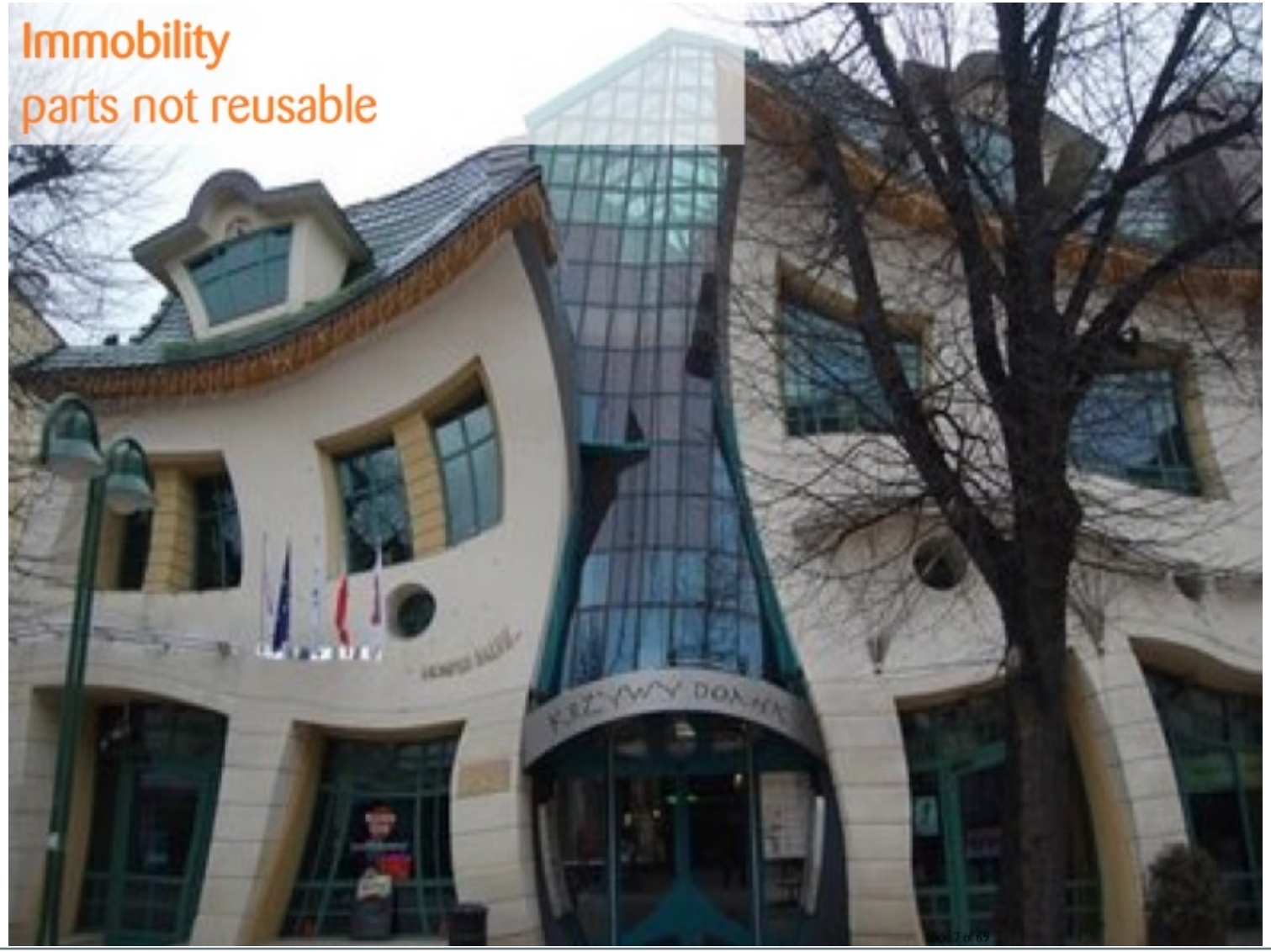
Design Smells (by Robert C. Martin)

- Rigidity: difficult to change, chain of changes

Design Smells (by Robert C. Martin)



Design Smells (by Robert C. Martin)



Design Smells (by Robert C. Martin)

Design Smells (by Robert C. Martin)


Needless Repetition
no reuse, copy & paste

D

Opacity
difficult to understand

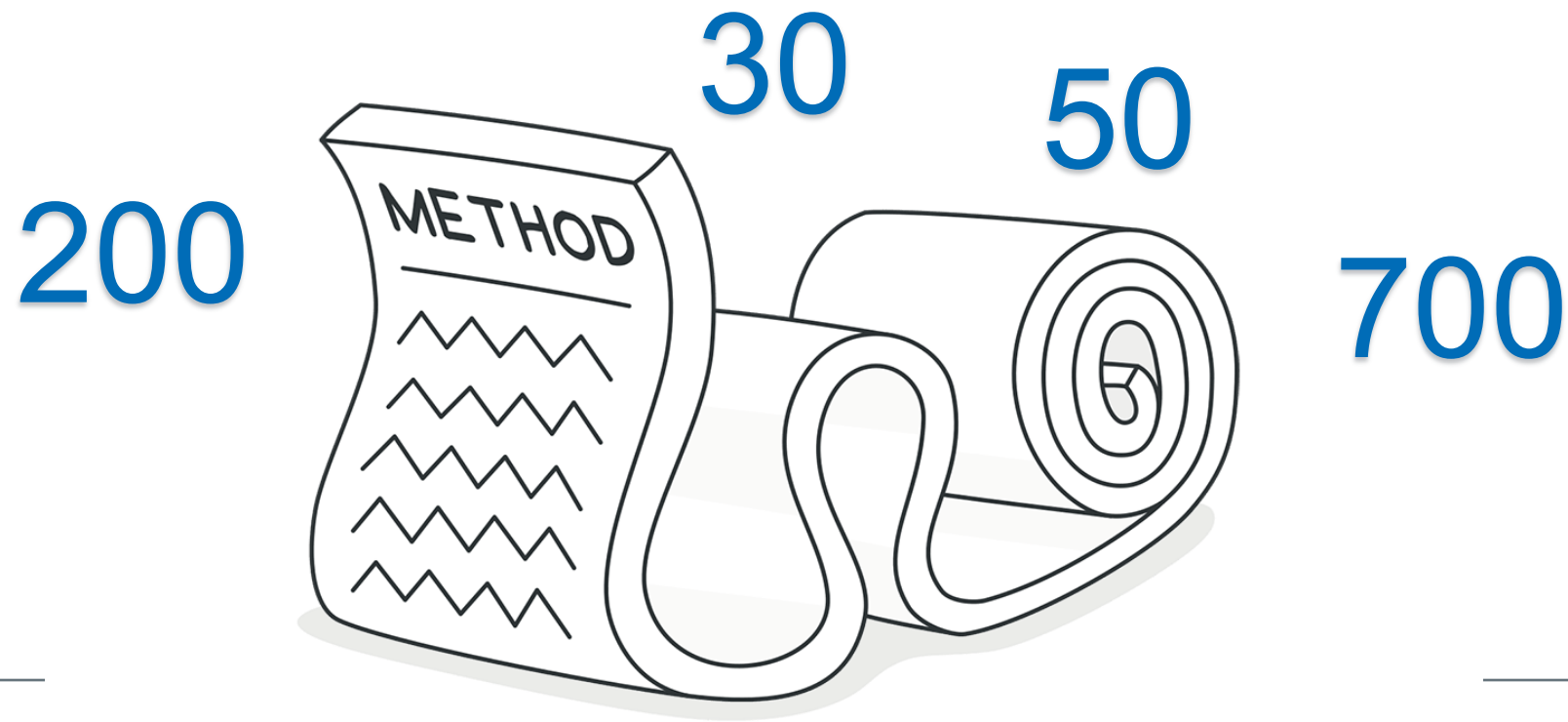
Typical Smells in Code

Long Method

- https://refactoring.guru/smells/long-method

Signs and Symptoms

A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions.



Long Method

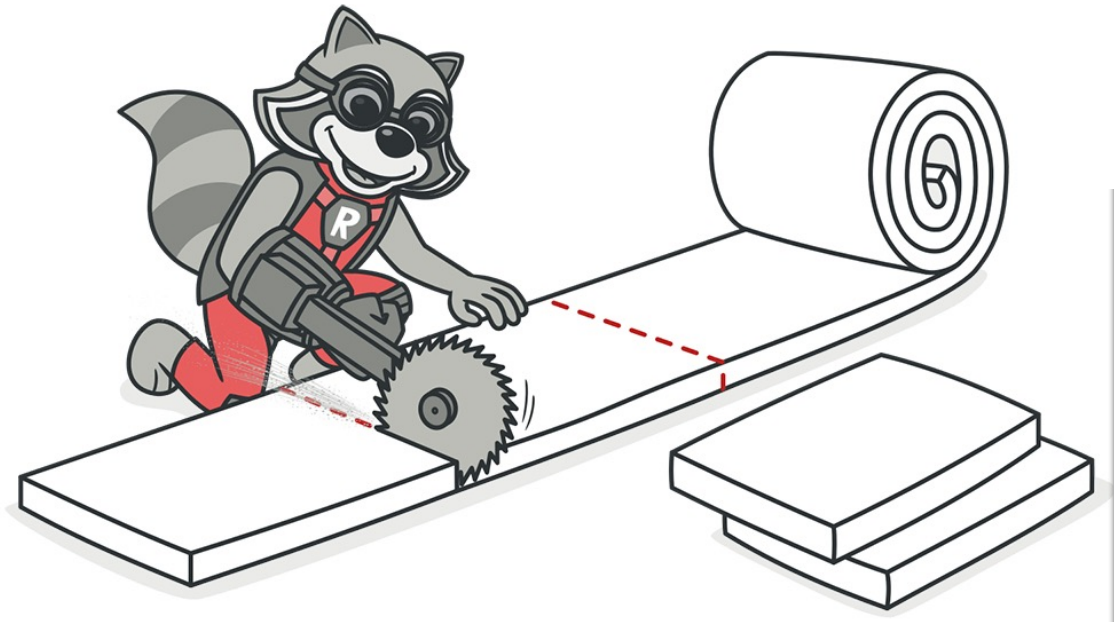
```
public void UpdateQuality()
{
        for (var i = 0; i < Items.Count; i++)
        {
                if (Items[i].Name != "Aged Brie" && Items[i].Name != "Backstage passes to a TAFKAL80ETC
concert")
                {
                        if (Items[i].Quality > 0)
                        {
                                if (Items[i].Name != "Sulfuras, Hand of Ragnaros")
                                {
                                        Items[i].Quality = Items[i].Quality - 1;
                                }
                        }
                }
                else
                {
                        if (Items[i].Quality < 50)
                        {
                                Items[i].Quality = Items[i].Quality + 1;

                                if (Items[i].Name == "Backstage passes to a TAFKAL80ETC concert")
                                {
                                        if (Items[i].SellIn < 11)
                                        {
                                                if (Items[i].Quality < 50)
                                                {
                                                        Items[i].Quality = Items[i].Quality + 1;
                                                }
                                        }

                                        if (Items[i].SellIn < 6)
                                        {
                                                if (Items[i].Quality < 50)
                                                {
                                                        Items[i].Quality = Items[i].Quality + 1;
                                                }
                                        }
                                }
                        }
                }
```

https://makolyte.com/refactoring-the-long-method-code-smell/

Long Method

Treatment

As a rule of thumb, if you feel the need to comment on something inside a method, you should take this code and put it in a new method. Even a single line can and should be split off into a separate method, if it requires explanations. And if the method has a descriptive name, nobody will need to look at the code to see what it does.



- To reduce the length of a method body, use **Extract Method**.
- If local variables and parameters interfere with extracting a method, use **Replace Temp with Query**, **Introduce Parameter Object** or **Preserve Whole Object**.
- If none of the previous recipes help, try moving the entire method to a separate object via **Replace Method with Method Object**.
- Conditional operators and loops are a good clue that code can be moved to a separate method. For conditionals, use **Decompose Conditional**. If loops are in the way, try **Extract Method**.

Long Method

- https://makolyte.com/refactoring-the-long-method-code-smell/

```
public void UpdateQuality()
{
        for (var i = 0; i < Items.Count; i++)
        {
                if (Items[i].Name != "Aged Brie" && Items[i].Name != "Backstage passes to a TAFKAL80ETC
concert")
                {
                        DecrementQualityForNormalItems(i);
                }
                else
                {
                        UpdateQualityForItemsThatAgeWell(i);
                }

                if (Items[i].Name != "Sulfuras, Hand of Ragnaros")
                {
                        Items[i].SellIn = Items[i].SellIn - 1;
                }

                if (Items[i].SellIn < 0)
                {
                        UpdateQualityForExpiredItems(i);
                }
        }
}
```

Long Parameter List

- https://refactoring.guru/smells/long-parameter-list

```
sendPriorizedEmail(true, null, ‘boss@that.com’, false, 2);
userService.create(USER_NAME, group, "joshua", USER_NAME, false, false, new Date(),
                   "blah», true);
```

Signs and Symptoms

More than three or four parameters for a method.

Long Parameter List

- https://www.arhohuttunen.com/long-parameter-list/

Symptoms

A long parameter list is easy to spot. Symptoms of too many parameters include:

- It is hard to use a method call or to get the parameters in correct order.
- It is hard to read and interpret what a method call does.
- A method call has boolean parameters.
- A method call has null parameters as optional parameters.

Consider the following example:

```
calculateStatistics(customer, unit, null, true, false);
```

It is not really clear what each parameter does. To find out you are forced to read the documentation.

- More Examples: https://www.informit.com/articles/article.aspx?p=102271&seqNum=5

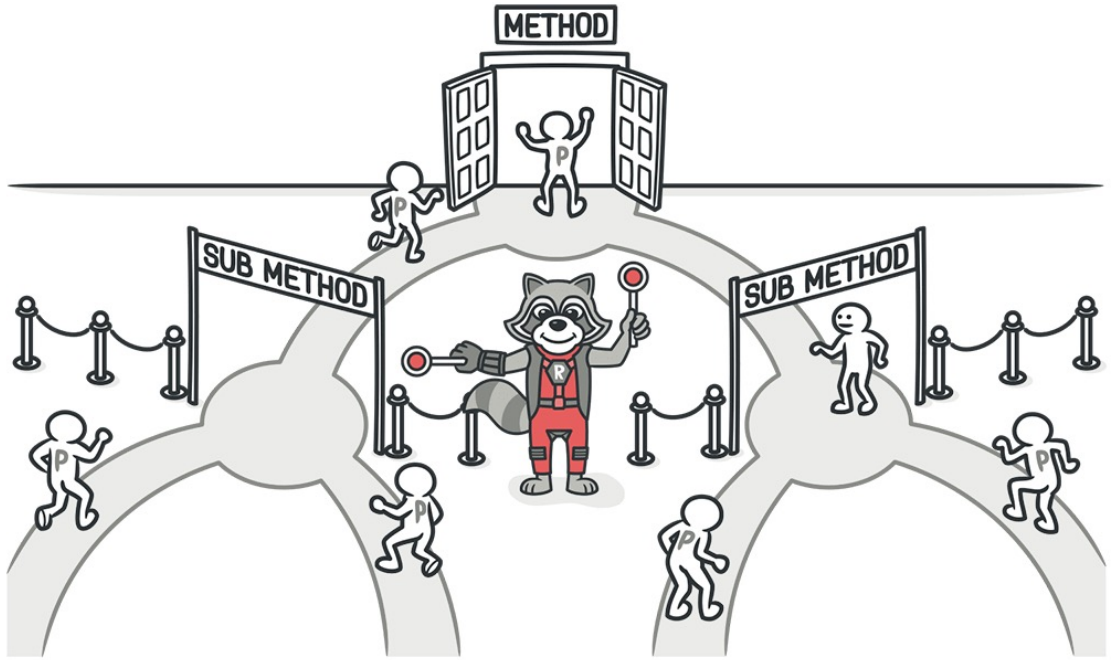
Long Parameter List

Treatment

- Check what values are passed to parameters. If some of the arguments are just results of method calls of another object, use **Replace Parameter with Method Call**. This object can be placed in the field of its own class or passed as a method parameter.

- Instead of passing a group of data received from another object as parameters, pass the object itself to the method, by using **Preserve Whole Object**.

- If there are several unrelated data elements, sometimes you can merge them into a single parameter object via **Introduce Parameter Object**.



Data Clumps

"Whenever two or three values are gathered together, turn them into a $%#$%#^ object" **– Martin Fowler**

https://medium.com/multinetinventiv/refactoring-code-smells-data-clumps-1d9823c7198e

Data Clumps

Treatment

- If repeating data comprises the fields of a class, use **Extract Class** to move the fields to their own class.
- If the same data clumps are passed in the parameters of methods, use **Introduce Parameter Object** to set them off as a class.
- If some of the data is passed to other methods, think about passing the entire data object to the method instead of just individual fields. **Preserve Whole Object** will help with this.
- Look at the code used by these fields. It may be a good idea to move this code to a data class.

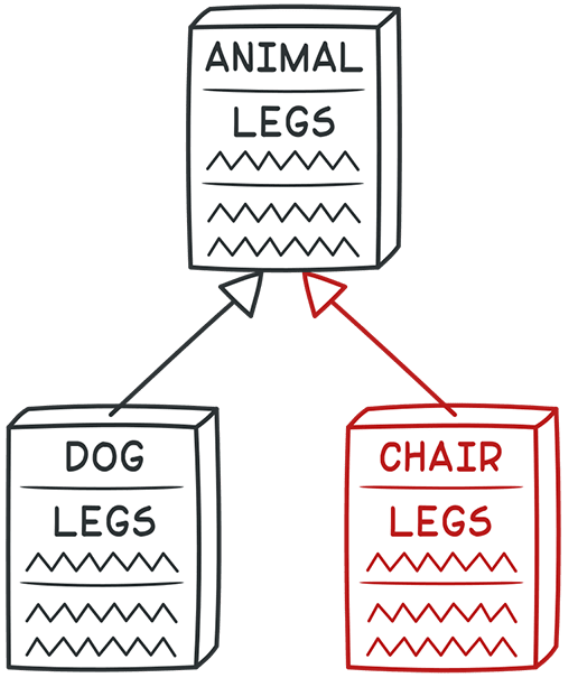
Make cohesive primitive objects travel together

Refused Bequest – Wrong (implementation) inheritance

- https://refactoring.guru/smells/refused-bequest

Reasons for the Problem

Someone was motivated to create inheritance between classes only by the desire to reuse the code in a superclass. But the superclass and subclass are completely different.

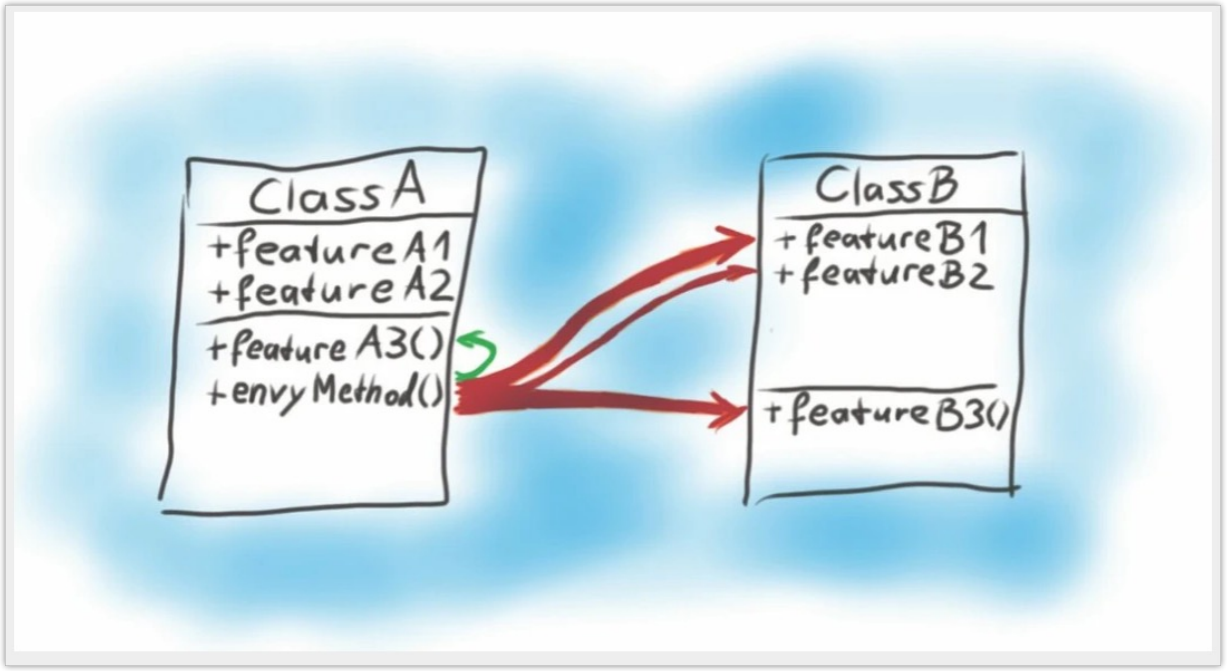


VIOLATION OF "is-a" condition for inheritance

Who knows «is-a»?

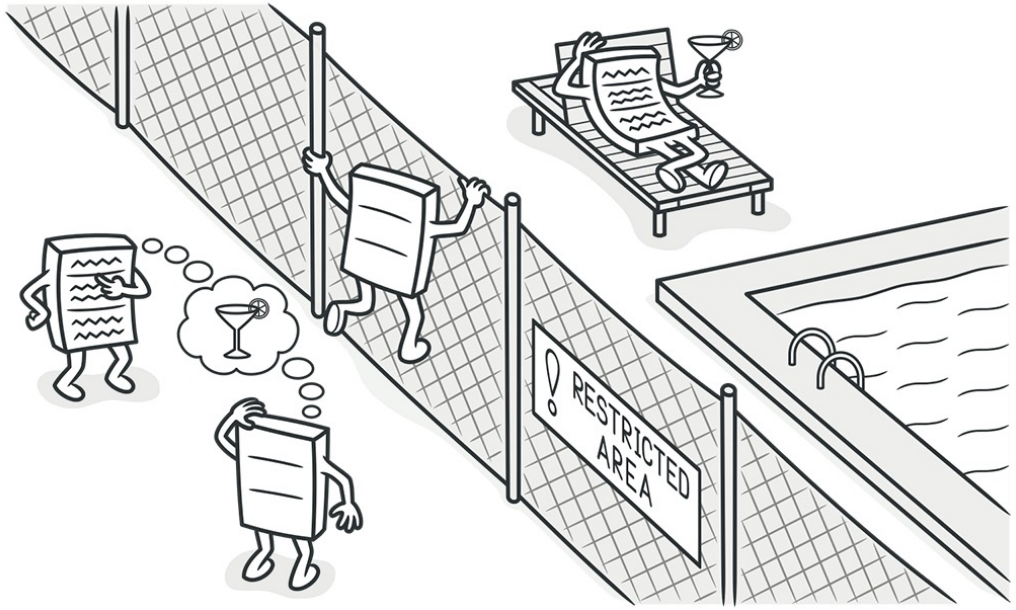
Feature Envy – Wrong class design

- https://refactoring.guru/smells/feature-envy



Signs and Symptoms

A method accesses the data of another object more than its own data.



Reasons for the Problem

This smell may occur after fields are moved to a data class. If this is the case, you may want move the operations on data to this class as well.

https://waog.wordpress.com/2014/08/25/code-smell-feature-envy/

Feature Envy – Wrong class design

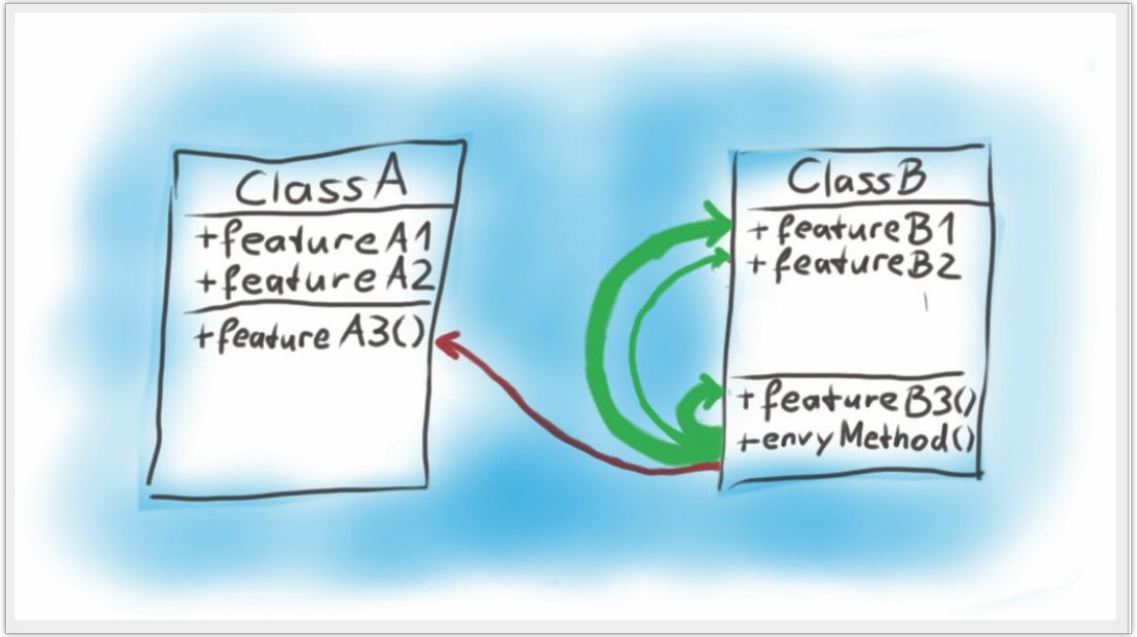
- https://maximilianocontieri.com/code-smell-63-feature-envy

```
class Candidate {

 void printJobAddress(Job job) {

   System.out.println("This is your position address");

   System.out.println(job.address().street());
   System.out.println(job.address().city());
   System.out.println(job.address().ZipCode());
 }
}
```

What's wrong here?

```
class Candidate {

 void printJobAddress(Job job) {

   System.out.println("This is your position address");

   System.out.println(job.address().street());
   System.out.println(job.address().city());
   System.out.println(job.address().ZipCode());
 }
}
```

Feature Envy – Wrong class design



Treatment

As a basic rule, if things change at the same time, you should keep them in the same place. Usually data and functions that use this data are changed together (although exceptions are possible).

- If a method clearly should be moved to another place, use **Move Method**.
- If only part of a method accesses the data of another object, use **Extract Method** to move the part in question.
- If a method uses functions from several other classes, first determine which class contains most of the data used. Then place the method in this class along with the other data. Alternatively, use **Extract Method** to split the method into several parts that can be placed in different places in different classes.

https://waog.wordpress.com/2014/08/25/code-smell-feature-envy/

Feature Envy – Corrected class design

```
class Job {

 void printAddress() {

   System.out.println("This is your job position address"

   System.out.println(this.address().street());
   System.out.println(this.address().city());
   System.out.println(this.address().ZipCode());

  //We might even move this responsability directly to th
  //Some address information is relevant to a job and not
 }
}

class Candidate {
  void printJobAddress(Job job) {
    job.printAddress();
  }
}
```

https://maximilianocontieri.com/code-smell-63-feature-envy

Inappropriate Intimacy – Wrong class design

- https://refactoring.guru/smells/inappropriate-intimacy

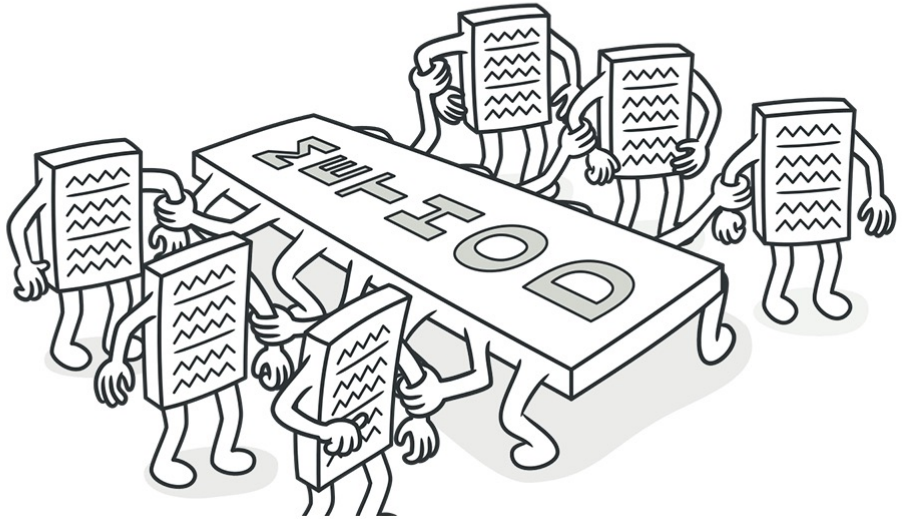
Signs and Symptoms

One class uses the internal fields and methods of another class.

What is violated?

Signs and Symptoms

One class uses the internal fields and methods of another class.



Inappropriate Intimacy – Wrong class design

Treatment

- The simplest solution is to use **Move Method** and **Move Field** to move parts of one class to the class in which those parts are used. But this works only if the first class truly doesn't need these parts.

Inappropriate Intimacy vs Feature Envy

- **Feature Envy** is that methods in one object **invoke** a lot of (getting) **methods on another** object or uses more **public features of another class** than it does of its own.

  => Features are not correctly disributed
  => Low cohesion
  => Tighly coupled

- **Inappropriate Intimacy** is that a class depends on **anothers' private parts** too often.This means **compromising** the other **class's encapsulation**, by directly accessing instance variables that aren't meant to be directly accessed

  => Bad encapsulation
  => Massivly coupled

Inappropriate Intimacy vs Feature Envy

- Inappropriate Intimacy means compromising the other class's encapsulation, and this causes strong coupling and harder maintainibility.

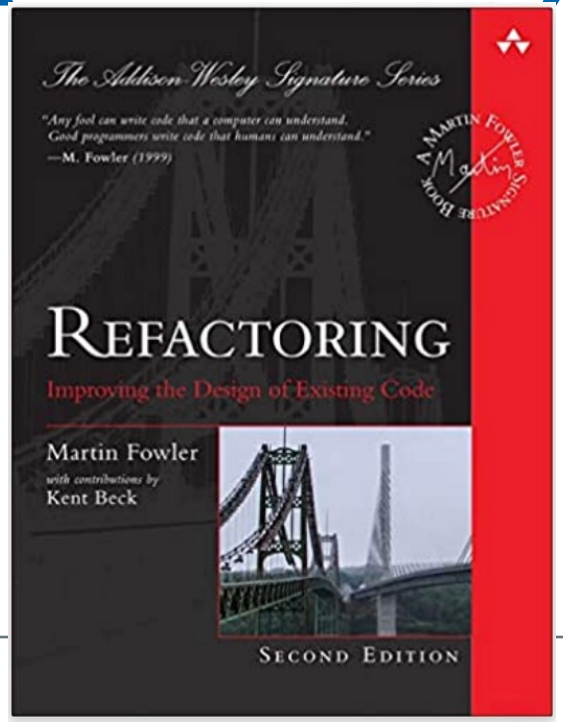
  => ensure / introduce encapsulation (encapsulate field refactoring)

  => rely only in public api

- Feature Envy is not as bad as inappropriate Initimacy, because (assuming the other class's public features are safe to use) it won't lead to bugs. But it does lead to design entangling between the two classes.

  => move methods to other class

Intro Basic Refactorings (Fowler)



Most important basic refactorings

- Extract Method
- Extract Local Variable*
- Extract Field (Convert Local Variable to Field)*
- Extract Constant
- Encapsulate Field (Generate getters and setters)
- Rename
- Inline
- Change Method Signature
- Introduce Parameter Object

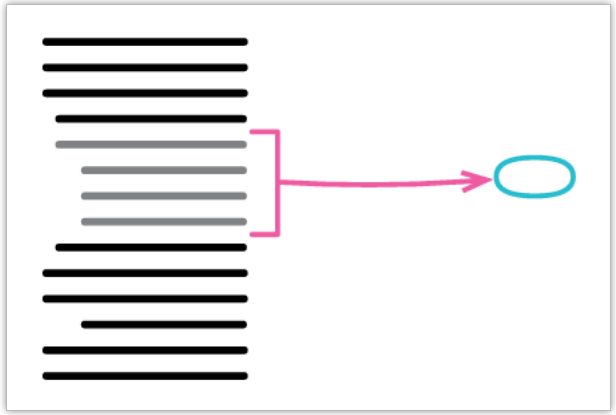
*Extract XYZ == Introduce XYZ

Important basic refactorings

- Extract Interface
- Extract Class
- Extract Superclass
- Move
- Pull Up
- Push Down

Basic refactorings at a glance (https://refactoring.com/catalog/)

Extract Method



```
function printOwing(invoice) {
  printBanner();
  let outstanding  = calculateOutstanding();

  //print details
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);
}
```

```
function printOwing(invoice) {
  printBanner();
  let outstanding  = calculateOutstanding();
  printDetails(outstanding);

  function printDetails(outstanding) {
    console.log(`name: ${invoice.customer}`);
    console.log(`amount: ${outstanding}`);
  }
}
```

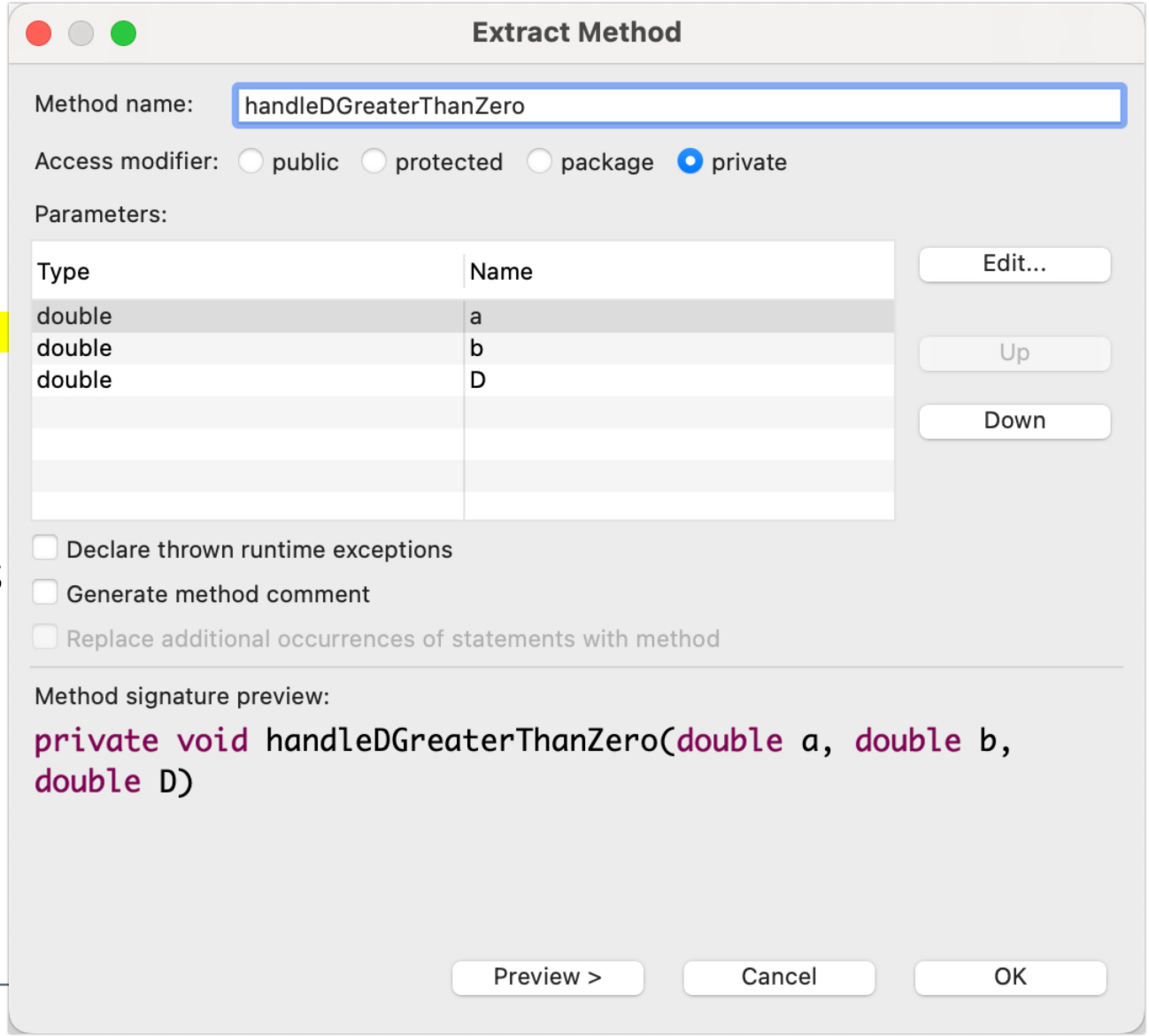
Extract Method

```
public void calcQuadraticEq(double a, double b, double c) {
    double D = b * b - 4 * a * c;
    if (D > 0) {
        double x1, x2;
        x1 = (-b - Math.sqrt(D)) / (2 * a);
        x2 = (-b + Math.sqrt(D)) / (2 * a);
        System.out.println("x1 = " + x1 + ", x2 = " + x2);
    } else if (D == 0) {
        double x;
        x = -b / (2 * a);
        System.out.println("x = " + x);
    } else {
        System.out.println("Equation has no roots");
    }
}
```

https://codegym.cc/groups/posts/196-how-refactoring-works-in-java

Extract Method

```
public void calcQuadraticEq(double a, double b,
                            double c) {
    double D = b * b - 4 * a * c;
    if (D > 0) {
        double x1, x2;
        x1 = (-b - Math.sqrt(D)) / (2 * a);
        x2 = (-b + Math.sqrt(D)) / (2 * a);
        System.out.println("x1 = " + x1 + ", x2 = "
    } else if (D == 0) {
        double x;
        x = -b / (2 * a);
        System.out.println("x = " + x);
    } else {
        System.out.println("Equation has no roots");
    }
}
```



https://codegym.cc/groups/posts/196-how-refactoring-works-in-java

D E M O

Extract Variable

```
return order.quantity * order.itemPrice -
  Math.max(0, order.quantity - 500) * order.itemPrice * 0.05 +
  Math.min(order.quantity * order.itemPrice * 0.1, 100);
```

```
const basePrice = order.quantity * order.itemPrice;
const quantityDiscount = Math.max(0, order.quantity - 500) * order.itemPrice * 0.05;
const shipping = Math.min(basePrice * 0.1, 100);
return basePrice - quantityDiscount + shipping;
```

Inline Variable

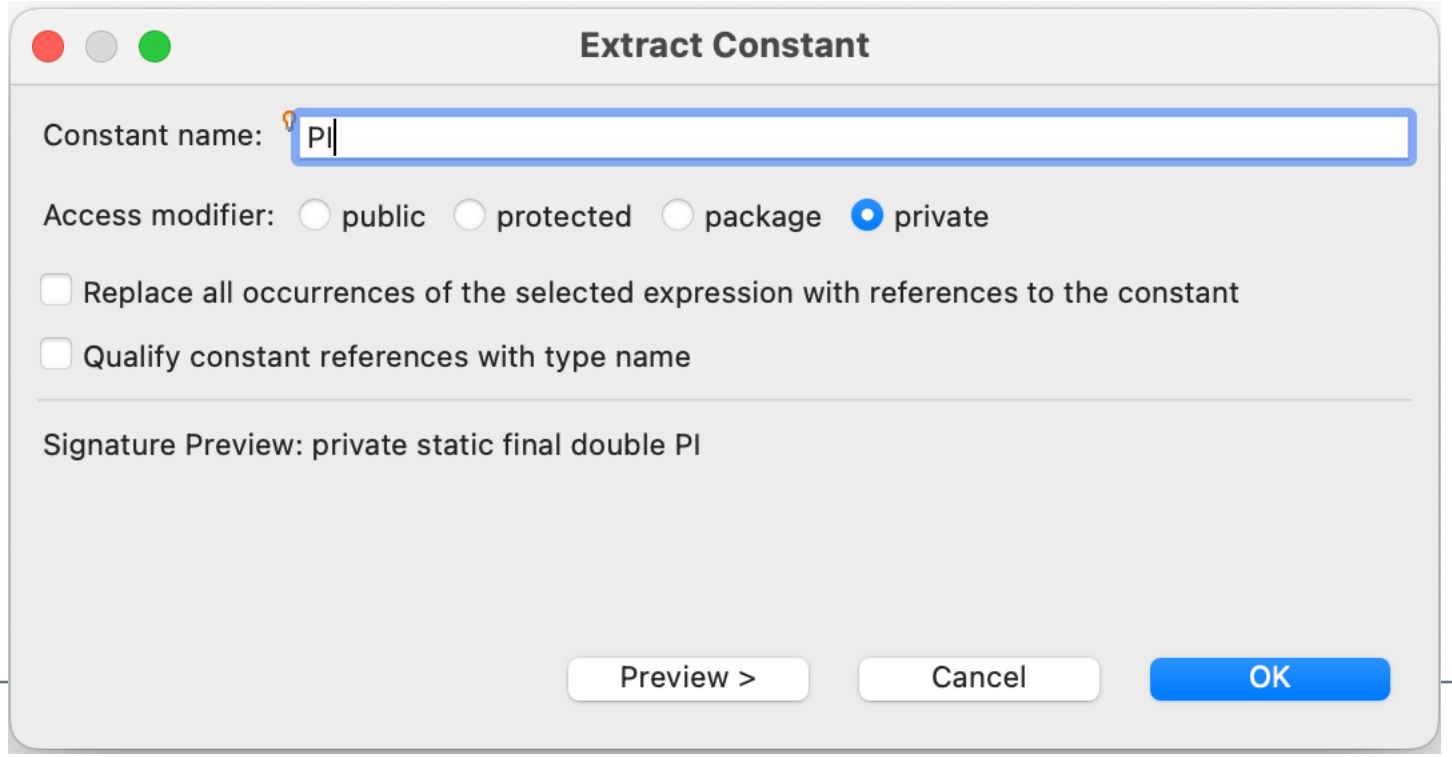
```
let basePrice = anOrder.basePrice;
return (basePrice > 1000);
```

```
return anOrder.basePrice > 1000;
```

Extract Constant

```
public double calc(int x, int y)
{
    return x * y * 3.1415;
}
```

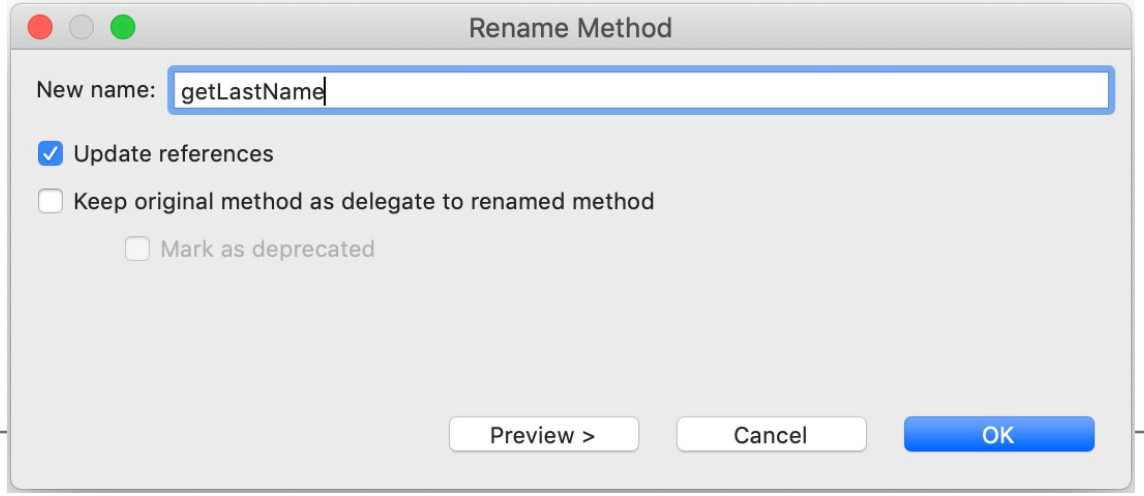
```
private static final double PI = 3.1415;


public double calc(int x, int y)
{
    return x * y * PI;
}
```



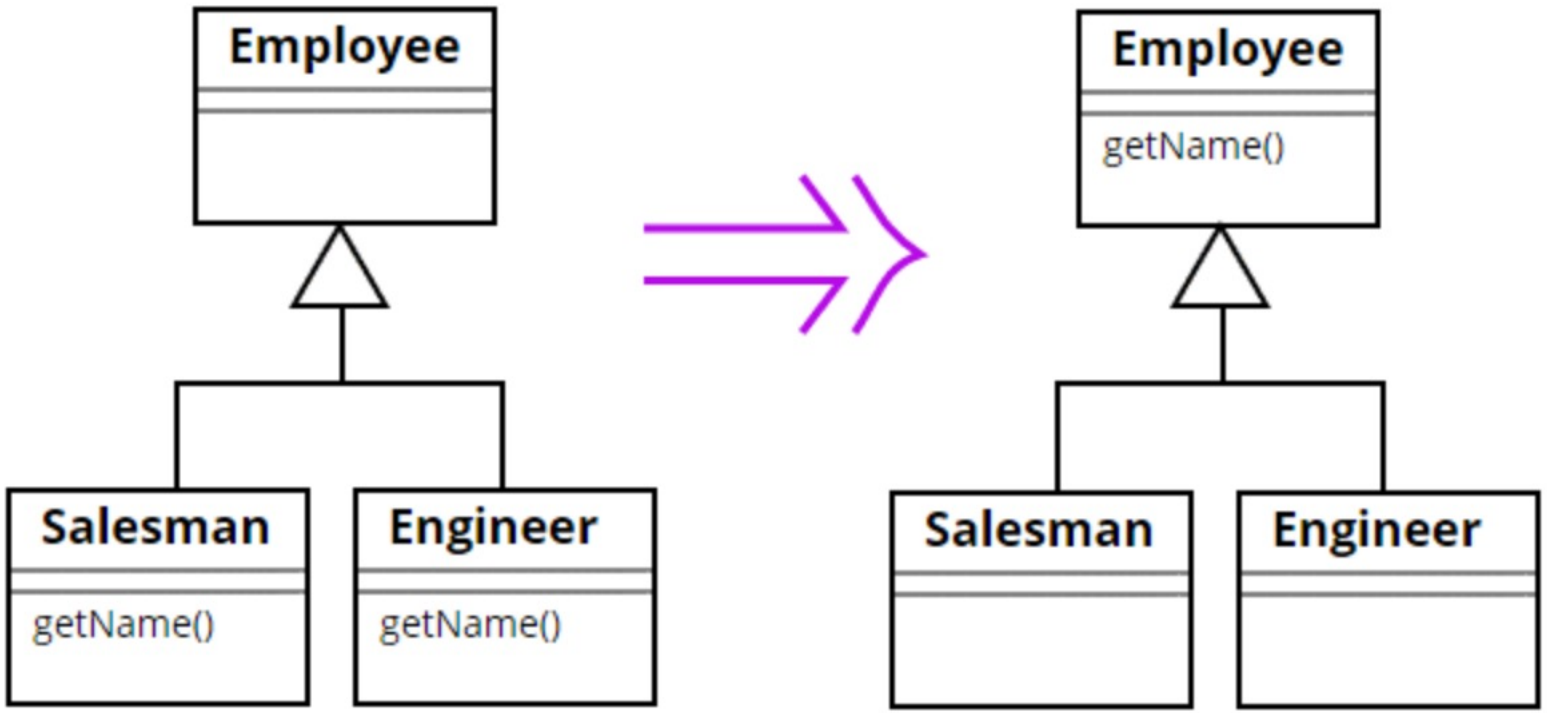
Rename

```java
class Customer
{
    private String lastName = "lastname";

    /**
     * Method returns customer's lastname.
     */
    String getlnm()
    {
        return lastName;
    }

    // ...
}
```

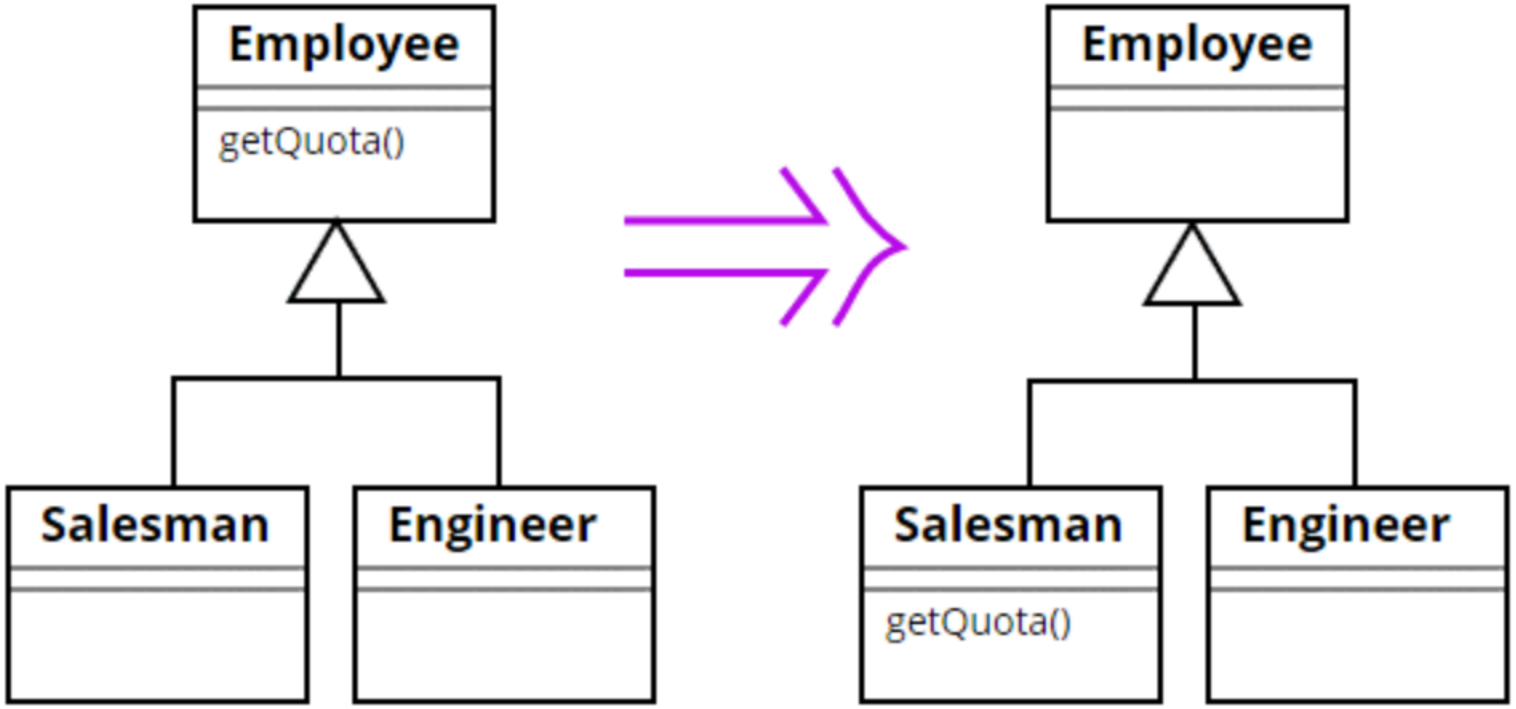
```java
class Customer
{
    private String lastName = "lastname";

    /**
     * Method returns customer's lastname.
     */
String getLastName()
{
    return lastName;
}
```



Pull up

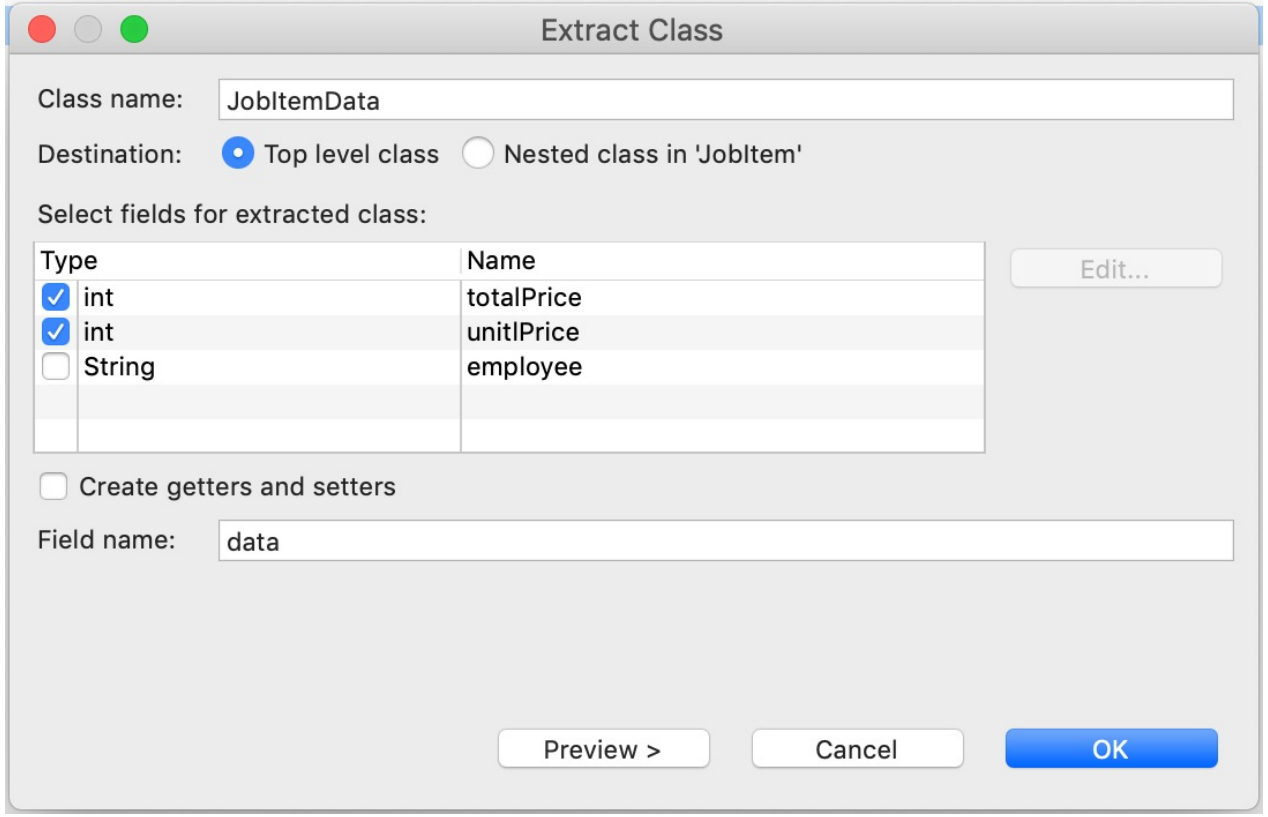

Employee
Salesman
getName()
Engineer
getName()
Employee
getName()
Salesman
Engineer

Push down


Employee
getQuota()
Salesman
Engineer
Employee
Salesman
getQuota()
Engineer

Extract Class

```
public class JobItem
{
    int totalPrice;
    int unitPrice;
    String employee;

    public int getTotalPrice()
    {
        return totalPrice;
    }

    public int getUnitPrice()
    {
        return unitlPrice;
    }


    public String getEmployee()
    {
        return "PETER";
    }
}
```



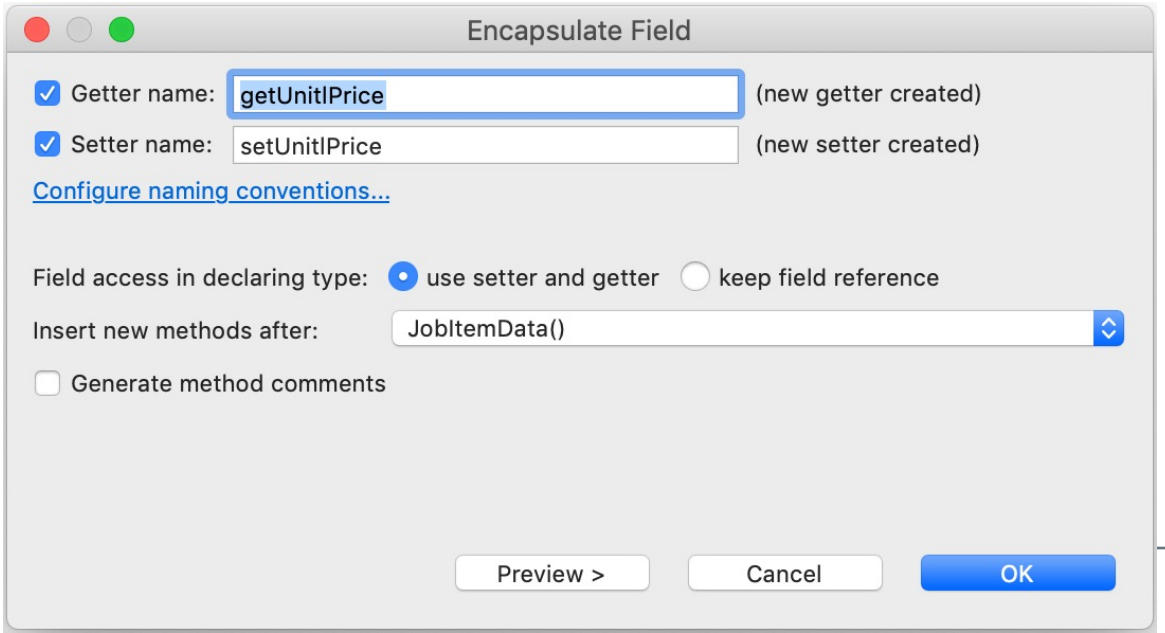
Extract Class

```
public class JobItem
{
    int totalPrice;
    int unitPrice;
    String employee;

    public int getTotalPrice()
    {
        return totalPrice;
    }

    public int getUnitPrice()
    {
        return unitPrice;
    }

    public String getEmployee()
    {
        return "PETER";
    }
}
```

```
public class JobItem
{
    JobItemData data = new JobItemData();
    String employee;

    public int getTotalPrice()
    {
        return data.totalPrice;
    }

    public int getUnitPrice()
    {
        return data.unitPrice;
    }

    public String getEmployee()
    {
        return "PETER";
    }
}

public class JobItemData
{
    public int totalPrice;
    public int unitPrice;

    public JobItemData()
    {
    }
}
```

Encapsulate Field

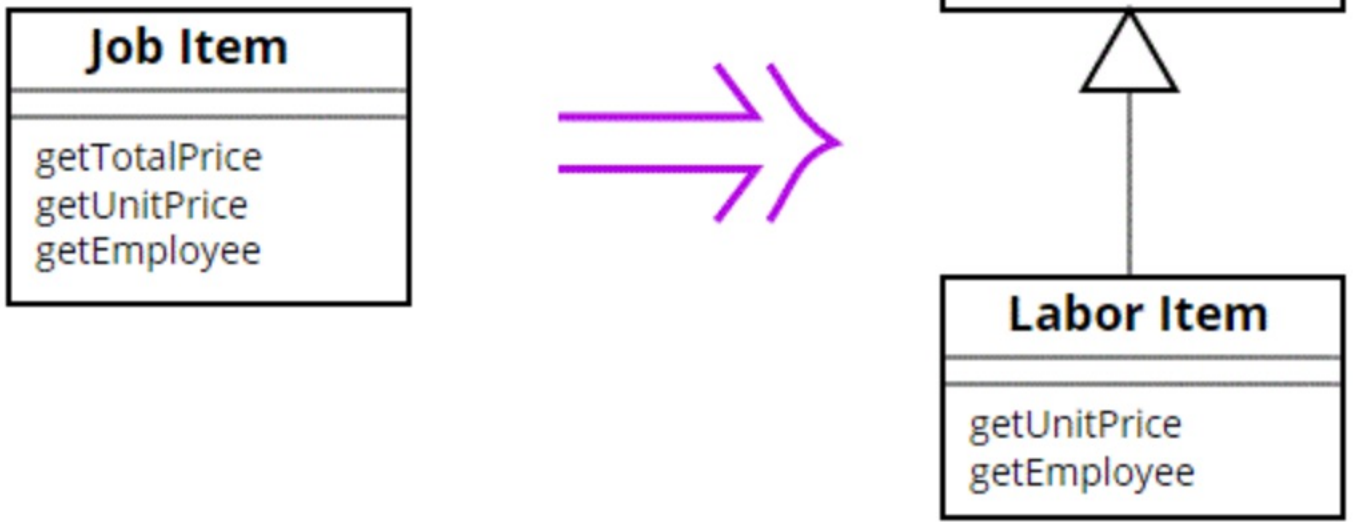
```
public class JobItemData
{
    public int totalPrice;
    public int unitPrice;

    public JobItemData()
    {
    }
}
```




```
public class JobItemData
{
    public int totalPrice;
    private int unitPrice;

    public JobItemData()
    {
    }

    public int getUnitPrice()
    {
        return unitPrice;
    }

    public void setUnitPrice(int unitPrice)
    {
        this.unitPrice = unitPrice;
    }
}
```

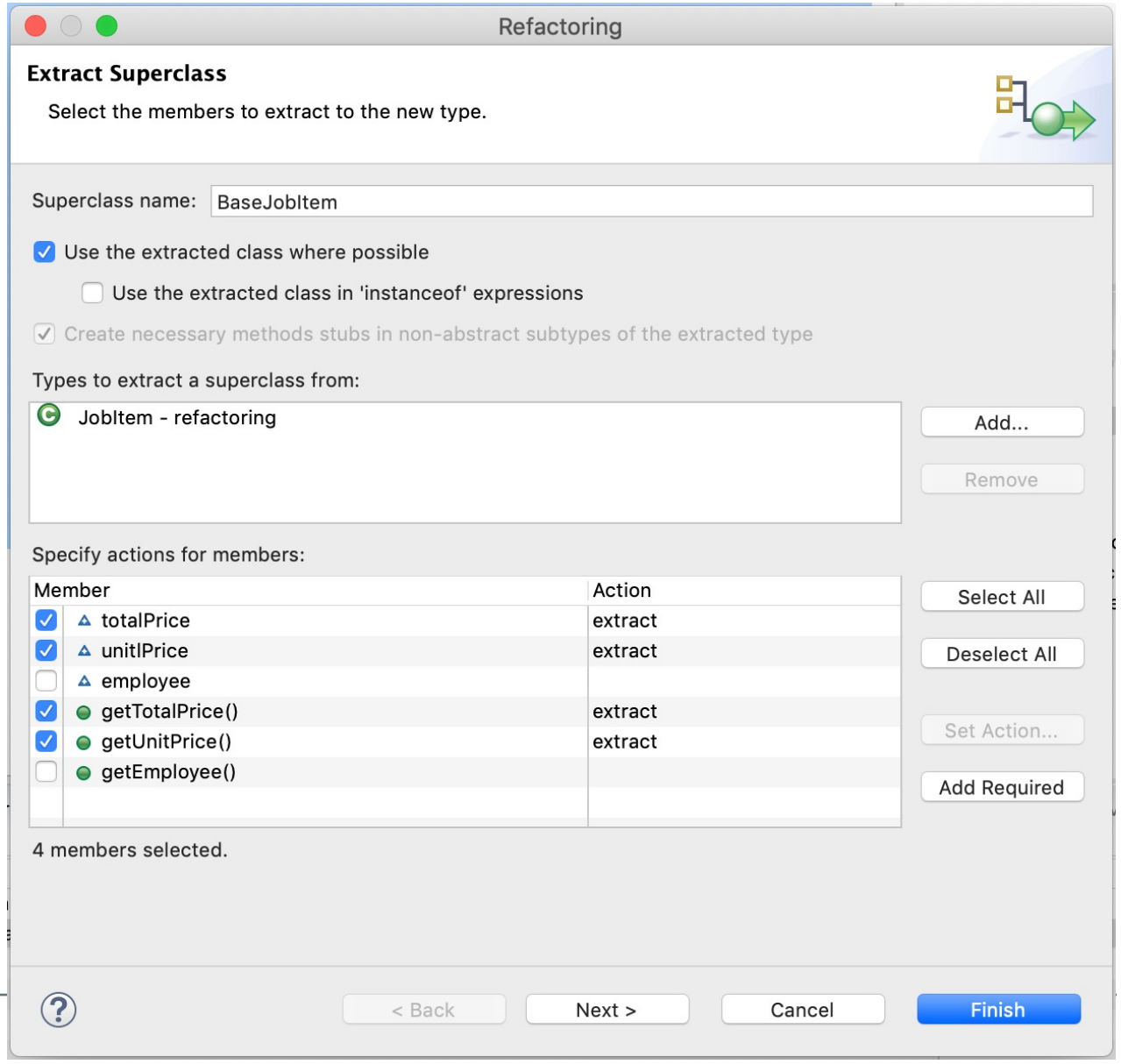
Extract Superclass

```
public class JobItem
{
    int totalPrice;
    int unitPrice;
    String employee;

    public int getTotalPrice()
    {
        return totalPrice;
    }

    public int getUnitPrice()
    {
        return unitPrice;
    }


    public String getEmployee()
    {
        return "PETER";
    }
}
```



Extract Superclass

```
public class JobItem
{
    int totalPrice;
    int unitPrice;
    String employee;

    public int getTotalPrice()
    {
        return totalPrice;
    }

    public int getUnitPrice()
    {
        return unitPrice;
    }


    public String getEmployee()
    {
        return "PETER";
    }
}
```



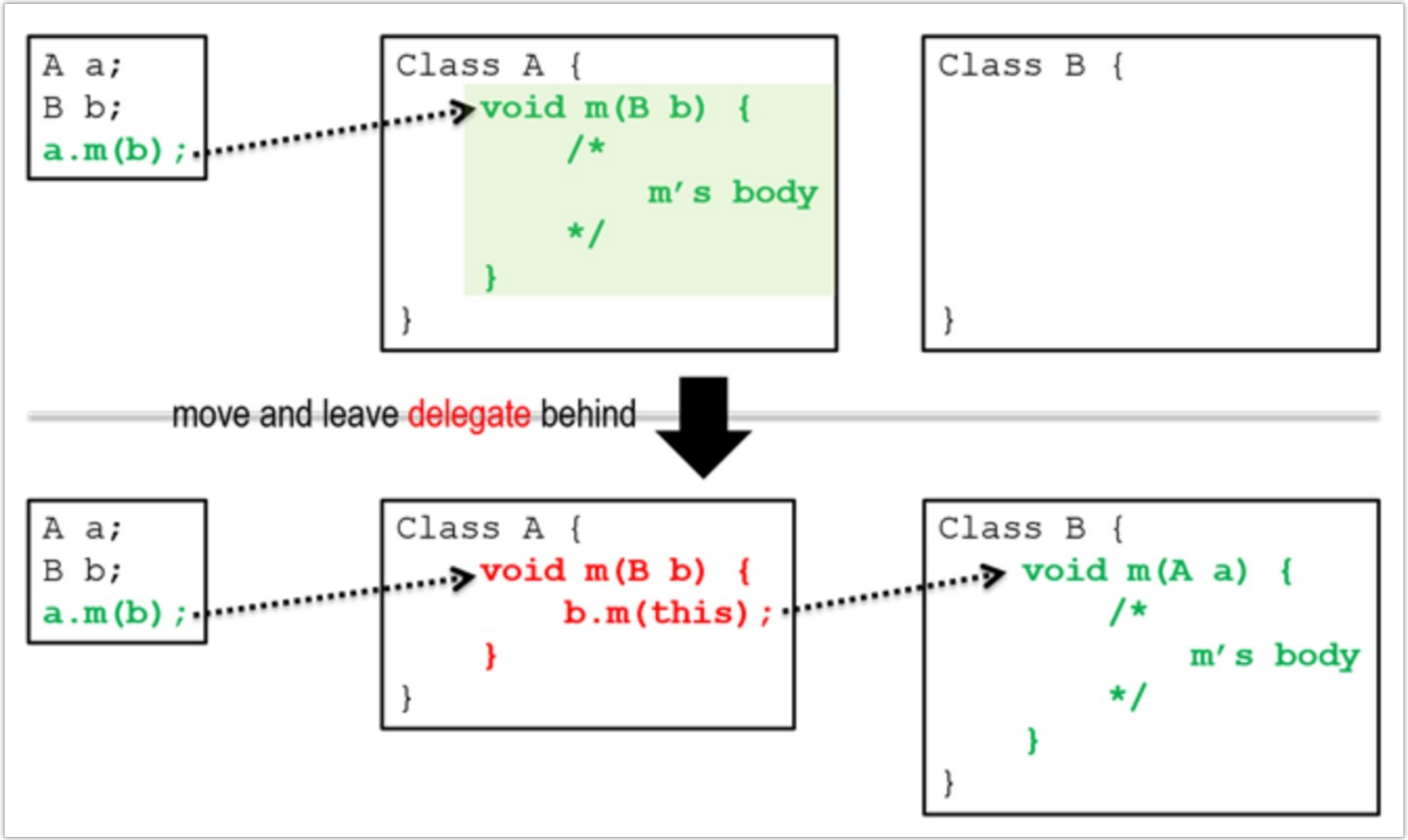
Extract Superclass

```
public class JobItem
{
    int totalPrice;
    int unitPrice;
    String employee;

    public int getTotalPrice()
    {
        return totalPrice;
    }

    public int getUnitPrice()
    {
        return unitPrice;
    }


    public String getEmployee()
    {
        return "PETER";
    }
}
```

```
public class BaseJobItem
{
    int totalPrice;
    int unitPrice;

    public int getTotalPrice()
    {
        return totalPrice;
    }

    public int getUnitPrice()
    {
        return unitPrice;
    }
}

public class JobItem extends BaseJobItem
{
    String employee;

    public String getEmployee()
    {
        return "PETER";
    }
}
```

Follow-up actions still needed: Rename, Override, ..

Move Method

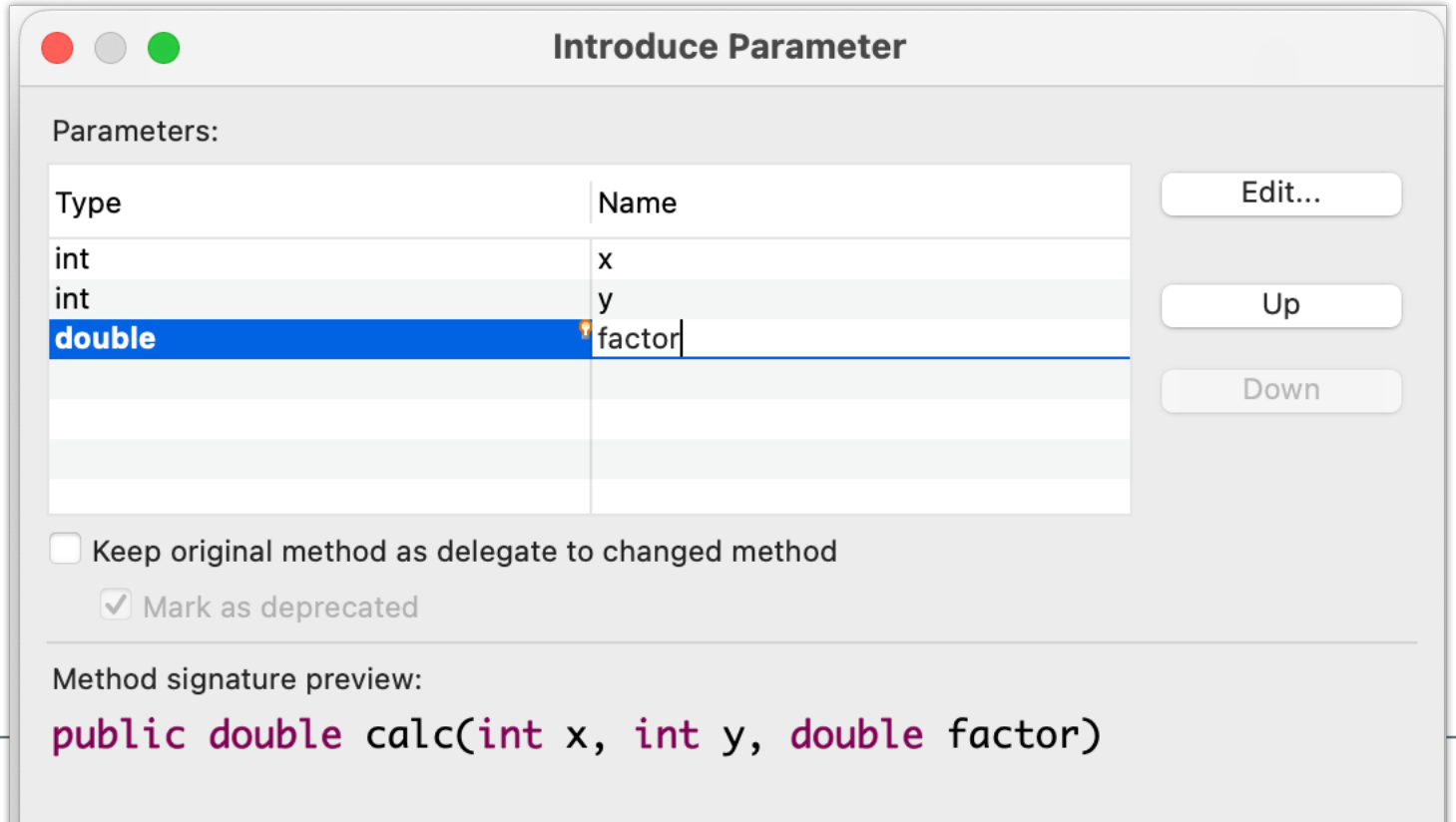


https://www.cs.utexas.edu/~dsb/CS373S/P/R3Assignment/Tutorial/eclipserefactorings.html#deleteClass

Introduce Parameter

```java
public double calc(int x, int y)
{
    return x * y * PI;
}
```

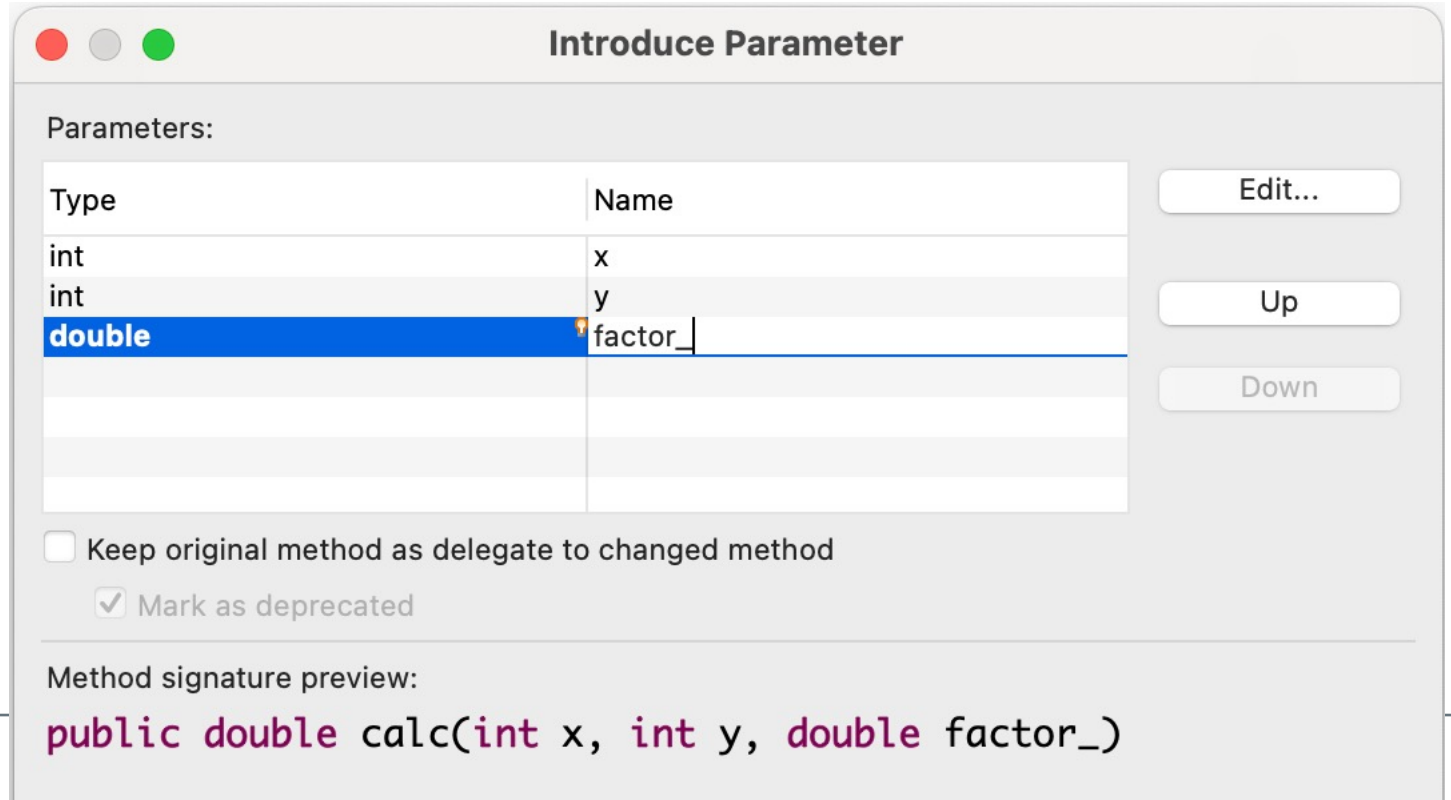
```java
public double calc(int x, int y,
                   double factor)
{
    return x * y * factor;
}
```



Introduce Parameter (from local var – easier with IntelliJ)

```
public double calc(int x, int y)
{
    double factor = PI;
    return x * y * factor;
}
```

```
public double calc(int x, int y,
                   double factor_)
{
    double factor = factor_;
    return x * y * factor;
}
```



Introduce Parameter … Inline + Rename

```
public double calc(int x, int y,
                   double factor_)
{
    double factor = factor_;
    return x * y * factor;
}
```

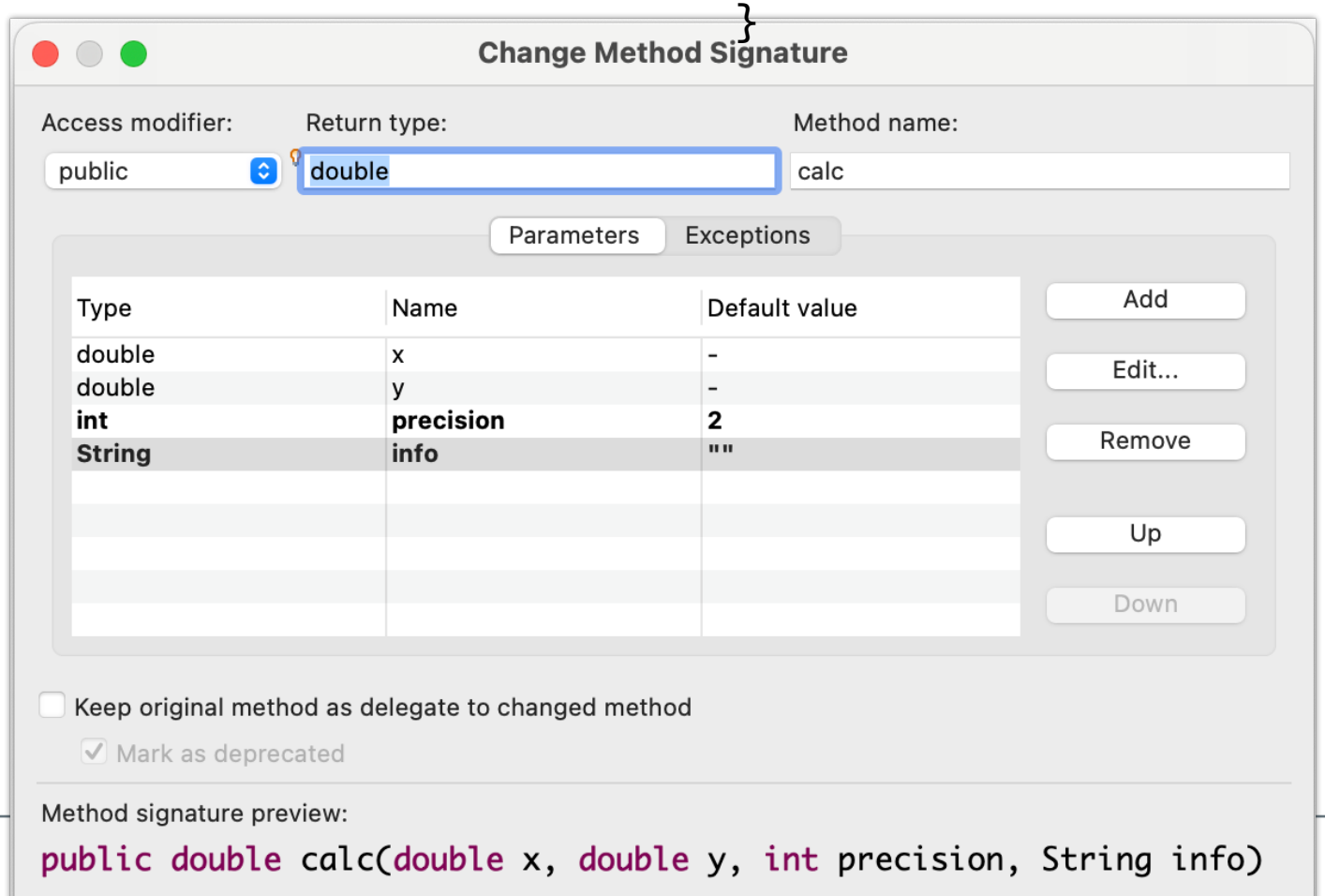
```
public double calc(int x, int y,
                   double factor_)
{
    return x * y * factor_;
}
```

```
public double calc(int x, int y,
                   double factor)
{
    return x * y * factor;
}
```

Change Method Signature

```
public double calc(int x, int y)
{
    return x * y * 3.1415;
}
```

```
public double calc(double x, double y,
                   int precision, String info)
{
    return x * y * 3.1415;
}
```



Refactoring Support in IDEs

Refactoring

Refactor Navigate Search Project Run

- Rename...
- Move... ⌥⌘V
- Change Method Signature... ⌥⌘C
- Extract Method... ⌥⌘M
- Extract Local Variable... ⌥⌘L
- Extract Constant...
- Inline... ⌥⌘I
- Convert Local Variable to Field...
- Convert Anonymous Class to Nested...
- Move Type to New File...
- Extract Interface...
- Extract Superclass...
- Use Supertype Where Possible...
- Pull Up...
- Push Down...
- Extract Class...
- Introduce Parameter Object...
- Introduce Indirection...
- Introduce Factory...
- Introduce Parameter...
- Encapsulate Field...
- Generalize Declared Type...
- Infer Generic Type Arguments...
- Migrate JAR File...
- Create Script...
- Apply Script...
- History...

Refactor Build Run Tools VCS Window Help

rBeginners – Ex5_Quadrate.java

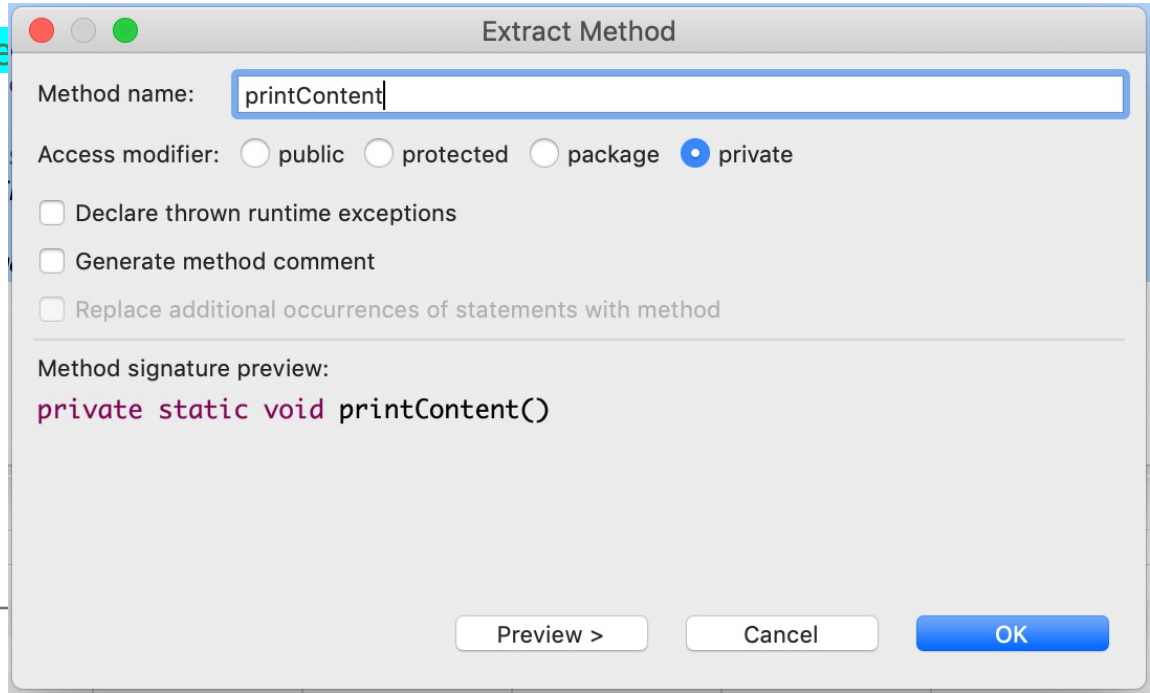
- Refactor This... ^T
- Rename... ⇧F6
- Rename File...
- Change Signature... ⌘F6
- Extract/Introduce ›
  - Variable... ⌥⌘V
  - Constant... ⌥⌘C
  - Field... ⌥⌘F
  - Parameter... ⌥⌘P
  - Functional Parameter... ⌥⇧⌘P
  - Functional Variable...
  - Parameter Object...
  - Method... ⌥⌘M
  - Replace Method With Method Object...
  - Delegate...
  - Interface...
  - Superclass...
  - Subquery as CTE
- Inline... ⌥⌘N
- Find and Replace Code Duplicates...
- Move File... F6
- Copy File... F5
- Safe Delete... ⌘⌦
- Pull Members Up...
- Push Members Down...
- Type Migration... ⇧⌘F6
- Make Static...
- Convert To Instance Method...
- Use Interface Where Possible...
- Replace Inheritance with Delegation...
- Encapsulate Fields...
- Migrate Packages and Classes ›
- Convert Raw Types to Generics...
- Invert Boolean...
- Internationalize...
- Migrate to AndroidX...
- Migrate to Non-Transitive R Classes...

Refactoring in Practice – Starting point

```
private static void printLetterOrig()
{
    printBanner();

    // print content
    for (int i = 0; i < content.size(); i++)
    {
        ContentLine cl = content.get(i);
        printContentline(cl);
    }

    // add happy christmas if in december and before / on christmas eve
    if (LocalDate.now().getMonth() == Month.DECEMBER &&
        LocalDate.now().getDayOfMonth() < 25)
    {
        printHappyChristmas();
    }

    printFooter();
}
```

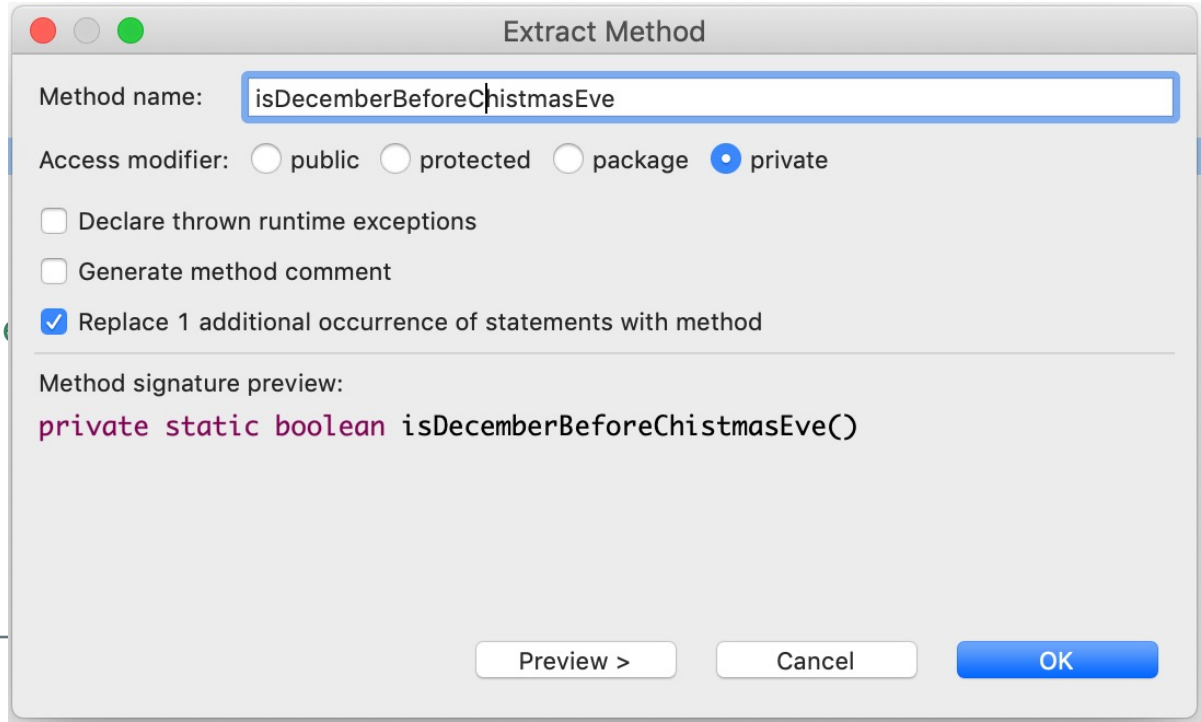
Refactoring in Practice – Applying EXTRACT METHOD

```
private static void printLetterOrig()
{
    printBanner();

    // print conent
    for (int i = 0; i < content.size(); i++)
    {
        ContentLine cl = content.get(i);
        printContentline(cl);
    }

    // add happy christmas if ... before / on christmas eve
    if (LocalDate.now().getMonth() == Month.DECEMBER &&
        LocalDate.now().getDayOfMonth() < 25)
    {
        printHappyChristmas();
    }

    printFooter();
}
```



Refactoring in Practice – Applying EXTRACT METHOD

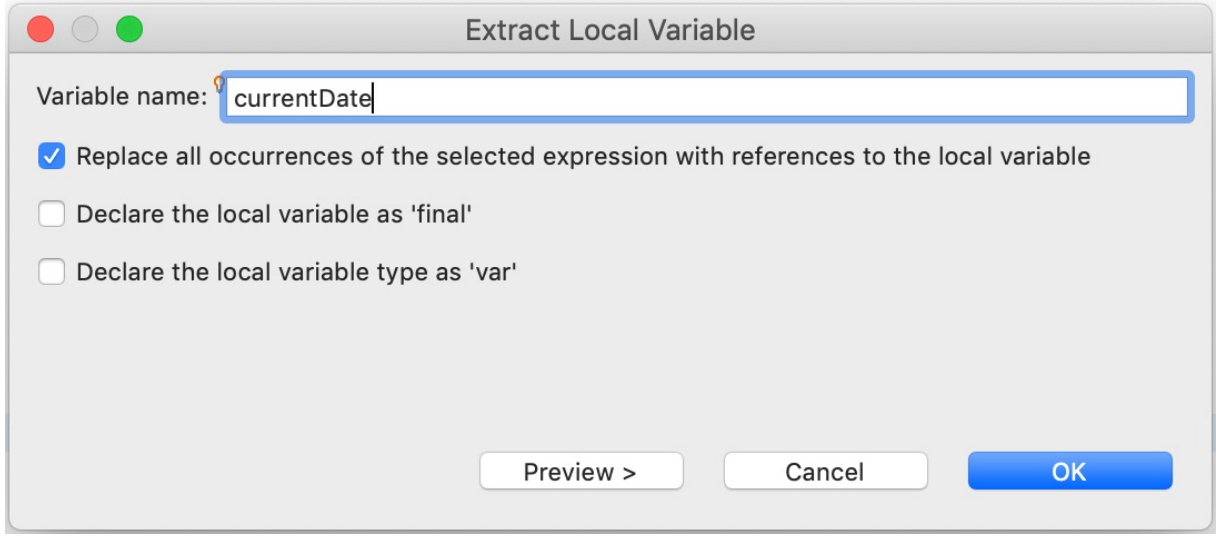
```
private static void printLetterImproved_Step1()
{
    printBanner();

    printContent();

    printFooter();
}

private static void printContent()
{
    for (int i = 0; i < content.size(); i++)
    {
        ContentLine cl = content.get(i);
        printContentline(cl);
    }

    // add happy christmas if ... before / on christmas
    if (LocalDate.now().getMonth() == Month.DECEMBER &&
        LocalDate.now().getDayOfMonth() < 25)
    {
        printHappyChristmas();
    }
}
```



Refactoring in Practice – Applying EXTRACT LOCAL VARIABLE

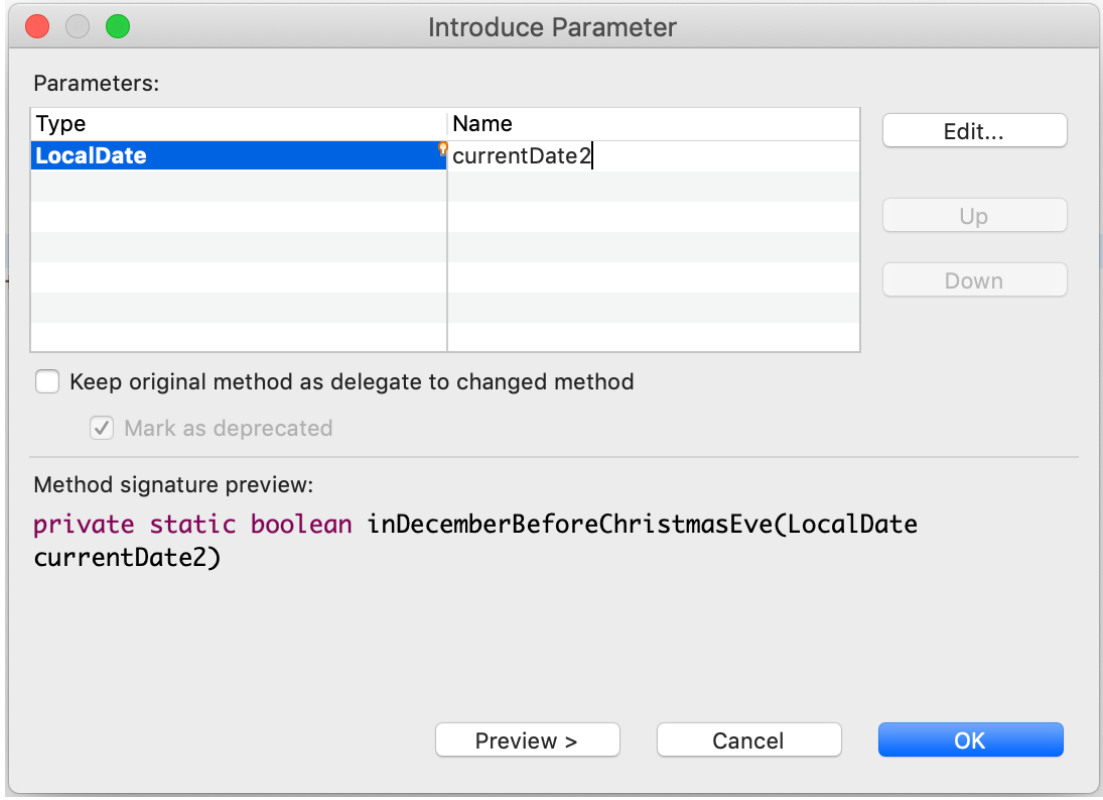
```
private static void printLetterImproved_Step2()
{
    printBanner();

    printContent();

    printFooter();
}

private static void printContent()
{
    for (int i = 0; i < content.size(); i++)
    {
        ContentLine cl = content.get(i);
        printContentline(cl);
    }

    if (inDecemberBeforeChristmasEve())
    {
        printHappyChristmas();
    }
}

private static boolean inDecemberBeforeChristmasEve()
{
    return LocalDate.now().getMonth() == Month.DECEMBER && LocalDate.now().getDayOfMonth() < 25;
}
```

Short Cut:
Introduce Parameter



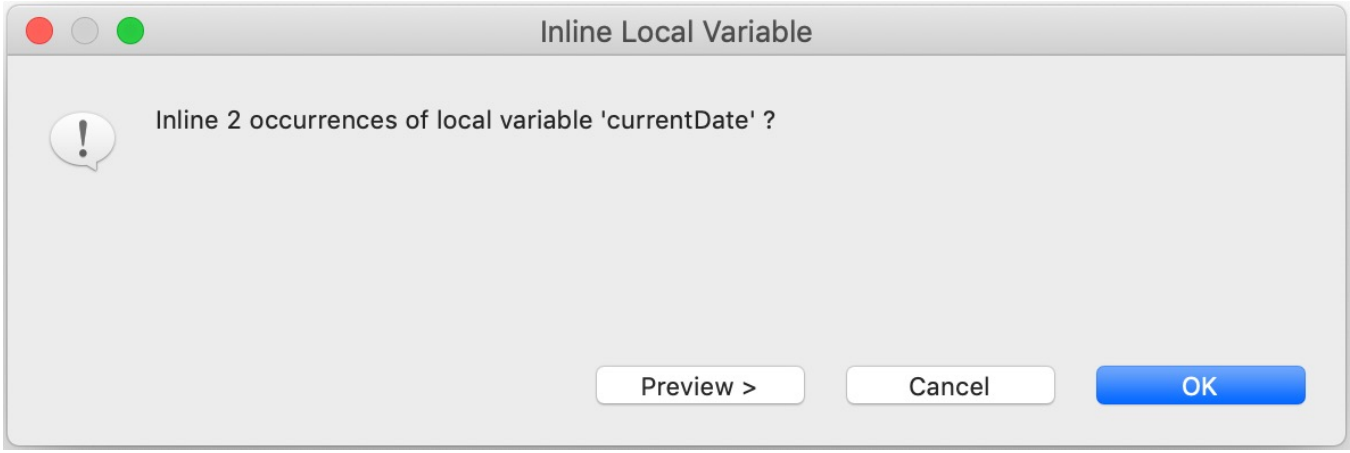
Refactoring in Practice – Applying INTRODUCE PARAMETER

```
private static void printLetterImproved_Step3()
{
    printBanner();
    printContent();

    printFooter();
}

private static void printContent()
{
    for (int i = 0; i < content.size(); i++)
    {
        ContentLine cl = content.get(i);
        printContentline(cl);
    }

    if (inDecemberBeforeChristmasEve())
    {
        printHappyChristmas();
    }
}

private static boolean inDecemberBeforeChristmasEve()
{
    LocalDate currentDate = LocalDate.now();
    return currentDate.getMonth() == Month.DECEMBER && currentDate.getDayOfMonth() < 25;
}
```



Refactoring in Practice – Applying INLINE LOCAL VARIABLE

```
private static void printLetterImproved_Step4()
{
    printBanner();
    printContent();

    printFooter();
}

private static void printContent()
{
    for (int i = 0; i < content.size(); i++)
    {
        ContentLine cl = content.get(i);
        printContentline(cl);
    }

    if (inDecemberBeforeChristmasEve(LocalDate.now()))
    {
        printHappyChristmas();
    }
}

private static boolean inDecemberBeforeChristmasEve(LocalDate currentDate2)
{
    LocalDate currentDate = currentDate2;
    return currentDate.getMonth() == Month.DECEMBER && currentDate.getDayOfMonth() < 25;
}
```



Refactoring in Practice – Applying RENAME

```
private static void printLetterImproved_Step5()
{
    printBanner();

    printContent();

    printFooter();
}

private static void printContent()
{
    for (int i = 0; i < content.size(); i++)
    {
        ContentLine cl = content.get(i);
        printContentline(cl);
    }

    if (inDecemberBeforeChristmasEve(LocalDate.now()))
    {
        printHappyChristmas();
    }
}

private static boolean inDecemberBeforeChristmasEve(LocalDate currentDate2)
{
    return currentDate2.getMonth() == Month.DECEMBER && currentDate2.getDayOfMonth() < 25;
}
```

Refactoring in Practice – FINISH ☺

```
private static void printLetterImproved_Step6()
{
    printBanner();
    printContent();
    printFooter();
}

private static void printContent()
{
    for (int i = 0; i < content.size(); i++)
    {
        ContentLine cl = content.get(i);
        printContentline(cl);
    }

    if (inDecemberBeforeChristmasEve(LocalDate.now()))
    {
        printHappyChristmas();
    }
}

private static boolean inDecemberBeforeChristmasEve(LocalDate currentDate)
{
    return currentDate.getMonth() == Month.DECEMBER && currentDate.getDayOfMonth() < 25;
}
```

Better in refactorings:
As mentioned, it can combine the last steps

Exercises

https://github.com/Michaeli71/ADC_BOOTCAMP_REFACTORINGS

Catalog of Refactoring Techniques

Refactoring Techniques

Composing

- https://refactoring.guru/extract-method
- https://refactoring.guru/inline-method
- https://refactoring.guru/inline-temp
- https://refactoring.guru/split-temporary-variable

Moving Features between Objects

- https://refactoring.guru/move-method
- https://refactoring.guru/move-field
- https://refactoring.guru/extract-class
- https://refactoring.guru/inline-class

Refactoring Techniques

Organizing Data

- https://refactoring.guru/encapsulate-field
- https://refactoring.guru/self-encapsulate-field
- https://refactoring.guru/encapsulate-collection
- https://refactoring.guru/replace-type-code-with-class
- https://refactoring.guru/replace-type-code-with-subclasses
- https://refactoring.guru/replace-subclass-with-fields
- https://refactoring.guru/replace-magic-number-with-symbolic-constant

Refactoring Techniques

Simplyfing Conditionals

- https://refactoring.guru/decompose-conditional
- https://refactoring.guru/consolidate-conditional-expression
- https://refactoring.guru/replace-nested-conditional-with-guard-clauses
- https://refactoring.guru/consolidate-duplicate-conditional-fragments
- https://refactoring.guru/replace-conditional-with-polymorphism

Refactoring Techniques

Simplyfing Method Calls

- https://refactoring.guru/rename-method
- https://refactoring.guru/add-parameter
- https://refactoring.guru/remove-parameter
- https://refactoring.guru/introduce-parameter-object
- https://refactoring.guru/hide-method
- https://refactoring.guru/replace-constructor-with-factory-method
- https://refactoring.guru/replace-error-code-with-exception
- https://refactoring.guru/preserve-whole-object

Refactoring Techniques

Dealing With Generalization

- https://refactoring.guru/pull-up-field
- https://refactoring.guru/pull-up-method
- https://refactoring.guru/push-down-field
- https://refactoring.guru/push-down-method
- https://refactoring.guru/extract-superclass
- https://refactoring.guru/extract-subclass
- https://refactoring.guru/extract-interface
- https://refactoring.guru/replace-inheritance-with-delegation

Composing

Extract Method

- https://refactoring.guru/extract-method

Problem

You have a code fragment that can be grouped together.

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

  // Print details.
  System.out.println("name: " + name);
  System.out.println("amount: " + getOuts
}
```

Solution

Move this code to a separate new method (or function) and replace the old code with a call to the method.

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

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

Inline Method

- https://refactoring.guru/inline-method

Problem

When a method body is more obvious than the method itself, use this technique.

Solution

Replace calls to the method with the method's content and delete the method itself.

```
class PizzaDelivery {
  // ...
  int getRating() {
    return moreThanFiveLateDeliveries() ?
  }
  boolean moreThanFiveLateDeliveries() {
    return numberOfLateDeliveries > 5;
  }
}
```

```
class PizzaDelivery {
  // ...
  int getRating() {
    return numberOfLateDeliveries > 5 ? 2
  }
}
```

Inline Temp

- https://refactoring.guru/inline-temp

Problem

You have a temporary variable that's assigned the result of a simple expression and nothing more.

```
boolean hasDiscount(Order order) {
  double basePrice = order.basePrice();
  return basePrice > 1000;
}
```

Solution

Replace the references to the variable with the expression itself.

```
boolean hasDiscount(Order order) {
  return order.basePrice() > 1000;
}
```

Split Temp

- https://refactoring.guru/split-temporary-variable

Problem

You have a local variable that's used to store various intermediate values inside a method (except for cycle variables).

```
double temp = 2 * (height + width);
System.out.println(temp);
temp = height * width;
System.out.println(temp);
```

Solution

Use different variables for different values. Each variable should be responsible for only one particular thing.

```
final double perimeter = 2 * (height + wi
System.out.println(perimeter);
final double area = height * width;
System.out.println(area);
```

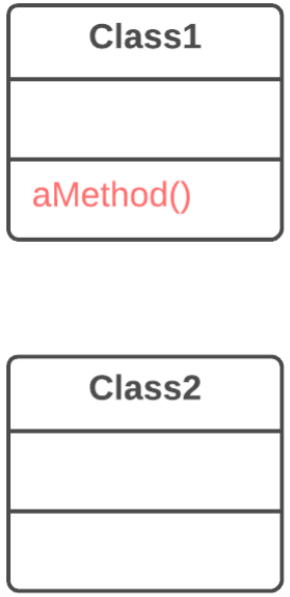
Moving Features between Objects

Move Method

- https://refactoring.guru/move-method

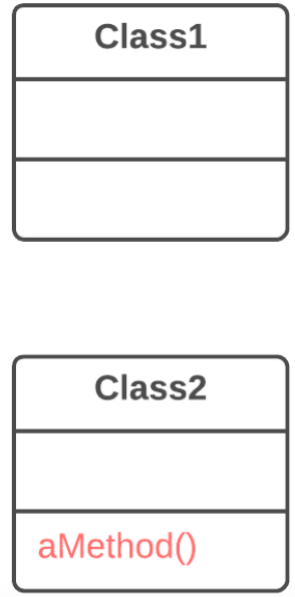
Problem

A method is used more in another class than in its own class.



Solution

Create a new method in the class that uses the method the most, then move code from the old method to there. Turn the code of the original method into a reference to the new method in the other class or else remove it entirely.

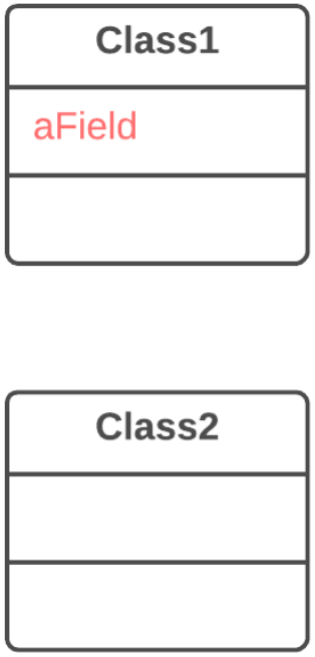


Move Field

- https://refactoring.guru/move-field

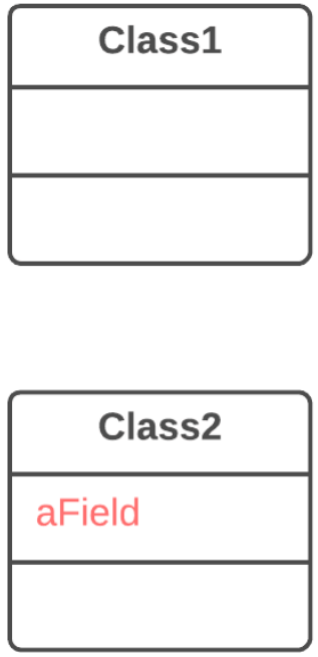
Problem

A field is used more in another class than in its own class.



Solution

Create a field in a new class and redirect all users of the old field to it.



Extract Class

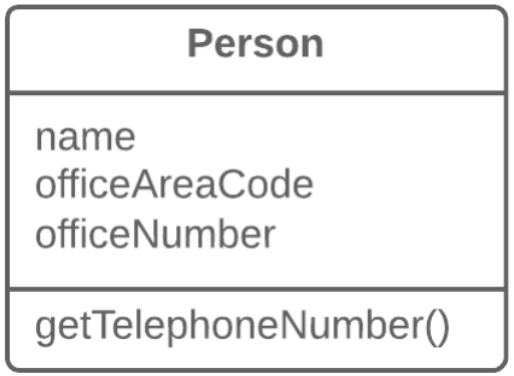
- https://refactoring.guru/extract-class

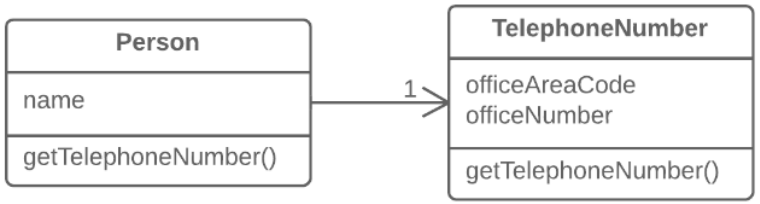
Problem

When one class does the work of two, awkwardness results.

Solution

Instead, create a new class and place the fields and methods responsible for the relevant functionality in it.





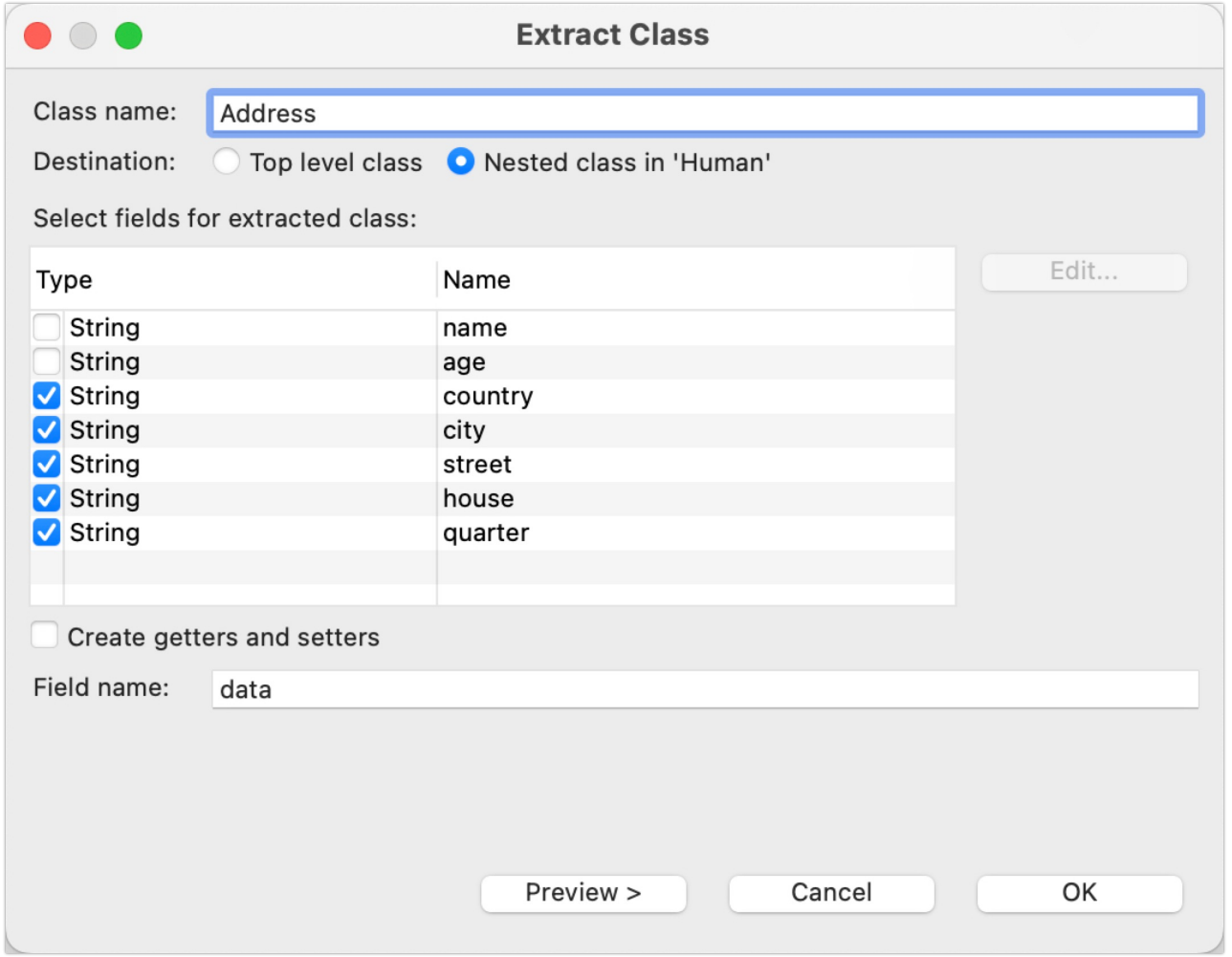
Extract Class

```
class Human {
    private String name;
    private String age;
    private String country;
    private String city;
    private String street;
    private String house;
    private String quarter;


    public String getFullAddress() {
        StringBuilder result = new StringBuilder();
        return result
                        .append(country)
                        .append(", ")
                        .append(city)
                        .append(", ")
                        .append(street)
                        .append(", ")
                        .append(house)
                        .append(" ")
                        .append(quarter).toString();
    }
 }
```

https://codegym.cc/groups/posts/196-how-refactoring-works-in-java

Extract Class

```
class Human {
    private String name;
    private String age;
    private String country;
    private String city;
    private String street;
    private String house;
    private String quarter;


    public String getFullAddress() {
        StringBuilder result = new StringBuilder();
        return result
                        .append(country)
                        .append(", ")
                        .append(city)
                        .append(", ")
                        .append(street)
                        .append(", ")
                        .append(house)
                        .append(" ")
                        .append(quarter).toString();
    }
}
```



https://codegym.cc/groups/posts/196-how-refactoring-works-in-java

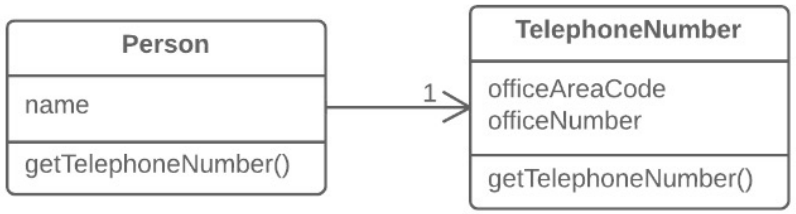
D E M O

Inline Class

- https://refactoring.guru/inline-class

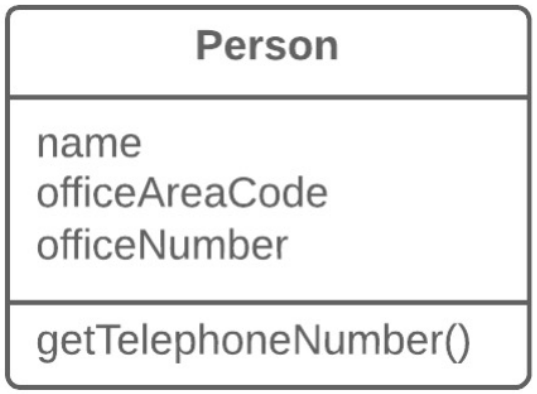
Problem

A class does almost nothing and isn't responsible for anything, and no additional responsibilities are planned for it.



Solution

Move all features from the class to another one.



Organizing Data

Encapsulate Field

- https://refactoring.guru/encapsulate-field

Problem

You have a public field.

```
class Person {
  public String name;
}
```

Solution

Make the field private and create access methods for it.

```
class Person {
  private String name;

  public String getName() {
    return name;
  }
  public void setName(String arg) {
    name = arg;
  }
}
```

Self Encapsulate Field

- https://refactoring.guru/self-encapsulate-field

Problem

You use direct access to private fields inside a class.

Solution

Create a getter and setter for the field, and use only them for accessing the field.

```
class Range {
  private int low, high;
  boolean includes(int arg) {
    return arg >= low && arg <= high;
  }
}
```

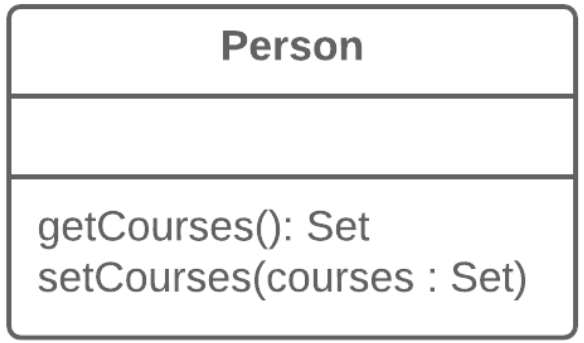
```
class Range {
  private int low, high;
  boolean includes(int arg) {
    return arg >= getLow() && arg <= getH
  }
  int getLow() {
    return low;
  }
  int getHigh() {
    return high;
  }
}
```

Encapsulate Collection

- https://refactoring.guru/encapsulate-collection

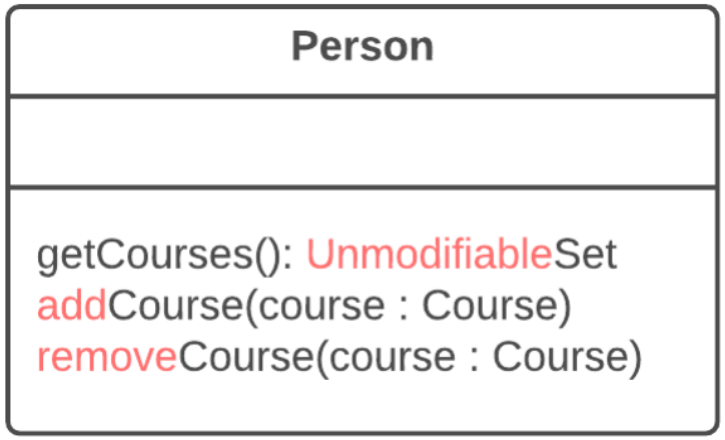
Problem

A class contains a collection field and a simple getter and setter for working with the collection.



Solution

Make the getter-returned value read-only and create methods for adding/deleting elements of the collection.

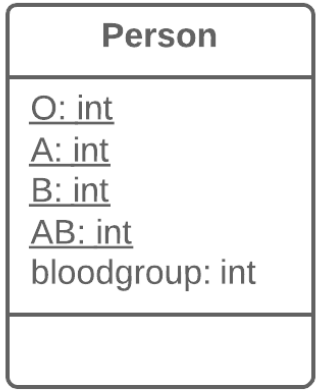


Replace Type Code with Class / Enum

- https://refactoring.guru/replace-type-code-with-class

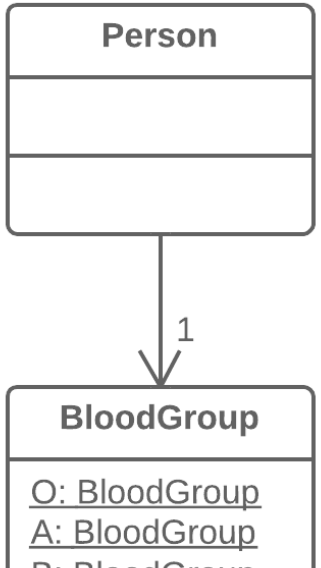
Problem

A class has a field that contains type code. The values of this type aren't used in operator conditions and don't affect the behavior of the program.



Solution

Create a new class and use its objects instead of the type code values.



Replace Type Code with Subclass

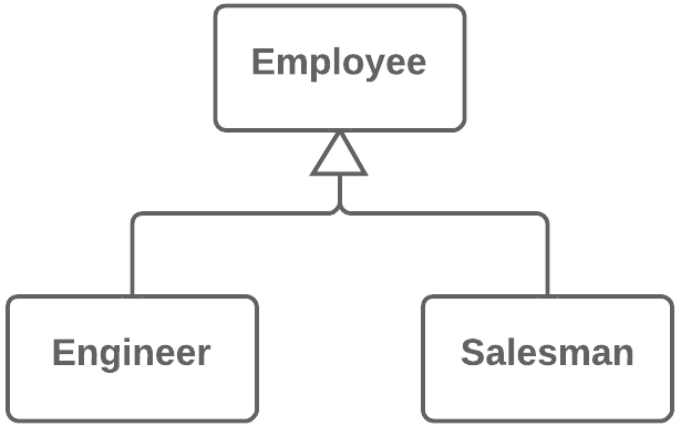
- https://refactoring.guru/replace-type-code-with-subclasses

Problem

You have a coded type that directly affects program behavior (values of this field trigger various code in conditionals).

Solution

Create subclasses for each value of the coded type. Then extract the relevant behaviors from the original class to these subclasses. Replace the control flow code with polymorphism.

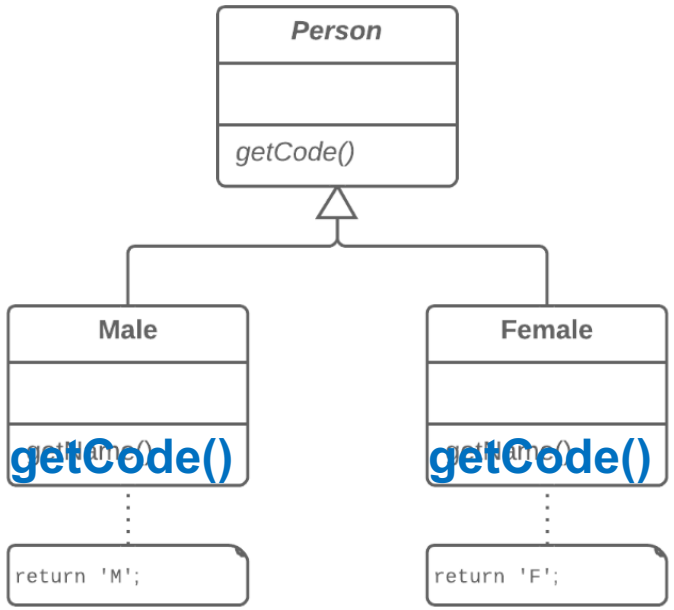


Replace Subclass with field(s)

- **https://refactoring.guru/replace-subclass-with-fields**

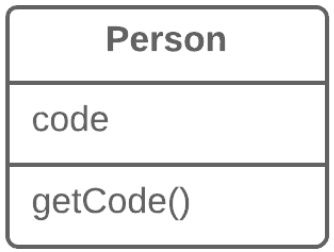
Problem

You have subclasses differing only in their (constant-returning) methods.



Solution

Replace the methods with fields in the parent class and delete the subclasses.



Replace Magic Number with Symbolic Constant

- https://refactoring.guru/replace-magic-number-with-symbolic-constant => Extract Constant

Problem

Your code uses a number that has a certain meaning to it.

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

Solution

Replace this number with a constant that has a human-readable name explaining the meaning of the number.

```
static final double GRAVITATIONAL_CONSTAN

double potentialEnergy(double mass, doubl
  return mass * height * GRAVITATIONAL_CO
}
```

Simplyfing Conditionals

Decompose Conditional

- https://refactoring.guru/decompose-conditional => n * Extract Method

Problem

You have a complex conditional (`if-then`/`else` or `switch`).

```
if (date.before(SUMMER_START) || date.aft
  charge = quantity * winterRate + winter
}
else {
  charge = quantity * summerRate;
}
```

Solution

Decompose the complicated parts of the conditional into separate methods: the condition, `then` and `else`.

```
if (isSummer(date)) {
  charge = summerCharge(quantity);
}
else {
  charge = winterCharge(quantity);
}
```

Consolidate Conditional Expression

- https://refactoring.guru/consolidate-conditional-expression => Extract Method

Problem

You have multiple conditionals that lead to the same result or action.

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

Solution

Consolidate all these conditionals in a single expression.

```
double disabilityAmount() {
  if (isNotEligibleForDisability()) {
    return 0;
  }
  // Compute the disability amount.
  // ...
}
```

Replace Nested Conditional with Guard

- https://refactoring.guru/replace-nested-c

Problem

You have a group of nested conditionals and it's hard to determine the normal flow of code execution.

```
public double getPayAmount() {
  double result;
  if (isDead){
    result = deadAmount();
  }
  else {
    if (isSeparated){
      result = separatedAmount();
    }
    else {
      if (isRetired){
        result = retiredAmount();
      }
      else{
        result = normalPayAmount();
      }
    }
  }
  return result;
}
```

Solution

Isolate all special checks and edge cases int separate clauses and place them before the main checks. Ideally, you should have a "flat list of conditionals, one after the other.

```
public double getPayAmount() {
  if (isDead){
    return deadAmount();
  }
  if (isSeparated){
    return separatedAmount();
  }
  if (isRetired){
    return retiredAmount();
  }
  return normalPayAmount();
}
```

https://refactoring.guru/replace-nested-conditional-with-guard-clauses

Consolidate Duplicate Conditional Fragments

- https://refactoring.guru/consolidate-duplicate-conditional-fragments

Problem

Identical code can be found in all branches of a conditional.

```
if (isSpecialDeal()) {
  total = price * 0.95;
  send();
}
else {
  total = price * 0.98;
  send();
}
```

Solution

Move the code outside of the conditional.

```
if (isSpecialDeal()) {
  total = price * 0.95;
}
else {
  total = price * 0.98;
}
send();
```

Replace Conditional with Polymorphism

- https://refactoring.guru/replace-conditional-with-polymorphism

Problem

You have a conditional that performs various actions depending on object type or properties.

Solution

Create subclasses matching the branches of the conditional. In them, create a shared method and move code from the corresponding branch of the conditional to it. Then replace the conditional with the relevant method call. The result is that the proper implementation will be attained via polymorphism depending on the object class.

```
class Bird {
  // ...
  double getSpeed() {
    switch (type) {
      case EUROPEAN:
        return getBaseSpeed();
      case AFRICAN:
        return getBaseSpeed() - getLoadFa
      case NORWEGIAN_BLUE:
        return (isNailed) ? 0 : getBaseSp
    }
    throw new RuntimeException("Should be
  }
}
```

```
abstract class Bird {
  // ...
  abstract double getSpeed();
}

class European extends Bird {
  double getSpeed() {
    return getBaseSpeed();
  }
}
class African extends Bird {
  double getSpeed() {
    return getBaseSpeed() - getLoadFactor
  }
}
class NorwegianBlue extends Bird {
  double getSpeed() {
    return (isNailed) ? 0 : getBaseSpeed(
  }
}

// Somewhere in client code
speed = bird.getSpeed();
```

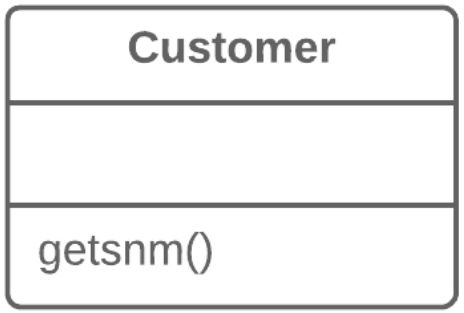
Simplyfing Method Calls

Rename Method

- https://refactoring.guru/rename-method => Rename

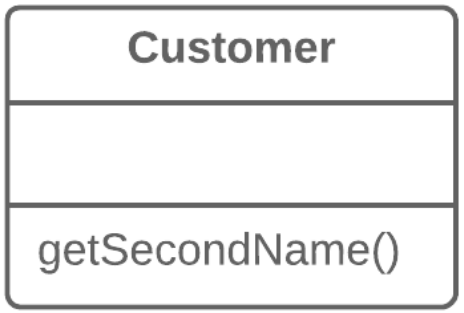
Problem

The name of a method doesn't explain what the method does.



Solution

Rename the method.



Add / Remove Parameter

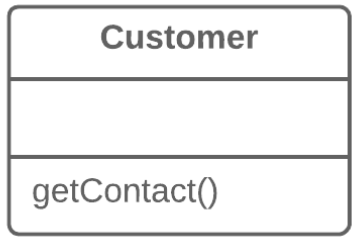
- https://refactoring.guru/add-parameter => Change Method Signature
- https://refactoring.guru/remove-parameter => Change Method Signature

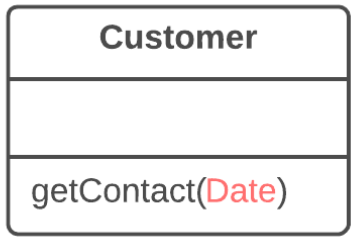
Problem

A method doesn't have enough data to perform certain actions.

Solution

Create a new parameter to pass the necessary data.



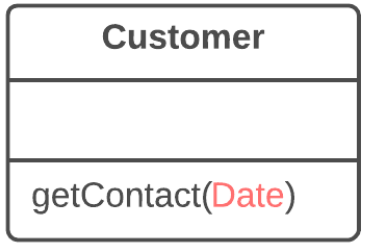


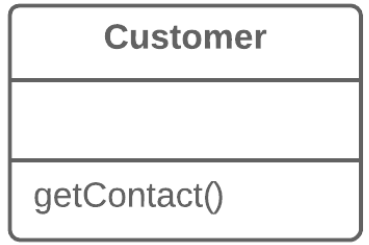
Problem

A parameter isn't used in the body of a method.

Solution

Remove the unused parameter.





Introduce Parameter Object

- https://refactoring.guru/introduce-parameter-object => Introduce Parameter Object

Problem

Your methods contain a repeating group of parameters.

Customer
amountInvoicedIn (start : Date, end : Date) amountReceivedIn (start : Date, end : Date) amountOverdueIn (start : Date, end : Date)

Solution

Replace these parameters with an object.

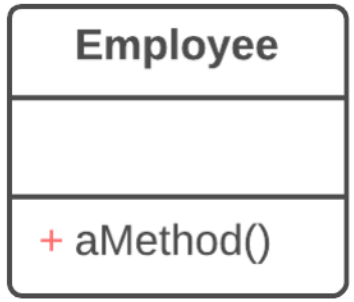
Customer
amountInvoicedIn (date : DateRange) amountReceivedIn (date : DateRange) amountOverdueIn (date : DateRange)

Hide Method

- https://refactoring.guru/hide-method

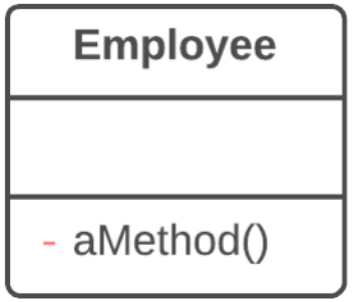
Problem

A method isn't used by other classes or is used only inside its own class hierarchy.



Solution

Make the method private or protected.



Replace Constructor with Factory Method

- https://refactoring.guru/replace-constructor-with-factory-method => Introduce Factory

Problem

You have a complex constructor that does something more than just setting parameter values in object fields.

```
class Employee {
  Employee(int type) {
    this.type = type;
  }
  // ...
}
```

Solution

Create a factory method and use it to replace constructor calls.

```
class Employee {
  static Employee create(int type) {
    employee = new Employee(type);
    // do some heavy lifting.
    return employee;
  }
  // ...
}
```

Replace Error Code with Exception

- https://refactoring.guru/replace-error-code-with-exception => Sanity Checks

Problem

A method returns a special value that indicates an error?

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

Solution

Throw an exception instead.

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

Preserve whole object*

- https://refactoring.guru/preserve-whole-object

Problem

You get several values from an object and then pass them as parameters to a method.

```
int low = daysTempRange.getLow();
int high = daysTempRange.getHigh();
boolean withinPlan = plan.withinRange(low
```

Solution

Instead, try passing the whole object.

```
boolean withinPlan = plan.withinRange(day
```

Preserve whole object*

```
public void employeeMethod(Employee employee) {
    // Some actions
    double yearlySalary = employee.getYearlySalary();
    double awards = employee.getAwards();
    double monthlySalary = getMonthlySalary(yearlySalary, awards);
    // Continue processing
}


public double getMonthlySalary(double yearlySalary, double awards) {
    return (yearlySalary + awards) / 12;
}
```

https://codegym.cc/groups/posts/196-how-refactoring-works-in-java

Preserve whole object*

```
public void employeeMethod(Employee employee) {
    // Some actions
    double yearlySalary = employee.getYearlySalary();
    double awards = employee.getAwards();
    double monthlySalary = getMonthlySalary(yearlySalary, awards);
    // Continue processing
}


public double getMonthlySalary(double yearlySalary, double awards) {
    return (yearlySalary + awards) / 12;
}
```

https://codegym.cc/groups/posts/196-how-refactoring-works-in-java

Preserve whole object*

```
public void employeeMethod(Employee employee) {
    // Some actions
    double monthlySalary = getMonthlySalary(employee);
    // Continue processing
}


public double getMonthlySalary(Employee employee) {
    return (employee.getYearlySalary() + employee.getAwards()) / 12;
}
```

https://codegym.cc/groups/posts/196-how-refactoring-works-in-java

D E M O

IntelliJ:
1) Change method signature
2+3) Inline on parameters!

Eclipse:
1) Inline method
2+3) Inline local variable
4) Extract method

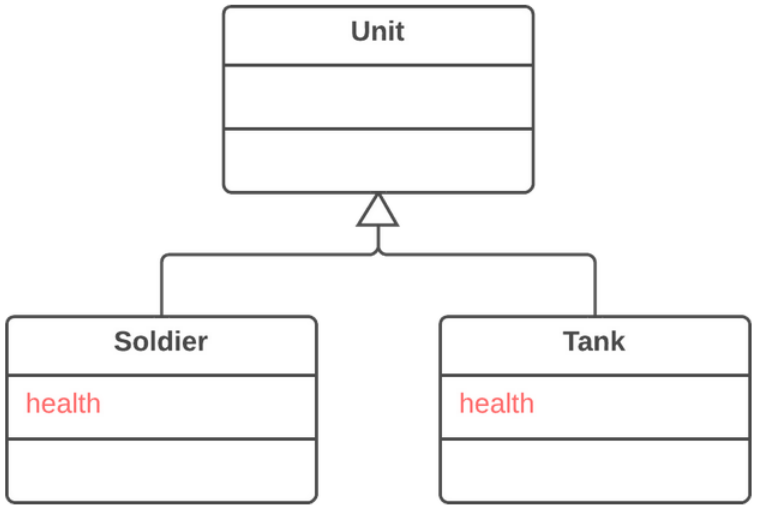
Dealing With Generalization

Pull Up Field

- https://refactoring.guru/pull-up-field => Pull Up

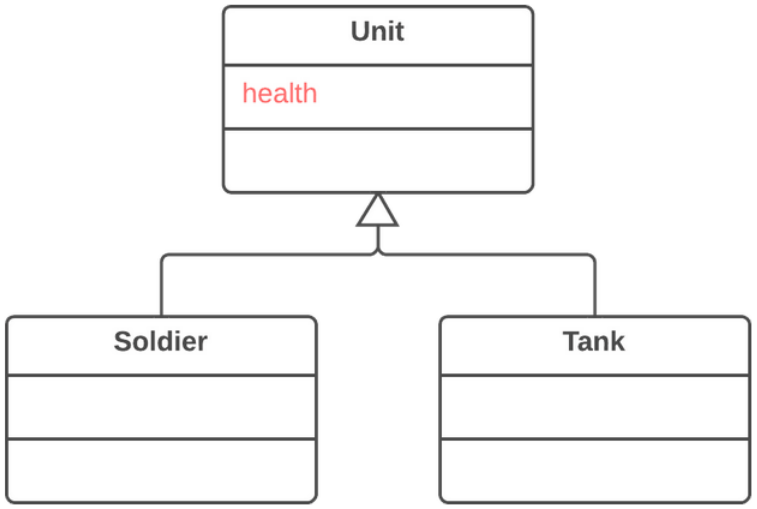
Problem

Two classes have the same field.



Solution

Remove the field from subclasses and move it to the superclass.



Pull Up Method

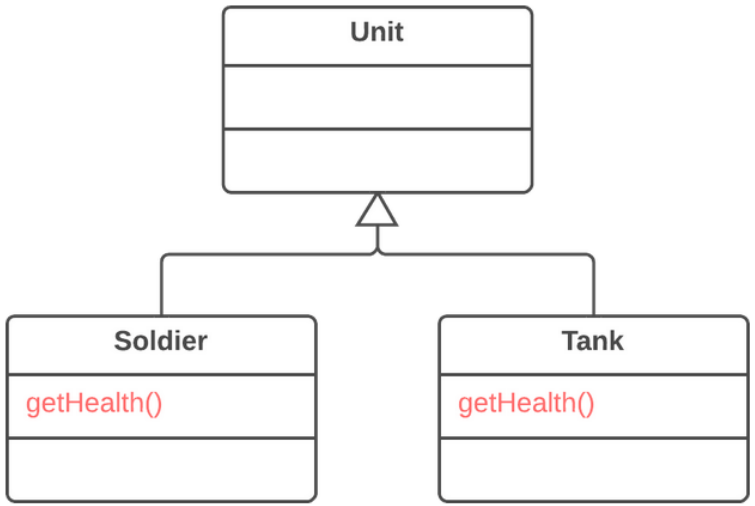
- https://refactoring.guru/pull-up-method => Pull Up

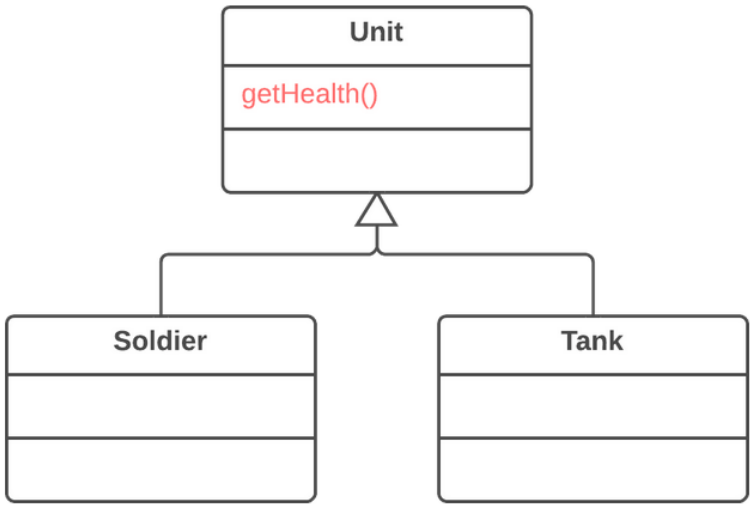
Problem

Your subclasses have methods that perform similar work.

Solution

Make the methods identical and then move them to the relevant superclass.



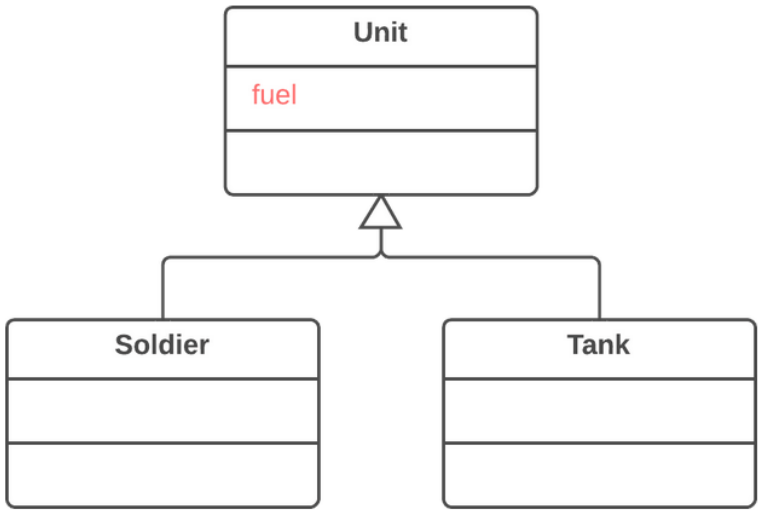


Push Down Field

- https://refactoring.guru/push-down-field => Push Down

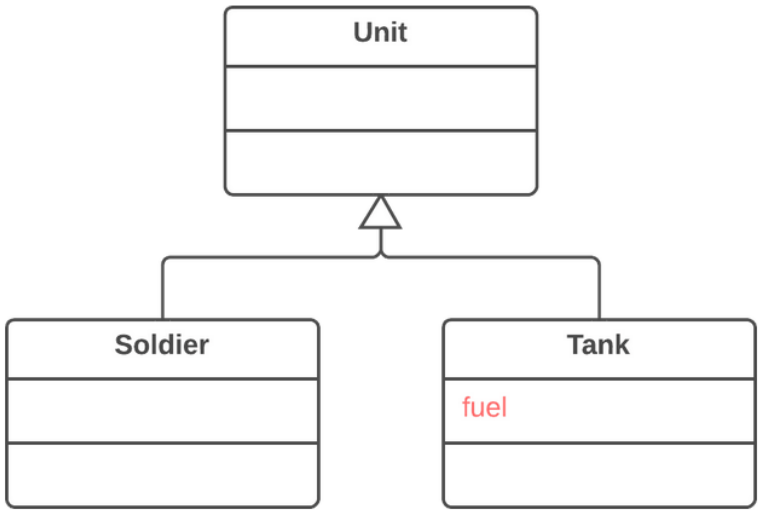
Problem

Is a field used only in a few subclasses?



Solution

Move the field to these subclasses.

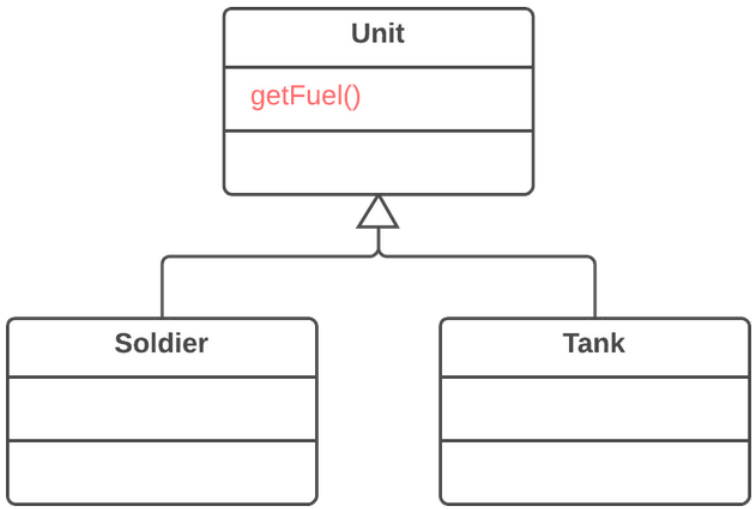


Push Down Method

- https://refactoring.guru/push-down-method => Push Down

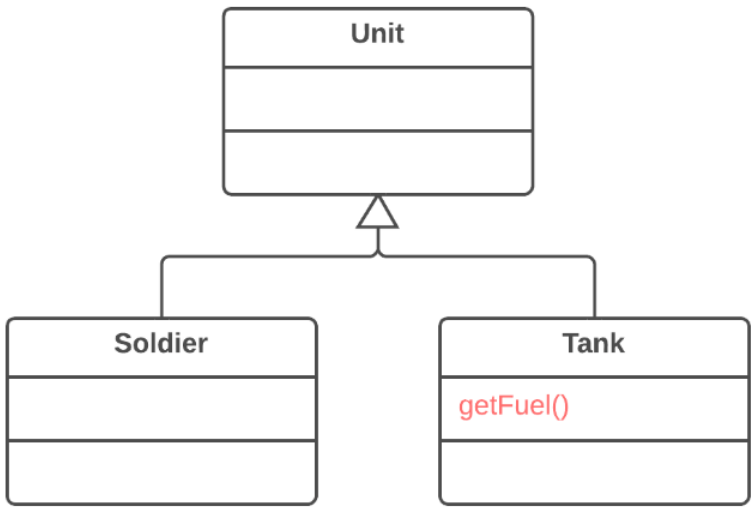
Problem

Is behavior implemented in a superclass used by only one (or a few) subclasses?



Solution

Move this behavior to the subclasses.

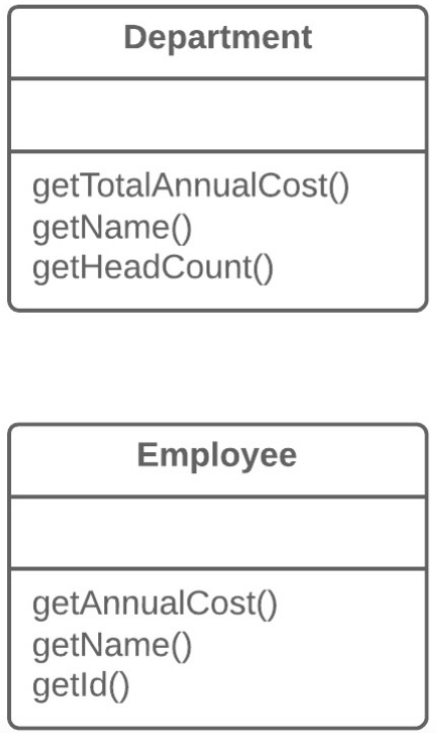


Extract Super Class

- https://refactoring.guru/extract-superclass
  => Extract Superclass

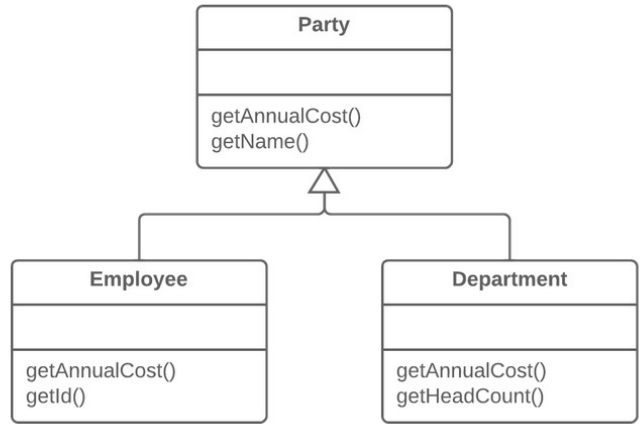
Problem

You have two classes with common fields and methods.



Solution

Create a shared superclass for them and move all the identical fields and methods to it.



Extract Sub Class

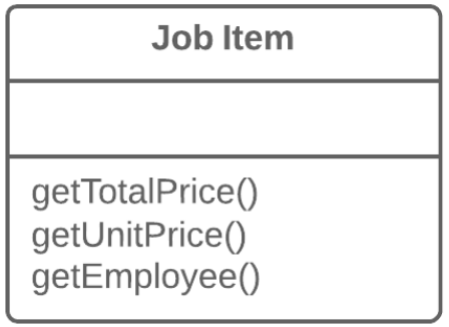
- https://refactoring.guru/extract-subclass

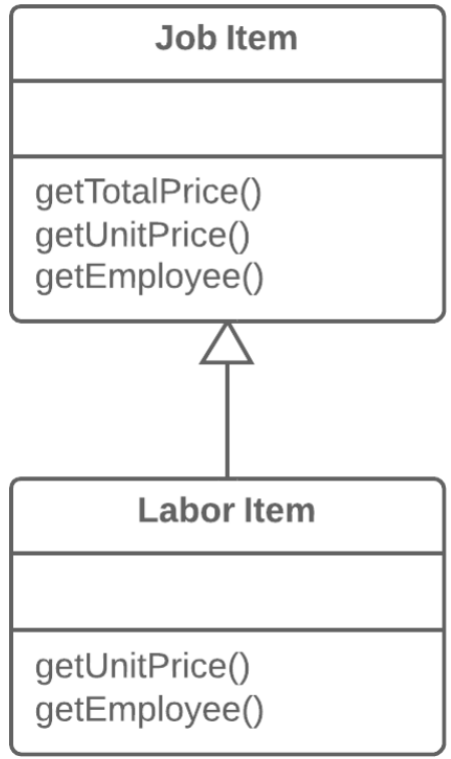
Problem

A class has features that are used only in certain cases.

Solution

Create a subclass and use it in these cases.



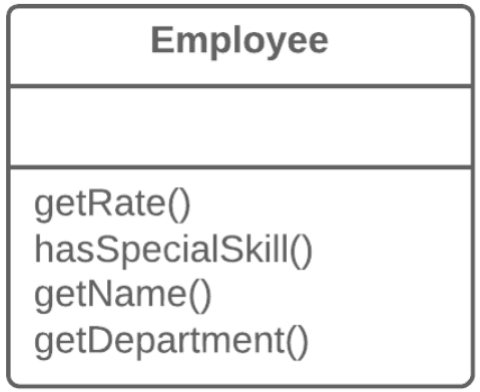


Extract Interface

- https://refactoring.guru/extract-interface:
  => Extract Interface

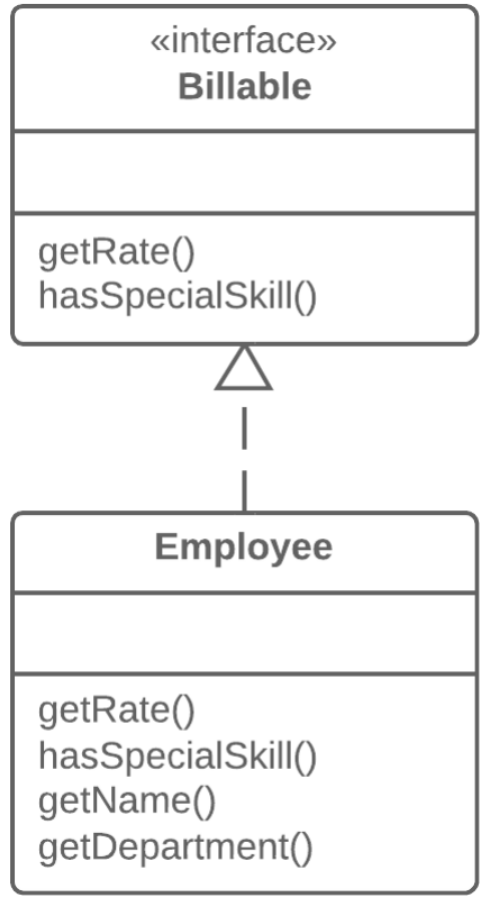
Problem

Multiple clients are using the same part of a class interface. Another case: part of the interface in two classes is the same.



Solution

Move this identical portion to its own interface.

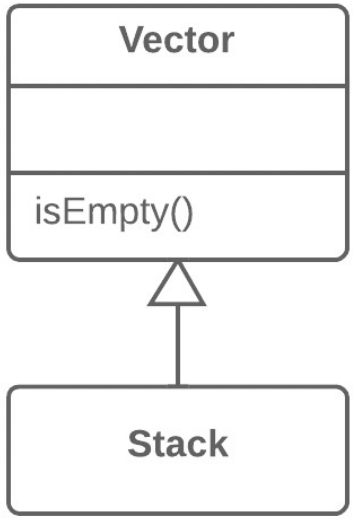


Replace Inheritance with Delegation

- https://refactoring.guru/replace-inheritance-with-delegation

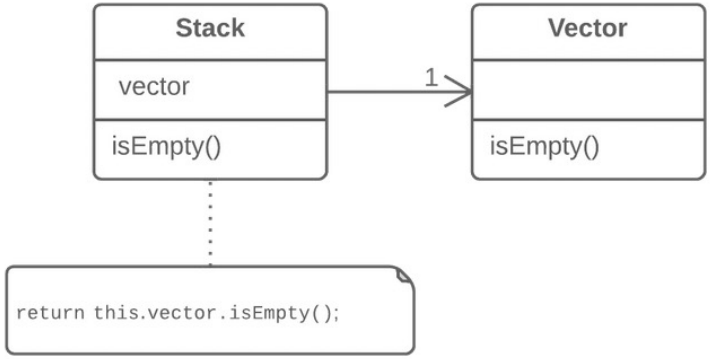
Problem

You have a subclass that uses only a portion of the methods of its superclass (or it's not possible to inherit superclass data).



Solution

Create a field and put a superclass object in it, delegate methods to the superclass object, and get rid of inheritance.



Take Aways and Pitfalls

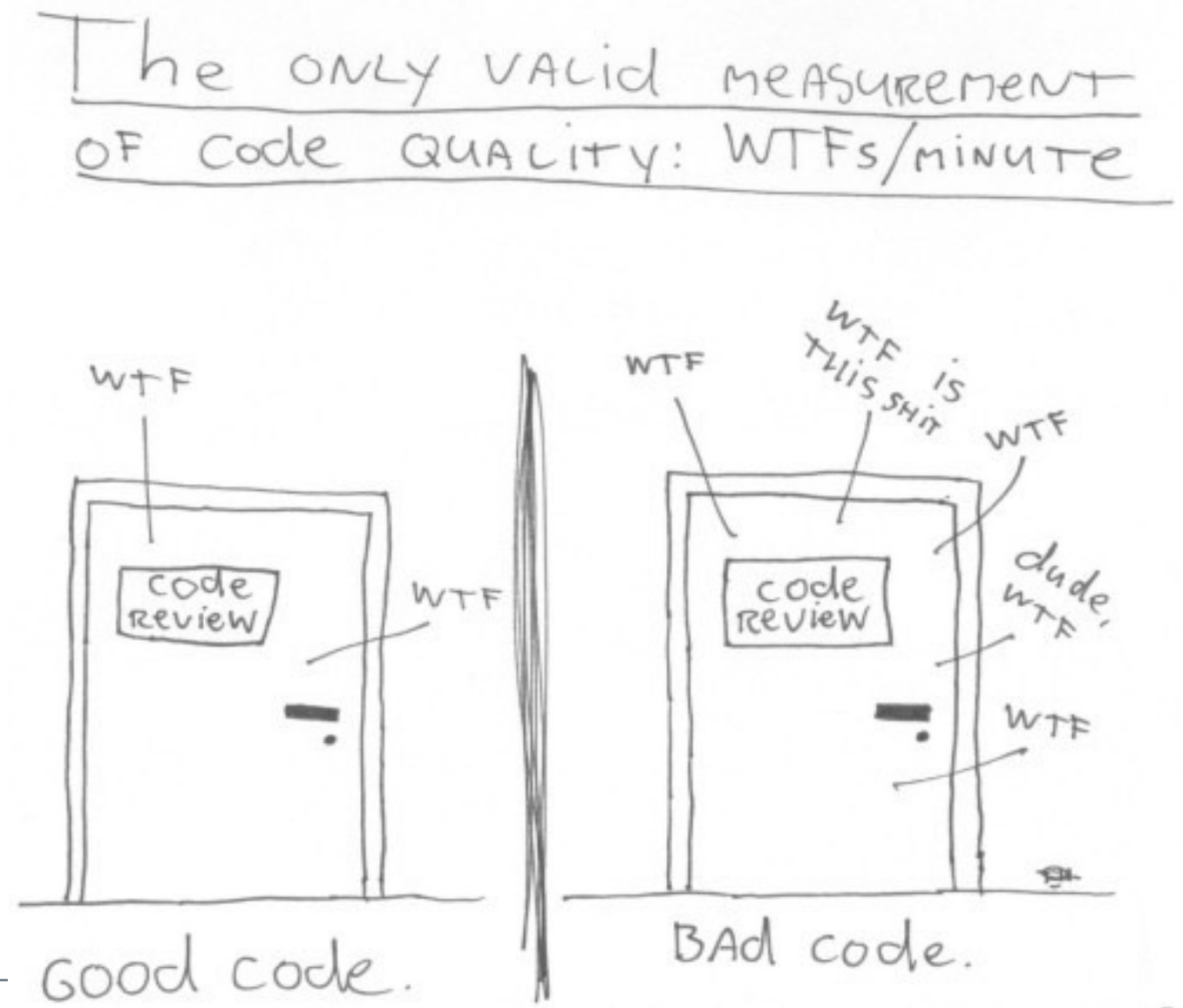
Take Away: Read vs. Write

We READ 10x more time than we WRITE

→ **Make it more readable**, even if it's harder to write

Take Away: Boyscout Rule
Leave the campground cleaner than you found it

Take Away: WTF – The TRUE measurement



Attention: Pitfalls even with Base Refactorings

- **While base refactorings should not change anything in the (externally visible) behavior, how do you ensure that?**
  - No freestyle refactorings
  - Always use the IDE automations if possible
  - **It is always advisable to use a (large) number of tests to ensure safety**
- **Be aware of possible side effects**
- **Know pitfalls like temporal coupling**

Attention: Base Refactorings & Temporal Coupling

- *Temporal coupling describes the dependencies between method calls, which must occur in a certain sequence, often in combination with side effect problematic for refactorings*

```
class TemporalCouplingExample
{
    private String text;
    private int count;

    public TemporalCouplingExample()
    {
        methodA();
        methodB();
    }

    private void methodA()
    {
        text = "Something";
    }

    private void methodB()
    {
        count = text.length();
    }
}
```

Do you spot the problem here?

Attention: Base Refactorings & Temporal Coupling

- *Temporal coupling describes the dependencies between method calls, which must occur in a certain sequence, often in combination with side effect problematic for refactorings*

```
class TemporalCouplingExample
{
    private String text;
    private int count;

    public TemporalCouplingExample()
    {
        methodA();
        methodB();
    }

    private void methodA()
    {
        text = "Something";
    }

    private void methodB()
    {
        count = text.length();
    }
}
```

```
methodB();
methodA();
```

D E M O

Temporal Coupling + side effect

```
public void doSomething()
{
    String NEW_INFO_MSG = "Very long text ....";
    setText(NEW_INFO_MSG);
    printSummary();    // seems to not fit thematically
    if (hasExpectedLength(NEW_INFO_MSG))
        System.out.println("TEXT CHANGED: " + NEW_INFO_MSG);

    // seemingly better place, but no longer the same behavior
    printSummary();


    String NEW_INFO_MSG_2 = "Another text ....";
    setText(NEW_INFO_MSG_2);
    if (hasExpectedLength(NEW_INFO_MSG_2))
        System.out.println("TEXT CHANGED: " + NEW_INFO_MSG_2);
}

private boolean hasExpectedLength(String text)
{
    return text.length() == count;
}
```

```
public void printSummary()
{
    this.count = text.length();
    System.out.println("Count: " + count);
}
```

D E M O

TAKE AWAYS

1. base refactorings are normally safe
2. but only if none of the following is involved
    1. No side effects
    2. No temporal coupling
    3. No reflection magic
3. Always use the IDE automations if possible to avoid careless mistakes
4. **Try to provide a safety net of unit tests ideally before, but definetifly during and after your refactorings**

Exercises

https://github.com/Michaeli71/ADC_BOOTCAMP_REFACTORINGS

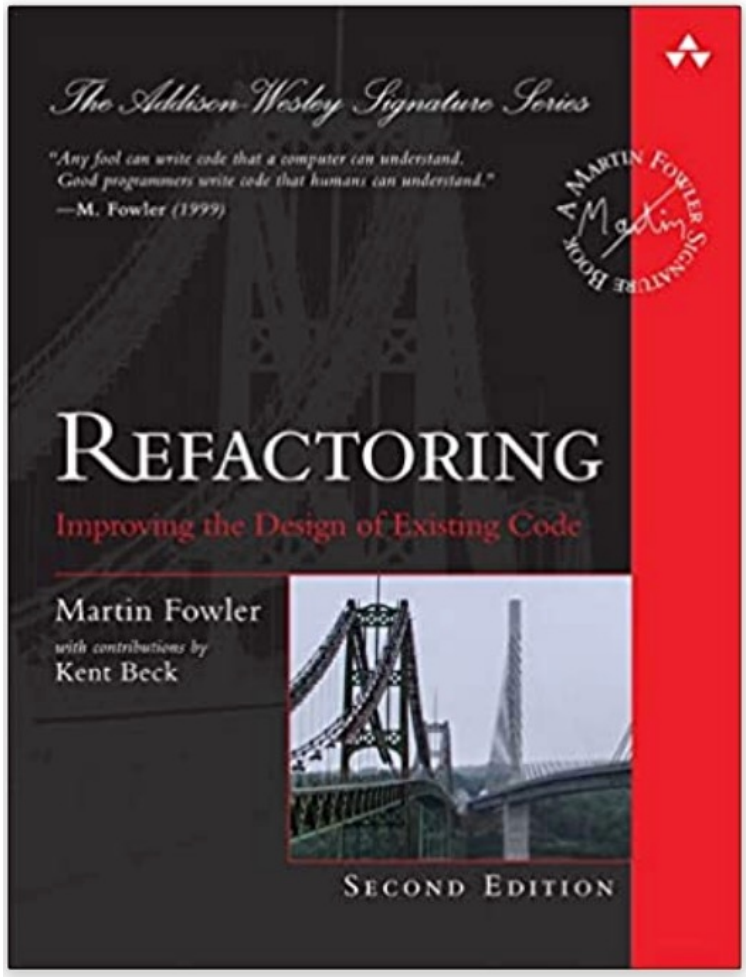
Further Info

Further Infos I

- https://www.ibm.com/developerworks/library/os-eclipse-refactoring/index.html
- https://www.jetbrains.com/help/idea/tutorial-introduction-to-refactoring.html#5db90

- https://refactoring.guru/refactoring
- http://www.tutego.de/java/refactoring/catalog/index.html
- https://www.baeldung.com/intellij-refactoring
- https://github.com/lamchau/refactoring-exercise

Further Info II



Thank You