

Refactorings

Introduction & Overview

<https://github.com/Michaeli71/RefactoringsWorkshop>

26.04.22

Inden Michael



Agenda

- Motivation und Intro / Definition
- Intro Basis-Refactorings
- **HANDS ON EXERCISES**
- Basis-Refactorings am Beispiel
- **EXERCISES!**
- Take Aways & Fallstricke

Motivation und Intro / Definition

«Jeder Dummkopf kann Code schreiben, der für Computer verständlich ist.»

«Gute Programmierer schreiben Code, den Menschen verstehen können.»

- Martin Fowler -

Refactoring Wie ist die Definition???



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.

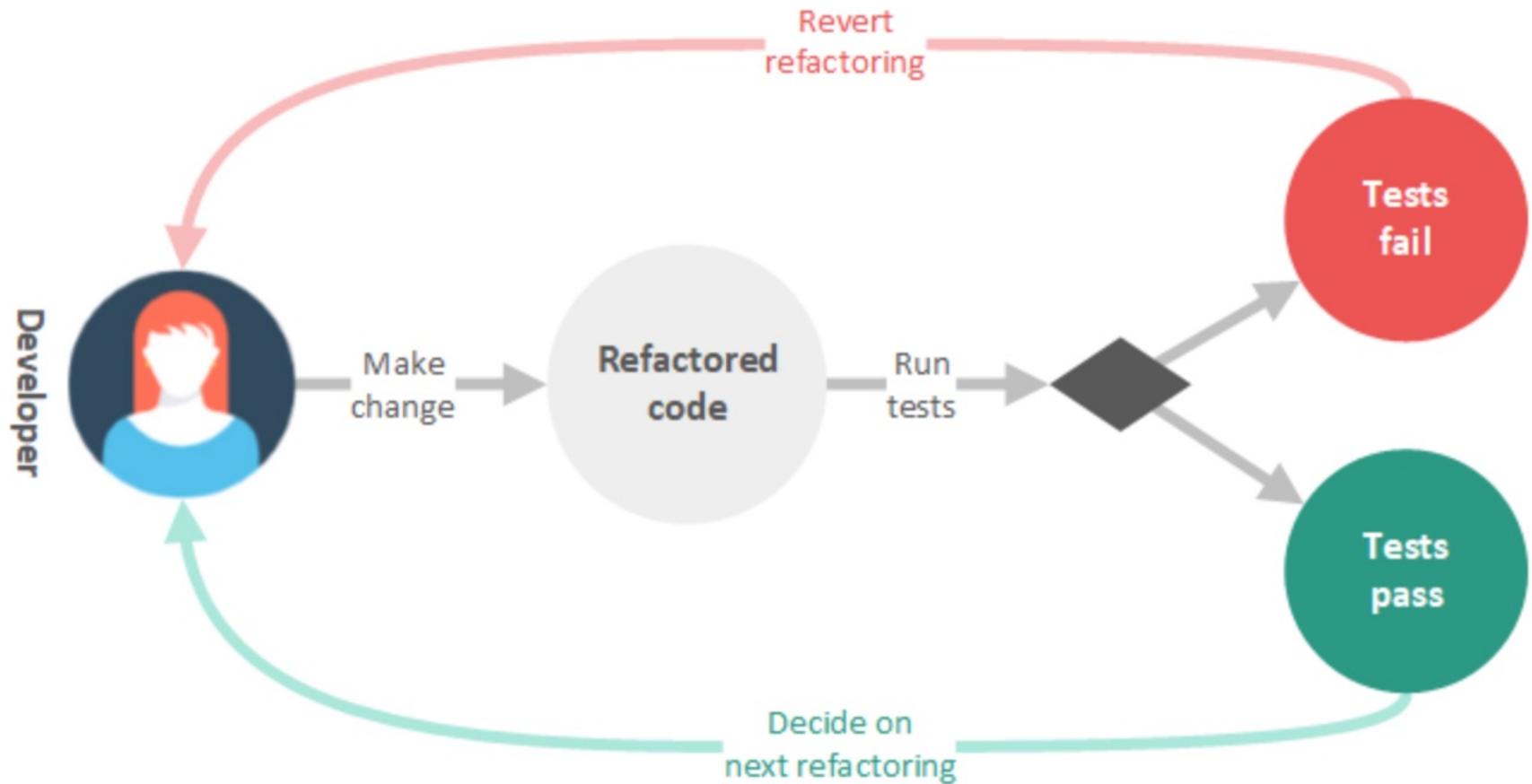
Beispiel Refactorings

```
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);
}
```

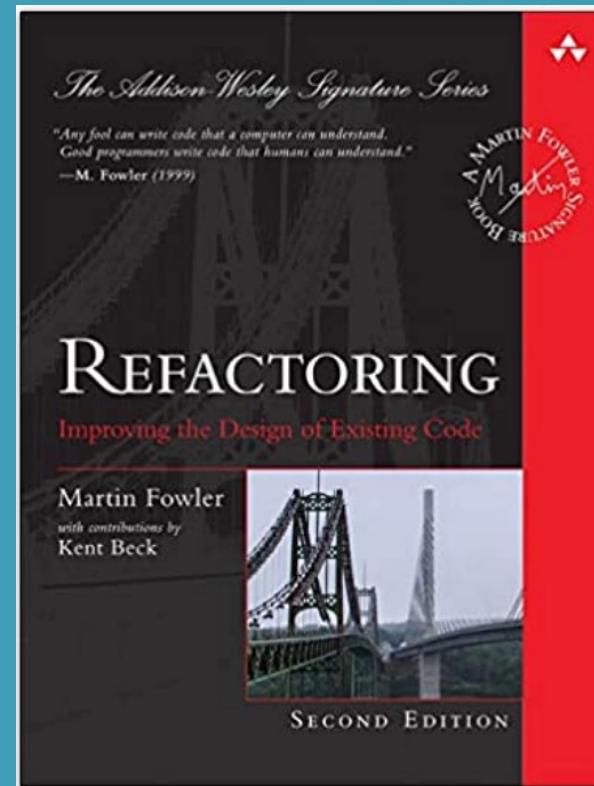
```
private static void inlined(int breite, int Höhe)
{
    System.out.println("Umfang: " + (double) (2 * (breite + Höhe)));
    System.out.println("Fläche: " + (double) (breite * Höhe));
}
```

Refactoring in Practice



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

Intro Basis-Refactorings



Allerwichtigste Basis-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

Wichtige Basis-Refactorings

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

Basis-Refactorings im Überblick

(<https://refactoring.com/catalog/>)



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;
```

Rename

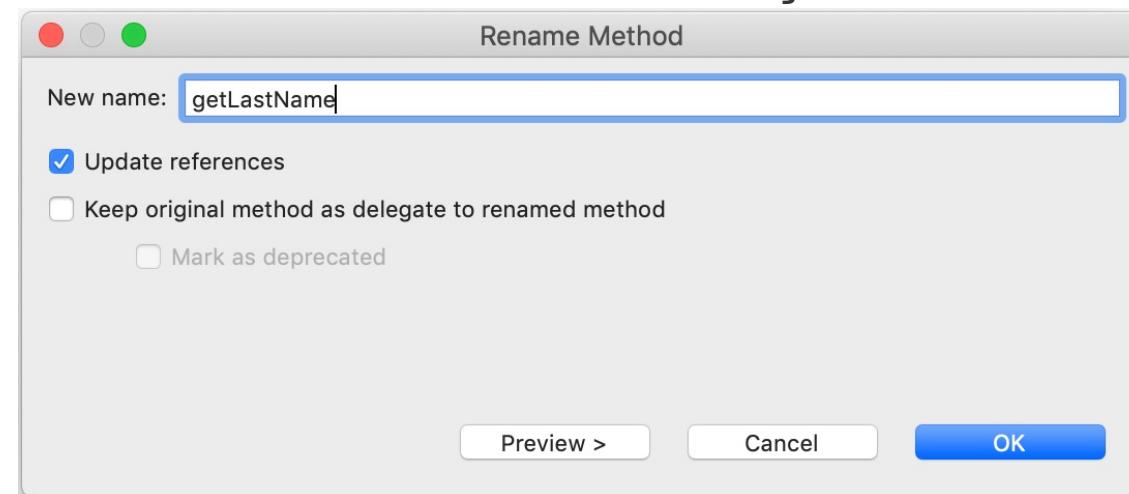
```
class Customer
{
    private String lastName = "lastname";

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

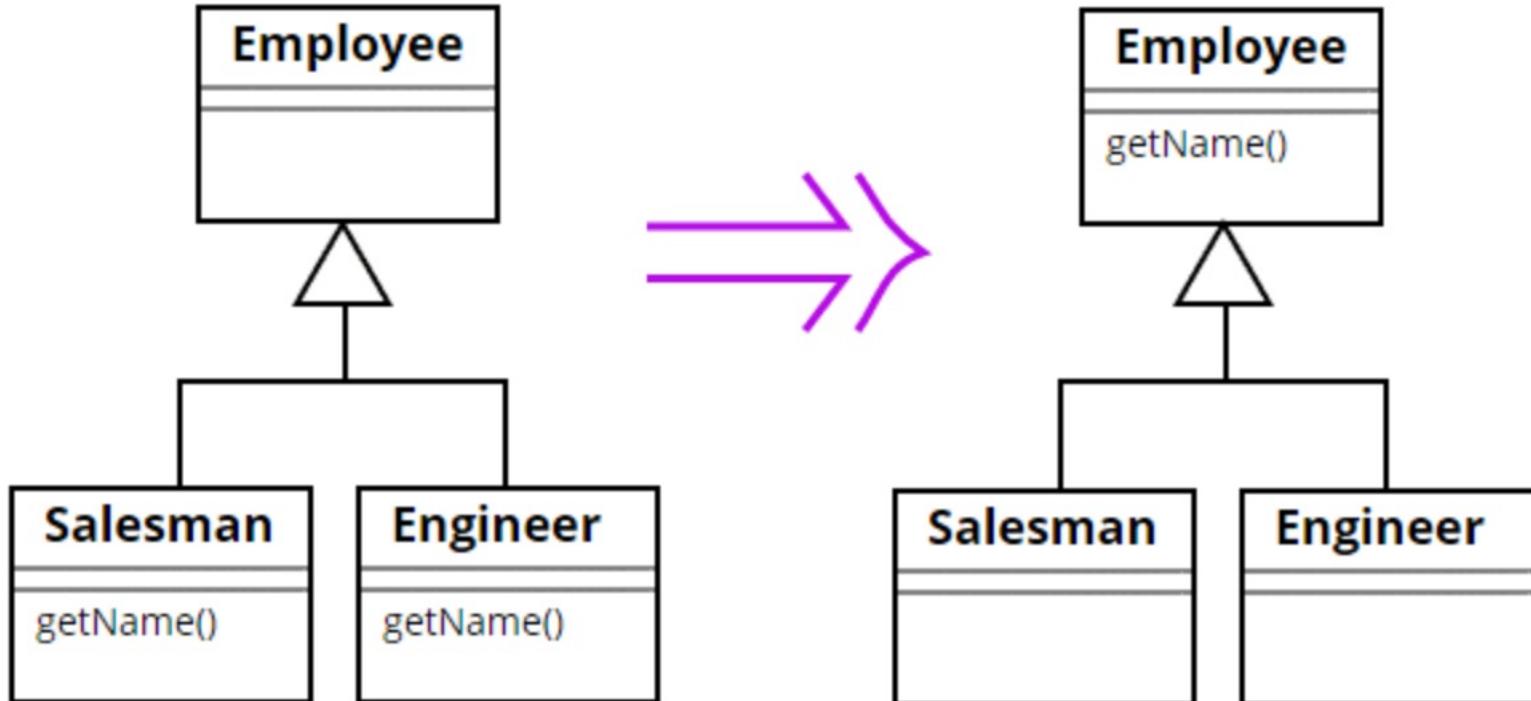
    // ...
}
```

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

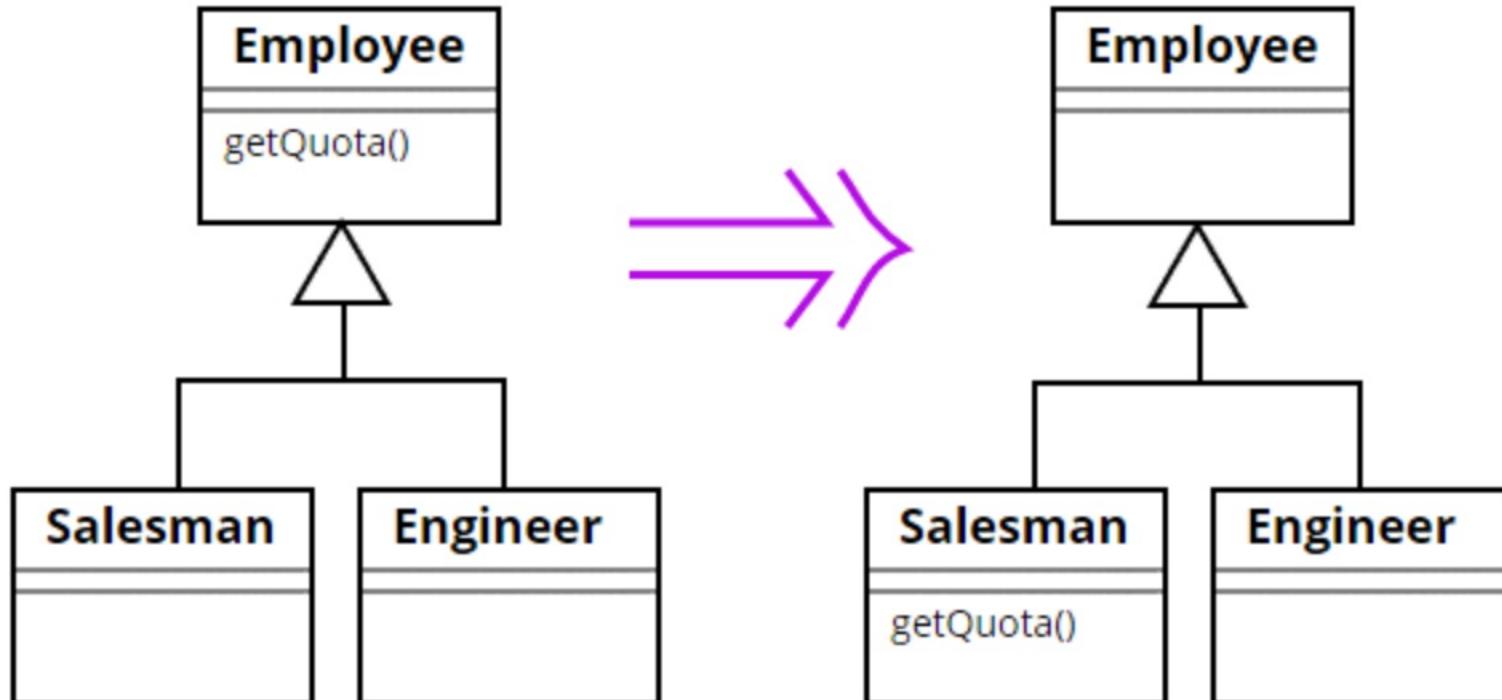
    /**
     * Method returns customer's lastname.
     */
    String getLastname()
    {
        return lastName;
    }
}
```



Pull up



Push down



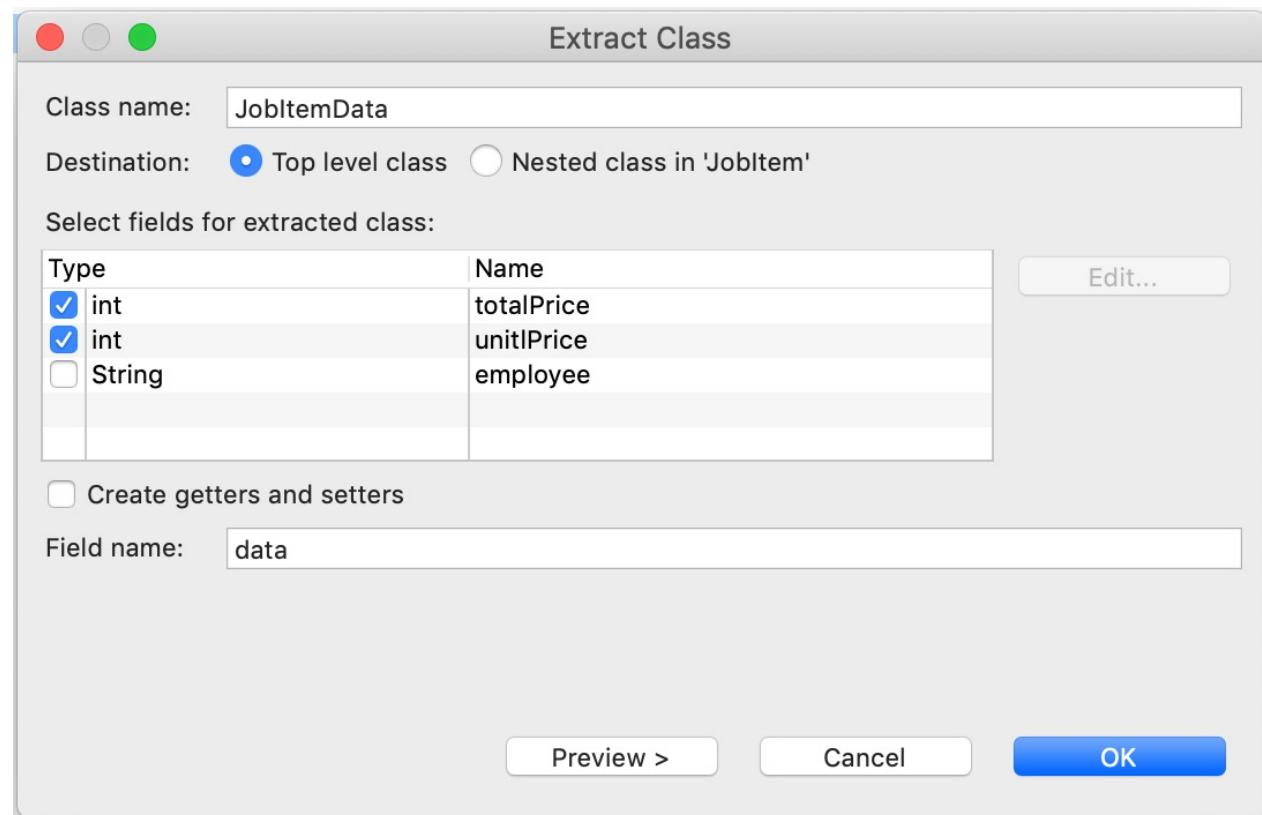
Extract Class

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

    public int getTotalPrice()
    {
        return totalPrice;
    }

    public int getUnitPrice()
    {
        return unitPrice;
    }

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



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";
    }

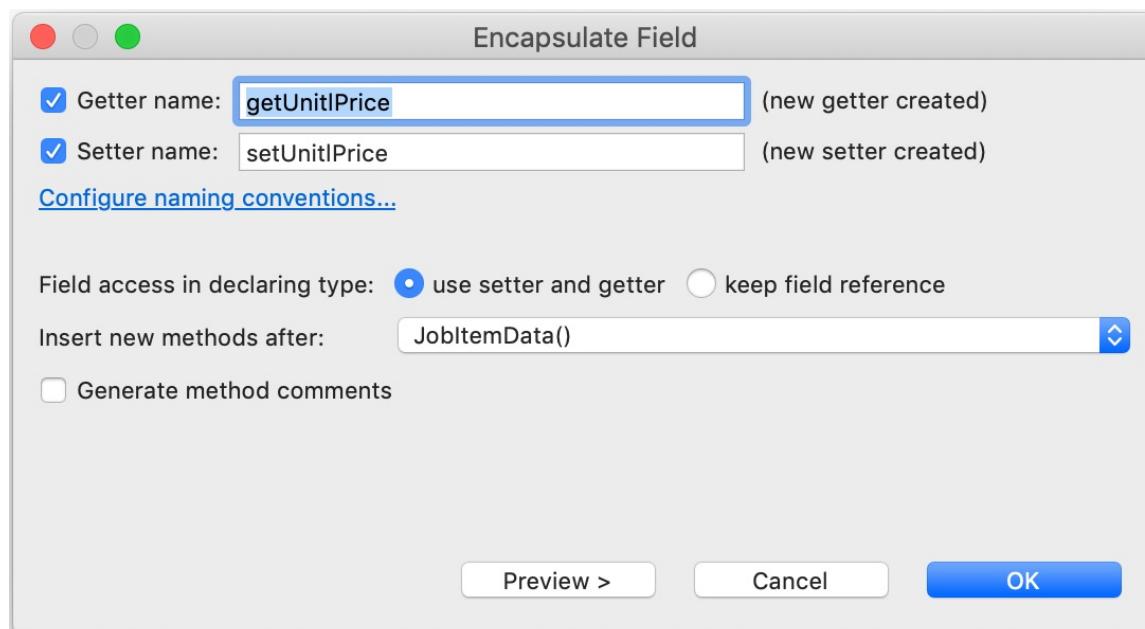
    public class JobItemData
    {
        public int totalPrice;
        public int unitPrice;

        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;
    }
}
```

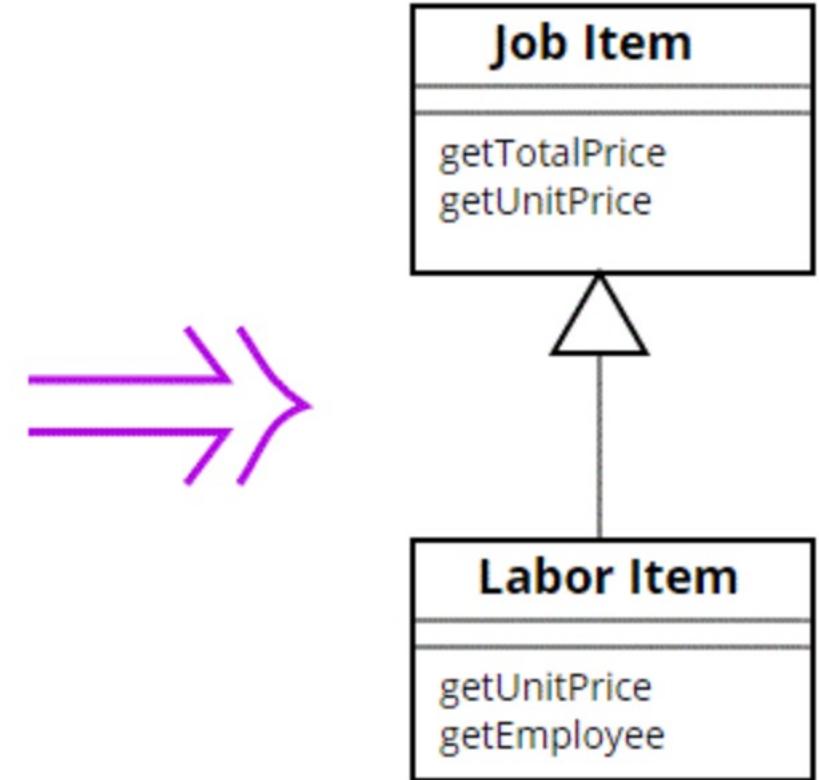
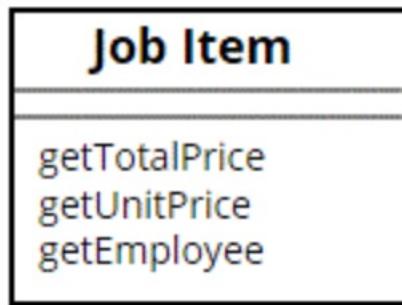
Extract Superclass

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

    public int getTotalPrice()
    {
        return totalPrice;
    }

    public int getUnitPrice()
    {
        return unitPrice;
    }

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



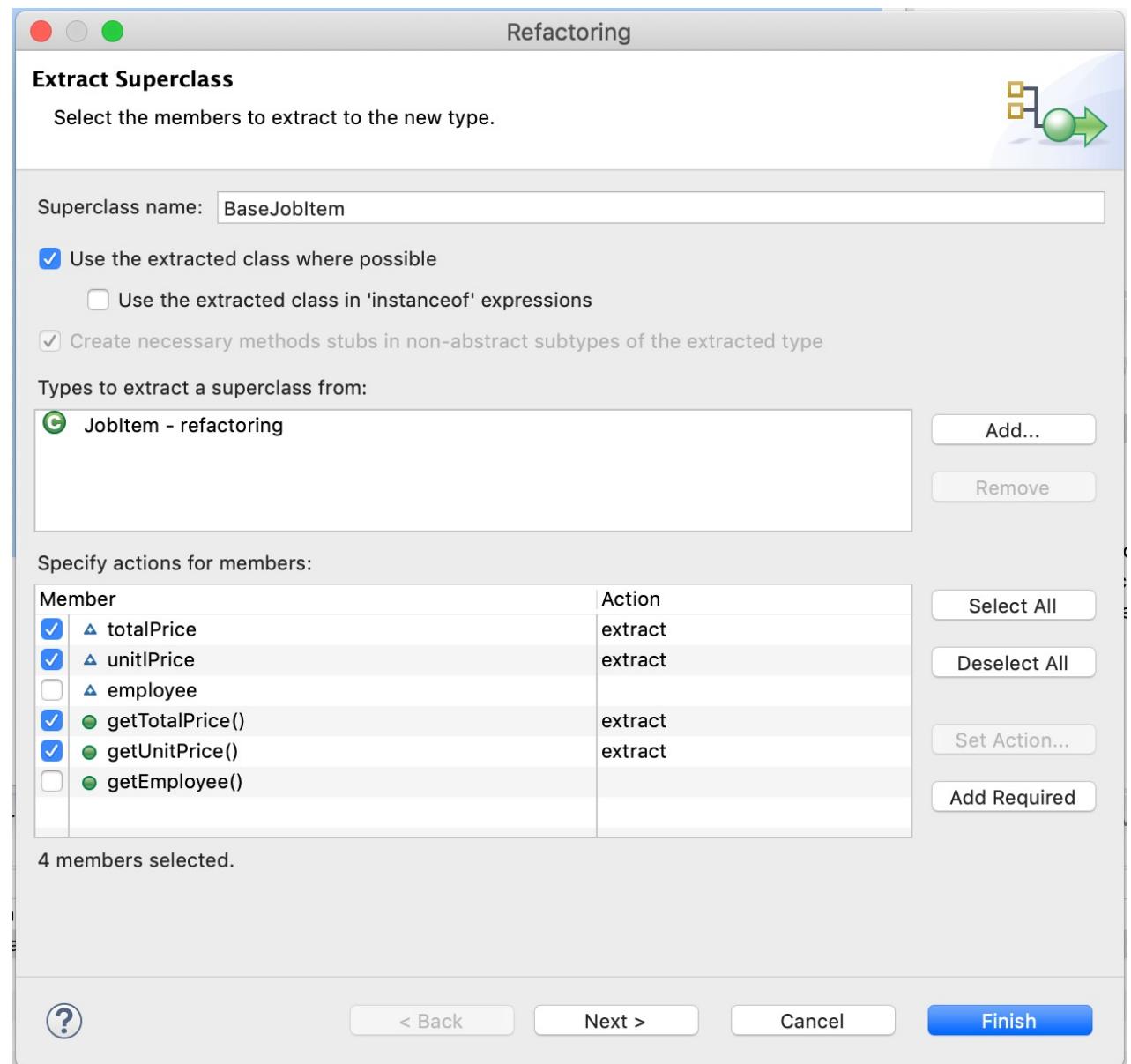
Extract Superclass

```
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";
        }
    }
}
```

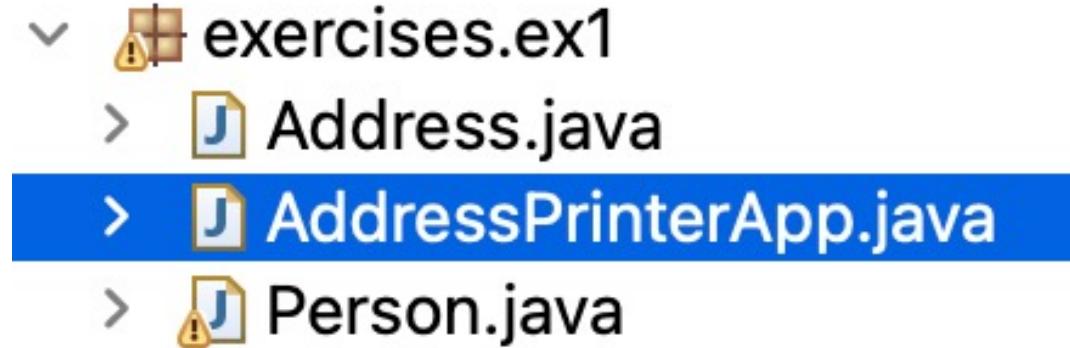
Noch Folgeaktionen nötig: Rename, Override, ...

HANDS ON – MOB REFACTORING

Vorgehen Aufgabe 1

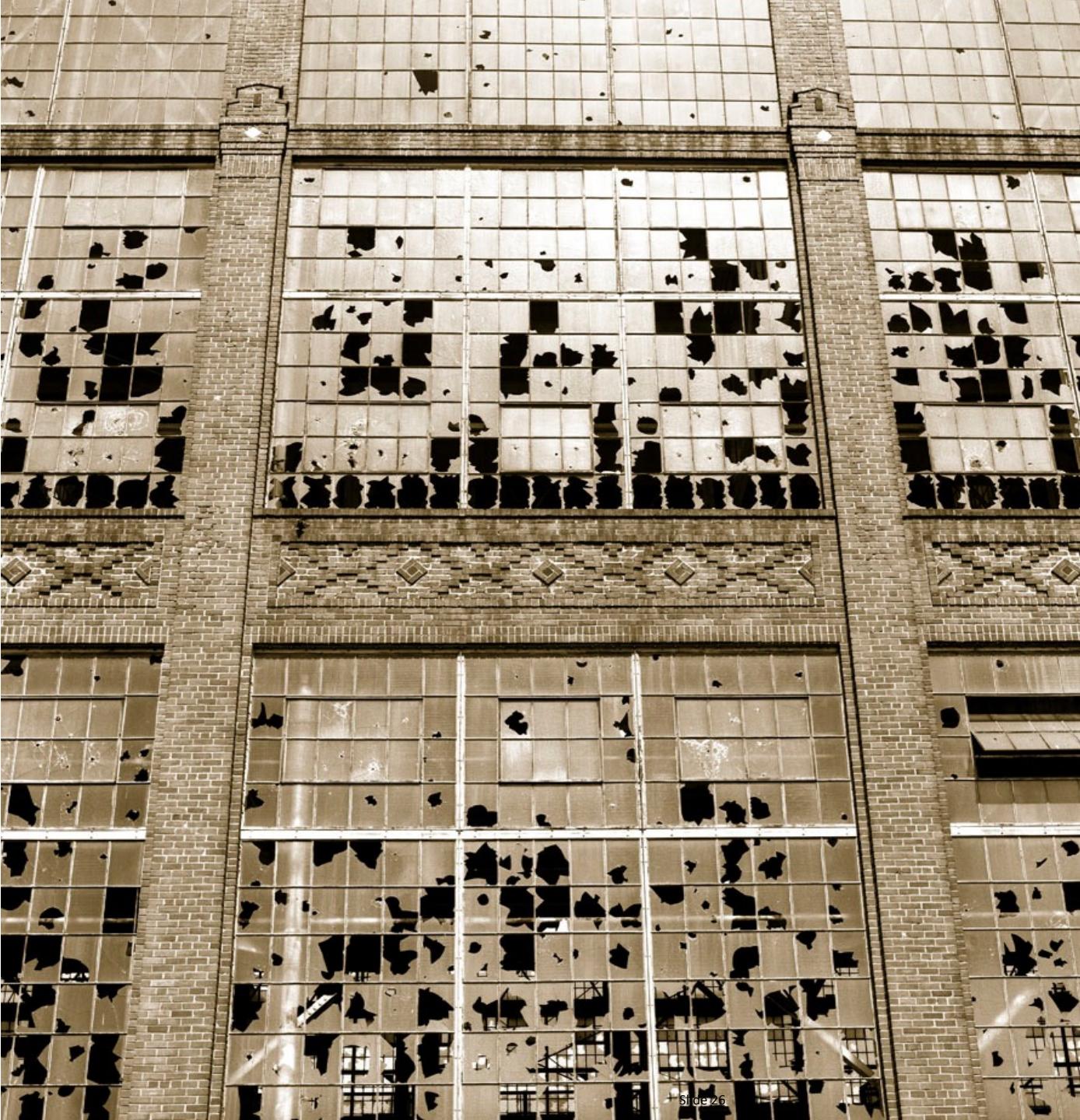
```
package exercises.ex1;
```

- Alle Klassen grob anschauen
- Klasse AddressPrinterApp ausführen
- Was sind die Probleme in Address und Person?
- Wie geht man schrittweise vor? Welche Refactorings sind nützlich?
- Wie finde ich Referenzierungen im Project (Cmd, Shift + G)?



Schlechte Angewohnheiten und Abhilfen & Refactoring-Beispiel

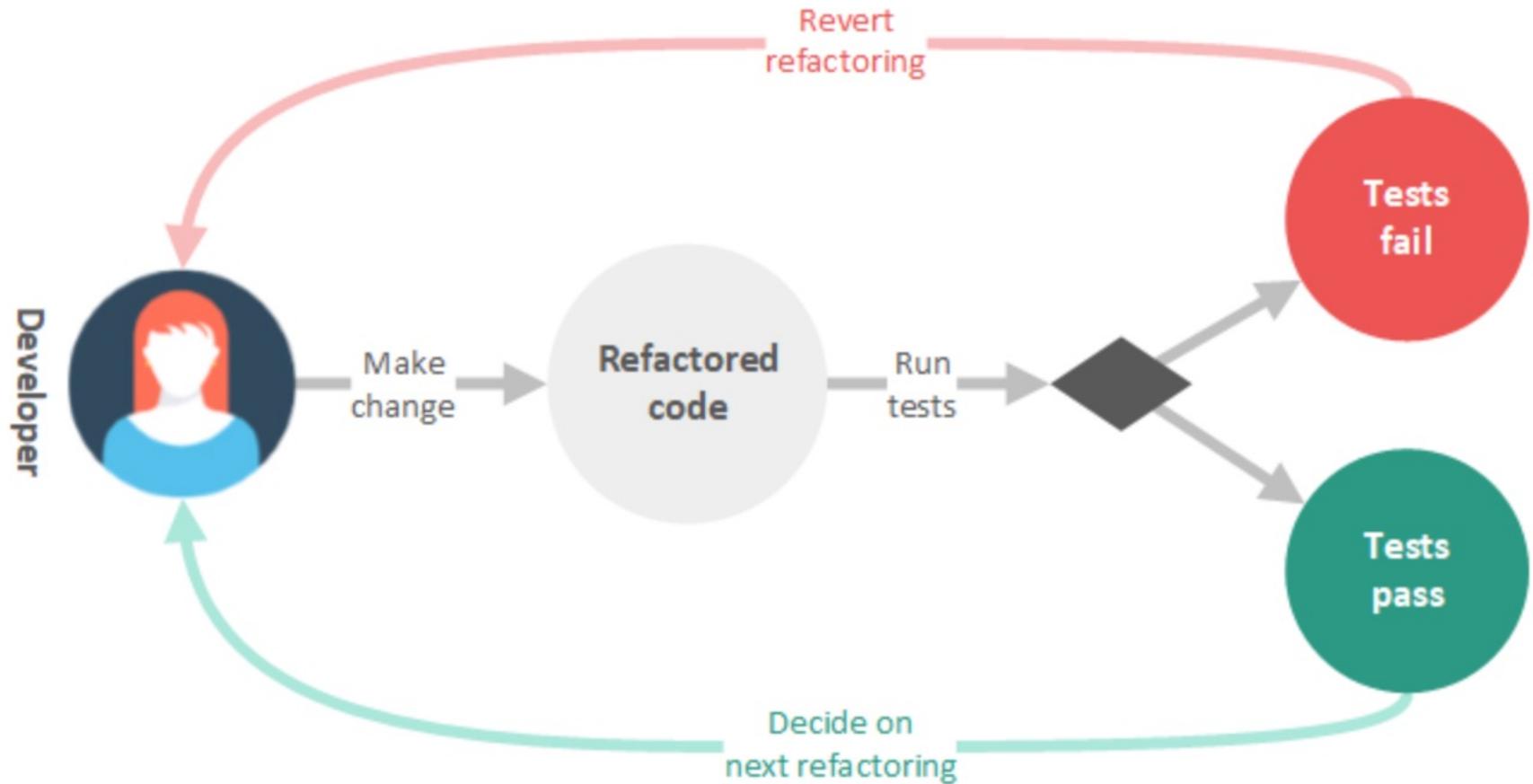
Broken Window Phenomena



Hotelroom Rule: Somebody is cleaning up for you



Refactoring in Practice



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

Refactoring in Practice – Ausgangsbasis

```
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();
}
```

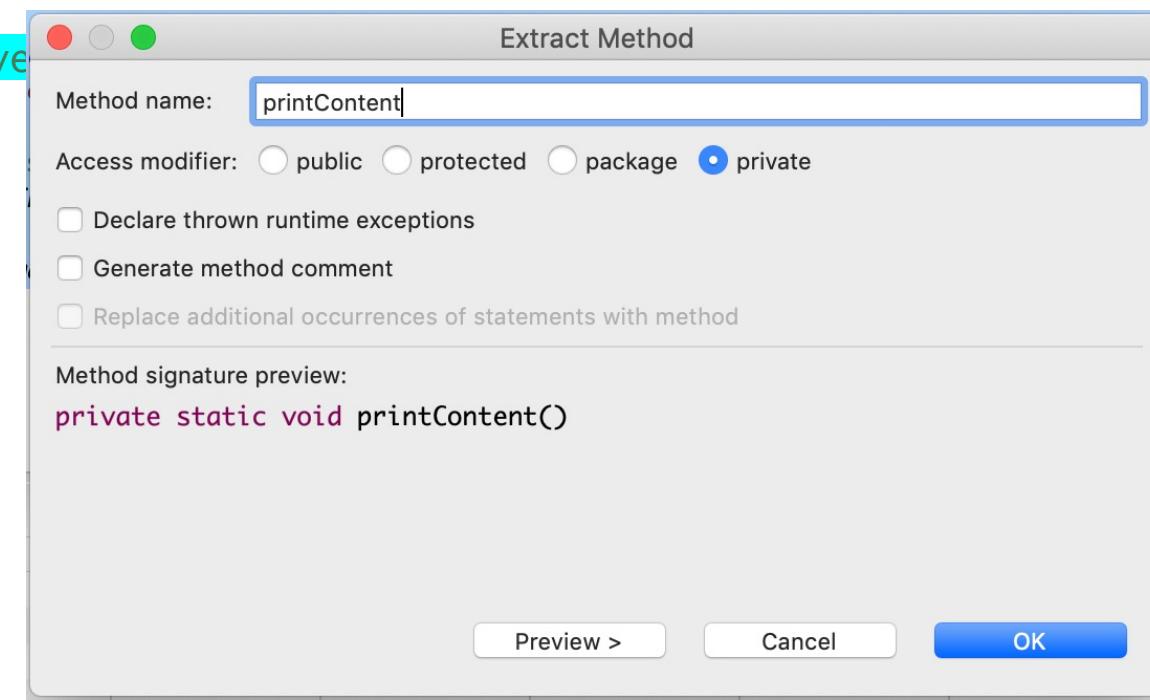
Refactoring in Practice – Beispiel EXTRACT METHOD

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

    printFooter();
}
```



Refactoring in Practice – Beispiel EXTRACT METHOD

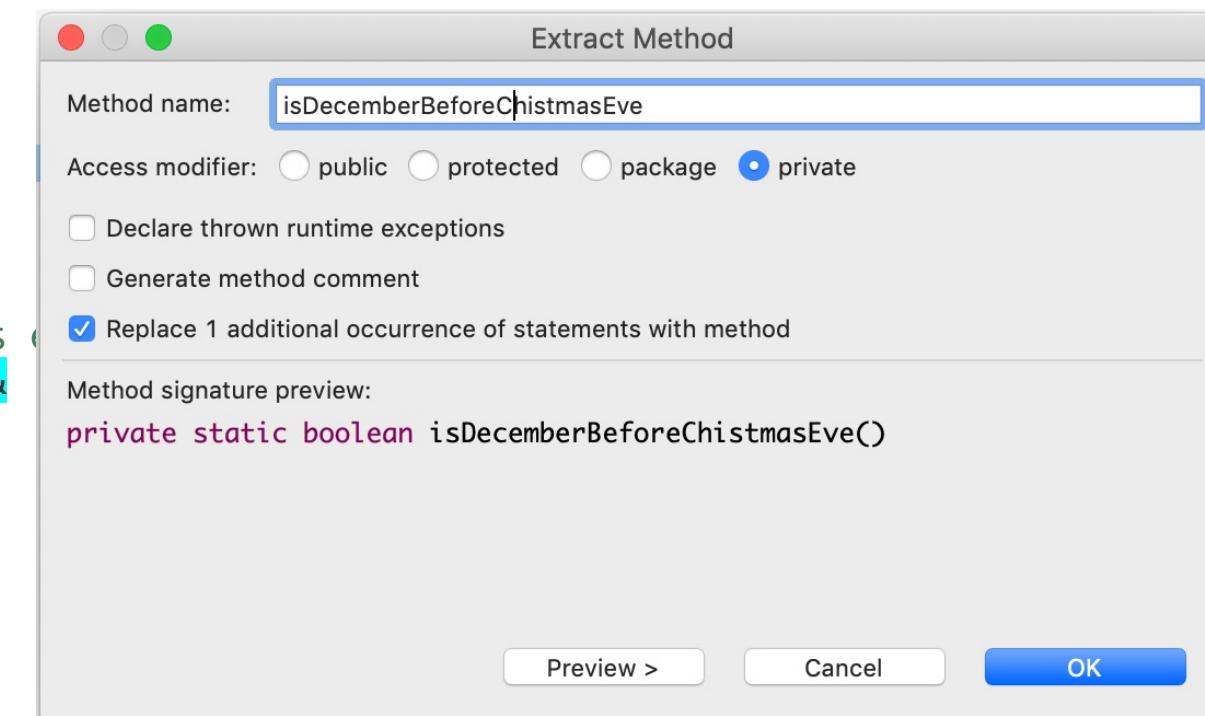
```
private static void printLetterImproved_Step1()
{
    printBanner();

    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 – Beispiel EXTRACT LOCAL VARIABLE

ad
cubum

```
private static void printLetterImproved_Step2()
{
    printBanner();
    printContent();
    printFooter();
}

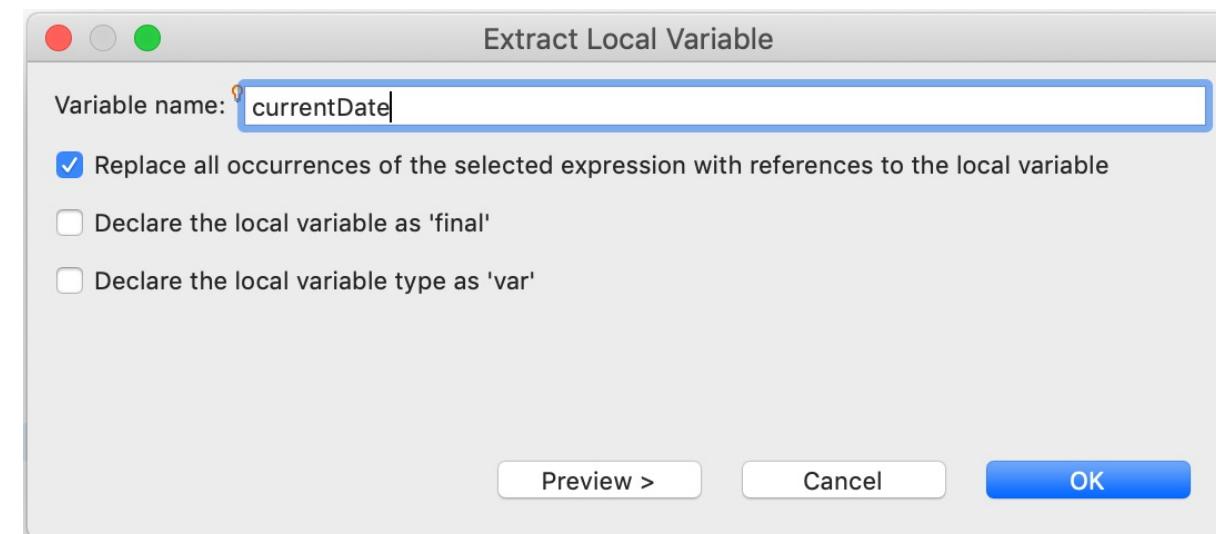
private static void printContent()
{
    for (int i = 0; i < content.size(); i++)
    {
        ContentLine cl = content.get(i);
        printContentline(cl);
    }

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

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



Short Cut:
Introduce Parameter



Refactoring in Practice – Beispiel INTRODUCE PARAMETER

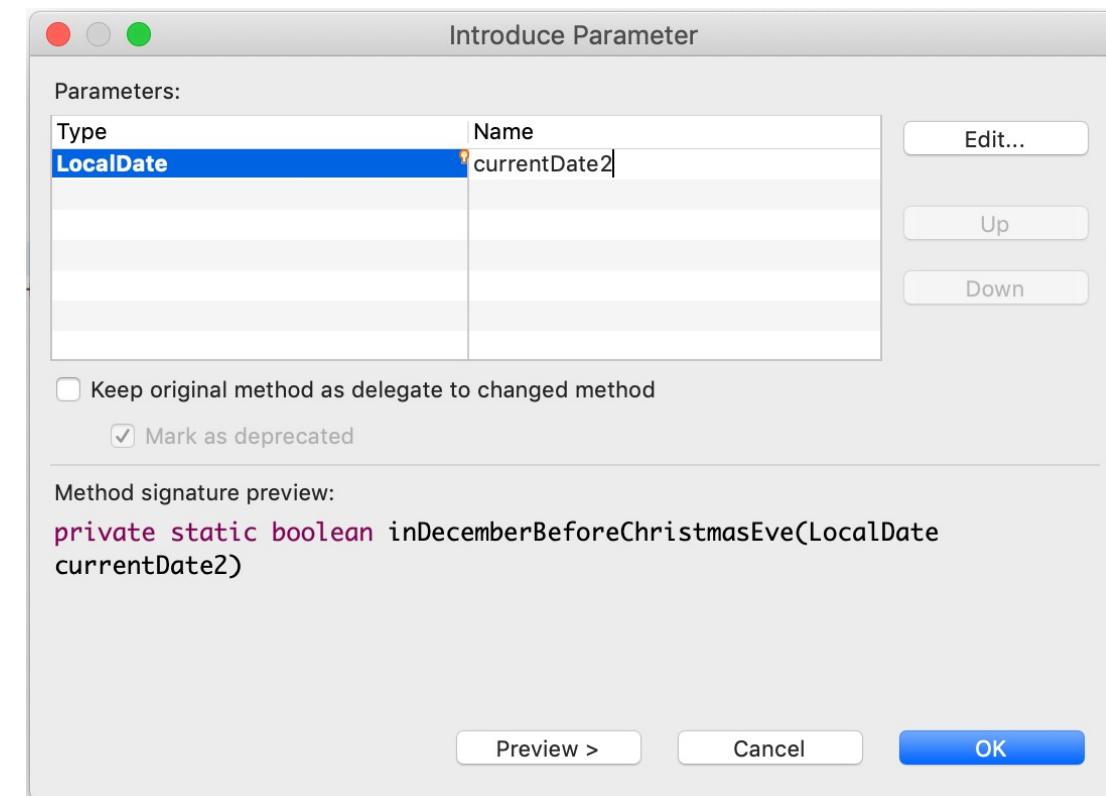
```
private static void printLetterImproved_Step3()
{
    printBanner();
    printContent();

    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 – Beispiel INLINE LOCAL VARIABLE

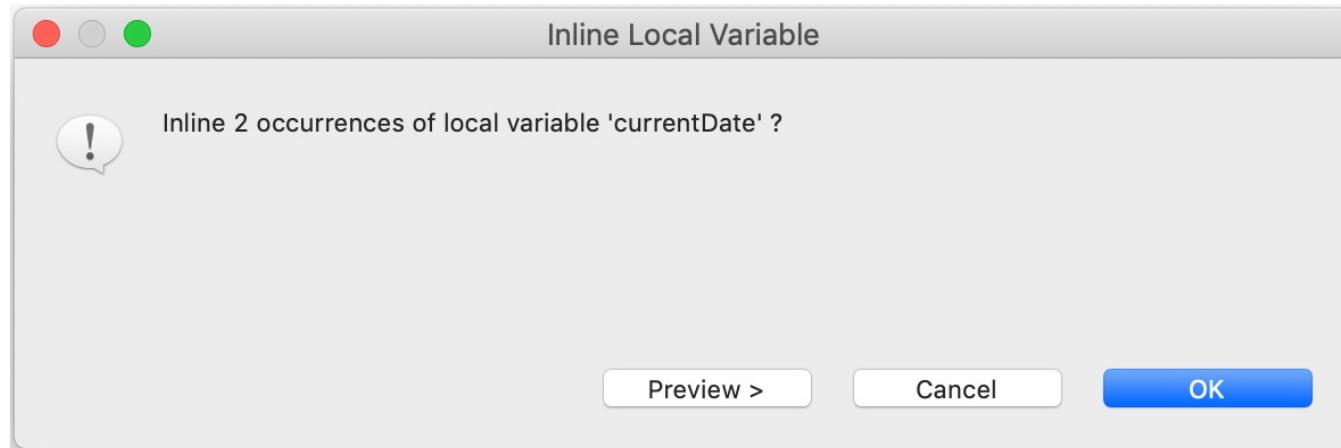
```
private static void printLetterImproved_Step4()
{
    printBanner();
    printContent();

    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 – Beispiel 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;
}
```

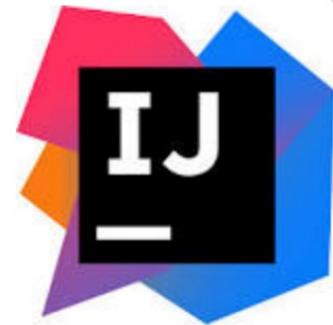
Refactoring in Practice – Beispiel FINISH 😊

```
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;
}
```



Besser in Refactorings:
Wie erwähnt, kann es die
letzten Schritte kombinieren

EXERCISES

Vorgehen Aufgabe 2

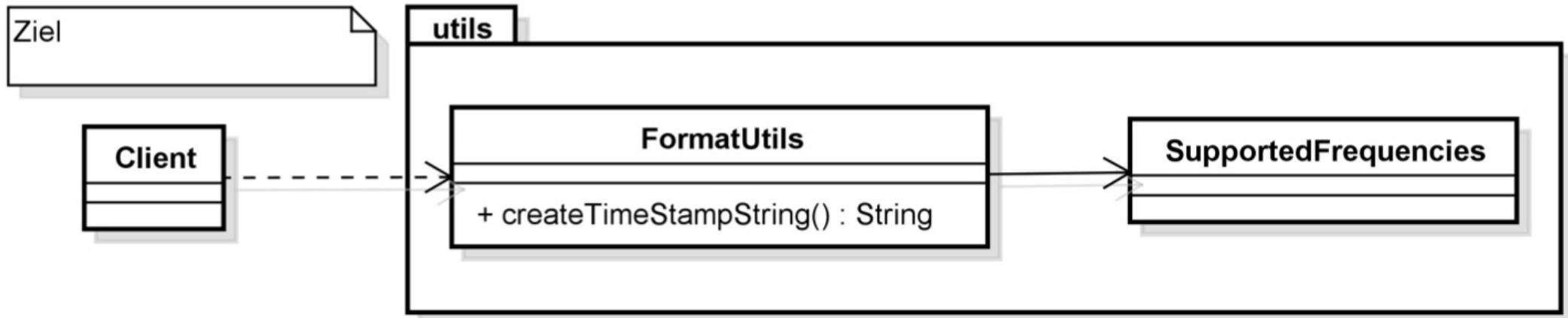
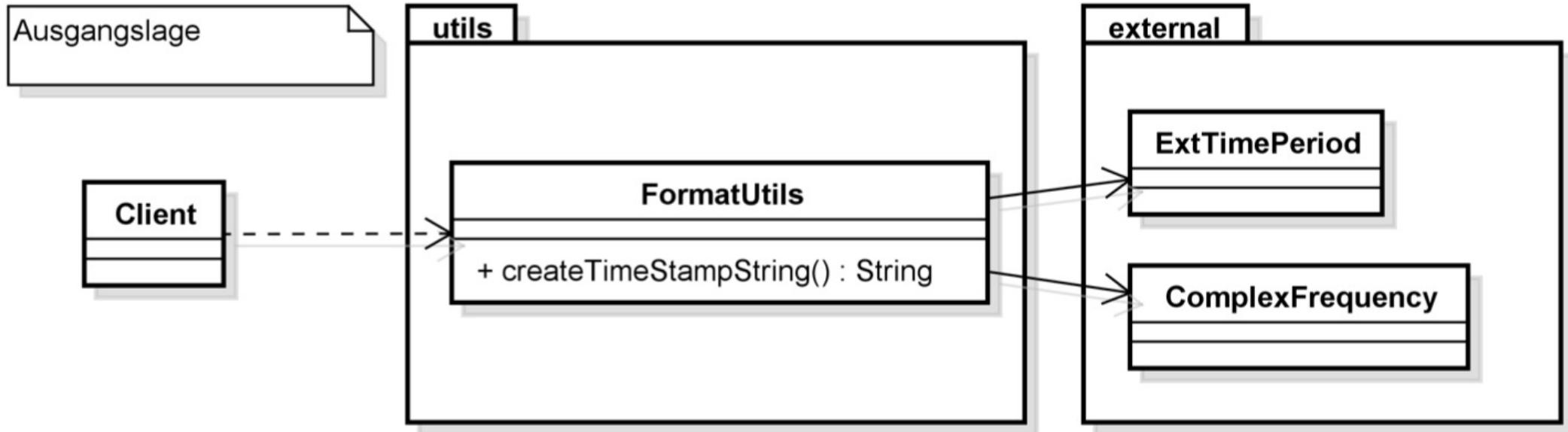
```
public class TimeStampUtils
{
    public static String createTimeStampString(final ExtTimePeriod currentPeriod,
                                              final ComplexyFrequency frequency)
    {
        final LocalDateTime start = currentPeriod.getDateTime();

        final int divisor = frequency == ComplexFrequency.P1M ? 1 : 3;
        final String addition = frequency == ComplexFrequency.P1M ? "" : "Q";
        final int value = ((start.getMonthValue() - 1) / divisor + 1);

        return start.getYear() + "-" + addition + value;
    }
}
```

Die Methode `createTimeStampString(ExtTimePeriod, ComplexFrequency)` soll nun mithilfe von Basis-Refactorings so umgestaltet werden, dass nur Abhängigkeiten auf Standards wie Klassen aus öffentlichen Bibliotheken oder besser noch dem JDK bestehen und der Sourcecode verständlicher wird.

Vorgehen Aufgabe 2



Vorgehen Aufgabe 2

Wir wollen nun folgende Schritte ausführen, um die angemerkteten Probleme zu beseitigen und das dargestellte Ziel zu erreichen:

- **Auflösen der Abhängigkeiten** – In einem ersten Schritt lösen wir die Abhängigkeiten zum Package external auf, indem wir einen enum namens SupportedFrequencies als Ersatz für ComplexFrequency einführen und anstelle der Klasse ExtTimePeriod die Klasse LocalDateTime aus dem Date and Time API (vgl. Kapitel 8) verwenden.
- **Vereinfachungen** – Einige der Berechnungen in der Methode sind etwas komplex und nicht gut zu lesen. Wir werden ein paar Vereinfachungen vornehmen.
- **Verlagern von Funktionalität** – Abschließend schauen wir, wie wir durch eine kleine Änderung von Zuständigkeiten für mehr Klarheit im Design sorgen.

Take Aways & Fallstricke

Take Away: Read vs. Write

We READ 10x more time than we WRITE

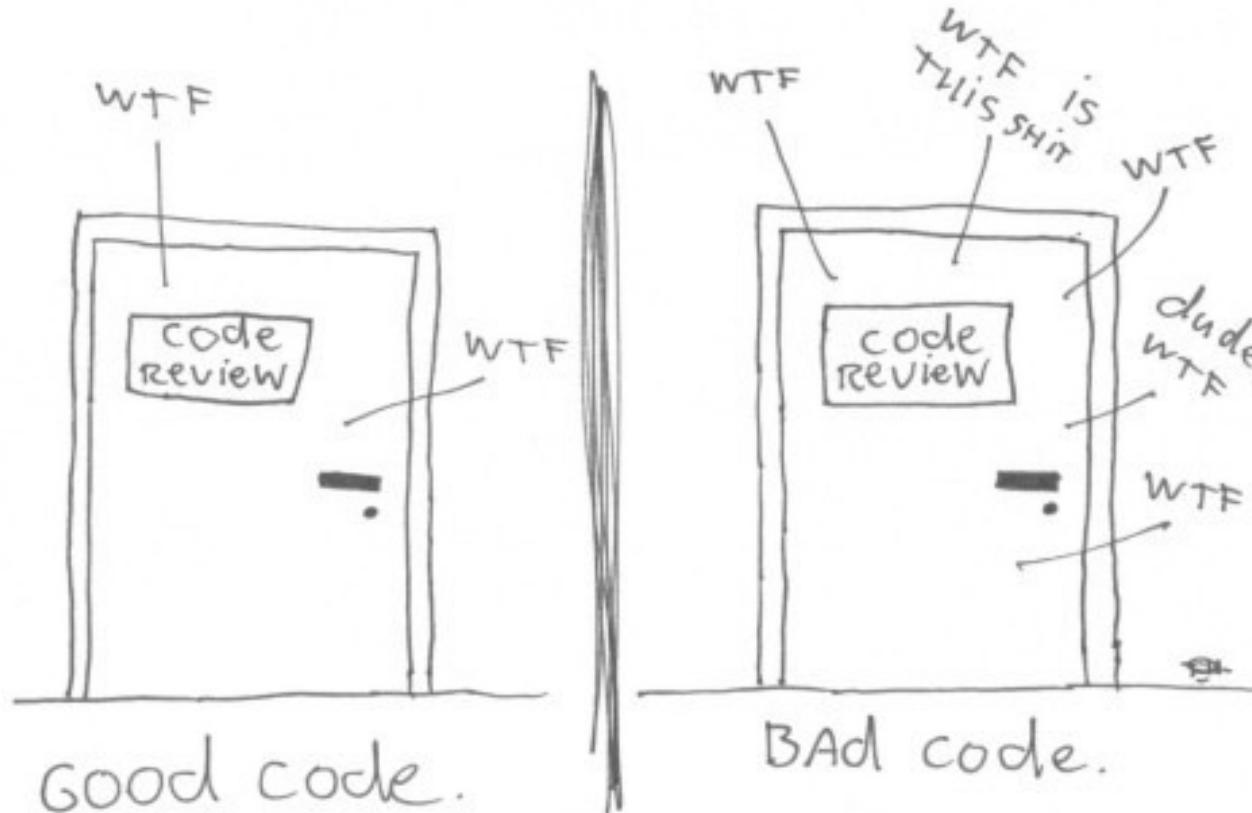
→ Make it more readable, even if it's harder to write

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



Take Away: WTF – The TRUE measurement

The ONLY VALID measurement
OF code QUALITY: WTFs/minute



Achtung: Basis-Refactorings

- Zwar sollten Basis-Refactorings nichts am (*nach außen sichtbaren*) Verhalten ändern, aber wie sichert man das?
 - Keine Freestyle-Refactorings
 - Immer möglichst die IDE-Automatiken nutzen
 - Mithilfe der (großen) Anzahl an Tests für Sicherheit sorgen
- Beachte möglicherweise vorhandene Seiteneffekte
- Kenne Fallstricke wie Temporal Coupling

Achtung: Basis-Refactorings & Temporal Coupling

- **Temporal coupling** beschreibt die Abhangigkeiten zwischen Methodenaufrufen, die in einer gewissen Abfolge erfolgen mussen, oftmals in Kombination mit Seiteneffekt problematisch fur Refactorings

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

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

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

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

Temporal Coupling + Seiteneffekt Fallstrick

```
public void doSomething()
{
    String NEW_INFO_MSG = "Very long text ....";
    setText(NEW_INFO_MSG);
    methodB(); // scheint thematisch nicht zu passen
    if (hasExpectedLength(NEW_INFO_MSG))
        System.out.println("TEXT CHANGED: " + NEW_INFO_MSG);

    // Scheinbar bessere Variante, aber nicht mehr funktionell gleich
    methodB();

    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);
}

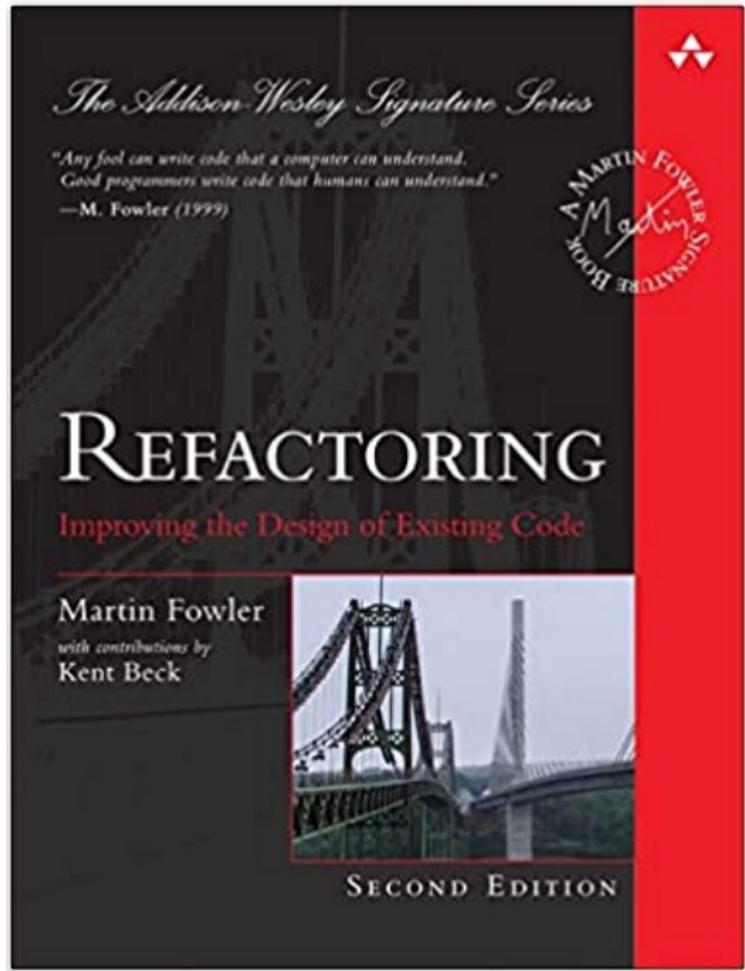
private boolean hasExpectedLength(String text)
{
    return text.length() == count;
}
```

Weitere Infos

Weitere 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>

Weitere Infos II



Ergänzendes Beispiel

Ausgangsbasis

```
public static boolean isNumber(final String value)
{
    if (Character.isDigit(value.charAt(0)))
    {
        for (int i = 1, n = value.length(); i < n; i++)
        {
            if (!Character.isDigit(value.charAt(i)))
            {
                return false;
            }
        }
    }
    else
    {
        return false;
    }
    return true;
}
```

Analysieren wir einmal, was uns daran stört!

Ausgangsbasis Analyse

```
public static boolean isNumber(final String value)
{
    if (Character.isDigit(value.charAt(0)))
    {
        for (int i = 1, n = value.length(); i < n; i++)
        {
            if (!Character.isDigit(value.charAt(i))))
            {
                return false;
            }
        }
    } else
    {
        return false;
    }
    return true;
}
```

1. **Unerwartete Exception bei leerer Eingabe** – Wenn die Eingabe nicht mindestens ein Zeichen enthält, wird eine `java.lang.StringIndexOutOfBoundsException` ausgelöst. Die Ursache ist der indizierte Zugriff per `charAt(0)`, ohne zuvor die Länge des Parameters `value` zu überprüfen. Ein solches Verhalten ist zu vermeiden. Dies gilt im Speziellen, wenn die Werte aus einer Benutzereingabe stammen.
2. **Fehleranfällig für null** – Bei Übergabe eines `null`-Werts löst die Methode erst bei der Verarbeitung und dem Aufruf von `charAt(0)` eine `NullPointerException` aus. Öffentliche Methoden sollten gemäß Design by Contract (vgl. Abschnitt 3.1.5) ihre Vorbedingungen sicherstellen und dazu vor der eigentlichen Verarbeitung die Eingabeparameter auf Gültigkeit prüfen. Damit wird eine Störung der Abarbeitung durch fehlerhafte Übergabewerte vermieden.
3. **Zu kompliziert** – Die initiale `if`-Bedingung und das `if` innerhalb der `for`-Schleife sorgen für eine weitere Schachtelungsebene und sind konträr zueinander formuliert. Bei flüchtigem Hinsehen könnte man die Abweichungen in der initialen Prüfung übersehen und das Ganze falsch zusammenführen.
4. **Missverständlicher Name bzw. unklares Verhalten** – Der Methodenname `isNumber(String)` suggeriert die Möglichkeit der Verarbeitung beliebiger Zahlen. Momentan werden aber weder Zahlen mit Vorzeichen noch solche mit Nachkommastellen unterstützt.
5. **Viele `return`-Anweisungen** – Für eine derart kurze Methode existieren mit drei `return`-Anweisungen schon recht viele Ausgänge.

Problem 1: Unerwartete Exception bei leerer Eingabe

Bevor wir uns dem eigentlichen Problem bei leeren Eingaben zuwenden, erstellen wir einige Funktionstests: Wir prüfen die Verarbeitung einer gültigen Eingabe, etwa "12345", und eines fehlerhaften Werts, etwa "ABC". Im ersten Fall erwarten wir `true` und im zweiten `false` als Ergebnis. Dies lässt sich mithilfe von JUnit und dessen Methoden `assertTrue(boolean)` und `assertFalse(boolean)` ausdrücken. Diese beiden Tests sollten bestanden werden.

```
@Test
public void testValidNumberInput()
{
    assertTrue(NumberUtilsV1.isNumber("12345"));
}

@Test
public void testInvalidInput()
{
    assertFalse(NumberUtilsV1.isNumber("ABC"));
}
```

Weitere Tests

```
@Test  
public void testNumberInputLength0()  
{  
    assertFalse(NumberUtilsV1.isNumber "");  
}  
  
@Test  
public void testNumberInputLength1()  
{  
    assertTrue(NumberUtilsV1.isNumber "1");  
}
```

Korrektur Bevor wir weitere Tests ergänzen, korrigieren wir die Funktionalität. Wir fügen folgende Sicherheitsprüfung am Anfang der Methode `isNumber()` hinzu:

```
if (value.isEmpty())  
{  
    return false;  
}
```

Problem 2: Fehleranfällig für null

In der initialen Analyse haben wir erkannt, dass die Eingabe von `null` unbehandelt ist und eine `NullPointerException` auslöst – allerdings nicht durch eine Parameterprüfung, sondern erst beim Zugriff auf den Parameter `value`. Wir wollen prüfen, dass eine explizite Behandlung erfolgt und ein Hinweis in der ausgelösten Exception mehr Informationen liefert. Mit JUnit 5 lässt sich das elegant mithilfe von `assertThrows()` formulieren:

```
@Test
public void testNullInput()
{
    NullPointerException npe = assertThrows(NullPointerException.class,
                                           () -> NumberUtilsV2EmptyInputCorrected.isNumber(null));

    assertFalse(StringUtils.isEmpty(npe.getMessage()));
}
```

Korrektur Wir haben in Kapitel 9 die Bibliothek Google Guava und deren Utility-Klasse Preconditions zum Sicherstellen von Vorbedingungen kennengelernt. Dementsprechend fügen wir eine Sicherheitsprüfung am Anfang der Methode hinzu:

```
public static boolean isNumber(final String value)
{
    Preconditions.checkNotNull(value, "parameter 'value' must not be null");

    if (value.isEmpty())
    {
        return false;
    }
    if (Character.isDigit(value.charAt(0)))
    {
        for (int i = 1, n = value.length(); i < n; i++)
        {
            if (!(Character.isDigit(value.charAt(i))))
            {
                return false;
            }
        }
    }
    else
    {
        return false;
    }
    return true;
}
```

Problem 3: Zu kompliziert

Wenn wir uns die Methode und insbesondere die Schleife und die dortigen `if`-Abfragen anschauen, ist das schon einigermaßen kompliziert. Bei genauem Hinsehen fällt auf, dass man Vereinfachungen erreichen kann: Die abgefragten Bedingungen sind semantisch gleich, aber negiert. Somit lassen sich die beiden `if`-Abfragen zusammenfassen, indem die `if`-Startbedingung in die `if`-Abfrage der `for`-Schleife integriert wird. Als Folge kann die Schleife bei 0 gestartet werden.

Auch die Formulierung der `for`-Schleife ist uns zu kompliziert und leicht unleserlich. Zur Vereinfachung entfernen wir die Zuweisung `n = value.length()` aus dem Initialisierungssteil der `for`-Schleife, die zur Optimierung des Vergleichs `i < n` diente.¹ Dies schreiben wir kürzer als `i < value.length()`. Das erhöht die Lesbarkeit und Verständlichkeit. Insgesamt ergibt sich folgende Korrektur:

```
public static boolean isNumber(final String value)
{
    Preconditions.checkNotNull(value, "parameter 'value' must not be null");

    if (value.isEmpty())
    {
        return false;
    }
    for (int i = 0; i < value.length(); i++)
    {
        if (!(Character.isDigit(value.charAt(i))))
        {
            return false;
        }
    }
    return true;
}
```

Problem 4: Missverständlicher Name bzw. unklares Verhalten

Nach Rücksprache mit dem Kunden oder einem Requirements Engineer wird deutlich, dass selbstverständlich erwartet wird, dass die Methode `isNumber(String)` neben positiven auch negative Ganzzahlen verarbeiten kann. Eventuell wird später sogar eine Erweiterung auf Gleitkommazahlen gewünscht.

```
@Test
public void testNumberPositive()
{
    assertTrue(NumberUtils.isNumber("+4711"), "plus sign should be accepted");
}

@Test
public void testNumberNegative()
{
    assertTrue(NumberUtils.isNumber("-4711"), "minus sign should be accepted");
}
```

```
public static boolean isNumber(final String value)
{
    Preconditions.checkNotNull(value, "parameter 'value' must not be null");

    if (value.isEmpty())
    {
        return false;
    }
    // Verarbeite Vorzeichen
    String number = value;
    if (value.startsWith("-") || value.startsWith("+"))
    {
        number = value.substring(1, value.length());
    }
    // Weitere Prüfung auf Zahl wie zuvor
    for (int i = 0; i < number.length(); i++)
    {
        if (!(Character.isDigit(number.charAt(i))))
        {
            return false;
        }
    }
    return true;
}
```

Wo stehen wir jetzt? Mehr Code, dafür sicherer und flexibler

Problem 5: Viele `return`-Anweisungen

Die Methode `isNumber(String)` sollte auf Ganzzahlen prüfen. Statt diese Funktionalität, wie initial geschehen, selbst zu programmieren, bietet sich der Einsatz der Java-Bibliotheken zum Parsen von Zahlen an. Da unsere Tests bereits recht ausgereift sind, müssen wir hier nichts ergänzen. In der Methode `isNumber(String)` verwenden wir nun die Methode `Integer.parseInt(String)` (vgl. Abschnitt 5.2.2). Durch deren Einsatz können im Gegensatz zu der eigenen Realisierung sowohl positive als auch negative Zahlen verarbeitet werden.

```
public static boolean isNumber(final String value)
{
    Preconditions.checkNotNull(value, "parameter 'value' must not be null");

    try
    {
        Integer.parseInt(value); // Rückgabe ignorieren, nur prüfen
        return true;
    }
    catch (final NumberFormatException ex)
    {
        return false;
    }
}
```

```
public static boolean isNumber(final String value)
{
    Preconditions.checkNotNull(value, "parameter 'value' must not be null");

    try
    {
        Integer.parseInt(value); // Rückgabe ignorieren, nur prüfen
        return true;
    }
    catch (final NumberFormatException ex)
    {
        return false;
    }
}
```

Was haben wir erreicht? Die Intention und Realisierung der Methode ist nun deutlich klarer, da nicht mehr auf Basis einzelner Zeichen geprüft wird. Vielmehr erfolgt auf logischer Ebene eine Umwandlung in eine Zahl. Die Lesbarkeit hat dadurch enorm zugenommen. Wir kommentieren zudem die bewusst fehlende Auswertung des Rückgabewerts von `Integer.parseInt(String)`, um möglicherweise aufkommende Fragen anderer Entwickler sofort zu klären.