Inhaltsverzeichnis

Refactoring Patterns

Coding Akademie München

Refactoring Grundlage in Action

- Refactoring Workshop
- Die versteckten Klassen
- TestableHtml Methode
- Legacy Code
- Refactoring Patterns
- P01 Extract an inner Class from a big Function
  - IntelliJ Refactor - Extract - Method Object
    - Body von der Big Funktion Markieren
    - Refactoring Extract Method Object
    - Einen guten Namen für die innere Klasse geben
    - Alle Einstellungen übernehmen und auf OK anklicken
  - Eclipse: Extract an inner Class from a big Function
    - Schritt 1: private static inner Class in der außen Klasse definieren
    - Schritt 2: Aus der Parameterliste die Instanzvariablen definiere
    - Schritt 3: invoke Methode in der inneren Class definieren
    - Schritt 4: Big Function Body in invoke() kopieren
    - Schritt 5: invoke() in Big Function Aufrufen
- P02 Extract Instance Variable from a Local Variable
  - Aus einer lokalen Variablen eine Instanzvariable Extrahieren
    - IntelliJ - Refactor - Extract - Field...
  - Schritt 2: die neuen Instanzvariablen im Konstruktor initialisieren
- P03 Duplikate Eliminieren
  - Aus einem Methodenaufruf eine Instanzvariable Extrahieren
    - IntelliJ - Methodenaufruf Markieren - Refactor - Extract - Field
  - Aus einem Konstanten eine Variable Extrahieren
  - Aus der Parameterisierte Repeating Structure eine Methode Extrahieren.
- Clean the Extracted Function

Refactoring Workshop

In diesem Workshop geht es darum, dass Sie die wichtigsten Refactoring Patterns durch die Anwendung der wichtigsten Refactoring Techniken kennen lernen und in der Praxis optimal auch ohne IDE Unterstützung sogar einsetzen können. Gegeben ist eine public Klasse: MakeHTML

```
public class MakeHTML {
...
}
```

Diese Klasse hat eine lange Methode:

public static String testableHtml(PageData pageData, boolean includeSuiteSetup) throws Exception { ... }

Diese Methode hat folgende unschöne Eigenschaften: 1. Ist ziemlich lang (über 40 Zeilen) 2. Ihr Code sagt nicht sofort, was er tut (Opacity). 3. Hat viele geschachtelte if-Bedingungen 4. Hat viele ähnliche Code-Segmente (Duplikate) 5. Hat keinen aussagekräftigen Namen

Die vollständige Methode finden Sie in der Datei MakeHtml.java. Ihre Aufgabe besteht darin, aus diesem schlechten Code, der zwar richtig und korrekt funktioniert, einen "Clean Code" zu machen! Dies ist eine typische Aufgabe bei dem sogenannten "Legacy Code".

Die versteckten Klassen

Eine lange Funktion versteckt fast immer eine Klasse. Aber warum? Dafür sehen wir uns die Definition von einer Klasse mal an.

Was ist eine Klasse?

```
Eine Klasse ist eine Sammlung von Funktionen, die auf gemeinsame
Daten (Instanzvariablen) operieren.
```

Und was ist eine lange Funktion?

```
Eine lange Funktion ist eine Sammlung von Subfunktionen, die auf
gemeinsame Daten (Funktion Parameterliste) operieren.
```

Daher ist es eine gute Technik, immer aus einer langen Funktion eine neue Klasse zu extrahieren. Die Idee ist einfach:

1. Wir erzeugen eine innere class und geben ihr einen "guten" Namen
2. Wir machen aus der Parameterliste der langen Funktion die Instanzvariablen der inneren Klasse. Möglicherweise müssen sie in den Konstruktor der inneren Klasse definiert bzw. initialisiert.
3. Wir erzeugen eine Methode z.B. mit dem Namen invoke() in der innen Klasse. Diese invoke() Methode hat den gleichen Rückgabetypen von der ursprünglichen langen Funktion.
4. Wir schneiden den gesamten Body Code von der langen Funktion aus und fügen ihn n dem Body von invoke() ein. Die ursprüngliche lange Methode in der außen Klasse wird nach diesen Schritt keinen Code im Body mehr haben.
5. Wir erzeugen in dem Body von der ursprünglichen langen Methode ein Objekt aus der inneren Klasse und rufen mit diesem Objek invoke() auf.

TestableHtml Methode

Hier ist der Grundgerüst von dieser Methode:

```
public static String testableHtml(PageData pageData,
              boolean includeSuiteSetup) throws Exception {

        WikiPage wikiPage = pageData.getWikiPage();
        StringBuilder buffer = new StringBuilder();

// Erste If-Bedingung
  if(pageData.hasAttribute("Test"))  // If Level 1
  {
     if(includeSuiteSetup)           //  If Level 2
     {
       ...
        if(suiteSetup != null)       //  If Level 3
```

```
        {
            ...
         }
    ...
     if(setup != null)               //  If Level 2
     {
      ...
    }
      ..
// Zweite If-Bedingung
  if(pageData.hasAttribute("Test"))   // If Level 1
    {
    ... // weitere geschachtelte If
}

pageData.setContent(buffer.toString());
return pageData.getHtml();
}
```

Legacy Code

```
Ein Legacy Code ist ein Code ohne Tests oder ein Code mit nicht
gerade guten oder vollständigen Test Suite.
```

Wie gehen wir mit legacy Code um?

1. Man muss die Grundlagen von Clean Code verstehen
2. Man muss die Refactoring Patterns ohne IDE beherrschen
3. Man muss Working with Legacy Code Techniken beherrschen
4. Das ist ziemlich anspruchsvoll und meist nicht intuitiv
5. Dafür gibt es einen extra Kurs "Arbeiten mit Legacy Code".

Um einen hässlichen Code in einen Clean Code zu bringen, müssen wir viele "Refactoring Patterns" anwenden.

Refactoring Patterns

Was ist Refactoring?

```
Code Struktur ändern, ohne dabei die Funktionalität zu ändern.
```

- Wir fügen keine neue Funktionen, Erweiterungen oder Features hinzu,
- Wir überarbeiten den vorhanden Code und bringen ihn durch mehrfache Anwendungen von Refactoring Patterns in eine bessere Struktur (Clean Code)
- Meist führt die neue Struktur zu einer besseren und flexibleren Architektur.
- Zur Erinnerung: Clean Code ist ein Code, der leicht lesbar ist, einfach zu erweitern, anzupassen und zu modifizieren ist.

Die nötigen Refactoring Schritte für das Workshop:

1. Extract an (inner) class from a long Function
2. Extract an Instance Variable from a Local Variable
3. Initialise the Extracted Instance Variable in the Constructor
4. Eliminate Duplicate (Repeating Structure)
5. Clean up the Extracted Funktion
6. Extract Inline local Variable
7. Extract a Function form the If-Statement
8. StringBuffer Code Eliminieren
9. Extract the If Predicate of the If-Statement
10. Rename a Variable (Inner Class and Method)
11. Extract a separate Class from the Inner Class

P01 Extract an inner Class from a big Function

Dieses Pattern besteht aus 5 Schritten:

1. Private static inner Class Definieren
2. Instanzvariablen aus der Parameterliste Extrahieren
3. Eine invoke() Methode in der inneren Class Definieren
4. Big Function Body Ausschneiden und in invoke() kopieren
5. In Big Function Body ein Inner Class Objekt erzeugen und invoke() aufrufen.

Diese 5 Schritte werden von IntelliJ IDEA mit

```
Refactoring => Extract => Method Object..
```

komplett generiert. Während Sie in Eclipse alle diese 5 Schritte manuell machen sollten, wie die folgenden Abschnitte dieses Refactoring Pattern mit IntelliJ und Eclipse erklärt.

IntelliJ Refactor - Extract - Method Object

Um eine innere Klasse aus einer großen Funktion zu extrahieren, bietet IntelliJ die Extract Method Object ...an. In IntelliJ können Sie die oben genannten 5 Schritte (mit Refactoring => Extract => Method Object..) von IntelliJ generieren lassen. Dafür

Body von der Big Funktion Markieren

Wir markieren den gesamten Body-Code von der Big Funktion

In unserem Beispiel also von der *testableHtml(PageData pageData, boolean includeSuiteSetup)* Methode und zwar von der ersten Body Zeile *WikiPage wikiPage* bis zur letzten Body Zeile *return pageData.getHtml()*

```
    public static String testableHtml(PageData pageData,
           boolean includeSuiteSetup) throws Exception {
        // Markieren Sie von Hier
           WikiPage wikiPage = pageData.getWikiPage();
           StringBuilder buffer = new StringBuilder();

           ...
           pageData.setContent(buffer.toString());
           return pageData.getHtml();
        // Bis hier!
```

Refactoring Extract Method Object

Mit der rechten Maustaste:

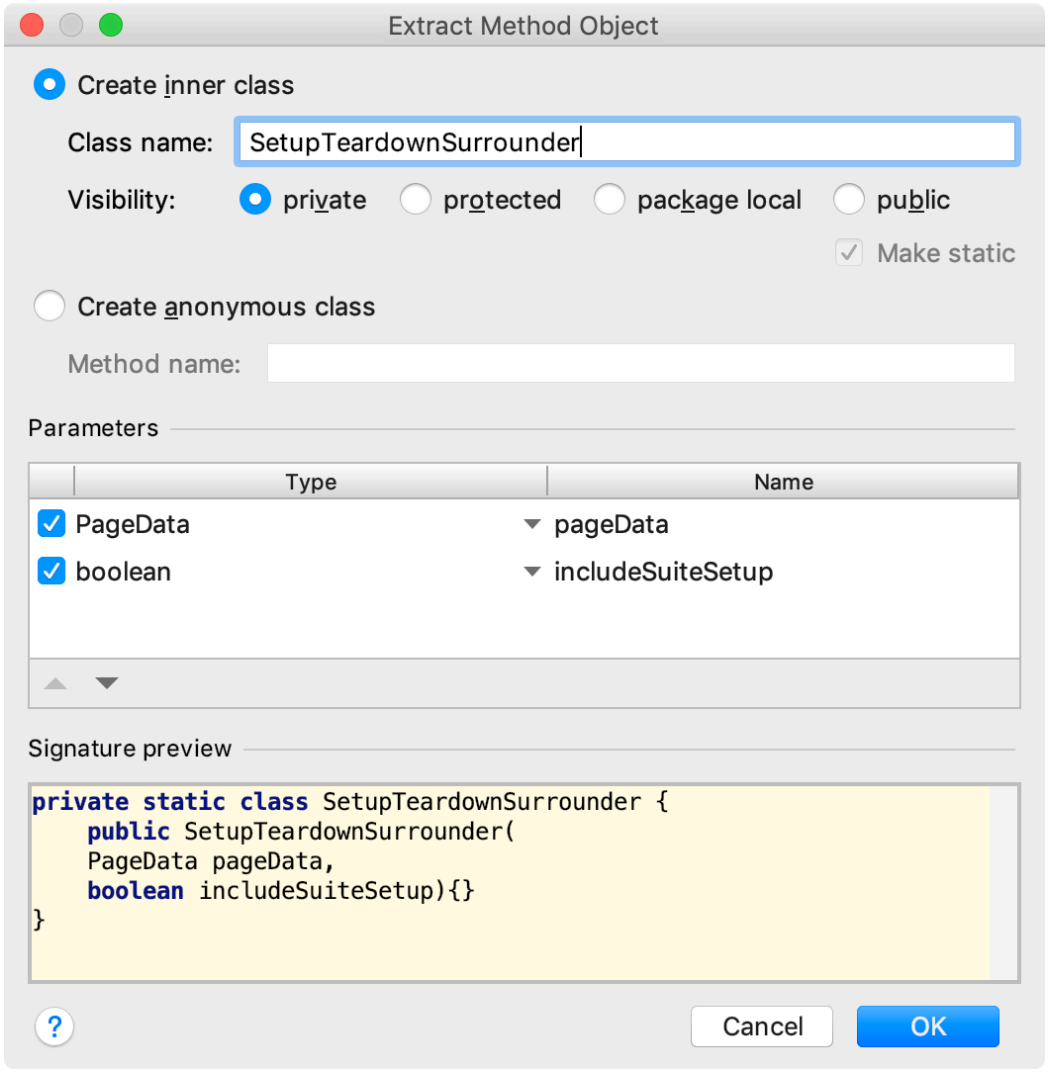
```
      Refactoring => Extract => Method Object..
```

Einen guten Namen für die innere Klasse geben

```
`class name: SetupTeardownSurrounder`
```

Bitte keine Zeit hier bei der Suche nach einem guten Namen verlieren, denn Sie werden diesen Namen später auch ändern!

Alle Einstellungen übernehmen und auf OK anklicken



IntelliJ generiert dann alle oben genannten 5 Schritte automatisch.

Eclipse: Extract an inner Class from a big Function

Eclipse bietet die Method Object.. Refactoring NICHT! Daher sollten Sie die folgenden 5 Schritte in Eclipse manuell machen.

Schritt 1: private static inner Class in der außen Klasse definieren

Definieren Sie eine private static inner Class innerhalb ihrer Klasse. Geben Sie hier einen guten Namen, aber machen Sie sich keine so großen Gedanken über den Namen. Denn er ändert sich sowieso später.

```
private static class SetupTeardownSurrounder {

}
```

Schritt 2: Aus der Parameterliste die Instanzvariablen definiere

Die Parameterliste der langen Funktion

```
testableHtml(PageData pageData, boolean includeSuiteSetup)
```

werden zu den Instanzvariablen der inneren Klasse:

```
private static class SetupTeardownSurrounder {
     private PageData pageData;
     private boolean includeSuiteSetup;
```

Schritt 3: invoke Methode in der inneren Class definieren

Diese invoke() Methode muss den gleichen Rückgabewert wie die lange Methode *testableHtml* haben:

```
public String testableHtml(PageData pageData,
                           boolean includeSuiteSetup)
```

Also muss invoke() in der inneren Klasse vom Typ String sein:

```
 private static class SetupTeardownSurrounder {
     private PageData pageData;
     private boolean includeSuiteSetup;
```

```
      public String invoke() {

      }
}
```

Schritt 4: Big Function Body in invoke() kopieren

Wir schneiden den kompletten Body Code von *testableHtml* und fügen ihn in *invoke()* Body ein. Nur der Body, nicht die Parameter!

```
public String invoke() {
  WikiPage wikiPage = pageData.getWikiPage();
  StringBuilder buffer = new StringBuilder();
   if(pageData.hasAttribute("Test")){
     if(includeSuiteSetup){
       WikiPage suiteSetup =
       PageCrowlerImpl.getInheritedPage(
          SuiteResponder.SUITE_SETUP_NAME, wikiPage);
        if(suiteSetup != null){
          WikiPagePath pagePath =
                          wikiPage.getPageCrawler()
                          .getFullPath(suiteSetup);
         String pagePathName = PathParser.render(pagePath);
         buffer.append("!include -setup .")
               .append(pagePathName).append("\n");
                }
           }
        WikiPage setup =
       PageCrowlerImpl.getInheritedPage("SetUp", wikiPage);

            if(setup != null){
             WikiPagePath setupPath =
                        wikiPage.getPageCrawler()
                        .getFullPath(setup);

             String setupPathName =
                        PathParser.render(setupPath);
            buffer.append("!include -setup . ")
                  .append(setupPathName).append("\n");
            }
        }


        buffer.append(pageData.getContent());
```

```
        if(pageData.hasAttribute("Test")) {
            WikiPage teardown =
                   PageCrowlerImpl
                   .getInheritedPage("TearDown", wikiPage);

            if(teardown != null){
                WikiPagePath tearDownPath =
                                    wikiPage
                                    .getPageCrawler()
                                    .getFullPath(teardown);

             String tearDownPathName =
                        PathParser.render(tearDownPath);
            buffer.append("!include -teardown . ")
                  .append(tearDownPathName).append("\n");
            }

            if(includeSuiteSetup){
              WikiPage suiteTeardown =
                PageCrowlerImpl
                .getInheritedPage(
             SuiteResponder.SUITE_TEARDOWN_NAME, wikiPage);

                if(suiteTeardown != null) {
                    WikiPagePath pagePath =
                       suiteTeardown
                      .getPageCrawler()
                      .getFullPath(suiteTeardown);

               String pagePathName =
                           PathParser.render(pagePath);

                buffer.append("!include -teardown . ")
                 .append(pagePathName).append("\n");
                }
            }
        }

        pageData.setContent(buffer.toString());
        return pageData.getHtml();
    }
```

Schritt 5: invoke() in Big Function Aufrufen

Ein Objekt aus der inneren Class in der langen Methode erzeugen und invoke() damit aufrufen. Also wir erzeugen in die *testableHtml* Methode ein Objekt von _ SetupTeardownSurrounder_ und rufen wir mit diesem Objekt die *invoke()* Methode auf.

public class MakeHTML {

```
public static String testableHtml(PageData pageData, boolean
includeSuiteSetup) throws Exception {

  return new SetupTeardownSurrounder(
        pageData,includeSuiteSetup).invoke();
}

private static class SetupTeardownSurrounder {
    ...
```

}

Achtung: Eclipse bietet leider keine Unterstützung für diese 5 Schritte an, daher müssen Sie dieses Refactoring Pattern Manuel wie in den Schritten 1-5 hier beschrieben durchführen!

P02 Extract Instance Variable from a Local Variable

```
Eine lokale Variable ist eine Variable, welche in dem Body einer
Funktion definiert ist. Man bezeichnet die Variablen, die in der
Parameterliste einer Funktion definiert sind als Parameter oder
manchmal Arguments. Mit lokalen Variablen sind nur die Variablen
gemeint, die in dem Body einer Funktion definiert sind.
```

Wenn eine lokale Variable durch mehrere Subfunktionen bzw. Subroutine benutzt wird, dann macht es Sinn, sie zu einer Instanzvariable zu machen. Meist werden dann diese Instanzvariablen in dem Konstruktor gesetzt bzw. mit den ursprünglichen Werten der vorherigen lokalen Variablen initialisiert.

Meist wird dieses Pattern unmittelbar nach Pattern 01 (Extract an inner Class from a big Function) angewendet. Zur Erinnerung: in dem Pattern 01 haben wir eine innere Klasse *SetupTeardownSurrounder* erzeugt. Die Parameter von *testableHtml* also die *(PageData pageData, boolean includeSuiteSetup)* zu den Instanzvariable der *SetupTeardownSurrounder* Klasse gemacht.

Pattern 02 besteht meist aus zwei Schritten:

Aus einer lokalen Variablen eine Instanzvariable Extrahieren

1. Lokale Variable Deklaration markieren und kopieren
2. Als *private* Instanzvariable in die Zielklasse einfügen.
3. Lokale Variable Deklaration entfernen (Nicht die Initialisierung!)
4. *setter* und *getter* für die neue Instanzvariable bei Bedarf anbieten.

Hier ein Beispiel aus dem Workshop

```
public String invoke() {
        WikiPage wikiPage = pageData.getWikiPage();
        StringBuilder buffer = new StringBuilder();
```

Die beiden lokalen Variablen *wikiPage* und *buffer* werden in dem gesamten Body von *invoke()* benutzt, daher macht es Sinn, beide zu Instanzvariablen zu machen.

```
private static class SetupTeardownSurrounder {
   …
   private WikiPage wikiPage;
   Private StringBuilder buffer;
```

In der *invoke()* Methode entfernen wir die Deklaration und behalten erst mal noch die Initialisierung. Die Initialisierung verschwindet aus *invoke()* im zweiten Schritt.

```
public String invoke() {
        wikiPage = pageData.getWikiPage();
        buffer = new StringBuilder();
```

IntelliJ - Refactor - Extract - Field...

Um aus einer lokalen Variable eine Instanzvariable zu extrahieren bietet IntelliJ Extract => Field... an.

```
Lokale Variable Markieren => Rechte Maustaste => Refactor => Extract
=> Field... => Choose Class to Introduce Field
```

Beispiel aus dem Workshop: *wikiPage* zu Instanzvariable

- *wikiPage* lokale Variable in *invoke()* auswählen
- Rechte Maustaste => Refactor => Extract => Field...
- *Choose Class to Introduce Field* : *SetupTeardownSurrounder* auswählen.
- Doppelklick auf wikiPage

Schritt 2: die neuen Instanzvariablen im Konstruktor initialisieren

Nach Schritt 1 entfernen wir die Deklaration aus einer Funktion.

```
public String invoke() {
        wikiPage = pageData.getWikiPage();
        buffer = new StringBuilder();
```

Die Initialisierung sollte aber in dem Konstruktor sein, damit diese Werte in allen möglichen Subfunktionen, die später zu eigenen Funktionen werden, sichtbar und überall benutzbar bleiben. Dafür sind zwei Schritte nötig:

1. Die Initialisierung der lokalen Variablen in den Konstruktor kopieren.
2. Die lokalen Variablen aus der langen Funktion entfernen.

Hier ein Beispiel aus dem Workshop:

```
   public SetupTeardownSurrounder(..) {
        ..
        this.wikiPage = pageData.getWikiPage();
        this.buffer = new StringBuilder();
    }
```

In der *invoke* Methode erst ma kommentieren wir die lokalen Variablen, wenn alles richtig ist, wird keine Fehlermeldung angezeigt, anschließend löschen wir die alten lokalen Variablen komplett weg.

```
public String invoke() {
        ...
}
```

P03 Duplikate Eliminieren

In dem Workshop sind die folgenden 4 Segmente ähnlich:

Code Segment 1

```
1) WikiPagePath pagePath =
2) wikiPage.getPageCrawler().getFullPath(suiteSetup);
3) String pagePathName = PathParser.render(pagePath);
4) buffer.append("!include -setup .")
5) .append(pagePathName).append("\n");
```

Code Segment 2

```
1) WikiPagePath setupPath =
2) wikiPage.getPageCrawler().getFullPath(setup);
3) String setupPathName = PathParser.render(setupPath);
4) buffer.append("!include -setup . ")
5) .append(setupPathName).append("\n");
```

Code Segment 3

```
1) WikiPagePath tearDownPath =
2) wikiPage.getPageCrawler().getFullPath(teardown);
3) String tearDownPathName= PathParser.render(tearDownPath);
4) buffer.append("!include -teardown . ")
5) .append(tearDownPathName).append("\n");
```

Code Segment 4

```
1) WikiPagePath pagePath =
2) suiteTeardown.getPageCrawler()
  .getFullPath(suiteTeardown);
3) String pagePathName = PathParser.render(pagePath);
4) buffer.append("!include -teardown . ")
5) .append(pagePathName).append("\n");
```

Duplikate kommen auf Grund des Symmetrie-Gesetzes ziemlich häufig vor.

- Duplikate verletzten das Software Prinzip: **Don’t Repeat Your Self**.
- **Jedoch kommen Duplikate selten mit 100% Übereinstimmung.**
- Sondern eher als ähnliche Varianten mit geringfügigem Unterschied.
- Duplikate kommen häufig als **Repeating Structure**
- Manchmal ergibt sich nach dem Eliminieren von Duplikaten neue Repeating Structue, die man mit der gleichen Technik wieder eliminiert.

Duplikate Eliminieren Techniken:

1. Aus einem Methodenaufruf eine Instanzvariable Extrahieren
2. Repeating Structure Mit Code Parameterization vorbereiten
3. Aus der Parameterisierte Repeating Structure eine Methode Extrahieren.

Aus einem Methodenaufruf eine Instanzvariable Extrahieren

Manchmal kommt in verschiedenen Segmenten der gleiche Methodenaufruf vor. Da macht es Sinn, aus diesem Aufruf, eine Instanzvariable vom Typ X zu extrahieren, falls der Methodenaufruf einen Wert vom Typ X zurück liefert:

```
   wikiPage.getPageCrawler()
```

Daraus extrahieren/definieren wir die Instanzvariable crawler:

```
private PageCrawler crawler;
```

Anschließend initialisieren wir diese *crawler* im Konstruktor und zwar mit dem Methodenaufruf *wikiPage.getPageCrawler()* als Wert

```
public SetupTeardownSurrounder(...) {
        ...
        this.crawler = wikiPage.getPageCrawler();
    }
```

IntelliJ - Methodenaufruf Markieren - Refactor - Extract - Field

Um aus einem Methodenaufruf eine Instanzvariable zu extrahieren, bietet IntelliJ die *Extract Field* Refactoring an

```
Methodenaufruf markieren => Refactor => Extract => Field..
```

Anschließend geben bzw. Wählen wir einen aussagekräftigen Namen für *Field* an und klicken *OK.* an. Hier ein Beispiel:

- *wikiPage.getPageCrawler()* Markieren
- Rechte Maustaste => Refactor => Extract => Field...
- *Choose class to introduce field* : Zielklasse auswählen
- Also *SetupTeardownSurrounder* auswählen.
- IntelliJ schlägt mehrere Namen vor.
- Den passenden Namen auswählen z.B. *crawler* und Doppelklick
- *crawler* im Konstruktiv initialisieren, also die Initialisierung von *crawler* in der Methode ausschneiden und in dem Konstruktor kopieren.
- Alle Methodenaufruf *wikiPage.getPageCrawler()* durch *crawler* ersetzen, wenn IntelliJ dies nicht automatisch tut!

Aus einem Konstanten eine Variable Extrahieren

Manchmal unterscheiden sich zwei Code Segmente durch Konstante. Damit wir die Segmente auf den gleichen Nenner bringen, müssen wir die Unterschiede parameterizieren. Bei Konstanten ist die einfachste Methode daraus lokale Variablen zu machen. Hier ein Beispiel von 4 ähnliche Segmente:

Segment 1

```
buffer.append("!include -setup .")
.append(pagePathName).append("\n");
```

Segment 2

```
buffer.append("!include -setup . ")
.append(setupPathName).append("\n");
```

Segment 3

```
buffer.append("!include -teardown . ")
.append(tearDownPathName).append("\n");
```

Und Segment 4

```
buffer.append("!include -teardown . ")
.append(pagePathName).append("\n");
```

- In Segment 1 und Segment 2 definieren wir *String mode = "setup"*
- In Segment 3 und Segment 4 definieren wir *String mode = "teardown"*
- Wir ersetzen dann "setup" und "teardown" durch mode.

Segment 1

```
String mode = "setup";
buffer.append("!include -" + mode + " . ")
.append(pagePathName).append("\n");
```

Segment 2

```
buffer.append("!include -" + mode + " . ")
.append(setupPathName).append("\n");
```

Des gleiche gilt für *String mode = "tearddown"*

IntelliJ Konstante Markieren - Extract - Variable

Um aus einem Konstanten eine Variable zu extrahieren, bietet IntelliJ Extract Variable Refactoring an.

```
Konstante ("setup") auswählen => Refactor => Extract => Variable =>
Variable of type: String auswählen => Name eingeben (mode) => Replace
all occurances aktivieren => OK
```

Hier ein Beispiel wie das Ergebnis danach aussieht:

```
public String invoke() {

        if(pageData.hasAttribute("Test"))
        {
            String mode = "setup";
            if(includeSuiteSetup)
            {
               ...
               if(suiteSetup != null)
                {
                    ..
                buffer.append("!include -" + mode + " . ")
                .append(pagePathName).append("\n");
                }
            }
```

Des gleiche gilt für die Konstante "teardown"

```
Konstante ("teardown") auswählen => Refactor => Extract => Variable
=> Variable of type: String auswählen => Name eingeben (mode) =>
Replace all occurances aktivieren => OK
```

Hier das Ergebnis:

```
String mode = "teardown";
    if(teardown != null){
                ..
        buffer.append("!include -" + mode + " . ")
```

```
        .append(tearDownPathName).append("\n");
     }
    if(includeSuiteSetup)
    {
      ..
       {
         ..
          buffer.append("!include -" + mode + " . ")
          .append(pagePathName).append("\n");
       }
```

Aus der Parameterisierte Repeating Structure eine Methode Extrahieren.

Dies ist ein Spezialfall von dem P04: Extract Method aus Anweisungen. Und wird im nächsten Abschnitt behandelt.

Clean the Extracted Function

Diese Schritt dient dazu, die extrahierte Funktion hübscher und kompakter zu machen. Dazu zählen z.B zwei Sachen:

1. Geschwefelte Klammer { } entfernen, die nur eine einzige Anweisung haben.
2. String.format Methode statt + String benutzen

Hier ein Beispiel für {anweisung} entfernen. Aus dem folgenden Code

```
 if(suiteSetup != null)
                {
                    includePage(mode, suiteSetup);
                }
```

Wird nach Entfernung von { }

```
if(suiteSetup != null)
        includePage(mode, suiteSetup);
```

Hier ein Beispiel für String.format Aus dem folgenden Code:

```
buffer.append("!include -" + mode + " .")
.append(pagePathName).append("\n");
```

Wir der Code:

```
buffer.append(String.format("!include -%s .%s\n",
                                        mode, pagePathName))
```